

# project2\_demo1

November 19, 2024

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from nbody import *
```

## 1 The Particles class

We could write a `Particles` python class to handle the particle information. The class contains several physical properties, including tag, mass, position, velocity, acceleration, and time.

For our own convenience, we want to have the below data type to handle the N-body simulation:

```
[2]: time          = 0      # the starting time
num_particles = 100 # number of particles
masses       = np.ones((num_particles,1))
positions    = np.zeros((num_particles,3)) # 3 directions
velocities   = np.zeros((num_particles,3))
accelerations = np.zeros((num_particles,3))
tags         = np.linspace(1,num_particles,num_particles)
```

Note that, the mass is setting to a Nx1 martrix.

The reason to use Nx1 matrix but not a 1D numpy array is because mass x veloicity is the momentum and only Nx1 matrix could multiple with an Nx3 matrix.

The particles class can be initialized by

```
[3]: particles = Particles(N=num_particles)
```

```
[4]: particles.masses = np.ones((num_particles,1))
particles.positions = np.random.rand(num_particles, 3)
particles.velocities = np.random.rand(num_particles, 3)
particles.accelerations = np.random.rand(num_particles, 3)
particles.tags = np.linspace(1,num_particles,num_particles)
```

Make sure your code will check the shape of your inputs. It must return errors when setting an incorrect shape.

```
[6]: # make sure the below codes will return an error. uncomment each line to test

# particles.masses = np.ones(num_particles)
```

```
# particles.positions = np.random.rand(199, 3)
# particles.velocities = np.random.rand(500, 3)
# particles.accelerations = np.random.rand(num_particles, 2)
# particles.tags = np.linspace(1,num_particles,500)
```

## 2 Add (remove) more particles

We could add more particles on the fly.

```
[5]: masses = np.ones((num_particles,1))
positions = np.random.rand(num_particles, 3)
velocities = np.random.rand(num_particles, 3)
accelerations = np.random.rand(num_particles, 3)

particles.add_particles(masses, positions, velocities, accelerations)
print(particles.nparticles)
```

200

### 2.0.1 Data IO

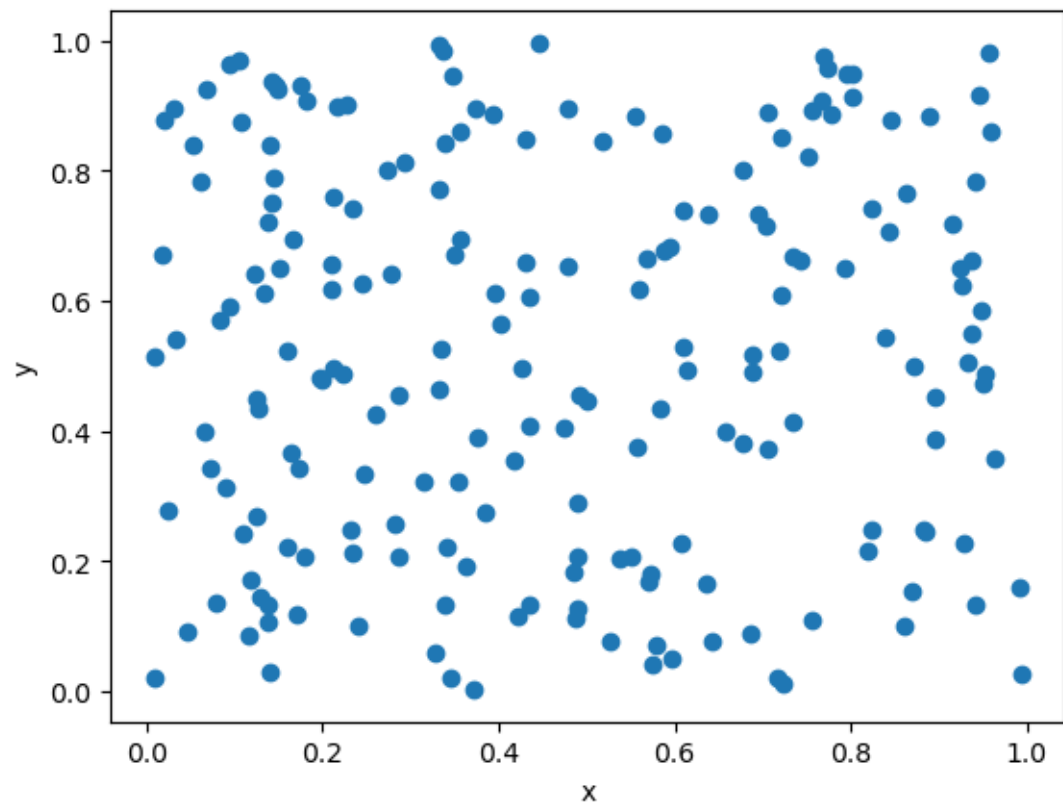
We could also dump the particle information into a text file.

```
[6]: particles.output(filename='data.txt')
```

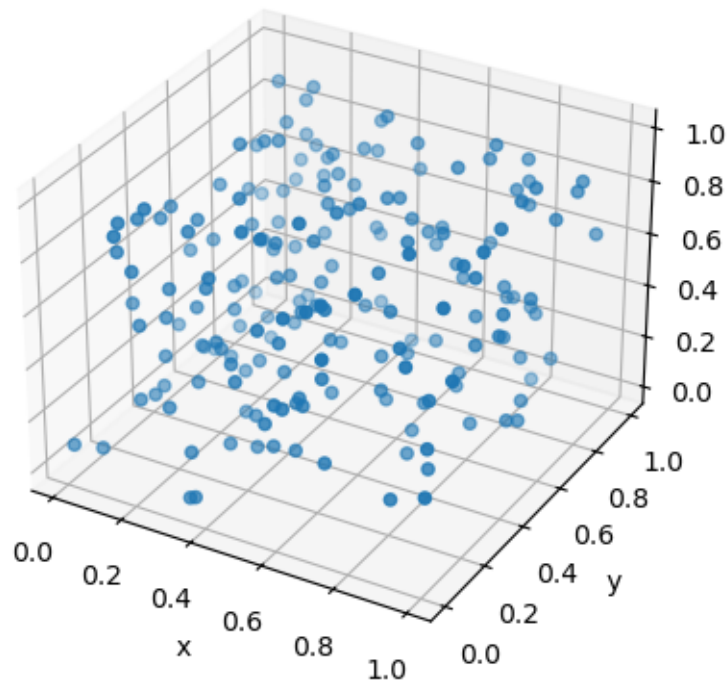
### 2.0.2 Visualization

We could also visualize (both 2D and 3D) these particles

```
[7]: particles.draw(dim=2)
```



```
[8]: particles.draw(dim=3)
```



### 3 Exercise 1

Implement the `Particles` class in `./nbody/particles.py`. Please make sure your `Particles` class has passed all the test in the above section.

```
[10]: # TODO: test your class here
```

### 4 Exercise 2

Once you have the `Particles` class implemented correctly.

You should be able to use it to initialize arbitrary distribution of  $N$  particles.

- (1) Initialize two particles that describe the Sun-Earth binary system.
- (2) Initialize a 3D particle cloud with  $N=1000$  particles in a normal distribution ( $\sigma=1$ ) and total mass equal to 10.

Hints: use `numpy.random.randn` (see <https://numpy.org/doc/stable/reference/random/generated/numpy.random.randn>).

```
[9]: def initialize_sun_earth_system():
    num_particles = 2
    particles = Particles(num_particles)
```

```

# Masses: Sun and Earth
masses = np.array([[1.989e30], [5.972e24]])

# Positions:
positions = np.array([[0, 0, 0], # Sun
                      [1.496e11, 0, 0]]) # Earth (1 AU away from Sun)

# Velocities:
velocities = np.array([[0, 0, 0], # Sun stationary
                       [0, 29.78e3, 0]]) # Earth's orbital velocity around
↳ the Sun (in m/s)

# Accelerations: Set to zero for simplicity
accelerations = np.array([[0, 0, 0], [5.93e-3, 0, 0]])

# Set particle properties
particles.masses = masses
particles.positions = positions
particles.velocities = velocities
particles.accelerations = accelerations

# Visualize the Sun-Earth system in 2D
particles.draw(2)

return particles

def initialize_particle_cloud(N=1000, total_mass=10):

    particles = Particles(N)
    masses = np.full((N, 1), total_mass / N)
    positions = np.random.randn(N, 3)
    velocities = np.zeros((N, 3))
    accelerations = np.zeros((N, 3))

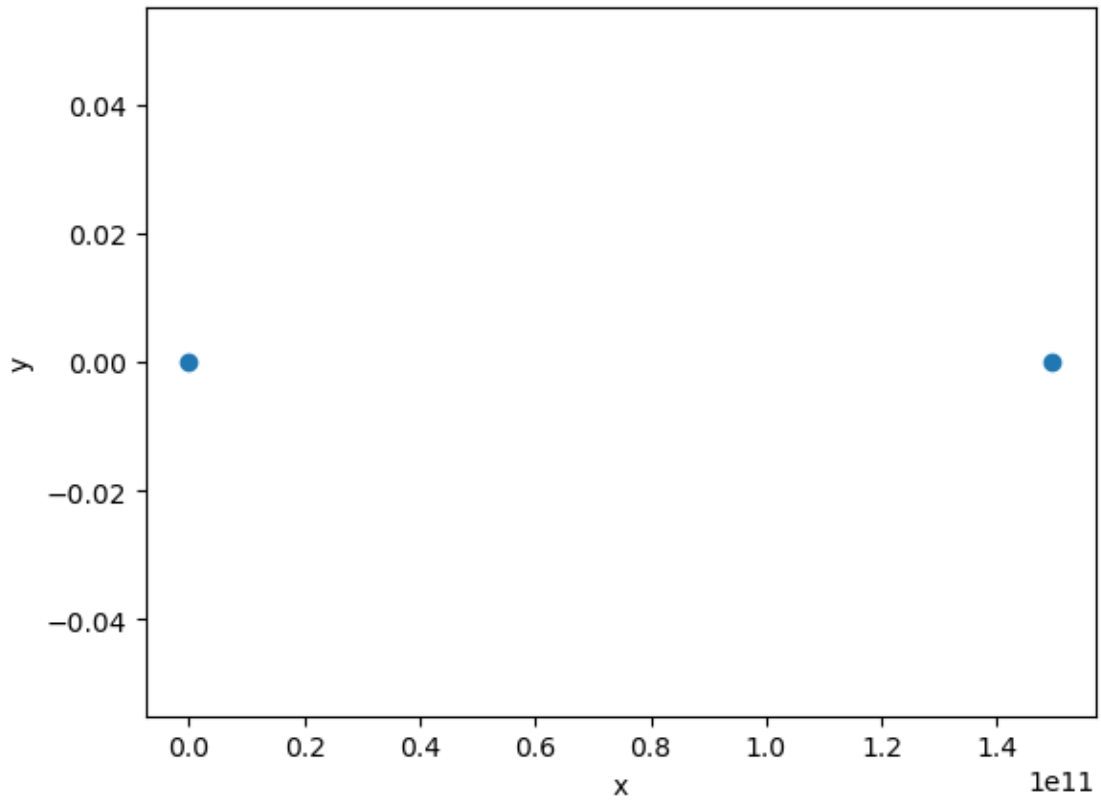
    # Set particle properties
    particles.masses = masses
    particles.positions = positions
    particles.velocities = velocities
    particles.accelerations = accelerations

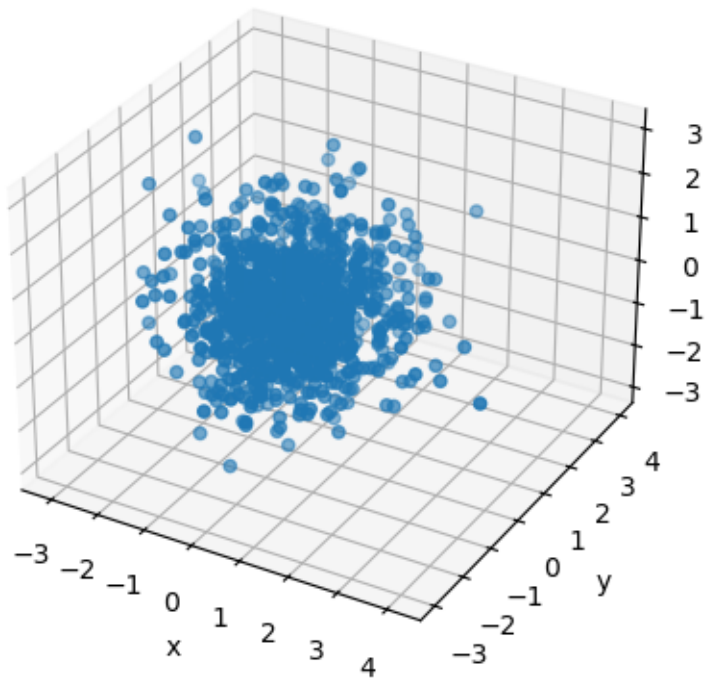
    particles.draw(3)

    return particles

```

```
[10]: if __name__ == "__main__":  
    # Task 1: Initialize the Sun-Earth system  
    sun_earth_particles = initialize_sun_earth_system()  
  
    # Task 2: Initialize a 3D particle cloud with N=1000 and total mass of 10  
    particle_cloud = initialize_particle_cloud(N=1000, total_mass=10)
```





# project2\_demo2

November 19, 2024

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from nbody import Particles, NBodySimulator
from nbody import load_files, save_movie
from numba import set_num_threads
```

```
[2]: # Uncomment the following line to install required packages if needed

# !conda install -y -c conda-forge ffmpeg, glob, numba
```

## 1 N-Body Simulation

In this notebook, we will test the NBodySimulator class in `./nbody/simulator.py`.

```
[3]: # Set the number of threads to use for numba
nthreads = 1
set_num_threads(nthreads)
```

## 2 Exercise 1: The Sun-Earth System

The first test is to implement the Sun-Earth system, which is a two body system. We need to make sure that you could simulate a circular motion first.

### 2.1 Step 1. The initial condition

Copy the initial condition of the Earth-Sun system from `project2_demo1.ipynb`.

```
[4]: # TODO:
particles = Particles(N=2)

# Masses: Sun and Earth
masses = np.array([[1.989e30], [5.972e24]])

# Positions:
positions = np.array([[0, 0, 0], # Sun
                     [1.496e11, 0, 0]]) # Earth (1 AU away from Sun)

# Velocities:
```



```

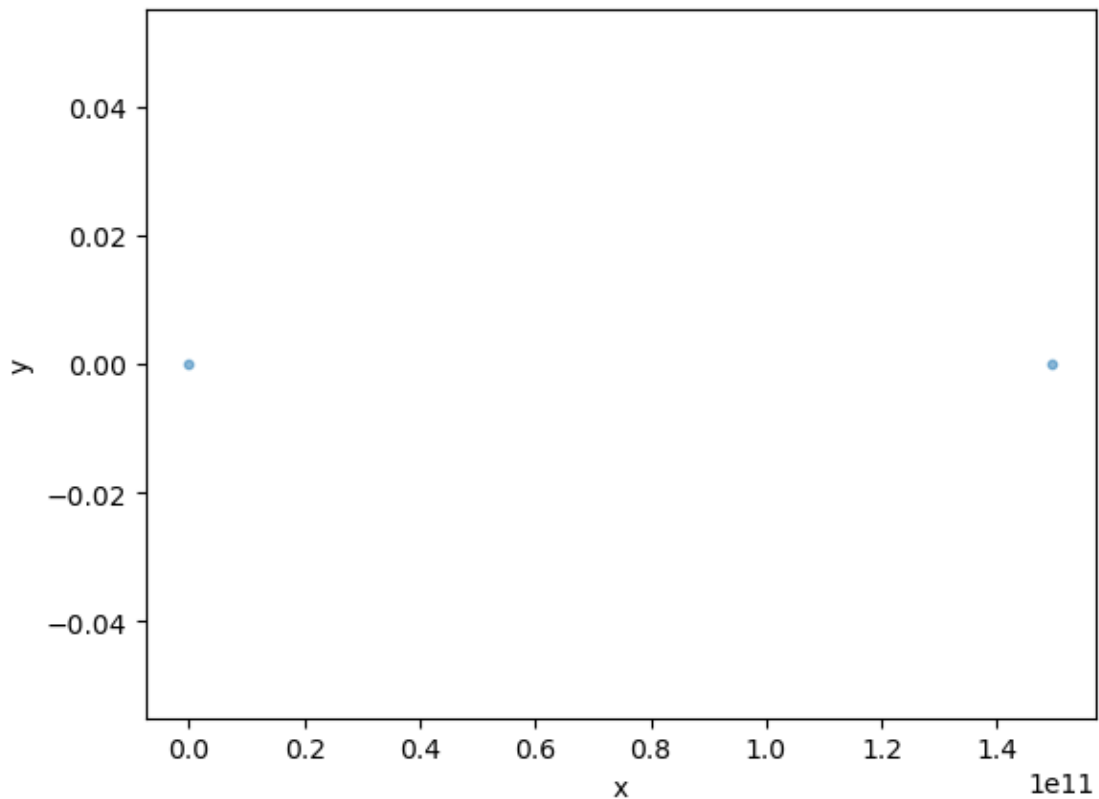
velocities = np.array([[0, 0, 0], # Sun stationary
                       [0, 29.78e3, 0]]) # Earth's orbital velocity around
↳ the Sun (in m/s)

# Accelerations: Set to zero for simplicity
accelerations = np.array([[0, 0, 0], [5.93e-3, 0, 0]])

# Set particle properties
particles.masses = masses
particles.positions = positions
particles.velocities = velocities
particles.accelerations = accelerations

particles.draw(2)

```



## 2.2 Step 2. N-body simulation

Make sure you have implemented the below two methods `setup()` and `evolve()`. Run the n-body simulation by:

```
[5]: simulation = NBodySimulator(particles=particles)
simulation.setup(G=6.674e-11,rsoft=1e9,method='RK4', io_freq=30)
simulation.evolve(dt=8640, tmax=365*86400)
```

```
Time: 0/31536000
Time: 259200/31536000
Time: 518400/31536000
Time: 777600/31536000
Time: 1036800/31536000
Time: 1296000/31536000
Time: 1555200/31536000
Time: 1814400/31536000
Time: 2073600/31536000
Time: 2332800/31536000
Time: 2592000/31536000
Time: 2851200/31536000
Time: 3110400/31536000
Time: 3369600/31536000
Time: 3628800/31536000
Time: 3888000/31536000
Time: 4147200/31536000
Time: 4406400/31536000
Time: 4665600/31536000
Time: 4924800/31536000
Time: 5184000/31536000
Time: 5443200/31536000
Time: 5702400/31536000
Time: 5961600/31536000
Time: 6220800/31536000
Time: 6480000/31536000
Time: 6739200/31536000
Time: 6998400/31536000
Time: 7257600/31536000
Time: 7516800/31536000
Time: 7776000/31536000
Time: 8035200/31536000
Time: 8294400/31536000
Time: 8553600/31536000
Time: 8812800/31536000
Time: 9072000/31536000
Time: 9331200/31536000
Time: 9590400/31536000
Time: 9849600/31536000
Time: 10108800/31536000
Time: 10368000/31536000
Time: 10627200/31536000
Time: 10886400/31536000
Time: 11145600/31536000
```

Time: 11404800/31536000  
Time: 11664000/31536000  
Time: 11923200/31536000  
Time: 12182400/31536000  
Time: 12441600/31536000  
Time: 12700800/31536000  
Time: 12960000/31536000  
Time: 13219200/31536000  
Time: 13478400/31536000  
Time: 13737600/31536000  
Time: 13996800/31536000  
Time: 14256000/31536000  
Time: 14515200/31536000  
Time: 14774400/31536000  
Time: 15033600/31536000  
Time: 15292800/31536000  
Time: 15552000/31536000  
Time: 15811200/31536000  
Time: 16070400/31536000  
Time: 16329600/31536000  
Time: 16588800/31536000  
Time: 16848000/31536000  
Time: 17107200/31536000  
Time: 17366400/31536000  
Time: 17625600/31536000  
Time: 17884800/31536000  
Time: 18144000/31536000  
Time: 18403200/31536000  
Time: 18662400/31536000  
Time: 18921600/31536000  
Time: 19180800/31536000  
Time: 19440000/31536000  
Time: 19699200/31536000  
Time: 19958400/31536000  
Time: 20217600/31536000  
Time: 20476800/31536000  
Time: 20736000/31536000  
Time: 20995200/31536000  
Time: 21254400/31536000  
Time: 21513600/31536000  
Time: 21772800/31536000  
Time: 22032000/31536000  
Time: 22291200/31536000  
Time: 22550400/31536000  
Time: 22809600/31536000  
Time: 23068800/31536000  
Time: 23328000/31536000  
Time: 23587200/31536000

```
Time: 23846400/31536000
Time: 24105600/31536000
Time: 24364800/31536000
Time: 24624000/31536000
Time: 24883200/31536000
Time: 25142400/31536000
Time: 25401600/31536000
Time: 25660800/31536000
Time: 25920000/31536000
Time: 26179200/31536000
Time: 26438400/31536000
Time: 26697600/31536000
Time: 26956800/31536000
Time: 27216000/31536000
Time: 27475200/31536000
Time: 27734400/31536000
Time: 27993600/31536000
Time: 28252800/31536000
Time: 28512000/31536000
Time: 28771200/31536000
Time: 29030400/31536000
Time: 29289600/31536000
Time: 29548800/31536000
Time: 29808000/31536000
Time: 30067200/31536000
Time: 30326400/31536000
Time: 30585600/31536000
Time: 30844800/31536000
Time: 31104000/31536000
Time: 31363200/31536000
Simulation is done!
```

## 2.3 Step 3. Visualization

Check the code in `./nbody/visualization.py`. Data loader is implemented in the function `load_files`.

```
[6]: fns = load_files('nbody')
      print(fns)
```

```
['data_nbody\\nbody_000000.dat', 'data_nbody\\nbody_000001.dat',
'data_nbody\\nbody_000002.dat', 'data_nbody\\nbody_000003.dat',
'data_nbody\\nbody_000004.dat', 'data_nbody\\nbody_000005.dat',
'data_nbody\\nbody_000006.dat', 'data_nbody\\nbody_000007.dat',
'data_nbody\\nbody_000008.dat', 'data_nbody\\nbody_000009.dat',
'data_nbody\\nbody_000010.dat', 'data_nbody\\nbody_000011.dat',
'data_nbody\\nbody_000012.dat', 'data_nbody\\nbody_000013.dat',
'data_nbody\\nbody_000014.dat', 'data_nbody\\nbody_000015.dat',
```



[illegible]



[illegible]



[illegible]









[illegible]







[illegible]

[illegible]





[illegible]





[illegible]











[illegible]

[illegible]

[illegible]

[illegible]











[illegible]

[illegible]









```

'data_nbody\\nbody_003568.dat', 'data_nbody\\nbody_003569.dat',
'data_nbody\\nbody_003570.dat', 'data_nbody\\nbody_003571.dat',
'data_nbody\\nbody_003572.dat', 'data_nbody\\nbody_003573.dat',
'data_nbody\\nbody_003574.dat', 'data_nbody\\nbody_003575.dat',
'data_nbody\\nbody_003576.dat', 'data_nbody\\nbody_003577.dat',
'data_nbody\\nbody_003578.dat', 'data_nbody\\nbody_003579.dat',
'data_nbody\\nbody_003580.dat', 'data_nbody\\nbody_003581.dat',
'data_nbody\\nbody_003582.dat', 'data_nbody\\nbody_003583.dat',
'data_nbody\\nbody_003584.dat', 'data_nbody\\nbody_003585.dat',
'data_nbody\\nbody_003586.dat', 'data_nbody\\nbody_003587.dat',
'data_nbody\\nbody_003588.dat', 'data_nbody\\nbody_003589.dat',
'data_nbody\\nbody_003590.dat', 'data_nbody\\nbody_003591.dat',
'data_nbody\\nbody_003592.dat', 'data_nbody\\nbody_003593.dat',
'data_nbody\\nbody_003594.dat', 'data_nbody\\nbody_003595.dat',
'data_nbody\\nbody_003596.dat', 'data_nbody\\nbody_003597.dat',
'data_nbody\\nbody_003598.dat', 'data_nbody\\nbody_003599.dat',
'data_nbody\\nbody_003600.dat', 'data_nbody\\nbody_003601.dat',
'data_nbody\\nbody_003602.dat', 'data_nbody\\nbody_003603.dat',
'data_nbody\\nbody_003604.dat', 'data_nbody\\nbody_003605.dat',
'data_nbody\\nbody_003606.dat', 'data_nbody\\nbody_003607.dat',
'data_nbody\\nbody_003608.dat', 'data_nbody\\nbody_003609.dat',
'data_nbody\\nbody_003610.dat', 'data_nbody\\nbody_003611.dat',
'data_nbody\\nbody_003612.dat', 'data_nbody\\nbody_003613.dat',
'data_nbody\\nbody_003614.dat', 'data_nbody\\nbody_003615.dat',
'data_nbody\\nbody_003616.dat', 'data_nbody\\nbody_003617.dat',
'data_nbody\\nbody_003618.dat', 'data_nbody\\nbody_003619.dat',
'data_nbody\\nbody_003620.dat', 'data_nbody\\nbody_003621.dat',
'data_nbody\\nbody_003622.dat', 'data_nbody\\nbody_003623.dat',
'data_nbody\\nbody_003624.dat', 'data_nbody\\nbody_003625.dat',
'data_nbody\\nbody_003626.dat', 'data_nbody\\nbody_003627.dat',
'data_nbody\\nbody_003628.dat', 'data_nbody\\nbody_003629.dat',
'data_nbody\\nbody_003630.dat', 'data_nbody\\nbody_003631.dat',
'data_nbody\\nbody_003632.dat', 'data_nbody\\nbody_003633.dat',
'data_nbody\\nbody_003634.dat', 'data_nbody\\nbody_003635.dat',
'data_nbody\\nbody_003636.dat', 'data_nbody\\nbody_003637.dat',
'data_nbody\\nbody_003638.dat', 'data_nbody\\nbody_003639.dat',
'data_nbody\\nbody_003640.dat', 'data_nbody\\nbody_003641.dat',
'data_nbody\\nbody_003642.dat', 'data_nbody\\nbody_003643.dat',
'data_nbody\\nbody_003644.dat', 'data_nbody\\nbody_003645.dat',
'data_nbody\\nbody_003646.dat', 'data_nbody\\nbody_003647.dat',
'data_nbody\\nbody_003648.dat', 'data_nbody\\nbody_003649.dat',
'data_nbody\\nbody_003650.dat']

```

Modify the matplotlib script

```

[ ]: au = 1.496e11
      save_movie(fns, lengthscale=2*au,filename='nbody_earth_sun.mp4', fps=10)

```

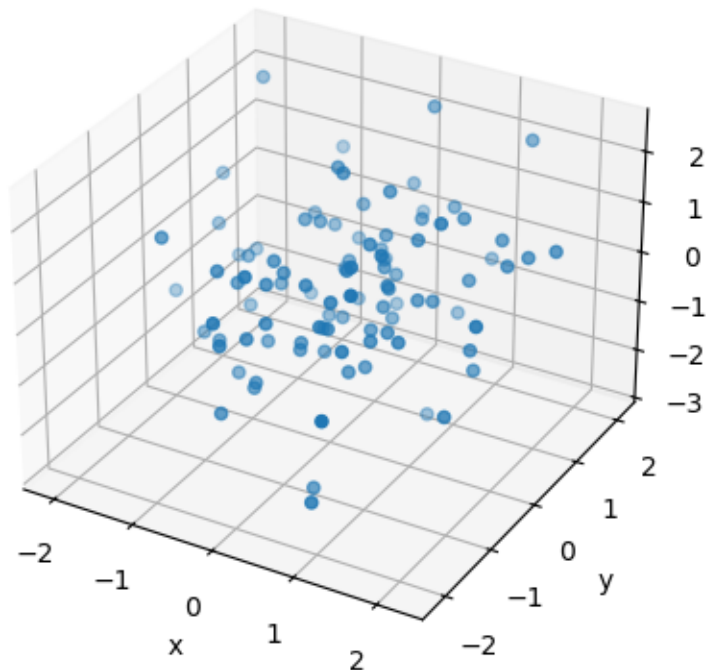
### 3 Exercise 2: N-Body simulation

Now, let's test our n-body solver with more particles. Copy the initial condition from the Exercise 2 in `project2_demo1.ipynb`.

```
[8]: # TODO
N = 100
particles = Particles(N)
masses = np.full((N, 1), 10 / N)
positions = np.random.randn(N, 3)
velocities = np.zeros((N, 3))
accelerations = np.zeros((N, 3))

# Set particle properties
particles.masses = masses
particles.positions = positions
particles.velocities = velocities
particles.accelerations = accelerations

particles.draw(3)
```



#### 3.1 Step 2. Simulation.

Visualize the system but use dimensionless units. Set  $G=1$  and  $r_{\text{soft}}=0.001$ .

```
[10]: simulation = NBodySimulator(particles=particles)
simulation.setup(G=1,rsoft=0.001,method='RK4', io_freq=10, io_header='cluster')
simulation.evolve(dt=0.01, tmax=1)
```

```
Time: 0/1
Time: 0.09999999999999999/1
Time: 0.20000000000000004/1
Time: 0.30000000000000001/1
Time: 0.40000000000000002/1
Time: 0.50000000000000002/1
Time: 0.60000000000000003/1
Time: 0.70000000000000004/1
Time: 0.80000000000000005/1
Time: 0.90000000000000006/1
Simulation is done!
```

It is VERY slow !!!!

Beacuse the accerlation calculations requires order of NxN calculations! Could we improve it?

First, we should avoid the two “for” loops in the calculation using python.

Second, we have many cup cores in our machine. Could we parallelize it?

## 4 numba

Step 1. Take out the function to calculate the accerlation from the class into a pure function that only counts math calculations (aka. a kernel).

Step 2. Use the numba’s @jit micro.

Results: In my experiments. I got roughly x140 speed up with the @jit(nopython=True).

Step 3. Try using @njit and prange in numba.

### 4.1 Performance

#### 4.1.1 Strong Scaling Test

**Strong scaling:** The number of processors is increased while the problem size remains constant. We could use %timeit to measure the performance. Let’s run 1000 particles for 10 step.

```
[20]: num_particles = 1000
pts = Particles(N=num_particles)
pts.masses = np.ones((num_particles, 1))* 10/num_particles
pts.positions = np.random.randn(num_particles, 3)
pts.velocities = np.random.randn(num_particles, 3)
pts.accelerations = np.zeros((num_particles, 3))

simulation = NBodySimulator(particles=pts)
simulation.setup(G=1,rsoft=0.001,method='RK4', io_screen=False, io_freq=1,
↳io_header='cluster')
```

```
#simulation.evolve(dt=0.02, tmax=1)
```

```
[ ]: %timeit simulation.evolve(dt=0.02, tmax=0.2)
```

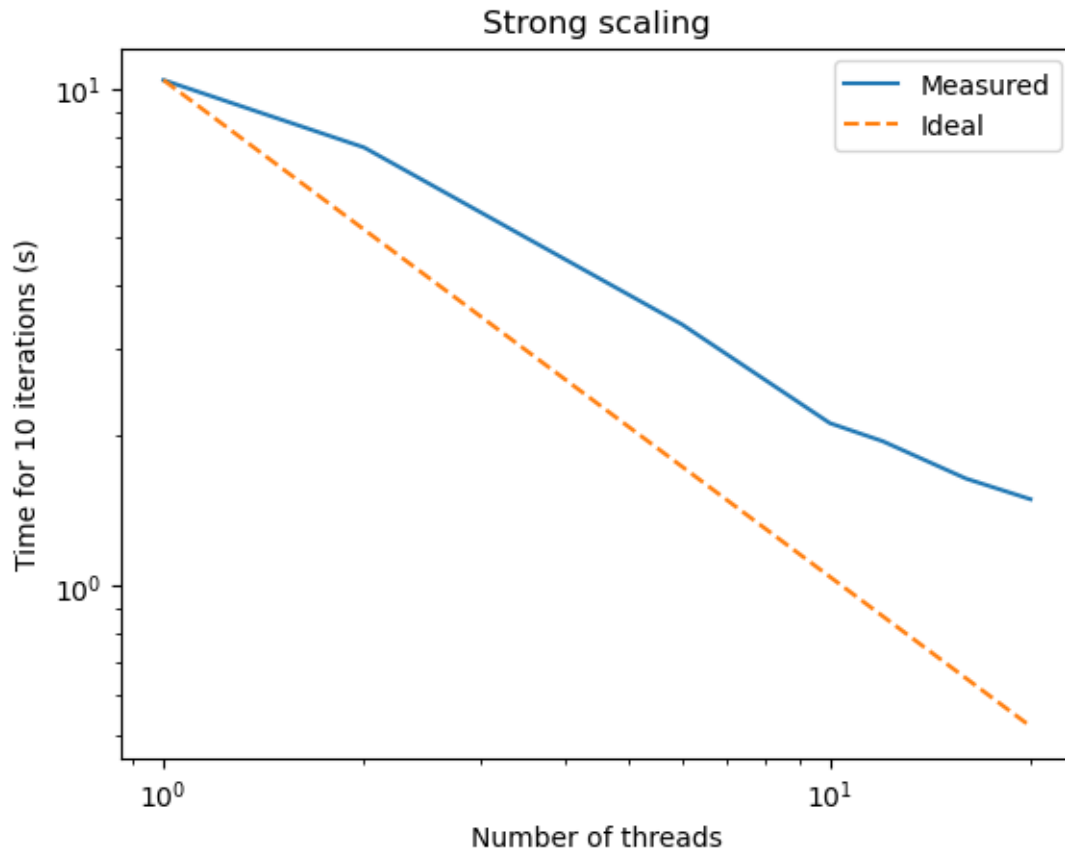
### Kuo-Chuan's measurements

CPU: 3 GHz 10-Core Intel Xeon W

1000 particles. 10 steps.

without numba: 5min 31s  $\pm$  2 s per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) with numba: \*  
1 core (jit): 17.7 s  $\pm$  328 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) \* 1 core (njit):  
10.4 s  $\pm$  337 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) \* 2 cores: 7.61 s  $\pm$  52.8 ms per  
loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) \* 6 cores: 3.35 s  $\pm$  50.9 ms per loop (mean  $\pm$  std.  
dev. of 7 runs, 1 loop each) \* 10 cores: 2.12 s  $\pm$  16.2 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1  
loop each) \* 12 cores: 1.95 s  $\pm$  36.1 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) \* 16  
cores: 1.64 s  $\pm$  46 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each) \* 20 cores: 1.49 s  $\pm$  55.1  
ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[23]: threads = [1,2,6,10,12,16,20] # Replace it to your measured values
      times = [10.4, 7.61, 3.35, 2.12, 1.95, 1.64,1.49] # Replace it to your measured values
      ideal = times[0]/np.array(threads)
      plt.loglog(threads, times)
      plt.loglog(threads, ideal, '--')
      plt.xlabel('Number of threads')
      plt.ylabel('Time for 10 iterations (s)')
      plt.legend(['Measured', 'Ideal'])
      plt.title('Strong scaling')
      plt.show()
```



## 4.2 Weak Scaling Test

In N-body simulation, the problem size is proportional to  $N^2$ .

**Weaking scaling** test measures the scaling with the same problem size per thread (core).

```
[24]: num_particles = int(225*np.sqrt(nthreads))
      print("N =", num_particles, ", threads = ", nthreads)
      pts = Particles(N=num_particles)
      pts.masses = np.ones((num_particles, 1))* 10/num_particles
      pts.positions = np.random.randn(num_particles, 3)
      pts.velocities = np.random.randn(num_particles, 3)
      pts.accelerations = np.zeros((num_particles, 3))

      simulation = NBodySimulator(particles=pts)
      simulation.setup(G=1,rsoft=0.001,method='RK4', io_screen=False, io_freq=0,
      ↪io_header='cluster')
```

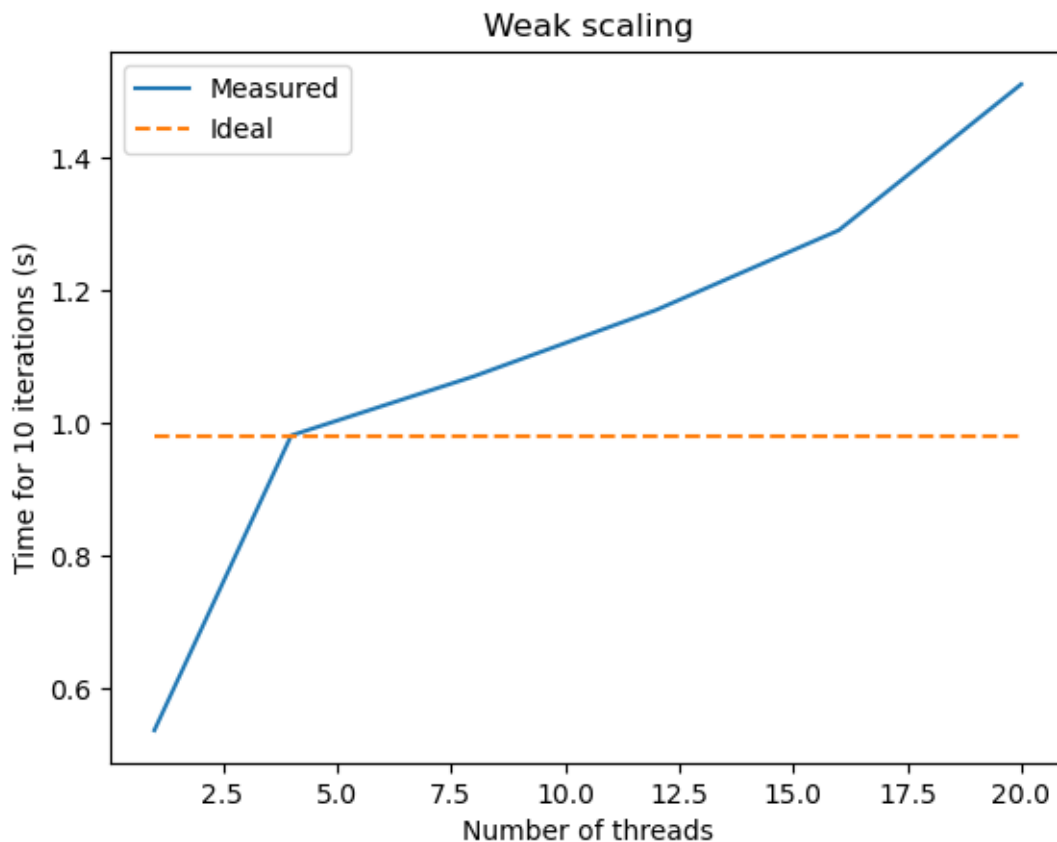
N = 225 , threads = 1

```
[ ]: %timeit simulation.evolve(dt=0.02, tmax=0.2)
```

Kuo-Chuan's measurements

- N = 225 threads = 1: 537 ms  $\pm$  13.9 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)
- N = 450 threads = 4: 981 ms  $\pm$  17.3 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)
- N = 636 threads = 8: 1.07 s  $\pm$  22.3 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)
- N = 779 threads = 12: 1.17 s  $\pm$  33.7 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)
- N = 900 threads = 16: 1.29 s  $\pm$  6.85 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)
- N = 1006 threads = 20: 1.51 s  $\pm$  35.3 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[26]: threads = [1,4,8,12,16,20] # Replace it to your measured values
times = [0.537, 0.981, 1.07, 1.17, 1.29, 1.51] # Replace it to your measured values
ideal = 0.981 * np.ones(len(threads))
plt.plot(threads, times)
plt.plot(threads, ideal, '--')
plt.xlabel('Number of threads')
plt.ylabel('Time for 10 iterations (s)')
plt.legend(['Measured', 'Ideal'])
plt.title('Weak scaling')
plt.show()
```







# HW2

November 21, 2024

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from nbody import Particles, NBodySimulator
from nbody import load_files, save_movie
from numba import set_num_threads
```

```
[2]: # Set the number of threads to use for numba
nthreads = 10
set_num_threads(nthreads)
```

## 1 Problem1

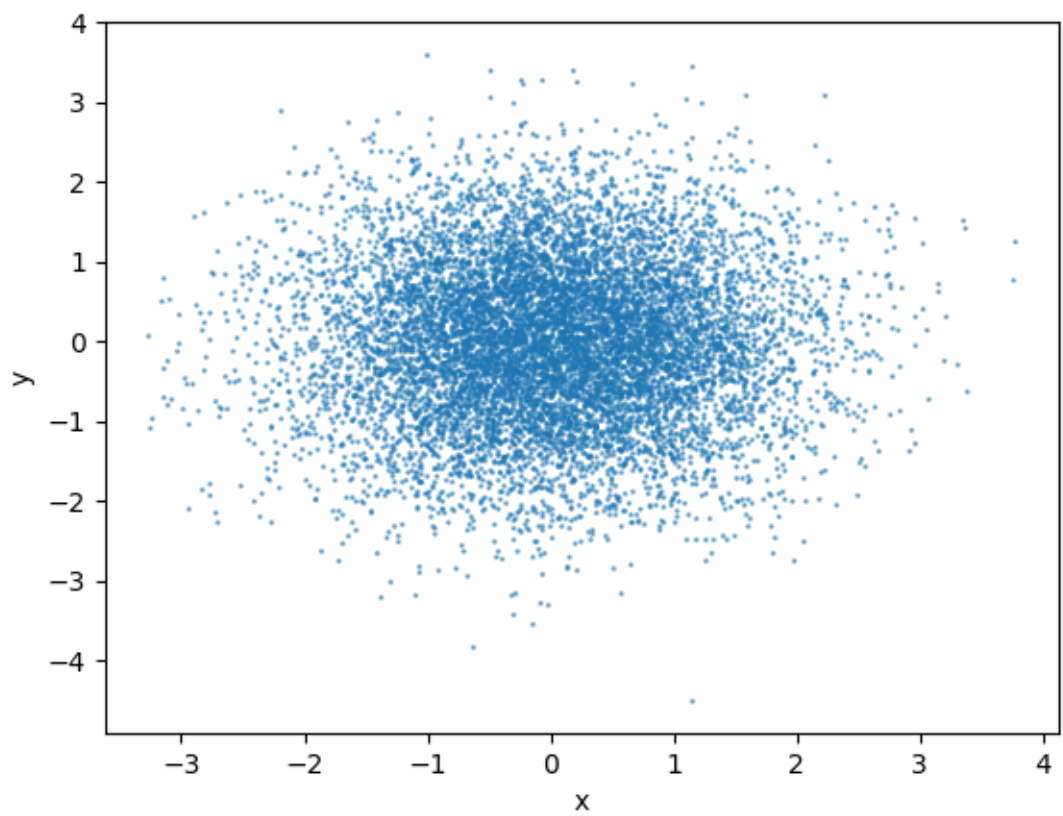
```
[11]: N = 10000
tot_mass = 20
t_max = 2
dt = 0.01
mean = 0
var = 1

particles = Particles(N)
masses = np.full((N, 1), tot_mass / N)
positions = np.random.normal(mean, np.sqrt(var), (N, 3))
velocities = np.random.normal(mean, np.sqrt(var), (N, 3))
accelerations = np.random.normal(mean, np.sqrt(var), (N, 3))

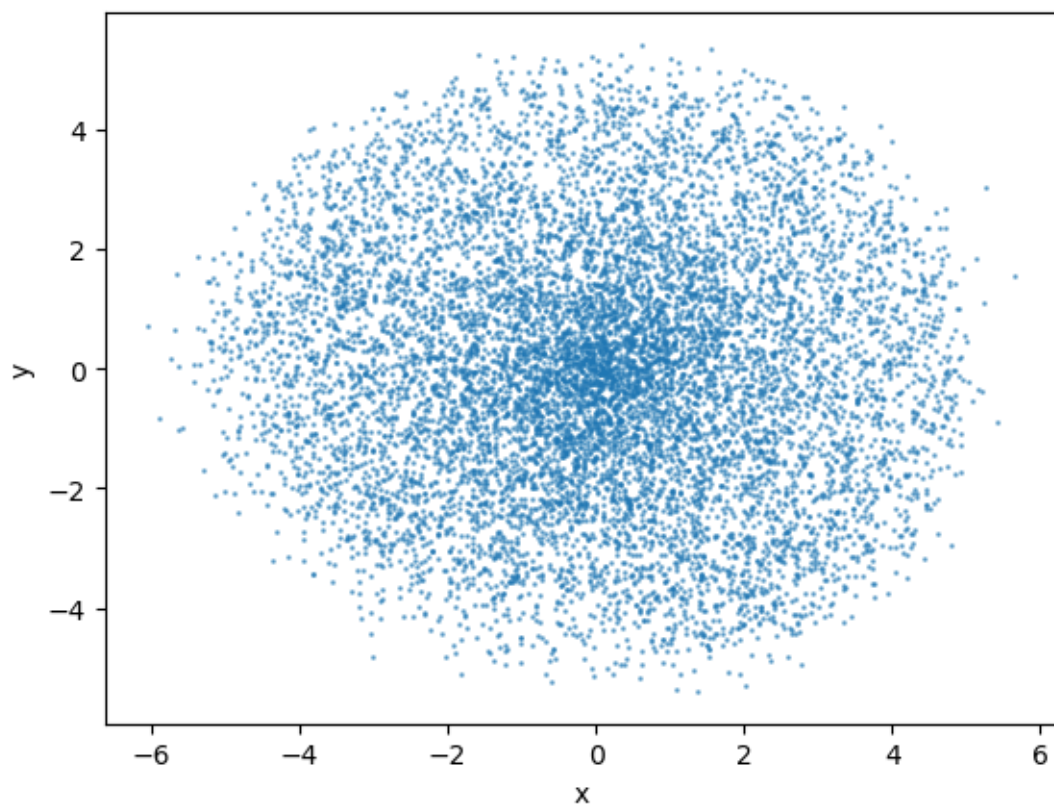
# Set particle properties
particles.masses = masses
particles.positions = positions
particles.velocities = velocities
particles.accelerations = accelerations
```

```
[12]: simulation = NBodySimulator(particles)
simulation.setup(G = 10, rsoft = 0.01, method = "Euler", io_freq = 50,
    ↪ visualization = True)
simulation.evolve(dt, t_max+dt)
```

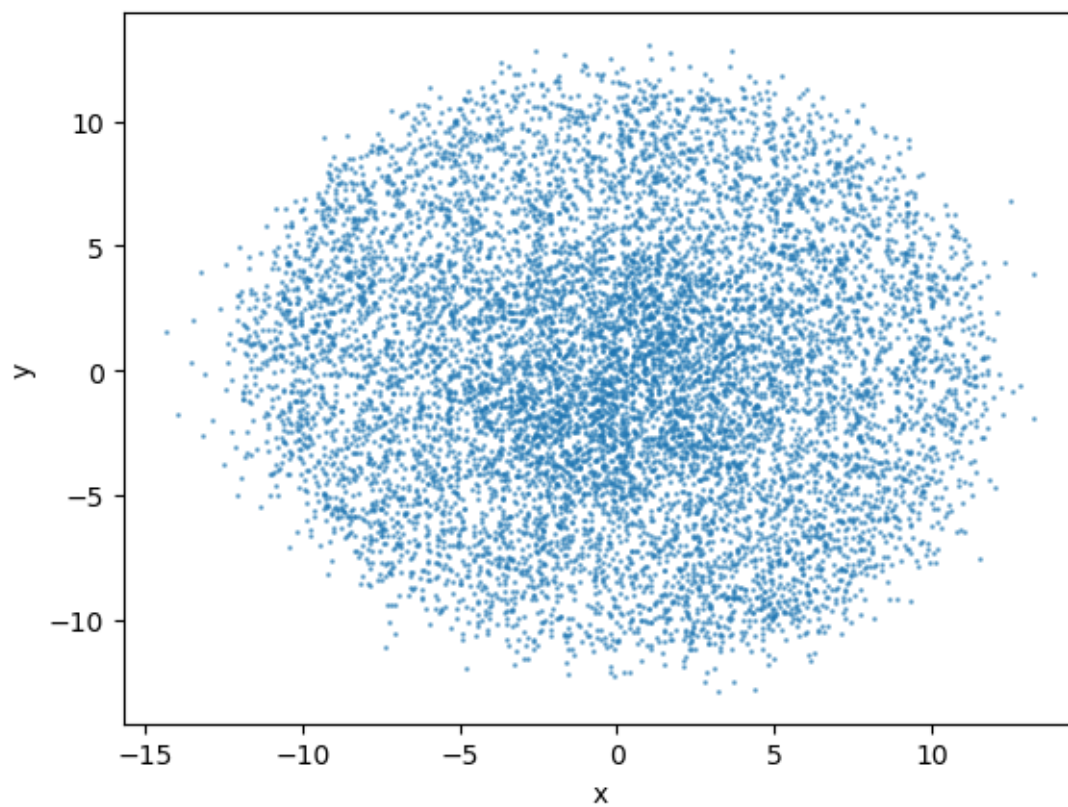
Time: 0.00/2.01



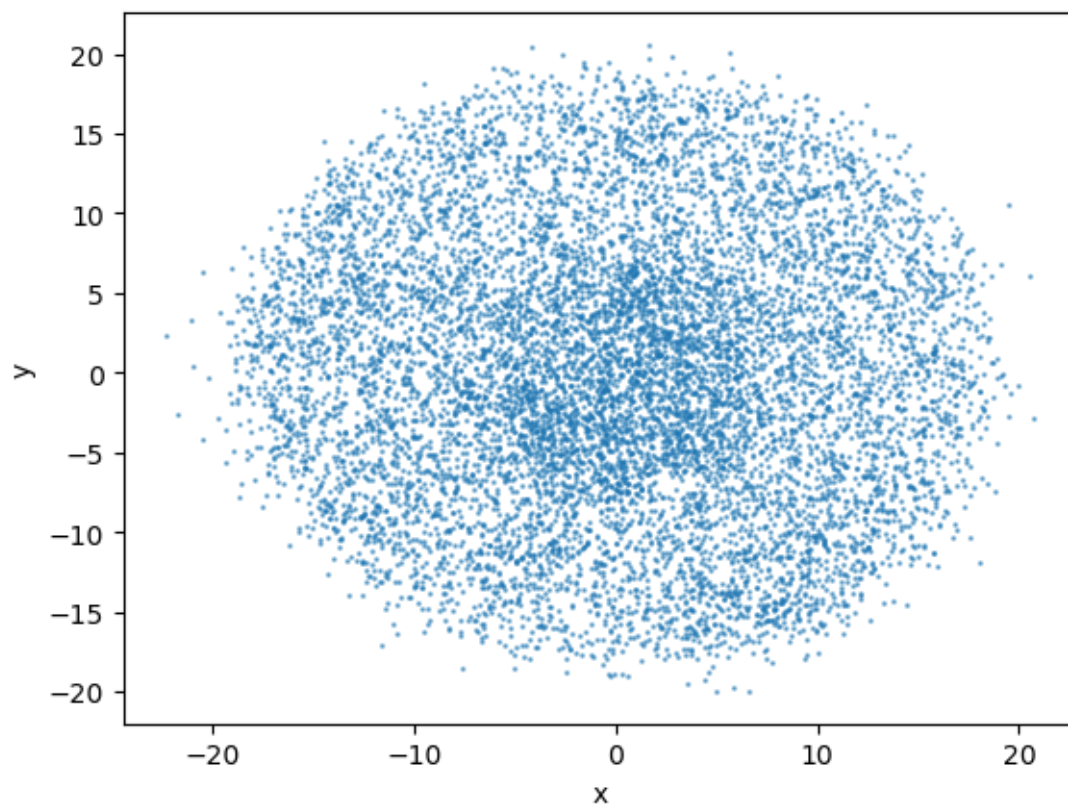
Time: 0.50/2.01



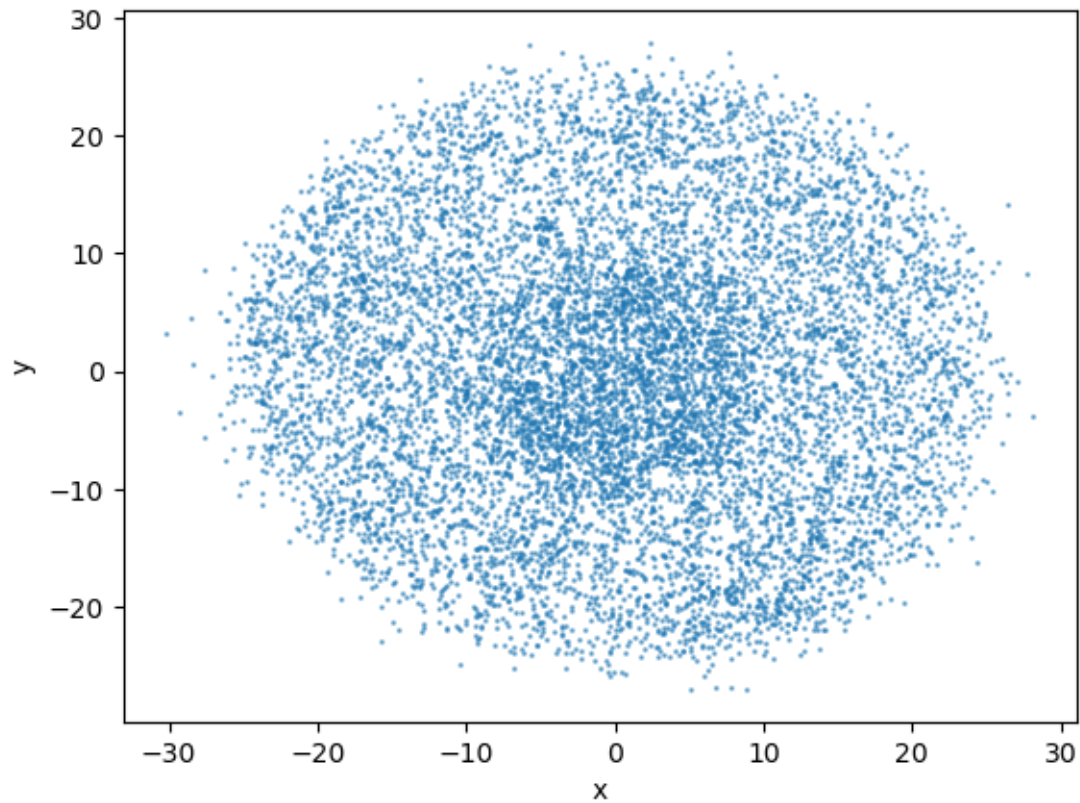
Time: 1.00/2.01



Time: 1.50/2.01



Time: 2.00/2.01



Simulation is done!

## 2 Problem2

```
[3]: import numpy as np
import matplotlib.pyplot as plt

def read_and_draw_energy(method):
    #
    file_names = [f"data_nbody/nbody_000{str(i).zfill(3)}.dat" for i in
↪range(0, 200)]

    #
    kinetic_energies = []
    potential_energies = []
    total_energies = []
    #
    for file_name in file_names:
        with open(file_name, 'r') as f:
            #
```

```

lines = f.readlines()
#
for line in lines:
    if "Total Kinetic Energy" in line:
        KE = float(line.split(":")[1].strip().split(" ")[0])
        kinetic_energies.append(KE)
    elif "Total Potential Energy" in line:
        PE = float(line.split(":")[1].strip().split(" ")[0])
        potential_energies.append(PE)
    elif "Total Energy" in line:
        TE = float(line.split(":")[1].strip().split(" ")[0])
        total_energies.append(TE)

#
time_steps = np.linspace(0, t_max, len(kinetic_energies))

#
plt.figure(figsize=(10, 6))

#
plt.subplot(3, 1, 1)
plt.plot(time_steps, kinetic_energies, label='Kinetic Energy', color='r')
plt.xlabel('Time (s)')
plt.ylabel('Kinetic Energy (J)')
plt.legend()

#
plt.subplot(3, 1, 2)
plt.plot(time_steps, potential_energies, label='Potential Energy',
color='b')
plt.xlabel('Time (s)')
plt.ylabel('Potential Energy (J)')
plt.legend()

#
plt.subplot(3, 1, 3)
plt.plot(time_steps, total_energies, label='Total Energy', color='g')
plt.xlabel('Time (s)')
plt.ylabel('Total Energy (J)')
plt.legend()

plt.title(f'Energy vs. Time ({method})')
plt.tight_layout()
plt.show()

```

```

[4]: methods = ["Euler", "RK2", "RK4"]
for method in methods:

```

```

N = 100
tot_mass = 20
t_max = 2
dt = 0.01
mean = 0
var = 1

particles = Particles(N)
masses = np.full((N, 1), tot_mass / N)
positions = np.random.normal(mean, np.sqrt(var), (N, 3))
velocities = np.random.normal(mean, np.sqrt(var), (N, 3))
accelerations = np.random.normal(mean, np.sqrt(var), (N, 3))

# Set particle properties
particles.masses = masses
particles.positions = positions
particles.velocities = velocities
particles.accelerations = accelerations

simulation = NBodySimulator(particles)
simulation.setup(G = 10, rsoft = 0.01, method = method, io_freq = 50,
↳ visualization = False)
simulation.evolve(dt, t_max+dt)
read_and_draw_energy(method)

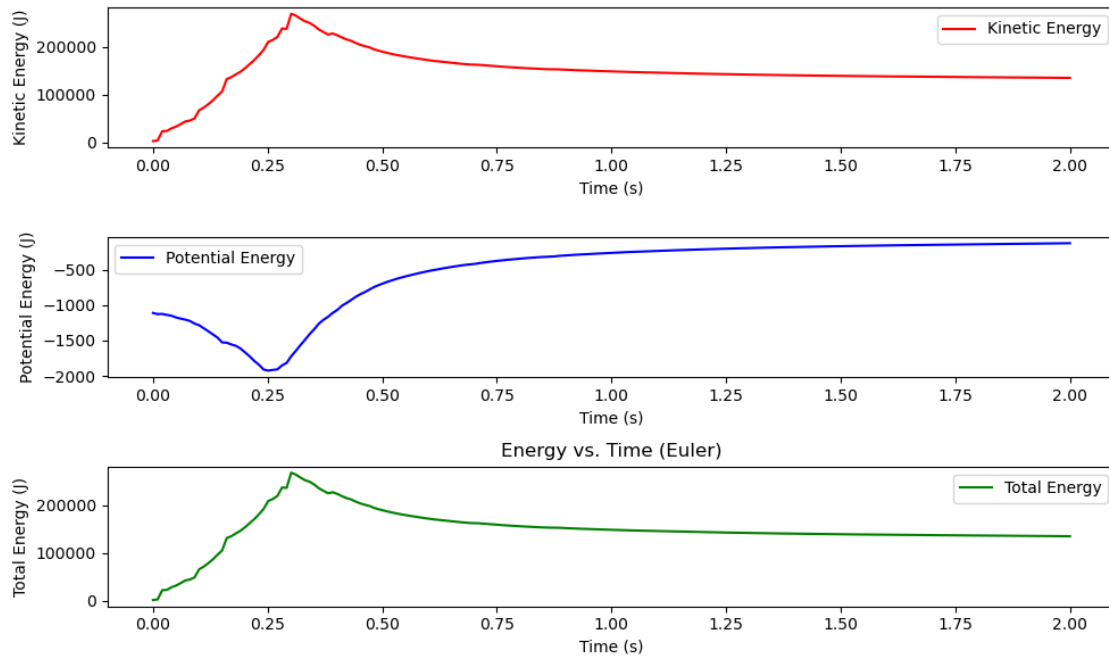
```

```

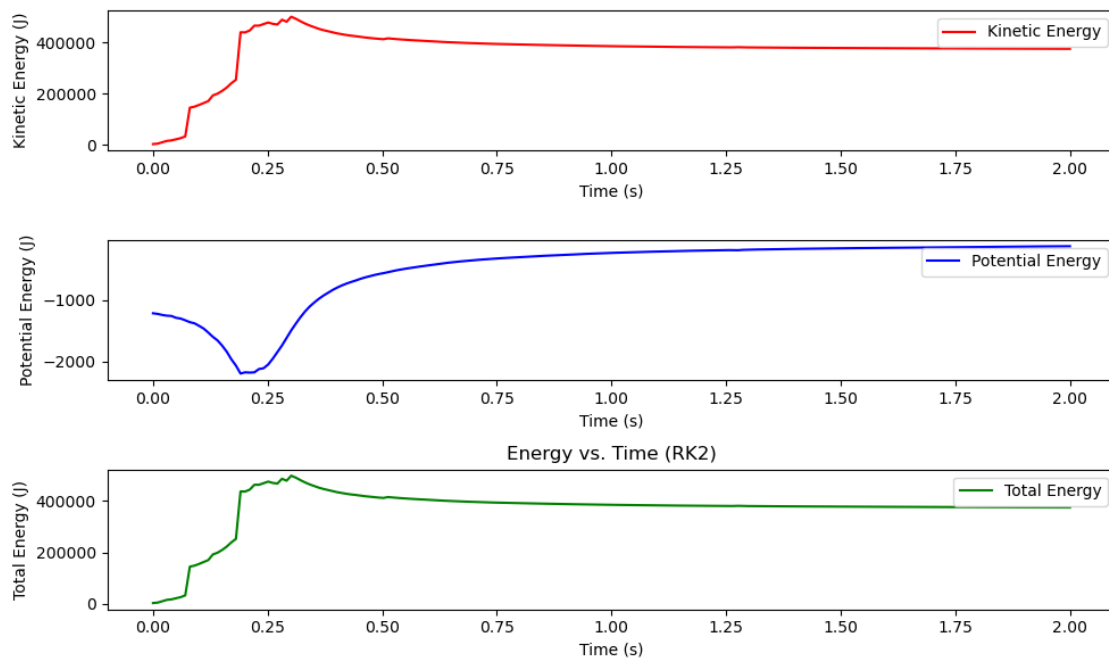
Time: 0.00/2.01
Time: 0.50/2.01
Time: 1.00/2.01
Time: 1.50/2.01
Time: 2.00/2.01
Simulation is done!

```

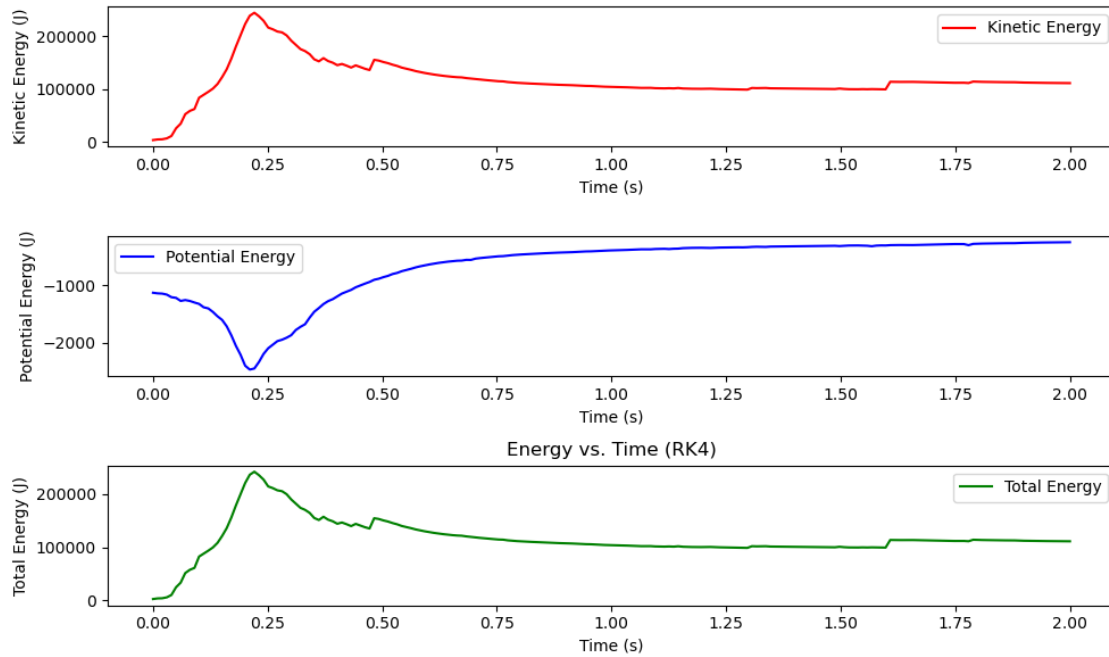




Time: 0.00/2.01  
Time: 0.50/2.01  
Time: 1.00/2.01  
Time: 1.50/2.01  
Time: 2.00/2.01  
Simulation is done!



Time: 0.00/2.01  
Time: 0.50/2.01  
Time: 1.00/2.01  
Time: 1.50/2.01  
Time: 2.00/2.01  
Simulation is done!



It seems that energy is not conserved, which suggests there might be a calculation error somewhere, but it's hard to pinpoint. Additionally, if  $N=10000$ , the program runs extremely slowly because the total kinetic and potential energy must be calculated at each time step, making it even slower than the first problem.