Exercise 1: Particles in a box

(Made by: Sigurd Sørlie Rustad)
In this exercise we are going to simulate and animate particles in a box. We are going to do this in steps, so that the process becomes easier, but also to verify that our results makes sence. We will work with electrons inside a square box with sides L. Although we will be working in two dimensions, if you vectorize your code, expanding to 3D is quite straight forward.

a) Create a class `box`, and a `__init__` function that takes the amount of particles N, length of the side of the box L, the charge of the particles q, mass of the particles m and vacuum permeability ϵ_0. You also need to place the particles in the box, select their positions randomly inside the box with velocities selected from a normal distribution.

Hint. To place the particles inside the box you can use

```
import numpy as np
r = np.random.uniform(low, high, size)
v = np.random.uniform(loc, scale, size)
```

Solution.

```
class box:
    def __init__(self, N, L, q, m, eps0):
        """
        Set initial conditions
        """
        self.N = int(N)                     # num of particles
        self.L = L                          # length of side
        self.q = q                          # charge
        self.eps0 = eps0                    # permittivity of vacuum
        self.m = m                          # mass
        self.r = np.random.uniform(
            low=-L/2, high=L/2, size=(2, self.N)
        )                                   # initial positions
        self.v = np.random.normal(
            loc=0, scale=10, size=(2, self.N)
        )                                   # initial velocities
```

b) Now we want to create a function `force` that takes the current positions and returns the force on each particle. We are only looking at electric forces so it's sufficient to use Coulomb's law. It is possible to vectorize the code such that there are no loops. The solution shows a way to do it with one loop. NumPy contains many directories that are useful to vectorize code.

Solution.

```
def force(self, r = None):
    """
    find electric force between particles
    """
    #testing if we gave r a value
    if r is None:
        r_test = self.r
    else:
        r_test = r
    #force array we are going to fill
    F = np.zeros((2, self.N))
    #possible to drop this for loop
    for i in range(self.N):
        #calculating the force between every particle
        r_eval = r_test[:,i]
        ri = np.tile(r_eval, self.N-1)
        ri = np.reshape(ri, (self.N-1, 2)).T
        rm = np.delete(r_test, i, axis=1)
        r_mi = ri - rm
        F_mi = r_mi/np.linalg.norm(r_mi, axis=0)**2
        F[:, i] = np.sum(F_mi, axis=1)
    #need to add the natural constants
    F *= self.q**2/(4*np.pi*self.eps0)
    return F
```

c) To make sure the forces makes sence it can be a good idea visualize the forces. Make a vector plot using `quiver` from `matplotlib.pyplot`. Create a function `plot_arrows` that plots the force-arrows on each particle. In order to make the plot easier to understand plot the particles' positions and the borders of the box.

Hint. To plot points instead of lines you can use `plt.scatter`.

Solution.

```
def plot_border(self):
    """
    plot the box border
    """
    plt.plot([-self.L/2, self.L/2],
             [-self.L/2, -self.L/2], 'k')  # bottom side
    plt.plot([-self.L/2, self.L/2],
             [self.L/2, self.L/2], 'k')    # top side
    plt.plot([-self.L/2, -self.L/2],
             [-self.L/2, self.L/2], 'k')   # left side
    plt.plot([self.L/2, self.L/2],
```

```
            [-self.L/2, self.L/2], 'k',
            \label = 'Walls')              # right side
def plot_positions(self):
    """
    plot current positions as scatter-plot
    """
    plt.scatter(self.r[0,:], self.r[1,:], color='b', \label='Electrons')
def plot_arrows(self):
    """
    plot force arrows
    """
    F = self.force()
    plt.quiver(self.r[0,:], self.r[1,:], F[0, :], F[1, :], color='r', \label='Force')
```

d) Now we are ready to simulate the motion of the particles. First expand your `__init__` to take a small time step Δt and the period T you want to evaluate. Also define the number of steps N_{steps} this will require.

Solution. Expanding on your previous `__init__` function it should look something like this (changes are marked with `#!#`):

```
def __init__(self, N, L, q, m, eps0, dt, T):
    """
    Set initial conditions
    """
    self.N = int(N)                     # num of particles
    self.L = L                          # length of side
    self.q = q                          # charge
    self.eps0 = eps0                    # permittivity of vacuum
    self.m = m                          # mass
    self.T = T                          # time we are looking at #!#
    self.dt = dt                        # time step              #!#
    self.Nsteps = int(T/dt)                                      #!#
    self.r = np.random.uniform(
        low=-L/2, high=L/2, size=(2, self.N)
    )                                   # initial positions
    self.v = np.random.normal(
        loc=0, scale=10, size=(2, self.N)
    )
```

e) To find the motion of the particles it is good practice to separate the function that solves the differential equations from the numeric solver that solves for time. To solve the differential equations we are simply going to write a function `RHS` that takes a vector $r_0 = (r, v)$ containing positions and velocities and returns the derivative of that vector $r' = (r', v') = (v, a)$. You can assume elastic collision with the walls.

Hint. To find the particles hitting the walls you can use `np.where()`.

Solution. The code can look something like this:

```
def RHS(self, r0):
    """
    returns right hand side of the ode
    """
    #the array we will return
    drdt = np.zeros((2,2,self.N))
    #finding what particles are hitting the walls
    index = np.where(np.abs(r0[0, :])>self.L/2)
    #changing the sign of the velocity (elastic collision)
    r0[1][index] *= -1
    #placing the particles outside the box inside again
    np.where(r0[0, :] > self.L/2, r0[0, :], self.L/2)
    np.where(r0[0, :] < -self.L/2, r0[0, :], -self.L/2)
    #filling the array
    drdt[0, :, :] = r0[1, :]
    drdt[1, :, :] = self.force(r = r0[0, :])/self.m
    return drdt
```

f) Choosing what solver we are going to use is important and you should ponder a while on what you should prioritize. If the simulation is going on for a while should we prioritize energy-conservation? Write a function `solver` that solves and returns the motion of the particles. The solution solves the motion using Euler-Chromer's method, but you should try another one and compare.

Solution.

```
def solver(self):
    """
    solves the motions of the particles with Euler-Chromer
    """
    #the solution we will fill
    sol = np.zeros((self.Nsteps, 2, 2, self.N))
    #setting initial conditions
    sol[0, 0, :, :] = self.r[:, :]
    sol[0, 1, :, :] = self.v[:, :]
    for i in range(self.Nsteps-1):
        drdt = self.RHS(sol[i, :, :, :])
        sol[i+1, :, :] = drdt*dt + sol[i, :, :, :]
    #the time we evaluated
    t = np.linspace(0, self.T, self.Nsteps)
    return sol, t
```

g) Finally animate the motion of the particles. To do this you can use `matplotlib.animate`.

Solution.

```
#animation
sol, t = system.solver()
Nsteps = len(t)
#Creating the figure we are going to use
fig = plt.figure()
ax = plt.axes(xlim=(-L/2+0*1*L, L/2+0*1*L), ylim=(-L/2+0*1*L, L/2+0*1*L))
#the particles we are goint to plot
particles, = ax.plot([], [], 'bo', \label='Electrons')
#the walls
ax.plot([-L/2, L/2],
        [-L/2, -L/2], 'k')  # bottom side
ax.plot([-L/2, L/2],
        [L/2, L/2], 'k')    # top side
ax.plot([-L/2, -L/2],
        [-L/2, L/2], 'k')   # left side
ax.plot([L/2, L/2],
        [-L/2, L/2], 'k',
        \label = 'Walls')   # right side
def init():
    """
    init function that clears axis
    """
    ax.clear
    return particles
speed = 1
def animate(i):
    """
    this function animates
    """
    ax.set_title('Tid =%fs' %(i*dt*speed))
    particles.set_data(sol[i*speed, 0, :, :])
    return particles
#plot
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=int(Nsteps/speed), interval=1, blit=False)
plt.suptitle('Animation of particles', fontsize=14)
ax.set_y\label('Position y-axis [m]')
ax.set_x\label('Position x-axis [m]')
plt.axis('equal')
plt.legend(loc=1)
plt.show()
```

0.1 The entire code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as const
from matplotlib import animation
from matplotlib import style
style.use('seaborn') #this is just visual
class box:
    def __init__(self, N, L, q, m, eps0, dt, T):
        """
        Set initial conditions
        """
        self.N = int(N)                     # num of particles
        self.L = L                          # length of side
        self.q = q                          # charge
        self.eps0 = eps0                    # permittivity of vacuum
        self.m = m                          # mass
        self.T = T                          # time we are looking at ###
        self.dt = dt                        # time step              ###
        self.Nsteps = int(T/dt)                                      ###
        self.r = np.random.uniform(
            low=-L/2, high=L/2, size=(2, self.N)
        )                                   # initial positions
        self.v = np.random.normal(
            loc=0, scale=10, size=(2, self.N)
        )                                   # initial velocities
    def plot_border(self):
        """
        plot the box border
        """
        plt.plot([-self.L/2, self.L/2],
                 [-self.L/2, -self.L/2], 'k')  # bottom side
        plt.plot([-self.L/2, self.L/2],
                 [self.L/2, self.L/2], 'k')    # top side
        plt.plot([-self.L/2, -self.L/2],
                 [-self.L/2, self.L/2], 'k')   # left side
        plt.plot([self.L/2, self.L/2],
                 [-self.L/2, self.L/2], 'k',
                 \label = 'Walls')              # right side
    def plot_positions(self):
        """
        plot current positions as scatter-plot
        """
        plt.scatter(self.r[0,:], self.r[1,:], color='b', \label='Electrons')
    def force(self, r = None):
```

```
        """
        find electric force between particles
        """
        #testing if we gave r a value
        if r is None:
            r_test = self.r
        else:
            r_test = r
        #force array we are going to fill
        F = np.zeros((2, self.N))
        #possible to drop this for loop
        for i in range(self.N):
            #calculating the force between every particle
            r_eval = r_test[:,i]
            ri = np.tile(r_eval, self.N-1)
            ri = np.reshape(ri, (self.N-1, 2)).T
            rm = np.delete(r_test, i, axis=1)
            r_mi = ri - rm
            F_mi = r_mi/np.linalg.norm(r_mi, axis=0)**2
            F[:, i] = np.sum(F_mi, axis=1)
        #need to add the natural constants
        F *= self.q**2/(4*np.pi*self.eps0)
        return F
    def plot_arrows(self):
        """
        plot force arrows
        """
        F = self.force()
        plt.quiver(self.r[0,:], self.r[1,:], F[0, :], F[1, :], color='r', \label='Force')
    def RHS(self, r0):
        """
        returns right hand side of the ode
        """
        #the array we will return
        drdt = np.zeros((2,2,self.N))
        #finding what particles are hitting the walls
        index = np.where(np.abs(r0[0, :])>self.L/2)
        #changing the sign of the velocity (elastic collision)
        r0[1][index] *= -1
        #placing the particles outside the box inside again
        np.where(r0[0, :] > self.L/2, r0[0, :], self.L/2)
        np.where(r0[0, :] < -self.L/2, r0[0, :], -self.L/2)
        #filling the array
        drdt[0, :, :] = r0[1, :]
        drdt[1, :, :] = self.force(r = r0[0, :])/self.m
        return drdt
```

```
    def solver(self):
        """
        solves the motions of the particles with Euler-Chromer
        """
        #the solution we will fill
        sol = np.zeros((self.Nsteps, 2, 2, self.N))
        #setting initial conditions
        sol[0, 0, :, :] = self.r[:, :]
        sol[0, 1, :, :] = self.v[:, :]
        for i in range(self.Nsteps-1):
            drdt = self.RHS(sol[i, :, :, :])
            sol[i+1, :, :] = drdt*dt + sol[i, :, :, :]
        #the time we evaluated
        t = np.linspace(0, self.T, self.Nsteps)
        return sol, t
#defining constants
q = -const.e
m_e = const.m_e
eps0 = const.epsilon_0
dt = 1e-4
T = 1
N = 50
L = 1
#creating our box of particles
system = box(N=N, L=L, q=q, m=m_e, eps0=eps0, dt=dt, T=T)
#running som plot functions we made
system.plot_border()
system.plot_arrows()
system.plot_positions()
system.force()
#plot
plt.x\label('Position x-axis [m]')
plt.y\label('Position y-axis [m]')
plt.axis('equal')
plt.title('Positions and the force exerted on the particles')
plt.legend(loc=1)
plt.show()
#animation
sol, t = system.solver()
Nsteps = len(t)
#Creating the figure we are going to use
fig = plt.figure()
ax = plt.axes(xlim=(-L/2+0*1*L, L/2+0*1*L), ylim=(-L/2+0*1*L, L/2+0*1*L))
#the particles we are goint to plot
particles, = ax.plot([], [], 'bo', \label='Electrons')
#the walls
```

```
ax.plot([-L/2, L/2],
        [-L/2, -L/2], 'k')  # bottom side
ax.plot([-L/2, L/2],
        [L/2, L/2], 'k')    # top side
ax.plot([-L/2, -L/2],
        [-L/2, L/2], 'k')   # left side
ax.plot([L/2, L/2],
        [-L/2, L/2], 'k',
        \label = 'Walls')   # right side
def init():
    """
    init function that clears axis
    """
    ax.clear
    return particles
speed = 1
def animate(i):
    """
    this function animates
    """
    ax.set_title('Tid =%fs' %(i*dt*speed))
    particles.set_data(sol[i*speed, 0, :, :])
    return particles
#plot
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=int(Nsteps/speed), interval=1, blit=False)
plt.suptitle('Animation of particles', fontsize=14)
ax.set_y\label('Position y-axis [m]')
ax.set_x\label('Position x-axis [m]')
plt.axis('equal')
plt.legend(loc=1)
plt.show()
```