Ex8_ThermoACF-Copy1

January 19, 2021

1 Exercise 6 :

2 Lennard-Jones particles and Velocity Verlet integrator

- Léa Beaules
- 15 December 2020

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML
import random
import copy
import numpy as np
import scipy as scy
from tqdm.notebook import trange, tqdm
import time
from scipy.stats import maxwell

#used for the video
import subprocess
import glob
import os
```

2.1 Task I: Implementation of Berendsen thermostat

Berendsen thermostat (velocity rescaling factor)
Later set to T = 300 K with = 0.2 ps.

$$\lambda = \sqrt{1 + \frac{\Delta t}{\tau}(\frac{T_0}{T} - 1)}$$

[2]:
```
def Lambda_fact(Dlt_t, tau, Temp_o, Temp):
    lambda_f = np.sqrt( 1 + (Dlt_t/tau)*(Temp_o/Temp -1))
    return lambda_f
```

Maxwell Boltzmann distribution with a scale parameter $a = \sqrt{\frac{k_b T}{m}}$

```
[3]: def MaxBoltz ():
         a = np.sqrt(k_b * Temp_ini / mass)
         mx = maxwell(scale=a)
         x = np.linspace(mx.ppf(0.01),mx.ppf(0.99), 100)
         velocity_distribution  = mx.rvs(size=1000)
         return a, velocity_distribution
```

2.2 Task II: Force autocorrelation and friction coefficient calculation

Force autocorrelation function

With N out total number of steps in the simulation, we need $M + \tau$ to be smaller or equal to N. To avoid having 'overhanging steps' we sum only over $M = N - \tau$ steps.

$$C(\tau) = \ f(0)f(\tau) \ \ \frac{1}{M}\sum_{i}^{M}(f(t_i){-}\bar{f})(f(t_i+\tau){-}\bar{f})$$

```
[4]: def Force_t(t):
         Fx = 0
         Fy = 0
         for p_ind in range(Particule_Nbr):
             f = force_BIS(Relative_dist_arrays_X, Relative_dist_arrays_Y, t , p_ind)
             Fx += f[0]
             Fy += f[1]
         return [Fx,Fy]
```

```
[5]: def C_tau(tau, F_bar):
         M = steps - tau -1
         m = np.linspace(0,M, int(M/20))
         m = [int(m[i]) for i in range (len(m))]
         sum_value = 0

         count= 0
         for i in range(M):
             if i%20 == 0:
                 count +=1
                 sum_value +=␣
      ↪(Force_t(i)[0]-F_bar[0])*(Force_t(i+tau)[0]-F_bar[0])+(Force_t(i)[1]-F_bar[1])␣
      ↪* (Force_t(i+tau)[1]-F_bar[1])
         sum_value = sum_value / count

         return sum_value
```

Stokes friction coefficient

$$\Gamma = \frac{1}{k_b T}\int_0^{t_{end}} C(\tau)d\tau$$

```
[6]: #k_b = 8.314462 #J K^(-1).mol^(-1)
k_b = 1.380649e-23 #J.K-1
T = 300 #K
```

```
[7]: def Friction_coef(C_list):
    d_tau = time_list[3]-time_list[2]
    integral = 0
    for C in C_list :
        integral += C*d_tau
    gamma = integral/(k_b * T)
    return gamma
```

2.3 Task III : Simulation

Simulation of 49 particules in a 5x5nm box (with PBC).
The interparticle interaction is modeled as a Lennard-Jones potential.
The Velocity Verlet integrator is used to calculate the motion of the particles. The following constant are used :

```
[8]: box = (5,5)#nm²

#20000 total time steps in the simulation
steps = 500

#Number of particles in the box
Particule_Nbr = 49
mass = 18 #g/mol

#Time step (2*e-6 in nm)
Dlt_t = 2e-6 #ns = 2fs
tau = 2e-4 #ns = 0.2ps

k_b = 8.314462 #J K^(-1).mol^(-1)
Na = 6.02214086e23
Temp_o = 300 #K

#Temperature used for the initial Maxwell-Boltzmann velocity distribution
Temp_ini = 100 #K

#Constant used in the Lennard Jones potential
C_12 = 9.847044 *10**(-3) #kJ mol^-1 nm^12
C_6  = 6.2647225  #kJ mol^-1 nm^6
```

Position

$x_{k+1} = x_k + v_k \Delta t + \frac{1}{2} a_k \Delta t^2$

```python
[9]: def position (x_k, v_k, a_k, Dlt_t):
         x_k1 = x_k + v_k*Dlt_t + (1/2)*a_k*(Dlt_t**2)
         return x_k1
```

Velocity

$v_{k+1} = v_k + \frac{1}{2}(a_k + a_{k+1})\Delta t$

Maybe the units are wrong and my acceleration difference isso smallthat is does not affect my velocity

```python
[10]: def velocity (v_k, a_k, a_k1, Dlt_t):
          v_k1 = v_k + (1/2)*(a_k+a_k1)*Dlt_t
          return v_k1
```

Potential (Lennard Jones)

$V_{IJ}(r_{ij}) = \frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^6}$

```python
[11]: def potential(r_ij_vect):
          #distance between the two particules
          r_ij = np.sqrt(r_ij_vect[0]**2 + r_ij_vect[1]**2)

          if r_ij == 0:
              return 0
          else :
              V_ij = C_12/r_ij**(12) - C_6/r_ij**(6)
              V_ij = V_ij
              return V_ij
```

```python
[12]: def pot_total(time_t):
          V= 0

          for our_p_ind in range (Particule_Nbr):
              our_p = Particules_list[our_p_ind]
              for other_p_ind in range(our_p_ind+1 ,Particule_Nbr) :
                  other_p = Particules_list[other_p_ind]

                  r_ij_vect = get_vect_r(our_p,other_p)
                  V += potential(r_ij_vect)

          return V
```

```python
[13]: def pot_total_BIS(Relative_dist_arrays_X, Relative_dist_arrays_Y, t):
          V= 0

          for p_ind in range (Particule_Nbr):
              #list of relative distances along x and y with all the other particles␣
       ↪at time t
```

```
        r_vectx = Relative_dist_arrays_X[t][p_ind,:]
        r_vecty = Relative_dist_arrays_Y[t][p_ind,:]

        #Calculating the total potential on (p_ind)th particle
        for i in range (len(r_vectx)):
            V += potential([r_vectx[i],r_vecty[i]])

    return V
```

Relative distance between two particles, with periodic boundary conditions

- Using the particle class

[14]:
```
def get_vect_r(particul1, particul2):
    x_list =[]
    x_list += [particul2.x - particul1.x]
    x_list += [particul2.x - particul1.x + box[0]]
    x_list += [particul2.x - particul1.x - box[0]]
    x_part = min(x_list, key=abs)

    y_list =[]
    y_list += [particul2.y - particul1.y]
    y_list += [particul2.y - particul1.y + box[1]]
    y_list += [particul2.y - particul1.y - box[1]]
    y_part = min(y_list, key=abs)

    r = [x_part, y_part]

    return r
```

- Using x and y position of the two particles

[15]:
```
def get_vect_r_BIS(x,y, x2, y2):
    x_list =[]
    x_list += [x2 - x]
    x_list += [x2 - x + box[0]]
    x_list += [x2 - x - box[0]]
    x_part = min(x_list, key=abs)

    y_list =[]
    y_list += [y2 - y]
    y_list += [y2 - y + box[1]]
    y_list += [y2 - y - box[1]]
    y_part = min(y_list, key=abs)

    r = [x_part, y_part]

    return r
```

Force / acceleration

$f(t) = m(t) = mv(t) = ma(t)$

$F_{IJ}(r_{ij}) = (12\frac{C_{12}}{r_{ij}^{13}} - 6\frac{C_6}{r_{ij}^7})\frac{\vec{r_{ij}}}{r_{ij}}$

[16]:
```
def force_ij(r_ij_vect):
    #distance between the two particules
    r_ij = np.sqrt(r_ij_vect[0]**2 + r_ij_vect[1]**2)

    if r_ij != 0:
        factor = (12*C_12/r_ij**(13) - 6*C_6/r_ij**(7))/r_ij
        Fij_x = factor * r_ij_vect[0]
        Fij_y = factor * r_ij_vect[1]

        #Force given in J/mol*nm
        Fij_vect = [Fij_x, Fij_y]
        return Fij_vect
    else :
        return [0,0]
```

- Force on the (p_ind)th particle at time t using the Relative_dist_arrays

[17]:
```
def force_BIS(Relative_dist_arrays_X, Relative_dist_arrays_Y, t , p_ind):
    F= [0,0] #total force vector acting on our particle
    if t > (steps-1):
        return F[0,0]

    #list of relative distances along x and y with all the other particles
    r_vectx = Relative_dist_arrays_X[t][p_ind,:]
    r_vecty = Relative_dist_arrays_Y[t][p_ind,:]

    #Calculating the total force on (p_ind)th particle
    for i in range (len(r_vectx)):
        Fij_vect = force_ij([r_vectx[i],r_vecty[i]])

        #sum
        F[0] += Fij_vect[0]
        F[1] += Fij_vect[1]

    return F
```

Kinetic energy

$E_{kin} = \frac{1}{2}m < v^2 >$

[18]:
```
def Kinetic(Data_traj_list, time_t):
    K_list = []
    for p_ind in range (Particule_Nbr):
```

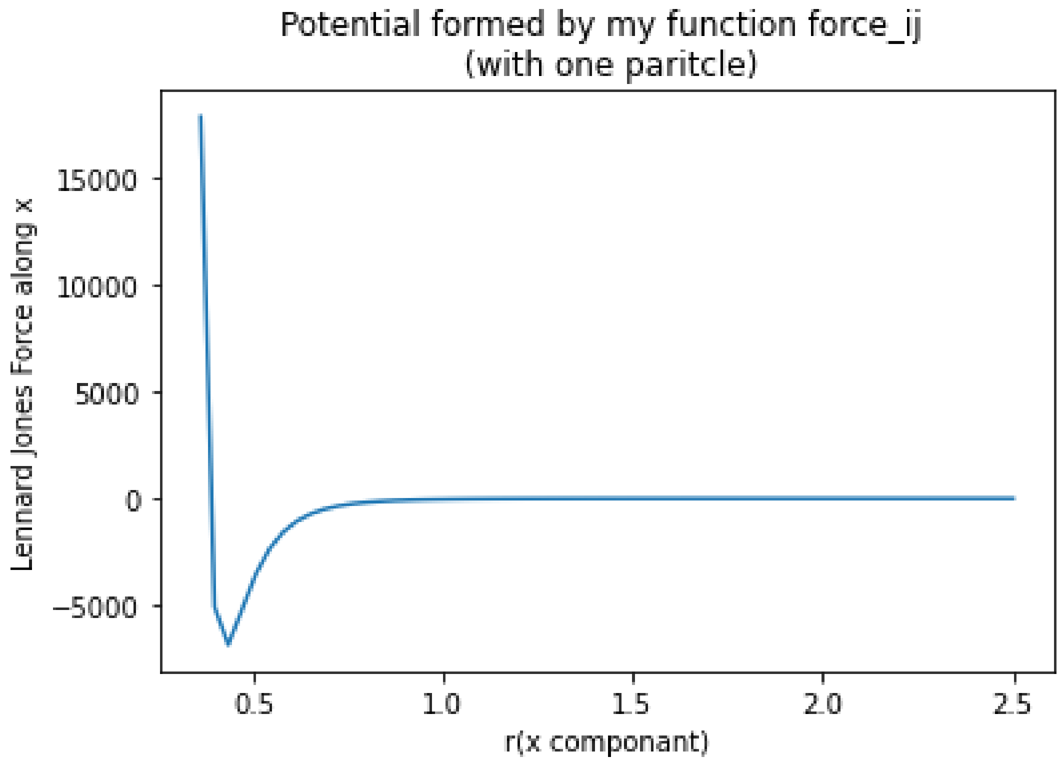
```
        p_vx = Data_traj_list[p_ind][2,time_t]
        p_vy = Data_traj_list[p_ind][3,time_t]
        v = np.sqrt(p_vx**2 + p_vy**2)
        K_list += [v**2]
    K = (1/2)*mass*np.mean(K_list)
    K = K * 10**(-3) #unit convertion into J/mol

    return K
```

Initialisation

- Check that the force function looks correct :

[19]:
```
a = np.linspace(0,2.5, 70)
f =[]
for i in range (70):
    f += [force_ij([a[i],0])[0]]

plt.plot(a[10:],f[10:])
plt.title('Potential formed by my function force_ij \n (with one paritcle)')
plt.ylabel('Lennard Jones Force along x')
u = plt.xlabel('r(x componant)')
mi = f.index(min(f)) #index of the minimum
print('There is a strong repulsive force if a particle gets closer then ',␣
 ↪a[mi], 'nm')
```

```
There is a strong repulsive force if a particle gets closer then
0.4347826086956522 nm
```



2.3.1 Relative distance between the particles

Creating a gereral list of arrays, with one array per time steps storing the relative distance of the particles. With r_{ij} being the vector r from particle i to particle j :

$$\begin{pmatrix} r_{11}=0 & r_{12} & \dots & r_{1N} \\ r_{21} & r_{22}=0 & \dots & r_{2N} \\ \dots & \dots & \ddots & \dots \\ r_{N1} & r_{N2} & \dots & r_{NN}=0 \end{pmatrix}_t$$

From the data traj list information it calculates the relative distance between all the particles of the system at the time t given in argument. And updates the gereral Relative_distance_arrays, adding the new arrays for time t

-Takes a bit of time to run-

[20]:
```
def Relative_dist(Relative_dist_arrays_X, Relative_dist_arrays_Y, t):

    Relative_dist_arrays_X += [np.zeros((Particule_Nbr,Particule_Nbr))]
    Relative_dist_arrays_Y += [np.zeros((Particule_Nbr,Particule_Nbr))]

    for p in range (Particule_Nbr):
        xp= Data_traj_list[p][0,:]
```

```
        yp= Data_traj_list[p][1,:]

        for p2 in range (p+1,Particule_Nbr):
            xp2 = Data_traj_list[p2][0,:]
            yp2 = Data_traj_list[p2][1,:]

            #vector r from particle p to particle p2
            r_vector = get_vect_r_BIS(xp[t],yp[t], xp2[t], yp2[t])
            Relative_dist_arrays_X[t][p,p2] = r_vector[0]
            Relative_dist_arrays_Y[t][p,p2] = r_vector[1]

            #opposite vector r from p2 to p
            Relative_dist_arrays_X[t][p2,p] = -r_vector[0]
            Relative_dist_arrays_Y[t][p2,p] = -r_vector[1]

    return (Relative_dist_arrays_X, Relative_dist_arrays_Y)


    #i_lower = np.tril_indices(n, -1)
    #matrix[i_lower] = matrix.T[i_lower]
```

Initialise simulation

The particles are placed on a regular grid of 7x7 particles. They are assigned their initial velocity $|v(x, y)|$ following the Maxwell-Boltzmann distribution at $T = 100K$, with random directions.

[21]:
```
class Particle:
    def __init__(self, x, y, vx, vy, ax, ay):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.ax = ax
        self.ay = ay

    def __repr__(self):
        return str("This is a particle at %0.2f, %0.2f with v=%0.2f,%0.2f" %␣
↪(self.x,self.y,self.vx,self.vy))
```

[22]:
```
def Particules_initialise (Particule_Nbr, steps):
    Particules_list = []
    Data_traj_list = []
    a, velocity_distribution = MaxBoltz()


    #regurlar grid 7*7
    nx, ny = (7, 7)
    x_grid = np.linspace(0, 5, nx+1)
```

```
    y_grid = np.linspace(0, 5, ny+1)

    d = x_grid[2]-x_grid[1]

    x_grid = x_grid[0:len(x_grid)-1] +d/2
    y_grid = y_grid[0:len(y_grid)-1] +d/2


    print('The space between two particle is : ', d, 'nm')

    p_count = 0
    for i in range (nx):
        for j in range (ny) :
            #Initial position on a uniform grid
            angle = np.random.uniform(0,2*np.pi)
            x_p = x_grid[i]
            y_p = y_grid[j]

            #Initial velocities : Maxwell-Boltzmann distribution with random␣
↪direction
            vel = velocity_distribution[np.random.randint(0,␣
↪len(velocity_distribution))]
            vx_p = np.sin(angle)*vel
            vy_p = np.cos(angle)*vel
            Particules_list += [Particle(x_p, y_p, vx_p, vy_p, 0, 0)]
            Data_traj_list += [np.zeros((4,steps))]

            Data_traj_list[p_count][:,0] = [x_p, y_p, vx_p, vy_p] #[particle␣
↪indice][data type, time step]

            p_count += 1
    print('Number of particles = ', p_count)
    return Particules_list, Data_traj_list
```

[23]:
```
Particules_list, Data_traj_list = Particules_initialise (Particule_Nbr, steps)
```

```
The space between two particle is :  0.7142857142857143 nm
Number of particles =  49
```

Run Simulation

At the start of the simulation we already have the initial positions, velocities and acceleration/force calculated. At each step we calculate :
FIRST LOOP : * Position at time t+1

Calculate all the relative distance at time t+1

SECOND LOOP : * Force at time t+1 * Velocities at time t+1

END OF TIME STEP * Rescaling the velocities with Berendsen thermostat

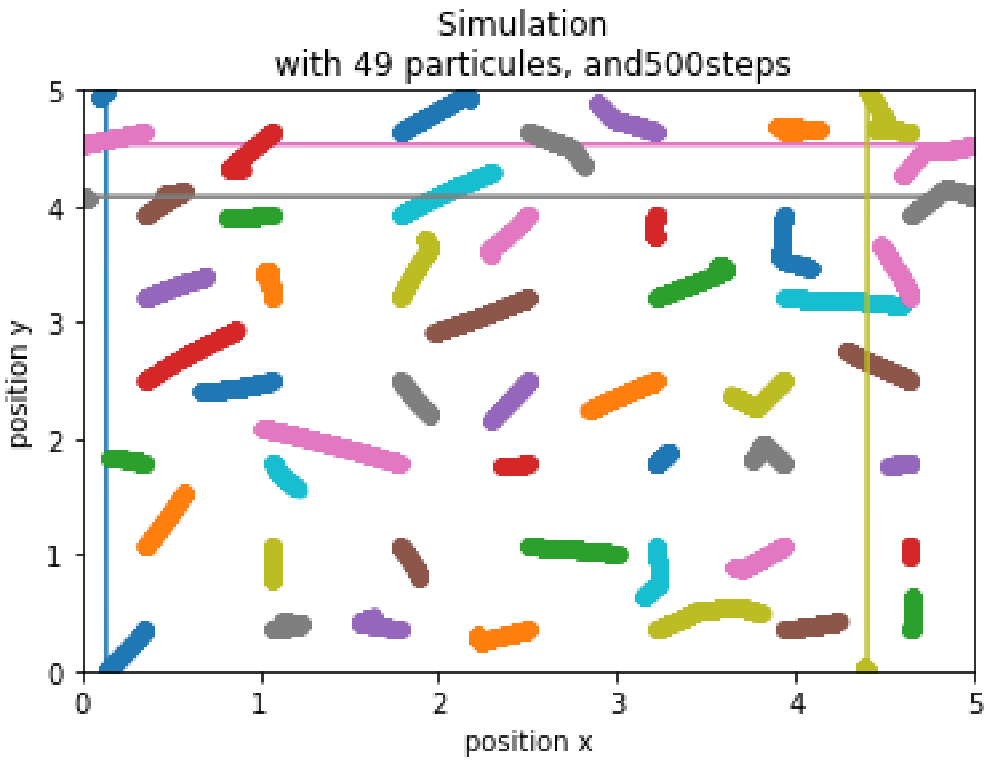
```
[24]: #arrays of relative distance between the particles
Relative_dist_arrays_X = []
Relative_dist_arrays_Y = []
Potential_list = [0]*steps

# At time = 0
Relative_dist_arrays_X, Relative_dist_arrays_Y =␣
 ↪Relative_dist(Relative_dist_arrays_X, Relative_dist_arrays_Y, 0)
Potential_list[0] = pot_total_BIS(Relative_dist_arrays_X,␣
 ↪Relative_dist_arrays_Y, 0)

for i in trange (steps-1):
    for p_ind in range (Particule_Nbr):
        our_P = Particules_list[p_ind]
        # calculating the next position
        x_1 = position (our_P.x, our_P.vx , our_P.ax, Dlt_t)%box[0]
        y_1 = position (our_P.y, our_P.vy, our_P.ay, Dlt_t)%box[1]
        # updating the particule position
        our_P.x = x_1
        our_P.y = y_1
        Data_traj_list[p_ind][0,i+1] = x_1
        Data_traj_list[p_ind][1,i+1] = y_1

    # Calculating the relative distance between all the particles at time i+1␣
 ↪---> the one that takes the most time !
    Relative_dist_arrays_X, Relative_dist_arrays_Y =␣
 ↪Relative_dist(Relative_dist_arrays_X, Relative_dist_arrays_Y, i+1)
    Potential_list[i+1] = pot_total_BIS(Relative_dist_arrays_X,␣
 ↪Relative_dist_arrays_Y, i+1)

    for ind in range (Particule_Nbr):
        our_P = Particules_list[ind]

        # calculating the force/acceleration at the next step
        F_1 = force_BIS (Relative_dist_arrays_X, Relative_dist_arrays_Y,i+1␣
 ↪,ind)
        ax_1, ay_1 = -F_1[0]*1000/(mass*1), -F_1[1]*1000/(mass*1)

        #velocity
        vx_1 = velocity (our_P.vx, our_P.ax, ax_1, Dlt_t)
        vy_1 = velocity (our_P.vy, our_P.ay, ay_1, Dlt_t)

        #updating velocity and acceleration
        our_P.ax = ax_1
        our_P.ay = ay_1
        our_P.vx = vx_1
        our_P.vy = vy_1
```

```
        Data_traj_list[ind][2,i+1] = vx_1
        Data_traj_list[ind][3,i+1] = vy_1

    #rescaling the velocities
    K = Kinetic(Data_traj_list, i+1)
    Temp = K / k_b
    lbda = Lambda_fact(Dlt_t, tau, Temp_o, Temp)
    for ind in range(Particule_Nbr):
        Rs_vx = (Data_traj_list[ind][2,i+1]) * lbda
        Rs_vy = (Data_traj_list[ind][3,i+1]) * lbda
        P = Particules_list[ind]

        P.vx, Data_traj_list[ind][2,i+1] = Rs_vx, Rs_vx
        P.vy, Data_traj_list[ind][3,i+1] = Rs_vy, Rs_vy
```

```
HBox(children=(FloatProgress(value=0.0, max=499.0), HTML(value='')))
```

[25]:
```
for i in trange(1,desc= 'Plot the graph of the last simulation'):

    #def plot_simulation (Particule_Nbr, Data_traj_list, Particules_list):
    for parti in range (Particule_Nbr):
        data_traj = Data_traj_list[parti]
        #Particule = Particules_list[parti]
        plt.plot(data_traj[0,:],data_traj[1,:], marker='.', markersize='10', ␣
 ↪linestyle = '-')

    plt.xlabel('position x')
    plt.ylabel('position y')
    plt.xlim(0,5)
    plt.ylim(0,5)

    Titles_graph1 = 'Simulation \n with ' + str(Particule_Nbr)+ ' particules,␣
 ↪and' + str(steps) + 'steps'
    plt.title(Titles_graph1)
```

```
HBox(children=(FloatProgress(value=0.0, description='Plot the graph of the last simulation', ma
```



- Analyse the force of the first particles coliding

[25]:
```
np.savez('03save_20.npz', Data_traj_list, Potential_list,␣
↪Relative_dist_arrays_X, Relative_dist_arrays_Y)
```

[26]:
```
#npzfile = np.load('save_array5.npz')
#Data_traj_list = npzfile['arr_0']
#Potential_list = npzfile['arr_1']
```

2.3.2 Video

[181]:
```
def creat_files():
    os.chdir("Image_storing_video")

    #def plot_simulation (Particule_Nbr, Data_traj_list, Particules_list):
    for t in trange (steps) :
        for parti in range (Particule_Nbr):
            data_traj = Data_traj_list[parti]
            plt.plot(data_traj[0,t],data_traj[1,t], marker='.',␣
↪markersize='10',  linestyle = '-', color = 'black')

            plt.xlabel('position x')
```

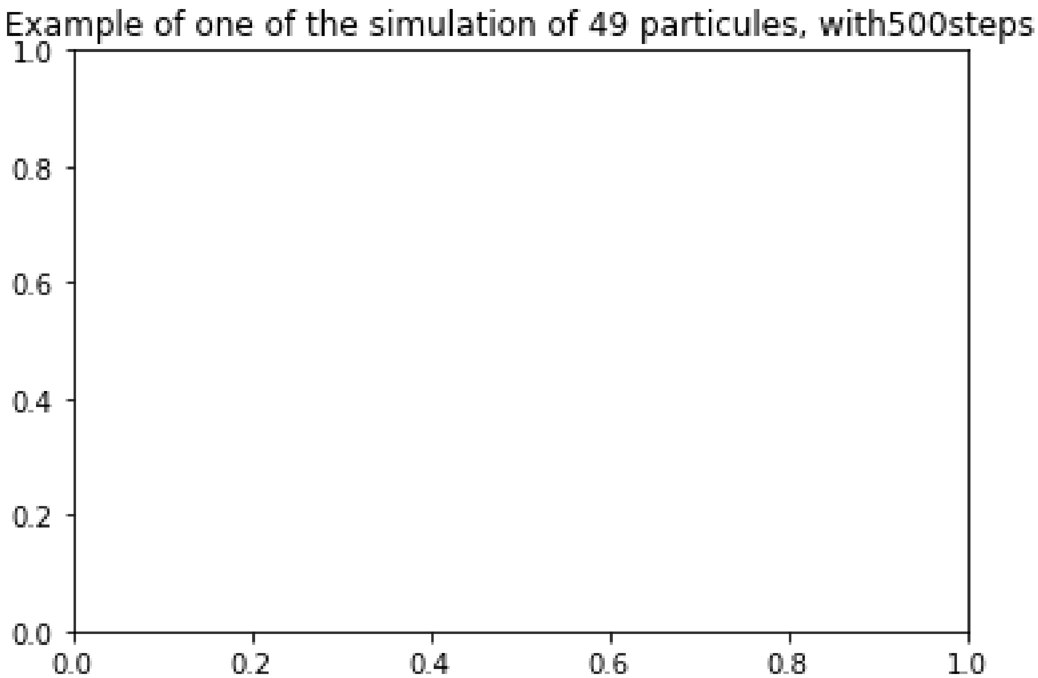
```
            plt.ylabel('position y')
            plt.xlim(0,box[0])
            plt.ylim(0,box[1])
        plt.savefig("File%02d.png" % t)
        #plt.show()
        #plt.savefig("file.png")
        plt.close()



    Titles_graph1 = 'Example of one of the simulation of ' +␣
↪str(Particule_Nbr)+ ' particules, with' + str(steps) + 'steps'
    plt.title(Titles_graph1)
    os.chdir("../")
```

[182]:
```
def creat_video():
    os.chdir("Image_storing_video")

    subprocess.call(['ffmpeg', '-framerate', '5', '-i', 'File%02d.png', '-r',␣
↪'30', '-pix_fmt', 'yuv420p', 'Contagion07.mp4'])
    #subprocess.call(['ffmpeg', '-framerate', '8', '-i', Titles_files, '-r',␣
↪'30', '-pix_fmt', 'yuv420p', Filename])


    for file_name in glob.glob("*.png"):
        os.remove(file_name)

    os.chdir("../")
```

[183]:
```
#os.chdir("ThermoACF")
retval = os.getcwd()
print ("Current working directory %s" % retval)
creat_files()
creat_video()
```

```
Current working directory
/home/lea/Bureau/Fac/Master/Simulating_the_physical_world/Simulations/ThermoACF
```

```
HBox(children=(FloatProgress(value=0.0, max=500.0), HTML(value='')))
```



2.4 Task III : Potential and Kinetic energy

[29]:
```
time_list_plot = np.linspace(0,steps,steps)
```

Total potential energy (per step)

Calculating the potential using the relative distance arrays

[26]:
```
from scipy.linalg import sqrtm
def pot (t):
    # All the other particles relative distances with the p_ind th particle at␣
↪time t
    r_vector_x = Relative_dist_arrays_X[t]
    r_vector_y = Relative_dist_arrays_Y[t]

    #Calculating the distance with all the other particuls (including over the␣
↪borders)
    rx_sqrd = np.linalg.matrix_power(r_vector_x , 2)
    ry_sqrd = np.linalg.matrix_power(r_vector_y , 2)
    norme_sqrd = np.add(rx_sqrd, ry_sqrd)
    radial_distance_list = sqrtm(norme_sqrd) # SQUARE ROOT
    #radial_distance_list = [np.sqrt(r_vector_x[i]**2 + r_vector_y[i]**2) for i␣
↪in range (len(r_vector_x))]

    RDist_12 = np.linalg.matrix_power(radial_distance_list, -12)*C_12
```

```
    RDist_6 = np.linalg.matrix_power(radial_distance_list, -6)*C_6

    pot = np.sum(np.subtract(RDist_12,RDist_6))

    return pot
```

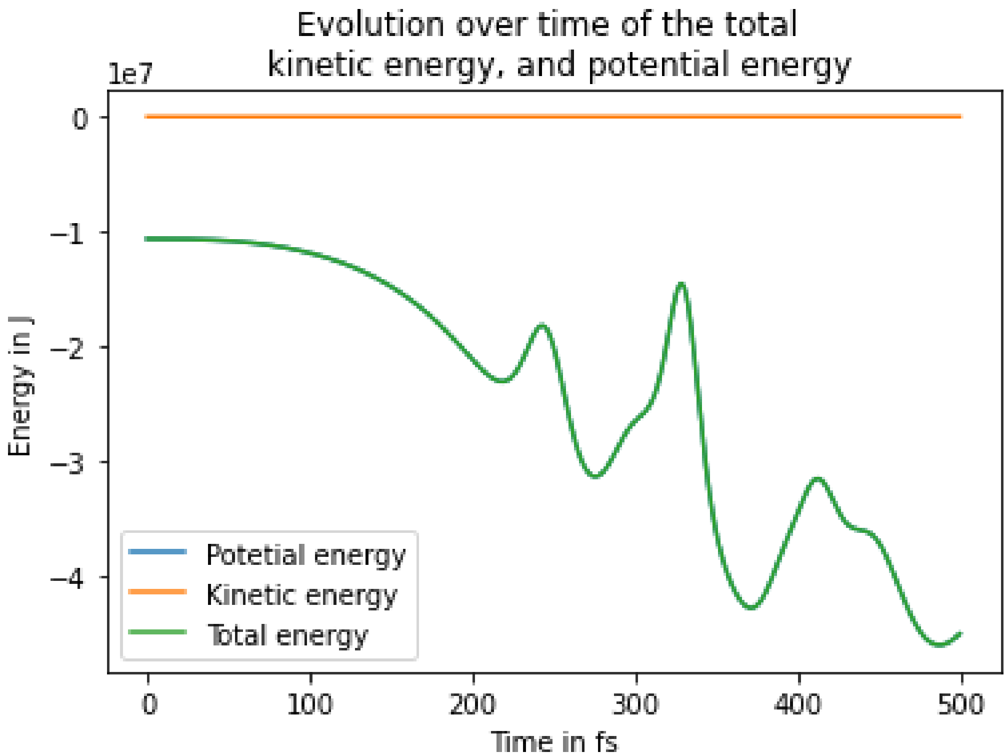
[27]:
```
#Potential_list = [0]*steps
#for t in trange(steps):
#    Potential_list[t] = [pot_total_BIS(Relative_dist_arrays_X,␣
 ↪Relative_dist_arrays_Y,t)]
```

```
HBox(children=(FloatProgress(value=0.0, max=500.0), HTML(value='')))
```

[33]:
```
plt.plot(time_list_plot, Potential_list, label='Potetial energy')
#plt.title('Evolution over time of the total potential energy')
#plt.ylabel('Potential energy in J')

Kin_list = []
Total = []
for i in trange (steps):
    k = Kinetic(Data_traj_list,i)
    Kin_list += [k]
    Total += [k+Potential_list[i]]

plt.plot(time_list_plot, Kin_list, label='Kinetic energy')
plt.plot(time_list_plot, Total, label='Total energy')

plt.title('Evolution over time of the total \n kinetic energy, and potential␣
 ↪energy')
plt.xlabel('Time in fs')
plt.ylabel('Energy in J')
plt.legend()
```

```
HBox(children=(FloatProgress(value=0.0, max=500.0), HTML(value='')))
```

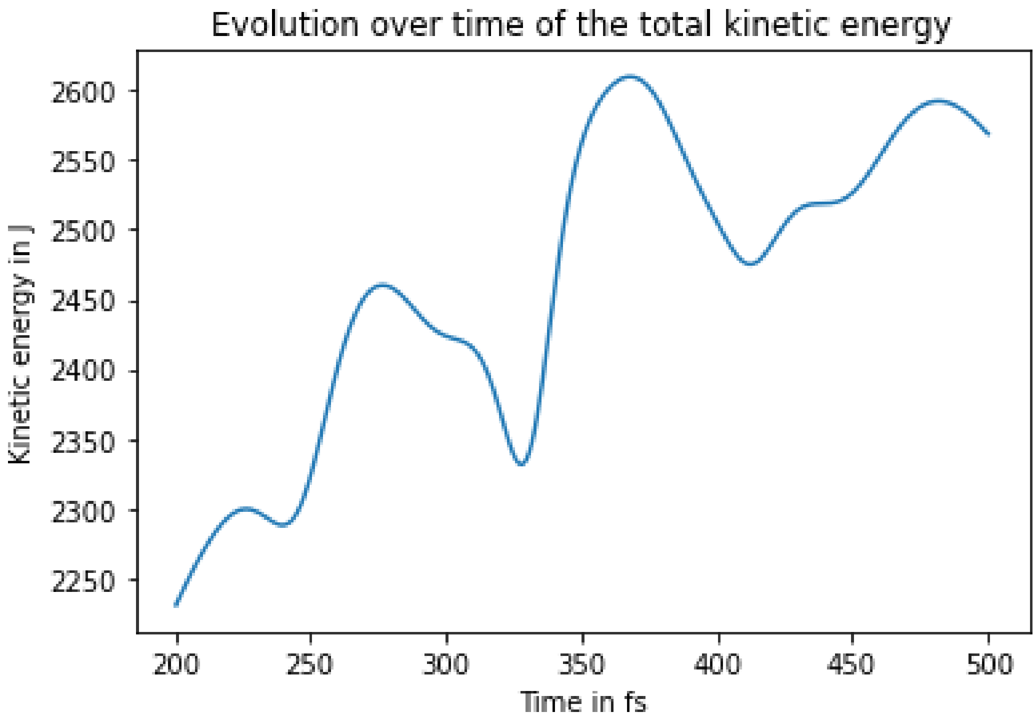
[33]:
```
<matplotlib.legend.Legend at 0x7f9e93049e20>
```



Total kinetic energy (per step)

[31]:
```
plt.plot(time_list_plot[200:], Kin_list[200:])
plt.title('Evolution over time of the total kinetic energy')
plt.xlabel('Time in fs')
plt.ylabel('Kinetic energy in J')
```

[31]:
```
Text(0, 0.5, 'Kinetic energy in J')
```



Interpretation of the results

There seems to be some problems in the calculation, as the Kinetic and potential energy do have similar and opposite shapes but they are in different order of magnitude.

2.5 Task V: Calculation and interpretation of friction coefficient

- Calculating f bar

[34]:
```
Fx_bar_list = [0]*steps
Fy_bar_list = [0]*steps

for t in trange(steps):
    Fx_bar_list[t] = Force_t(t)[0]
    Fy_bar_list[t] = Force_t(t)[1]
Fx_bar = np.mean(Fx_bar_list)
Fy_bar = np.mean(Fy_bar_list)

F_bar = [Fx_bar, Fy_bar]

print('The mean force over the fullsimulation', F_bar)
```

```
HBox(children=(FloatProgress(value=0.0, max=500.0), HTML(value='')))
```

The mean force over the fullsimulation [2.1040473148492557e-12,
-1.14104750648103e-12]

- Calculating $C(\tau)$

[35]:
```
time_list = []
C_list = []

count = 0
for tau in trange(0, int(3*steps/4)):
    #I had running time issues so I collected 1 out of 10 values for tau < 200
    # and 1 out of 20 values other tau
    if tau%20==0:
        count += 1
        C_list += [C_tau(tau, F_bar)]
        time_list += [tau]
```

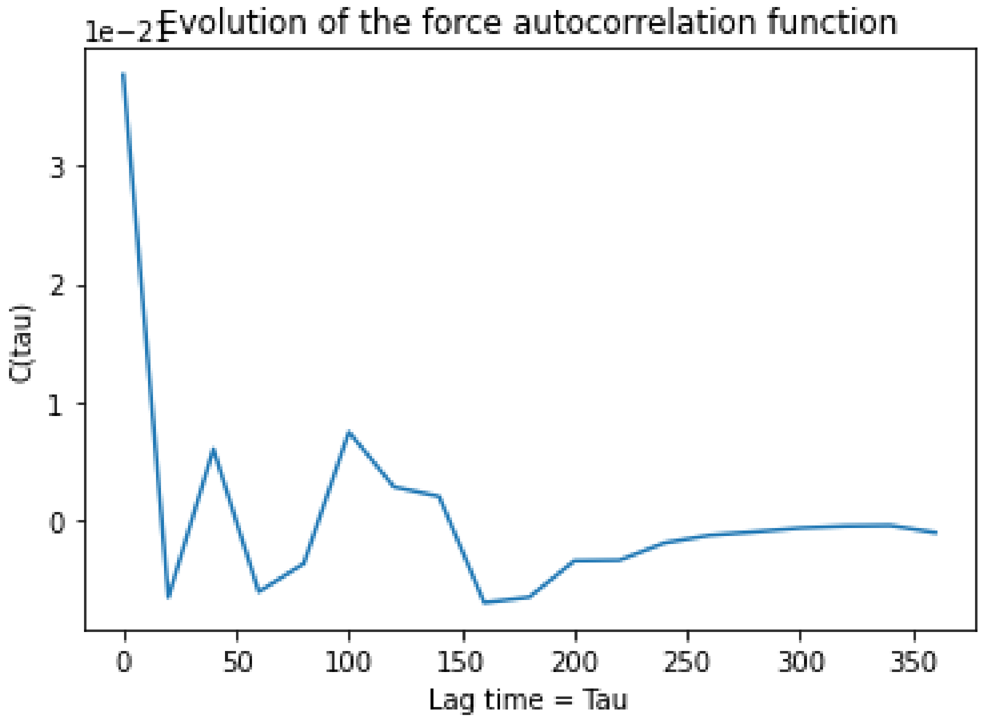
```
HBox(children=(FloatProgress(value=0.0, max=375.0), HTML(value='')))
```

[36]:
```
np.savez('02save_Ctau.npz', C_list, time_list)
np.savez('02save_Fbar.npz', F_bar)
```

I had issues with running the code on my computer so I ran it using the code in the above cells and saved the results in arrays.

[ ]:
```
#C_tau5000_File = np.load('save_Ctau03_5000.npz')
#C_tau5000to15000_File = np.load('save_Ctau04_5000to15000.npz')
#C_tau_list = C_tau5000_File['arr_0']+C_tau5000to15000_File['arr_0']
#time_list_C_tau = C_tau5000_File['arr_1']+C_tau5000to15000_File['arr_1']
```

[37]:
```
plt.plot(time_list, C_list)
plt.title('Evolution of the force autocorrelation function')
plt.xlabel('Lag time = Tau')
a = plt.ylabel('C(tau)')
```



[38]:
```
gamma = Friction_coef(C_list)
print(gamma)
```

1.042490541658736e-23

[ ]:

[ ]:

[ ]:

[ ]: