Simulation of Complex system HW-1 Molecular dynamics

Purusothaman Seenivasan

November 2023

1 Molecular Dynamics

1.1 Harmonic oscillator simulated using the Euler algorithm and Leapfrog method

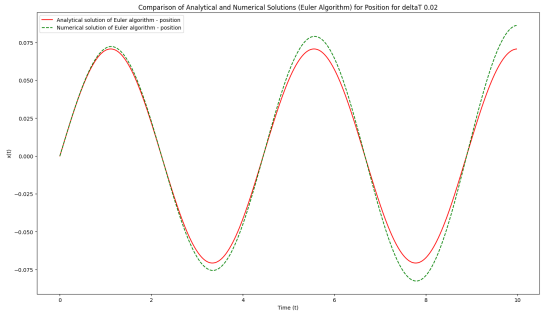
1.1.1 Coding

```
from matplotlib import pyplot as plt
import numpy as np

intial_position = 0.0
intial_velocity = 0.1
time = 0
delta_t = 0.002
spring_constant = 2  #k
mass = 1

time_list = []
time_list.append(time)
iteration = int(10 / delta_t)

positions = np.zeros(iteration)
velocity = np.zeros(iteration)
positions[0] = intial_position
velocity[0] = intial_velocity

total_energy = np.zeros(iteration)

for i in range(iteration - 1):
  omega_square = spring_constant / mass
  a = -omega_square * positions[i]
  positions[i + 1] = positions[i] + velocity[i] * delta_t
  # velocity = velocity + a*delta_t
  velocity[i + 1] = velocity[i] + a * delta_t

  time = time + delta_t
  time_list.append(time)

for i in range(iteration):
  kinetic_energy = 0.5 * mass * velocity[i]**2
  potential_energy = 0.5 * spring_constant * positions[i]**2
  total_energy[i] = kinetic_energy + potential_energy

#----------Leap frog method--------------

lf_position = np.zeros(iteration)
lf_velocity = np.zeros(iteration)
lf_total_energy = np.zeros(iteration)

lf_position[0] = intial_position
lf_velocity[0] = intial_velocity
```

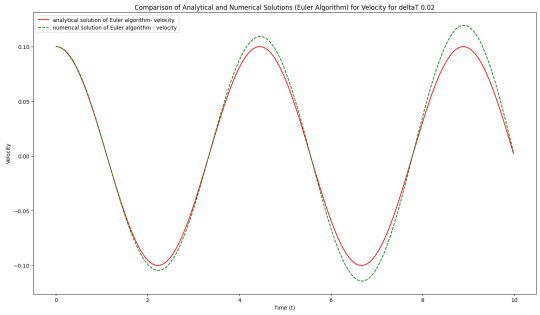
```
print('intial', lf_position)
for i in range(iteration - 1):
  middle_step = lf_position[i] + lf_velocity[i] * delta_t / 2
  print('middle', middle_step)
  a = -(spring_constant / mass) * middle_step
  lf_velocity[i + 1] = lf_velocity[i] + a * delta_t
  lf_position[i + 1] = middle_step + lf_velocity[i + 1] * delta_t / 2

for i in range(iteration):
  lf_kinetic_energy = 0.5 * mass * lf_velocity[i]**2
  lf_potential_energy = 0.5 * spring_constant * lf_position[i]**2
  lf_total_energy[i] = lf_kinetic_energy + lf_potential_energy

continous_time = np.arange(0, 10, 0.01)
print(len(continous_time))

r0 = 0.1
v0 = 0
omega_alalytical = np.sqrt(spring_constant / mass)
a_analytical = np.sqrt(intial_position**2 +
                       (intial_velocity / omega_alalytical)**2)
phi = np.arctan2((-intial_velocity / (a_analytical * omega_alalytical)),
                 (intial_position / a_analytical))

total_energy_analytical_value = []
functionvalues = []
velocity_analytical_list = []

for t in time_list:
  position_function = a_analytical * (np.cos(omega_alalytical * t + phi))
  functionvalues.append(position_function)

  velocity_function = -omega_alalytical * a_analytical * np.sin(
      omega_alalytical * t + phi)
  velocity_analytical_list.append(velocity_function)

  potential_energy_analytical = 0.5 * mass * omega_alalytical**2 * a_analytical**2 * np.sin(
      omega_alalytical * t + phi)**2
  kinetic_energy_analytical = 0.5 * spring_constant * a_analytical**2 * np.cos(
      omega_alalytical * t + phi)**2
  total_energy_analytical = potential_energy_analytical + kinetic_energy_analytical
  # total_energy_analytical = 0.5*spring_constant*a_analytical**2
  total_energy_analytical_value.append(total_energy_analytical)

plt.plot(time_list,
         functionvalues,
         color='r',
         linestyle='-',
         label='analytical solution of Euler algorithm- position')
plt.plot(time_list,
         positions,
         color='g',
         linestyle='--',
         label='numerical solution of Euler algorithm - position')
plt.legend()
plt.show()

plt.plot(time_list,
         velocity_analytical_list,
         color='r',
```

```
        linestyle='-',
        label='analytical solution of Euler algorithm- velocity')
plt.plot(time_list,
        velocity,
        color='g',
        linestyle='--',
        label='numerical solution of Euler algorithm - velocity')
plt.legend()
plt.show()

plt.plot(time_list,
        total_energy,
        color='r',
        linestyle='-',
        label='analytical solution of Euler algorithm - Total Energy')
plt.plot(time_list,
        total_energy_analytical_value,
        color='g',
        linestyle='--',
        label='numerical solution of Euler algorithm - Total Energy')
plt.legend()
plt.show()

plt.plot(time_list,
        positions,
        color='b',
        linestyle='dotted',
        label='numerical solution of Euler algorithm - position')
plt.plot(time_list,
        functionvalues,
        color='r',
        linestyle='-',
        label='analytical solution- position')
plt.plot(time_list,
        lf_position,
        color='g',
        linestyle='--',
        label='numerical solution of leaf frog algorithm - position')
plt.legend()
plt.show()

plt.plot(time_list,
        total_energy,
        color='b',
        linestyle='dotted',
        label='analytical solution of Euler algorithm - Total Energy')
plt.plot(time_list,
        lf_total_energy,
        color='r',
        linestyle='-',
        label='analytical solution of Leaf frog algorithm - Total Energy')
plt.plot(time_list,
        total_energy_analytical_value,
        color='g',
        linestyle='--',
        label='numerical solution of Euler algorithm - Total Energy')
plt.legend()
plt.show()
```

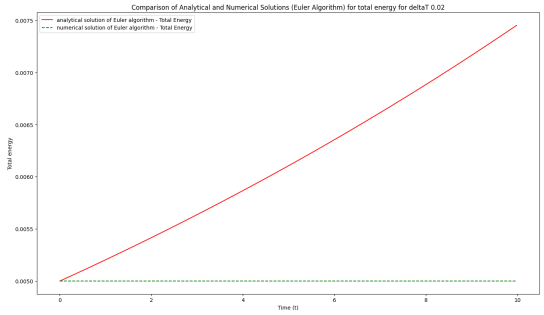
1.1.2 Results: Trajectory, velocity and total energy of particle at delta T = 20ms by Euler's method and by Leap frog method



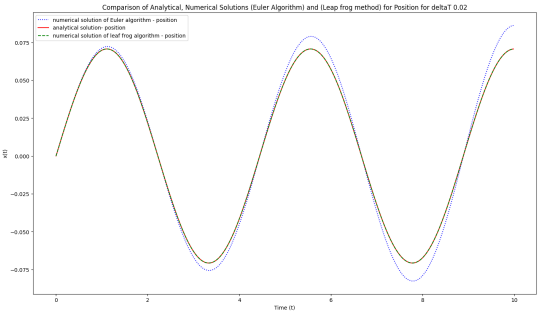
(a) Euler Position



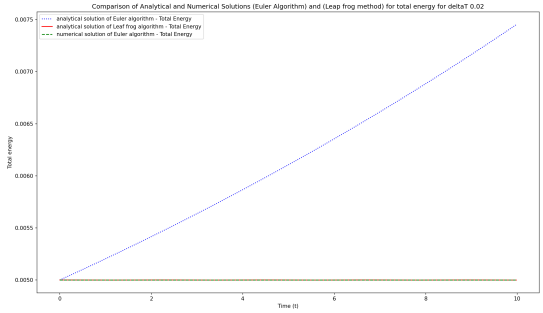
(b) Euler Velocity



(c) Euler TE



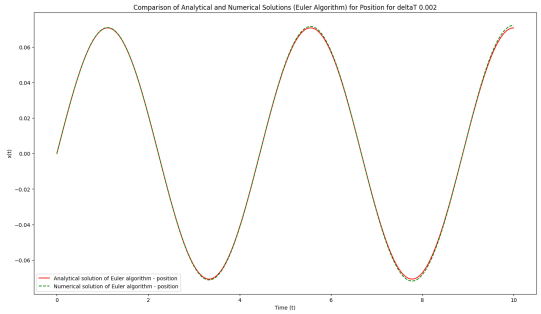
(d) Leap Position



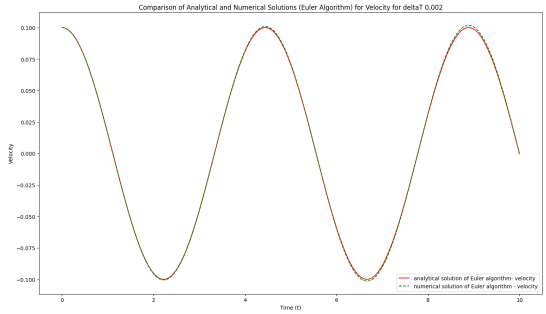
(e) Leap TE

Figure 1: Comparison of Euler and Leapfrog methods and analytical solution at 20ms

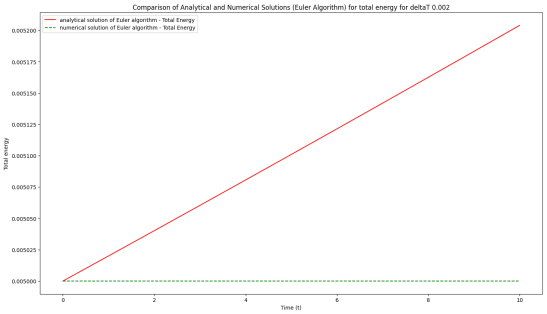
1.1.3 Results: Trajectory, velocity and total energy of particle at delta T = 2ms by Euler's method and by Leap frog method



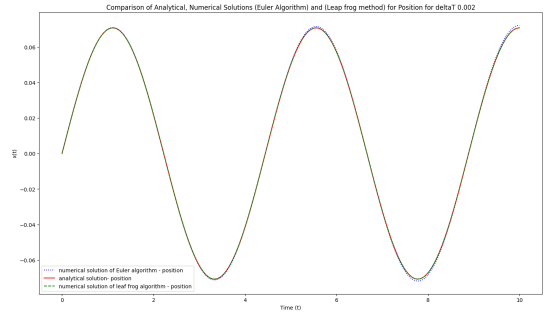
(a) Euler Position



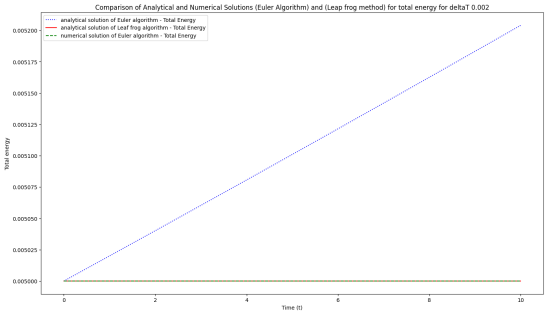
(b) Euler Velocity



(c) Euler TE



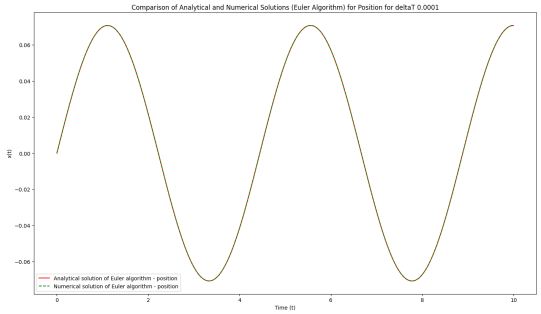
(d) Leap Position



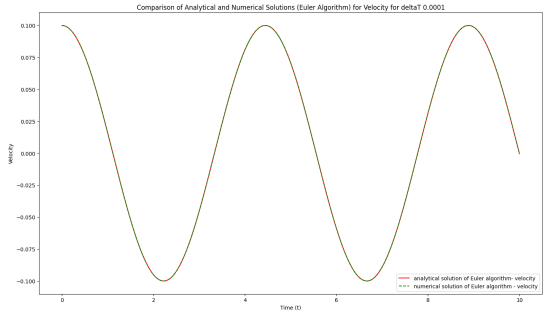
(e) Leap TE

Figure 2: Comparison of Euler and Leapfrog methods and analytical solution at 2ms

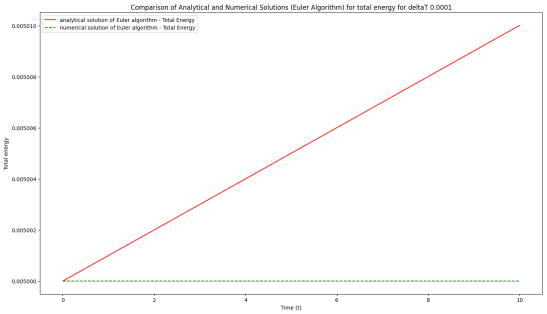
1.1.4 Results: Trajectory, velocity and total energy of particle at delta T = 0.1ms by Euler's method and by Leap frog method



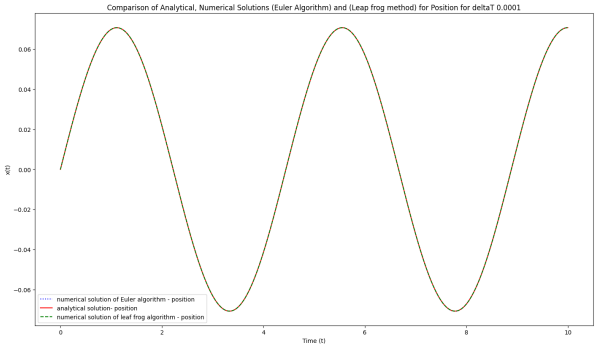
(a) Euler Position



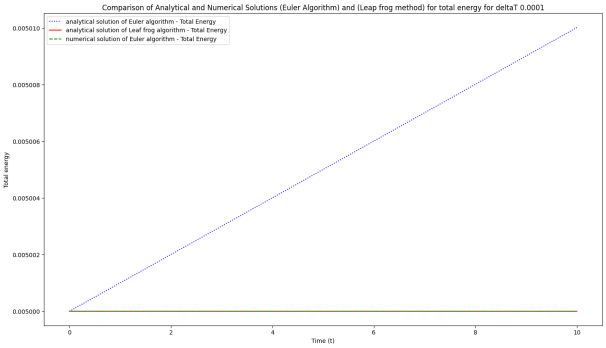
(b) Euler Velocity



(c) Euler TE



(d) Leap Position



(e) Leap TE

Figure 3: Comparison of Euler and Leapfrog methods and analytical solution at 0.1ms

Total mechanical energy is conserved by the analytical method and by the leapfrog method since total energy remains constant whereas, in Euler's method, the total energy is not conserved

2 Two Dimensional gas in a box

2.1 Program

```
import numpy as np
import random
from matplotlib import pyplot as plt

e0 = 0.001
sigma0 = 1
mass0 = 1

velocity0 = np.sqrt((2 * e0) / mass0)
t0 = sigma0 * np.sqrt(mass0 / (2 * e0))

deltaT = 0.001 * t0  #0.001/5
iteration = 100  #int(10000*deltaT)
print(iteration)
boundary = [0, 100 * sigma0]

n = 100
int_velocity = np.full((n, 2), 2 * velocity0)


def distance(point1, point2):
  return np.linalg.norm(point1 - point2)


int_positions = []
while len(int_positions) < n:
  new_position = np.random.uniform(boundary[0], boundary[1], size=2)
  valid = True
  for existing_position in int_positions:
    if distance(new_position, existing_position) < sigma0:
      valid = False
      break
  if valid:
    int_positions.append(new_position)

for i in range(n):
  angle = random.random() * 2 * np.pi
  int_velocity[i][0] = int_velocity[i][0] * np.cos(angle)
  int_velocity[i][1] = int_velocity[i][1] * np.sin(angle)

int_positions = np.array(int_positions)
print(int_positions[:][0])
plt.xlim(0, boundary[1])
plt.ylim(0, boundary[1])
plt.scatter(int_positions[:, 0], int_positions[:, 1])
plt.quiver(int_positions[:, 0],
           int_positions[:, 1],
           int_velocity[:, 0],
           int_velocity[:, 1],
           width=0.002)
plt.title('Initial Position and Velocity')
plt.show()


def distance_between(m):
  d = np.zeros((len(m), len(m)))
```

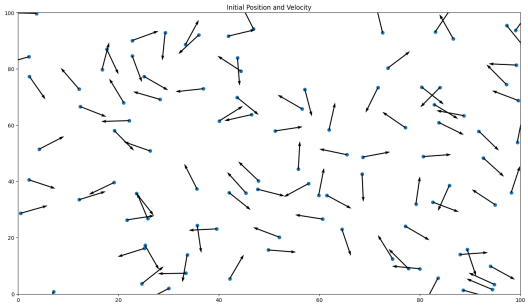
```
  for i in range(len(m)):
    for j in range(len(m)):
      if i != j:
        d[i, j] = distance(m[i], m[j])
  return d


def lennard_jones_force(positions):
  forces = np.zeros_like(positions)
  for i in range(n):
    for j in range(i + 1, n):
      r = np.linalg.norm(positions[i] - positions[j])
      direction = (positions[j] - positions[i]) / r
      magnitude = 4 * e0 * ((sigma0**12 / r**13) - (sigma0**6 / r**7))
      forces[i] -= magnitude * direction
      forces[j] += magnitude * direction
  return forces


def lennard_jones_potential(positions):
  v = np.zeros_like(positions)
  for i in range(n):
    for j in range(i + 1, n):
      r = np.linalg.norm(positions[i] - positions[j])
      direction = (positions[j] - positions[i]) / r
      magnitude = 4 * e0 * ((sigma0**12 / r**12) - (sigma0**6 / r**6))
      v[i] += magnitude * direction
      v[j] -= magnitude * direction
  return v


def potential(d):
  v = np.zeros((len(d), len(d)))
  for i in range(int(len(d))):
    for j in range(len(d)):
      if d[i][j] != 0:
        v[i][j] = 4 * e0 * ((sigma0 / d[i][j])**12 - (sigma0 / d[i][j])**6)
      else:
        v[i][j] = 0
  return v


positions = int_positions
velocity = int_velocity

final_KE = np.zeros((iteration, 1))
final_PE = np.zeros((iteration, 1))

final_TE = np.zeros((iteration, 1))
int_KE = 0
int_PE = 0
int_TE = 0
time_list = []
temp_time = 0
position_list = []
for t in range(iteration):
  temp_time = temp_time + deltaT
  time_list.append(temp_time)

  nxt_position = np.zeros((n, 2))
```

```
nxt_velocity = np.zeros((n, 2))
middle_step = np.zeros((n, 2))
d = distance_between(positions)
v = lennard_jones_potential(positions)

temp_KE = 0
temp_PE = 0
temp_TE = 0
for i in range(n):
  temp_KE += 0.5 * mass0 * (np.linalg.norm(velocity[i])**2)
  temp_PE += 0.5 * np.linalg.norm(v[i])**2

temp_TE = temp_PE + temp_KE
final_KE[t] = temp_KE
final_PE[t] = temp_PE
final_TE[t] = temp_TE

for i in range(n):
  middle_step[i] = positions[i] + velocity[i] * deltaT / 2

force_middle = lennard_jones_force(middle_step)

d_middle = distance_between(middle_step)
for i in range(n):
  nxt_velocity[i] = velocity[i] + force_middle[i] * deltaT / mass0
  nxt_position[i] = middle_step[i] + nxt_velocity[i] * deltaT / 2

  if nxt_position[i][0] < boundary[0]:
    difference = boundary[0] - nxt_position[i][0]
    nxt_position[i][0] = difference
    nxt_velocity[i][0] = -nxt_velocity[i][0]
  if nxt_position[i][0] > boundary[1]:
    difference = nxt_position[i][0] - boundary[1]
    nxt_position[i][0] = boundary[1] - difference
    nxt_velocity[i][0] = -nxt_velocity[i][0]

  if nxt_position[i][1] < boundary[0]:
    difference = boundary[0] - nxt_position[i][1]
    nxt_position[i][1] = difference
    nxt_velocity[i][1] = -nxt_velocity[i][1]
  if nxt_position[i][1] > boundary[1]:
    difference = nxt_position[i][1] - boundary[1]
    nxt_position[i][1] = boundary[1] - difference
    nxt_velocity[i][1] = -nxt_velocity[i][1]
positions = nxt_position
velocity = nxt_velocity
position_list.append(positions)
position_list = np.array(position_list)
if t % 1000 == 0:
  for i in range(100):
    x = position_list[:, i, 0]
    y = position_list[:, i, 1]
    plt.plot(x, y, label=f"Particle {i+1}")
  plt.scatter(positions[:, 0], positions[:, 1])
  plt.quiver(
      positions[:, 0],
      positions[:, 1],
      velocity[:, 0],
      velocity[:, 1],
      color='black',
```

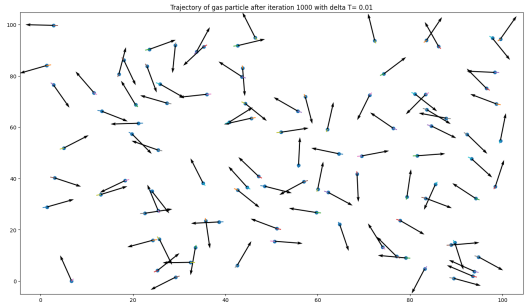
```
        width=0.002,
    )
    plt.title(
        f'Trajectory of gas particle after iteration {t} with delta T= {deltaT}'
    )
    plt.show()

    # for i in range (n):
    #     plt.subplot(121)
    #     plt.scatter(int_positions[i][0],int_positions[i][1],color='r')
    # plt.quiver(int_positions[:, 0], int_positions[:, 1], int_velocity[:, 0], int_velocity[:, 1], color='g',
    # plt.legend()
    # plt.title(f'inital position of paticles')

    # for i in range (n):
    #     plt.subplot(122)
    #     plt.scatter(positions[i][0],positions[i][1],color='r')
    # plt.quiver(positions[:, 0], positions[:, 1], velocity[:, 0], velocity[:, 1], color='r', label= f'velocit
    # plt.legend()
    # plt.title(f'position of patricles after iteration:{t} with time step= {deltaT}')
    # plt.show()
  position_list = position_list.tolist()

position_list = np.array(position_list)
for i in range(100):
  x = position_list[:, i, 0]
  y = position_list[:, i, 1]
  plt.plot(x, y, label=f"Particle {i+1}")
plt.scatter(positions[:, 0], positions[:, 1])
plt.quiver(
    positions[:, 0],
    positions[:, 1],
    velocity[:, 0],
    velocity[:, 1],
    color='black',
    width=0.002,
)
plt.title(
    f'Trajectory of gas particle after iteration {iteration} with delta T= 0.01*t0'
)
plt.show()

fig, ax = plt.subplots(3, 1)
ax[0].plot(time_list, final_KE)
ax[0].set_title('Kinetic Eenergy')
ax[1].plot(time_list, final_PE)
ax[1].set_title('Potential Energy')
ax[2].plot(time_list, final_TE)
ax[2].set_title('Total Energy')
plt.tight_layout()

plt.show()
```

2.2 Results

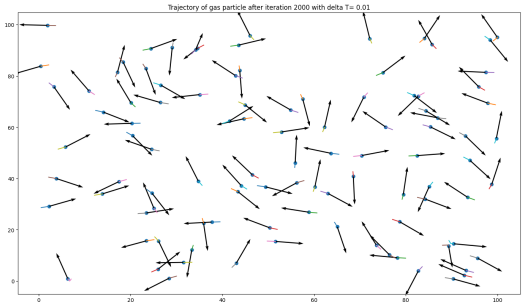
2.2.1 Results: plotting of Instantaneous positions and velocity



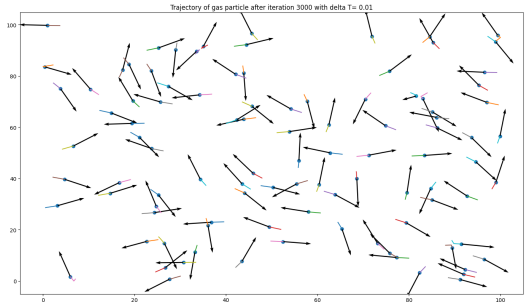
(a) Initial position and velocity of 100 particles



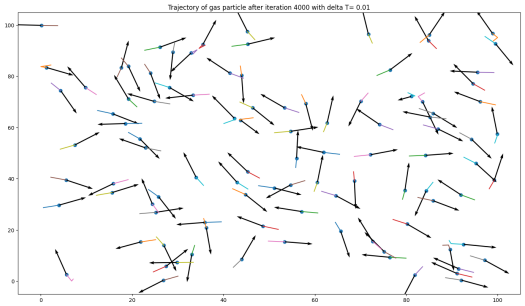
(b) Position and velocity after 1000 iterations



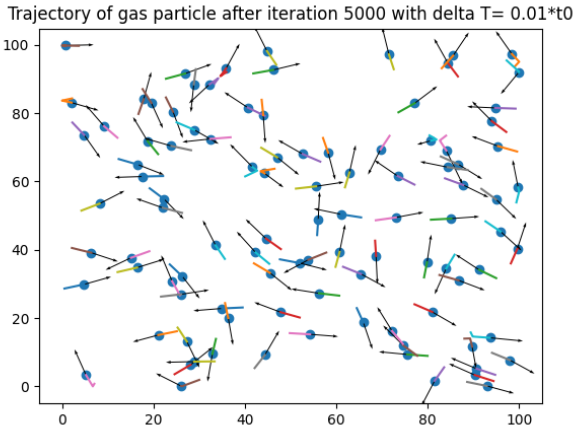
(c) Position and velocity after 2000 iterations



(d) Position and velocity after 3000 iterations



(e) Position and velocity after 4000 iterations



(f) Position and velocity after 5000 iterations

Figure 4: Trajectory images at different iteration steps

2.2.2 Results: Initial and final position and velocity of 100 gas particle and its trajectory when delta t = 0.01*t0

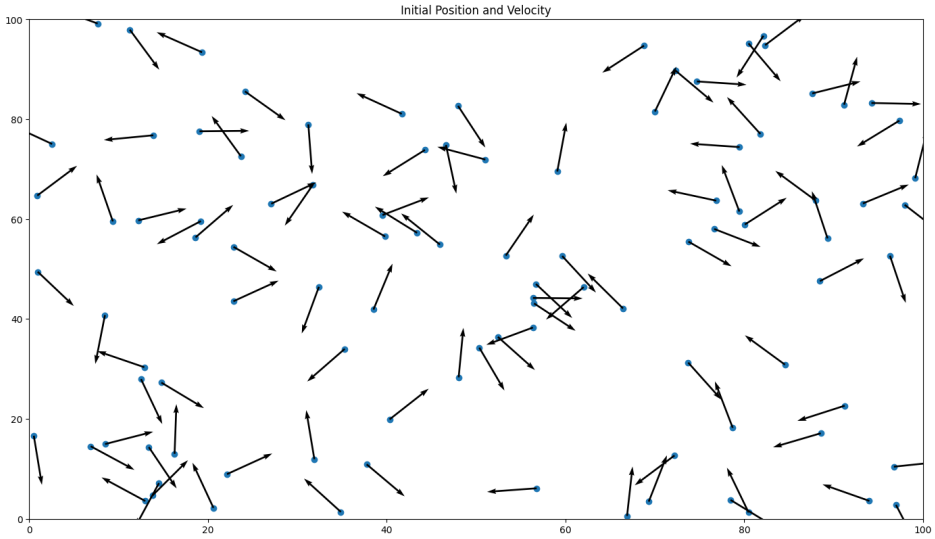


Figure 5: Initial position and velocity of 100 particles

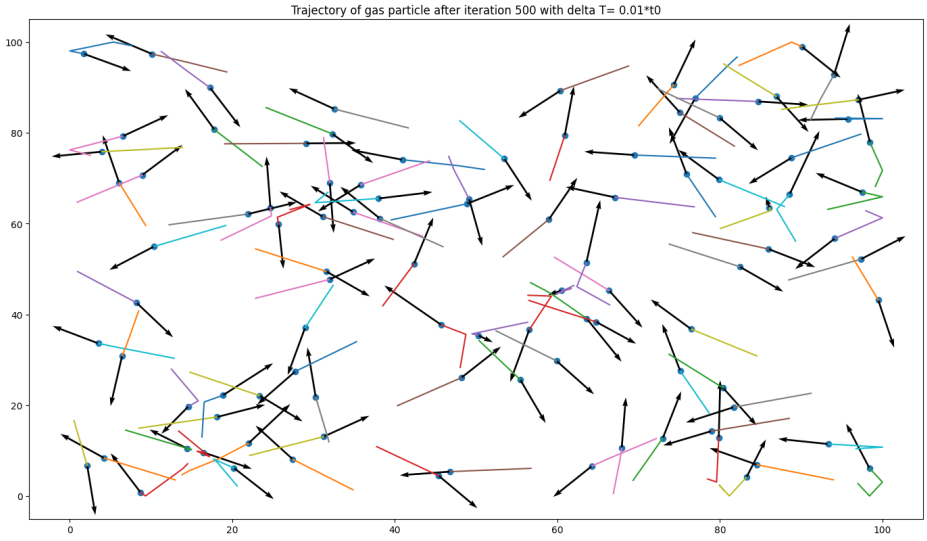


Figure 6: Trajectory of gas particles

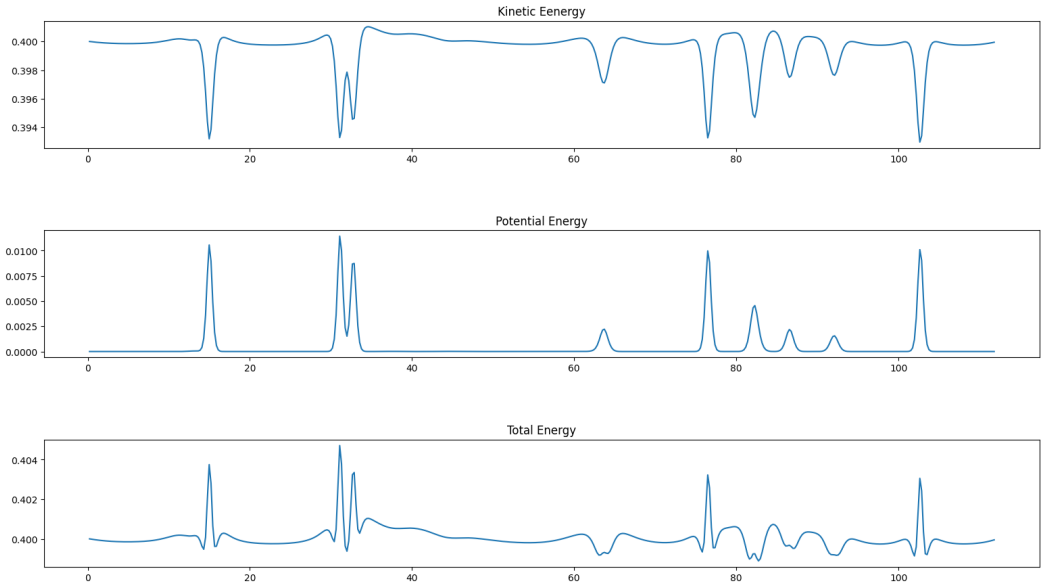


Figure 7: Energy over the iteration

2.2.3 Results: Initial and final position and velocity of 100 gas particle and its trajectory when delta t = 0.001*t0

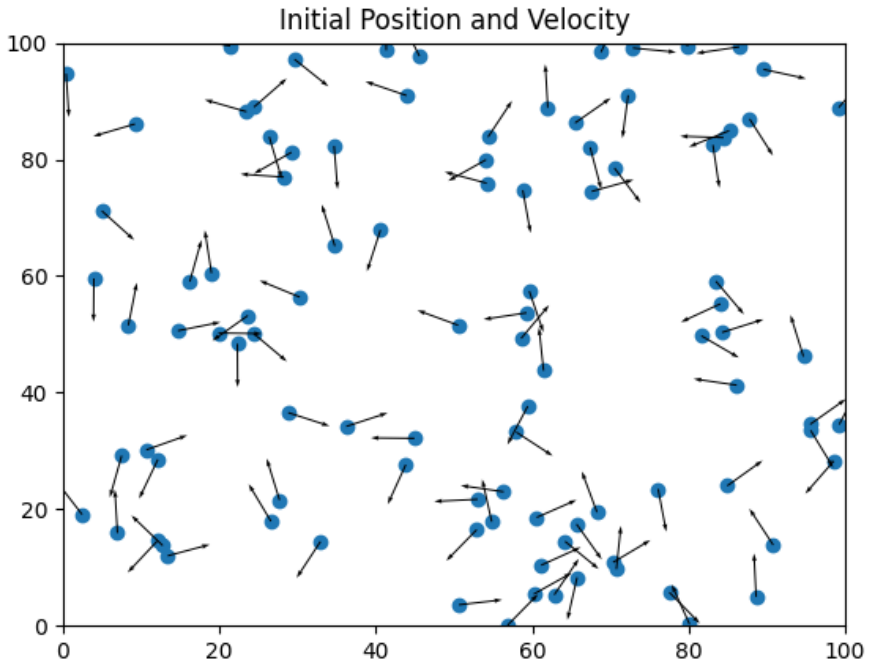


Figure 8: Initial position and velocity of 100 particles

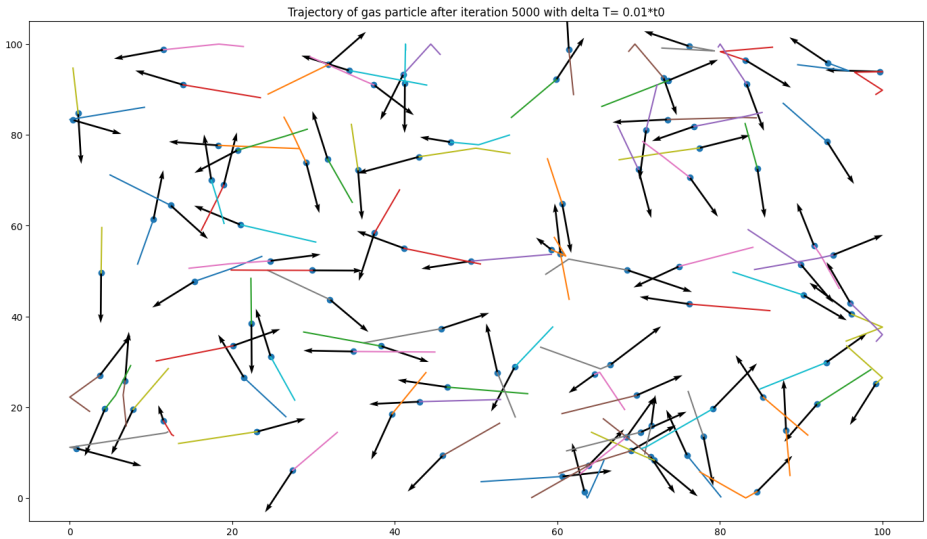


Figure 9: Trajectory of gas particles

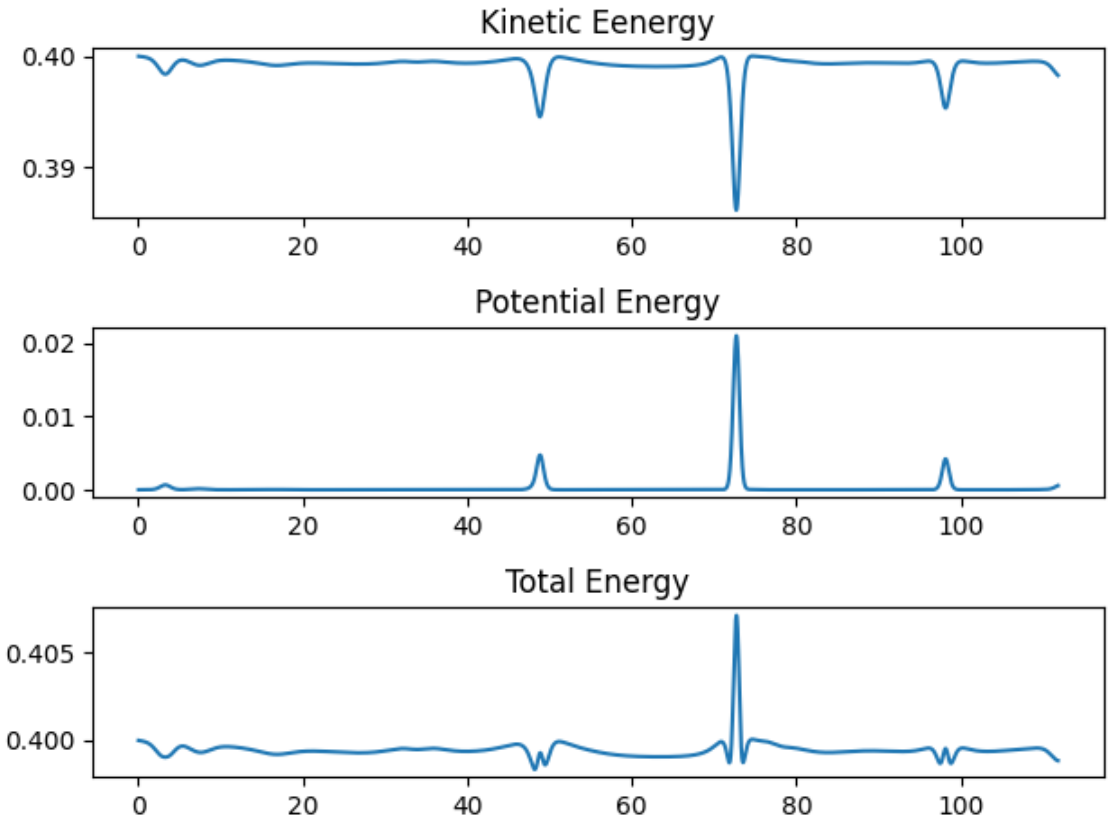


Figure 10: Energy over the iteration

3 Brownian motion - Simulation of a large particle with 36 non-interacting gas particles with time step 0.00005*t0 in confined boundary

3.1 Program

```
    import numpy as np
import random
from matplotlib import pyplot as plt

e0 = 0.01
sigma0=1
mass0=1

velocity0 = np.sqrt((2*e0)/mass0)
t0= sigma0*np.sqrt(mass0/(2*e0))

deltaT=0.00005*t0
boundary=[0,100*sigma0]
n=100
iteration =int(5/deltaT)

int_velocity = np.full((n,2),30*velocity0)
print(int_velocity)
for i in range(n):
    angle= random.random()*2*np.pi
    int_velocity[i][0]= np.sin(angle)*int_velocity[i][0]
    int_velocity[i][1]=np.cos(angle)*int_velocity[i][1]
print(int_velocity)
```

```
LP_int_position= [boundary[1]/2, boundary[1]/2]
LP_int_velocity=np.array([0,0])
LP_radius= 10
LP_mass= 40*mass0

int_positions = []
def distance(point1, point2):
    return np.linalg.norm(point1 - point2)


def is_inside_circle(x, LP_int_position, LP_radius):
    return np.linalg.norm(x-LP_int_position)<=LP_radius

def distance_between(x, position):
    d=np.zeros((n,1))
    for i in range (n):
        d[i]= np.linalg.norm(x-position[i])-LP_radius
    return d

# def force(d):
#     f = np.zeros((n,1))
#     for i in range (n):
#         if d[i]>sigma0:
#             # f[i]= 4*e0*(((sigma0**12/d[i]**12)) - (sigma0**6/d[i]**6))
#             f[i]= 48*e0*((2*(sigma0**12/d[i]**13)) - (0.5*sigma0**6/d[i]**7))
#         else:
#             f[i]=0
#     return f

def lennard_jones_force(positions, x):
    forces = np.zeros_like(positions)
    for i in range(len(positions)):
        r = np.sqrt(np.sum((positions[i] - x) ** 2)) - LP_radius
        magnitude = 4 * e0 * (12 * np.power(sigma0, 12) * np.power(r, -13) - 6 * np.power(sigma0, 6) * np.powe
        direction = (x - positions[i]) / r
        forces[i] = magnitude * direction

    return forces



while len(int_positions) < n:
    new_position = np.random.uniform(boundary[0], boundary[1], size=2)
    valid = True
    for existing_position in int_positions:
        if distance(new_position, existing_position) < sigma0 or is_inside_circle(new_position, LP_int_positio
            valid = False
            break
    if valid:
        int_positions.append(new_position)

int_positions= np.array(int_positions)
print(int_positions[:][0])
plt.xlim(0, boundary[1])
plt.ylim(0, boundary[1])
plt.quiver(int_positions[:, 0], int_positions[:, 1], int_velocity[:, 0], int_velocity[:, 1])

plt.show()

for i in range(n):
```

```
    angle = np.random.uniform(0,2*np.pi) #random.random()*2*np.pi
    int_velocity[i][0]= int_velocity[i][0]*np.cos(angle)
    int_velocity[i][1]= int_velocity[i][1]*np.sin(angle)


position= int_positions
velocity = int_velocity
Lp_position = LP_int_position
Lp_velocity = LP_int_velocity
Lp_position_list=[]
# Lp_position_list.append(Lp_position)
position_list=[]


for t in range(iteration):
    nxt_position = np.zeros((n, 2))
    nxt_velocity = np.zeros((n, 2))

    Lp_middle = Lp_position + 0.5 * Lp_velocity * deltaT
    f_LP = lennard_jones_force(position, Lp_middle)

    total_f = np.sum(f_LP, axis=0)

    Lp_next_velocity = Lp_velocity + total_f * deltaT / LP_mass
    # Lp_next_position = Lp_middle+ Lp_next_velocity*deltaT/2
    Lp_next_position = Lp_position + Lp_velocity * deltaT + 0.5 * total_f * deltaT**2 / LP_mass

    if Lp_next_position[0]< boundary[0]+LP_radius:
        difference = boundary[0] - Lp_next_position[0]
        Lp_next_position[0]= difference
        Lp_next_velocity[0]= - Lp_next_velocity[0]
    if Lp_next_position[0]> boundary[1]-LP_radius:
        difference = Lp_next_position[0] - boundary[1]
        Lp_next_position[0]= boundary[1] - difference
        Lp_next_velocity[0] = -Lp_next_velocity[0]

    if Lp_next_position[1]< boundary[0]+LP_radius:
        difference = boundary[0] - Lp_next_position[1]
        Lp_next_position[1]= difference
        Lp_next_velocity[1]= - Lp_next_velocity[1]
    if Lp_next_position[1]> boundary[1]-LP_radius:
        difference = Lp_next_position[1] - boundary[1]
        Lp_next_position[1]= boundary[1] - difference
        Lp_next_velocity[1] = -Lp_next_velocity[1]
    Lp_position = Lp_next_position
    Lp_velocity = Lp_next_velocity
    Lp_position_list.append(Lp_next_position)

    for i in range (n):
        nxt_position[i] = position[i]+ velocity[i]*deltaT
        nxt_velocity[i] = velocity[i]

        if nxt_position[i][0] < boundary[0] or nxt_position[i][0] > boundary[1]:
            nxt_position[i][0] = max(min(nxt_position[i][0], boundary[1]), boundary[0])
            nxt_velocity[i][0] *= -1  # Reverse the velocity for reflection

        # Boundary reflection for y-coordinate
        if nxt_position[i][1] < boundary[0] or nxt_position[i][1] > boundary[1]:
            nxt_position[i][1] = max(min(nxt_position[i][1], boundary[1]), boundary[0])
            nxt_velocity[i][1] *= -1
```

```
    position=nxt_position
    velocity = nxt_velocity
    position_list.append(position)

# #-----------------------------
position_list= np.array(position_list)
Lp_position_list=np.array(Lp_position_list)

#---------------------------------------



# Plotting the circle and particles
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))

# Plotting Circle and Particles
circle = plt.Circle(LP_int_position, LP_radius, edgecolor='black', facecolor='none')
ax1.set_aspect('equal')
ax1.add_patch(circle)
ax1.quiver(LP_int_position[0], LP_int_position[1], LP_int_velocity[0], LP_int_velocity[1])
for i in range(n):
    ax1.scatter(int_positions[i][0], int_positions[i][1], color='blue')

ax1.set_xlabel('X-axis')
ax1.set_ylabel('Y-axis')
ax1.set_title('gas particle and Large Particles Plot')
ax1.grid(True)

# Plotting the trace of a particle
ax2.set_aspect('equal')
circle = plt.Circle(Lp_position, LP_radius, edgecolor='black', facecolor='none')
ax2.add_patch(circle)
ax2.scatter(position[:, 0], position[:, 1])
ax2.plot(Lp_position_list[:, 0], Lp_position_list[:, 1], label='Trace of Particle')
ax2.set_xlabel('X-axis')
ax2.set_ylabel('Y-axis')
ax2.set_title('Trace of Large Particle')
ax2.grid(True)

plt.tight_layout()
plt.show()


msd= np.zeros((iteration,1))
iteration_list=[]
for t in range (iteration):
    dif_square=0
    for i in range(iteration-t):
        dif_square+= np.square(np.linalg.norm(Lp_position_list[i+t]-Lp_position_list[i]))
    msd[t]= 1/(iteration-t) * dif_square
    iteration_list.append(t)

slope, _ = np.polyfit(iteration_list, msd, 1)
# conversion_factor = 1e-6
diffusion_coefficient = slope / 4
print('diff', diffusion_coefficient)
# D = msd/(4*1)
# print(D)
plt.plot(iteration_list,msd, 'o')
plt.plot(iteration_list, slope*iteration_list, '-', color='black' )
```

```
plt.xlim(0,iteration)
plt.show()
```

3.2 Result

3.2.1 Simulation of gas particle

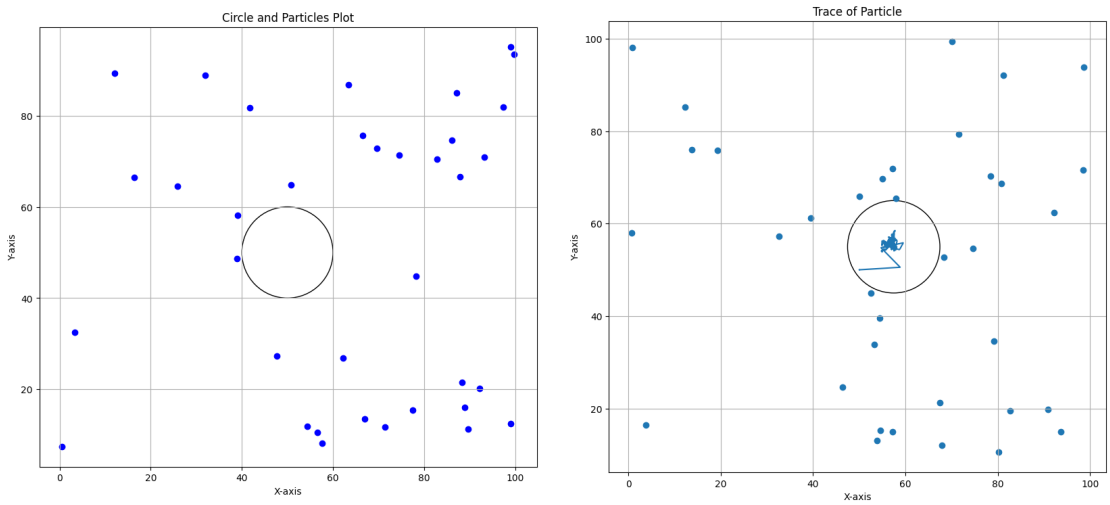


Figure 11: Trajectory of Large particles with 36 gas particle

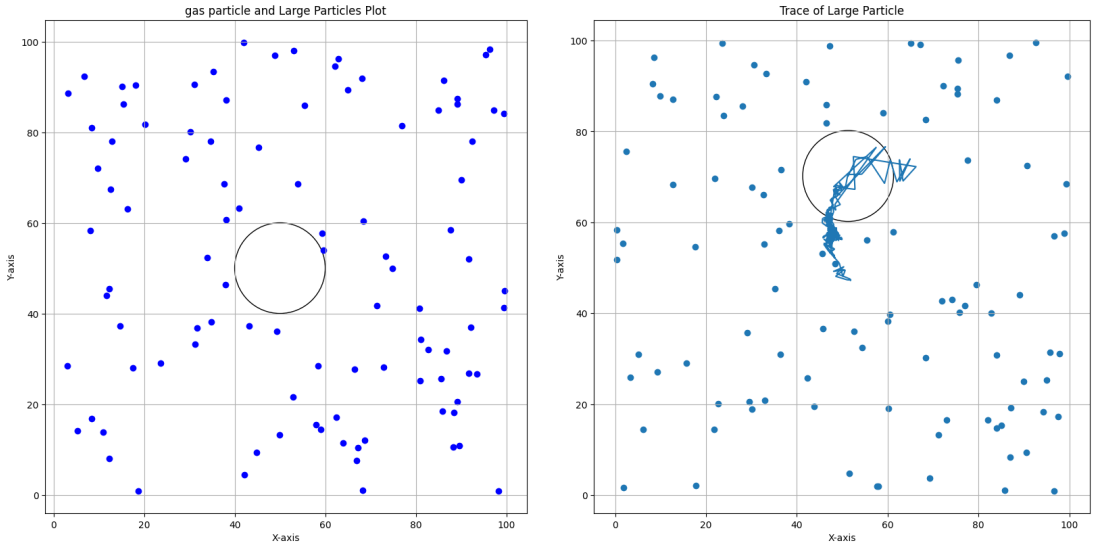


Figure 12: Trajectory of Large particles with 100 gas particle

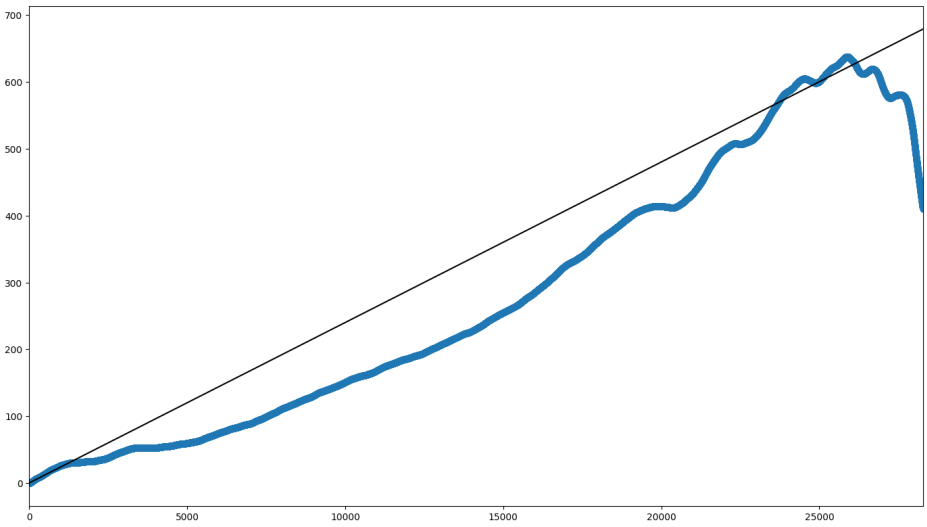


Figure 13: τ vs MSD, where τ represents the time lag

3.2.2 Effective diffusion coefficient of large particle

For 100 particles D=0.006600187

Initial Speed	Mass	Radius	D
20v0	40m0	10	0.006600187
20v0	30m0	10	0.00050325
30v0	40m0	10	0.0051685
20v0	40m0	5	0.00018823

Table 1: Diffusion coefficient