Computation Physic HW2

Kang-Shih Liu,109022102

This pdf file is for computation physic HW2.

I. Introduction

This time, there is no writing assignment, so I will just post all the code I write, and if I have discover any interesting things, I will write those at the bottom of the code. For below I will first put on the code of my solver(particles.py and simulator.py), and put on the result of each question. I will also send all the codes (python file and jupyter notebook) to the google classroom.

II. Code for particles.py

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit, njit, prange, set_num_threads
@njit(parallel=True)
def calculate_PE_kernel(nparticles,mass, position,G=0.1,rsoft=0.01):
    pe = np.zeros(nparticles)
    for i in prange(nparticles):
        for j in prange(i+1,nparticles): #避免double counting，亦可算出全部粒子的位能再除以2
            rij = position[i,:] - position[j,:]
            r = np.sqrt(np.sum(rij**2) + rsoft**2)
            pe[i] = pe[i] - G * mass[i,0] * mass[j,0] / (r)
    return pe

class Particles:
    #定義粒子的各屬性,tag為第幾顆粒子，ke及pe分別為動能及位能
    def __init__(self,N):
        self.nparticles = N
        self.mass = np.zeros((N,1))
        self.position = np.zeros((N,3))
        self.velocity = np.zeros((N,3))
        self.acceleration = np.zeros((N,3))
        self.tag = np.arange(N)
        self.time = 0
        self.ke = np.zeros(N)
        self.pe = np.zeros(N)
        return

    #給粒子設定(初始)狀態
    def setting_particles(self, pos, vel, acc):
        self.position = pos
        self.velocity = vel
        self.acceleration = acc

        return

    #將新的粒子加進系統中
    def adding_particles(self, mass, pos, vel, acc):
        self.nparticles = self.nparticles + len(mass)
        self.mass = np.vstack((self.mass, mass))
        self.position = np.vstack((self.position, pos))
        self.velocity = np.vstack((self.velocity, vel))
        self.acceleration = np.vstack((self.acceleration, acc))
        return

    #計算動能，並直接帶入self.ke
    def calculate_KE(self):
        self.ke = np.zeros(self.nparticles)
        for i in range(self.nparticles):
            self.ke[i] = 0.5*self.mass[i,0]*(np.sum(self.velocity

    #計算位能，並直接帶入self.pe
    def calculate_PE(self,G=0.1,rsoft=0.01):
        self.pe = np.zeros(self.nparticles)
        nparticles = self.nparticles
        mass = self.mass
        position = self.position
        self.pe = calculate_PE_kernel(nparticles,mass,positic


    #將疊代的結果數據output出來
    def output(self, filename):
        mass = self.mass
        pos = self.position
        vel = self.velocity
        acc = self.acceleration
        tag = self.tag
        time = self.time
        self.calculate_KE()
        self.calculate_PE()
        header = """
                ----------------------------------------
                Data from a 3D direct N-body simulation.

                rows are i-particle;
                coumns are :mass, tag, x ,y, z, vx, vy, vz, ax, ay, az, ke, pe
                The data is for Homework 2.
                ----------------------------------------
                """
        header += "Time = {}".format(time)
        np.savetxt(filename,(tag[:],mass[:,0],pos[:,0],
                             vel[:,0],vel[:,1],vel[:,2],
                             acc[:,0],acc[:,1],acc[:,2],self.
        return

    #將疊代出來的particle顯示在塗上✓
    def draw(self, dim=2):
        fig = plt.figure()
```

```
        if dim == 2:
            ax = fig.add_subplot(111)
            ax.scatter(self.position[:,0],
                self.position[:,1])
            ax.set_xlabel('X')
            ax.set_ylabel('Y')

        elif dim == 3:
            ax = fig.add_subplot(111,
                projection='3d')
            ax.scatter(self.position[:,0],
                self.position[:,1],
                self.position[:,2])
            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.set_zlabel('Z')
        else:
            print("Invalid dimension!")
            return

        ax.set_aspect('equal')
        plt.tight_layout()
        plt.show()
        return fig, ax
```

A. *note for particle.py*

Above is all the code in my Particle.py file. I have meet several problem during writing the code, but I have solved them all. Since I am not very familiar with the @property(which can check if the input dimension is wrong), I didn't put it in the particle.py file.

III. CODE FOR SIMULATOR.PY

```

import numpy as np
import matplotlib as plt
from pathlib import Path
from .particles import Particles
from numba import jit, njit, prange,
    set_num_threads

#用kernel去跑加速度增加效率
@njit(parallel=True)
def calculate_accleration_kernel(nparticles,
    mass, position, acceleration, G, rsoft):
    for i in prange(nparticles):
        for j in prange(nparticles):
            if (j>i):
                rij = position[i,:] -
                    position[j,:]
                r = np.sqrt(np.sum(rij**2) +
                    rsoft**2)
                force = - G * mass[i,0] *
                    mass[j,0] * rij / r**3
                acceleration[i,:] += force[:] /
                    mass[i,0]
                acceleration[j,:] -= force[:] /
                    mass[j,0]

    return acceleration

class NBodySimulator:
    #定義例子的各屬性，並input particles.py的數據來
        進行疊代
    def __init__(self, particles: Particles):
        self.particles = particles
        self.time = particles.time
        self.setup()
        return

    #設置好(並可以透過函數input改變)必要的參數，並準備
        好寫output函數所需參數
    def setup(self, G=0.1, rsoft=0.01,
        method="RK4", outfreq=10,
            outheader="nbody", outscreen=True,
                visualiztion=False):
        self.G = G                          #G:重力
            常數
        self.rsoft = rsoft
            #rosft:避免出現極端值(r->0)而加的緩衝
            值
        self.method = method
            #method:決定要用哪一種疊代方
            法
        self.outfreq = outfreq
            #outfreq:多久Output一次數
            據
        self.outheader = outheader
        self.outscreen = outscreen
        self.visualiztion = visualiztion
        return

    #準備疊代，疊代方法由setup輸入的值決定，粒子的狀態
        來自particles.py
    def evolve(self, dt:float, tmax:float):

        self.dt = dt
        self.tmax = tmax
        nsteps = int(np.ceil(tmax/dt)) #np.ceil功
            能為無條件進入
        time = self.time
        method = self.method
        particles = self.particles

        #這邊我是用教授的RK2跟RK4 method，我亦有自己
            寫RK2跟RK4(用泰勒展開的方式)，不過算過跟老
            師的方法結果會差一點(有時比較準有時比較不
            準)。
        if method.lower() == "euler":
            simulation_method =
                self.simulation_Euler
        elif method.lower() == "rk2":
            simulation_method =
                self._advance_particles_RK2
        elif method.lower() == "rk4":
            simulation_method =
                self._advance_particles_RK4
        elif method.lower() == "frog":
            simulation_method =
                self.simulation_Leap_frog
        else:
            raise ValueError("Unknown method")

        #這裡是真正疊代的地方，下面的method只是計算一
            次dt，這裡才是對時間做疊代
        outfolder = "data_"+self.outheader
        Path(outfolder).mkdir(parents=True,
            exist_ok=True)
        for n in range(nsteps):
            if (dt > tmax-time):
                dt = tmax - time     #避免dt太大

            particles = simulation_method(dt,
                particles)
```

```
            if (n % self.outfreq == 0): #也就是在一定間隔內output一次數據ex. outfreq=200-->200*dt做一次疊代
                if self.outscreen:
                    print("n", n, "Time", time, "dt:", dt )
                fn = self.outheader + "_" + str(n).zfill(6)+".dat"
                fn = outfolder+"/"+fn
                particles.output(fn)

            time+=dt

        print("Simulation finished.")
        return

    #這裡計算加速度(先用kernel計算完再丟回來)，再用這個函數回傳的結果帶入我們的Euler,RK2,RK4疊代
    def calculate_accleration(self, nparticles, mass, position):
        accleration = np.zeros_like(position)
        G = self.G
        rsoft = self.rsoft

        accleration = calculate_accleration_kernel(nparticles, mass, position, accleration, G, rsoft)

        return accleration


    #Euler method(1st order)
    def simulation_Euler(self, dt, particles):
        nparticles = particles.nparticles
        mass = particles.mass
        position = particles.position
        velocity = particles.velocity
        accleration = self.calculate_accleration(nparticles, mass, position)

        #把更新後的值output
        position = position + velocity*dt
        velocity = velocity + accleration*dt
        accleration = self.calculate_accleration(nparticles, mass, position)
        particles.setting_particles(position,velocity,accleration)
        return particles

    #RK2 method(2nd order)(我自己寫的RK2疊代方法)
    def simulation_RK2(self, dt, particles):
        nparticles = particles.nparticles #input資料進去疊代
        mass = particles.mass
        position = particles.position
        velocity = particles.velocity
        accleration = self.calculate_accleration(nparticles, mass, position) #RK2其實就是泰勒展開到第2項，所以就position而言，新的position = position + velocity*dt + accleration*(dt)平方/2
        pos = position

            #而就velocity而言，新的velocity = velocity + accleration(input position)*dt + accleration(input position的一次微分,也就是velocity)*(dt)平方/2
        posd1 = velocity #position的一次微分
        posd2 = accleration
        vel = velocity
        veld1 = accleration #velocity的一次微分
        veld2 = self.calculate_accleration(nparticles, mass, posd1)

        #所以我得到新的position跟velocity
        position = pos + posd1*dt + posd2*(dt**2)/2
        velocity = vel + veld1*dt + veld2*(dt**2)/2
        accleration = self.calculate_accleration(nparticles, mass, position)
        particles.setting_particles(position, velocity, accleration)
        return particles

    #RK2 method(2nd order)
    def _advance_particles_RK2(self, dt, particles):

        nparticles = particles.nparticles
        mass = particles.mass

        pos = particles.position
        vel = particles.velocity
        acc = self.calculate_accleration(nparticles, mass, pos)


        pos2 = pos + vel*dt
        vel2 = vel + acc*dt
        acc2 = self.calculate_accleration(nparticles, mass, pos2)

        pos2 = pos2 + vel2*dt
        vel2 = vel2 + acc2*dt


        pos = 0.5*(pos + pos2)
        vel = 0.5*(vel + vel2)
        acc = self.calculate_accleration(nparticles, mass, pos)

        particles.setting_particles(pos, vel, acc)

        return particles

    #RK4 method(4th order)(我自己寫的RK4)
    def simulation_RK4(self, dt, particles):
        nparticles = particles.nparticles #input資料進去疊代
        mass = particles.mass
        position = particles.position
        velocity = particles.velocity
        accleration = self.calculate_accleration(nparticles, mass, position)
```

```
        #同上，RK4其實就是泰勒展開到第四項
        pos = position
        posd1 = velocity #position的一次微分
        posd2 = accleration
        posd3 =
            self.calculate_accleration(nparticles,
            mass, posd1)
        posd4 =
            self.calculate_accleration(nparticles,
            mass, posd2)

        vel = velocity
        veld1 = accleration #velocity的一次微分
        veld2 =
            self.calculate_accleration(nparticles,
            mass, posd1)
        veld3 =
            self.calculate_accleration(nparticles,
            mass, posd2)
        veld4 =
            self.calculate_accleration(nparticles,
            mass, posd3)

        position = pos + posd1*dt +
            posd2*(dt**2)/2 + posd3*(dt**3)/6 +
            posd4*(dt**4)/24
        velocity = vel + veld1*dt +
            veld2*(dt**2)/2 + veld3*(dt**3)/6 +
            veld4*(dt**4)/24
        accleration =
            self.calculate_accleration(nparticles,
            mass, position)

        particles.setting_particles(position,
            velocity, accleration) #更新數
            值

        return particles

    #RK4 method(4th order)
    def _advance_particles_RK4(self, dt,
        particles):

        nparticles = particles.nparticles
        mass = particles.mass

        # y0
        pos = particles.position
        vel = particles.velocity # k1
        acc =
            self.calculate_accleration(nparticles,
            mass, pos) #
            k1

        dt2 = dt/2
        # y1
        pos1 = pos + vel*dt2
        vel1 = vel + acc*dt2 # k2
        acc1 =
            self.calculate_accleration(nparticles,
            mass, pos1) #
            k2

        # y2
        pos2 = pos + vel1*dt2
        vel2 = vel + acc1*dt2 # k3
        acc2 =
            self.calculate_accleration(nparticles,
            mass, pos2) #
            k3

        # y3
        pos3 = pos + vel2*dt
        vel3 = vel + acc2*dt # k4
        acc3 =
            self.calculate_accleration(nparticles,
            mass, pos3) #
            k4

        # rk4
        pos = pos + (vel + 2*vel1 + 2*vel2 +
            vel3)*dt/6
        vel = vel + (acc + 2*acc1 + 2*acc2 +
            acc3)*dt/6
        acc =
            self.calculate_accleration(nparticles,
            mass, pos)

        # update the particles
        particles.setting_particles(pos, vel,
            acc)

        return particles

    #Leap frog method
    def simulation_Leap_frog(self, dt,
        particles):
        nparticles = particles.nparticles
                                #input資料進去疊
            代
        mass = particles.mass
        position = particles.position
        velocity = particles.velocity
        accleration =
            self.calculate_accleration(nparticles,
            mass, position)

        velocity = velocity + accleration*(dt/2)
        position = position + velocity*dt
        accleration =
            self.calculate_accleration(nparticles,
            mass, position) #先用新的position得出新
            的accleration
        velocity = velocity +
            accleration*(dt/2)              #再帶回
            去疊代velocity

        particles.setting_particles(position,
            velocity, accleration)

        return particles


if __name__ == "__main__":

    pass
```

A. *note for simulator.py*

For simulator.py, although the code is longer, the concept is actually easier. The main problem is face is my RK2 and RK4 method will have a little bit different from the teacher's method, but I feels they are the same.

IV. The Problem I encounter for the result and code

A. the gravitational constant and numba

When I was trying to output the result, I find that when I set gravitational constant =1 and use numba (nthreads> 1), I will get different result even when I use the same method and same input. However, when I didn't use numba, this problem will not occur. Also, when I change gravitational constant to 0.1, the problem will not occur either. I think the problem might be the numba will have a bigger error or less significant digits or something like that. Or maybe is because the rsoft is too big? I am not very sure.

To solve this problem, now I will simply just set G=0.1, which I can get the great result.

B. The running speed of the code

The other problem is that, although I already use numba to accelerate the speed of simulating, It still take me about 5 6 hours to simulate one method of result. So if I need to simulate all results using 100000 particles, it might take me over 1 week to get the result, so finally, I only use 2500 to do the simulation.

Of course, the reason my code run so slow might be my code is not efficient enough, if that is the case, please let me know so I can improve my code.

V. Result

A. result for problem 1

Here, I use 2500 particles, G=0.1, the rest of parameter is same as the question required. I will upload the jupyter notebook which contains the code I use to generate the picture on google classroom. Here I will just post the result (The method I use is RK2):

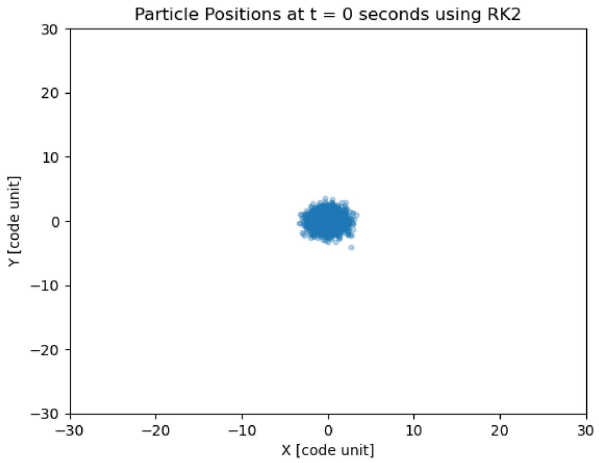

Fig. 1. particles project on x-y plane during t=0 (method:RK2)

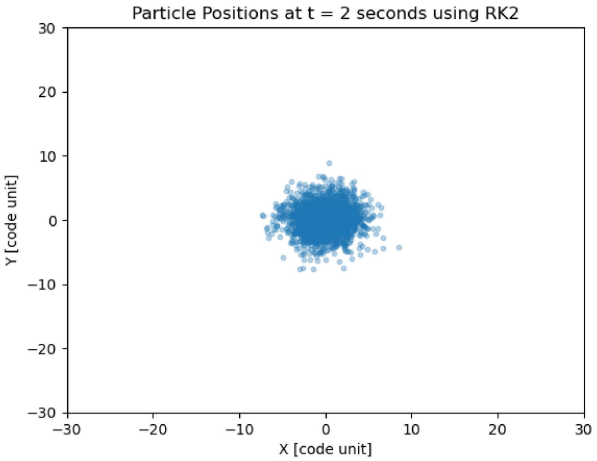

Fig. 2. particles project on x-y plane during t=2 (method:RK2)

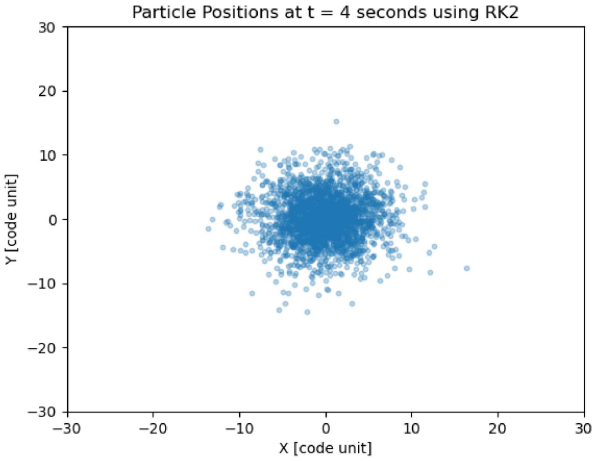

Fig. 3. particles project on x-y plane during t=4 (method:RK2)

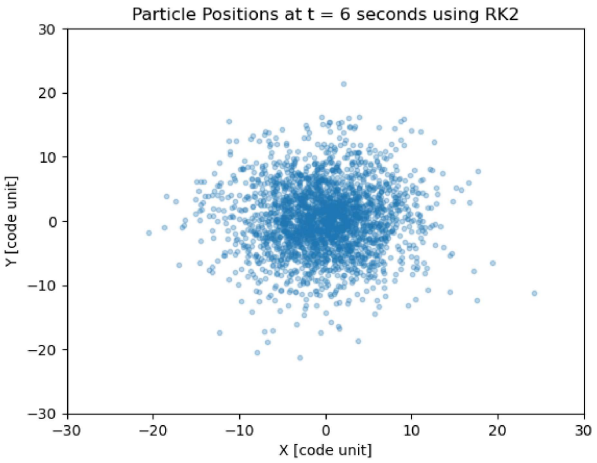

Fig. 4. particles project on x-y plane during t=6 (method:RK2)

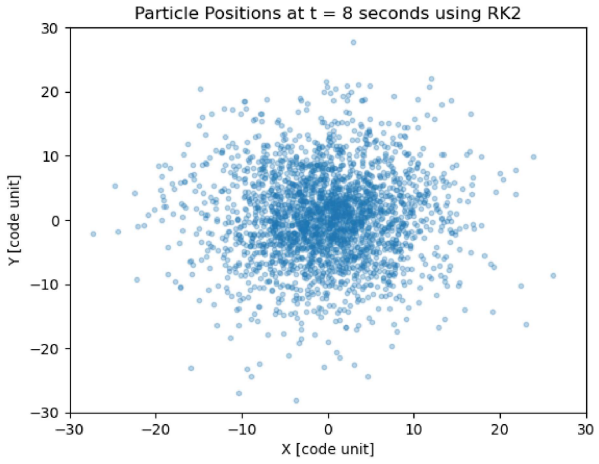

Fig. 5. particles project on x-y plane during t=8 (method:RK2)

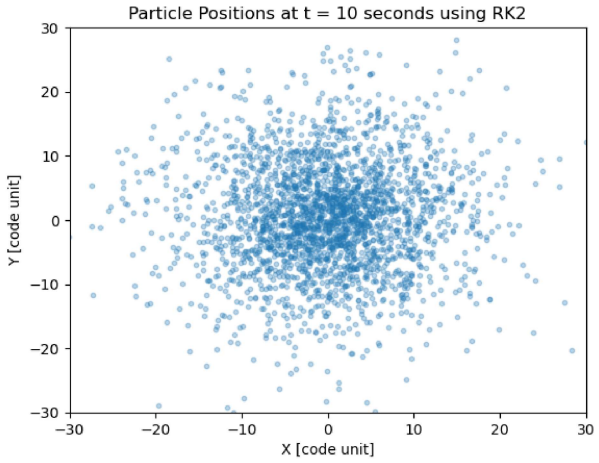

Fig. 6. particles project on x-y plane during t=10 (method:RK2)

and this is the requirement of problem 1.

VI. RESULT FOR PROBLEM 2

This question is just repeating problem 1 with leap frog method. One can see my lead frog method algorithm in my simulator.py, to see if it is right. Here I will just post my result:

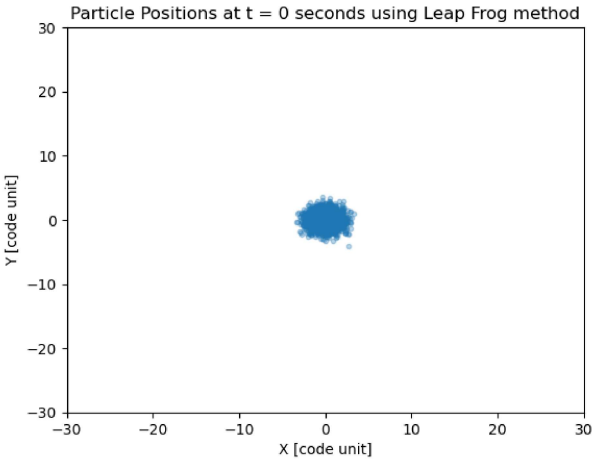

Fig. 7. particles project on x-y plane during t=0 (method:frog)

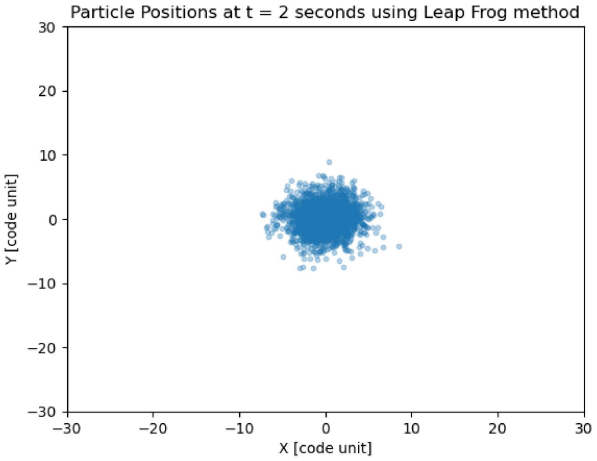

Fig. 8. particles project on x-y plane during t=2 (method:frog)

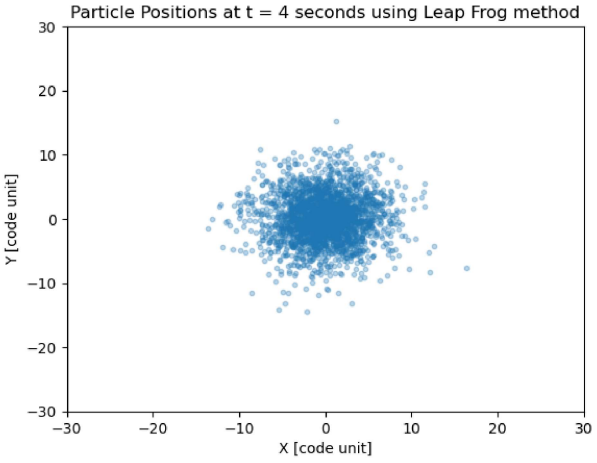

Fig. 9. particles project on x-y plane during t=4 (method:frog)

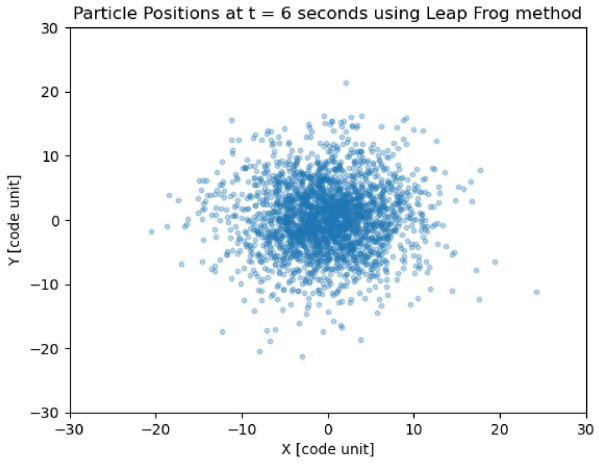

Fig. 10. particles project on x-y plane during t=6 (method:frog)

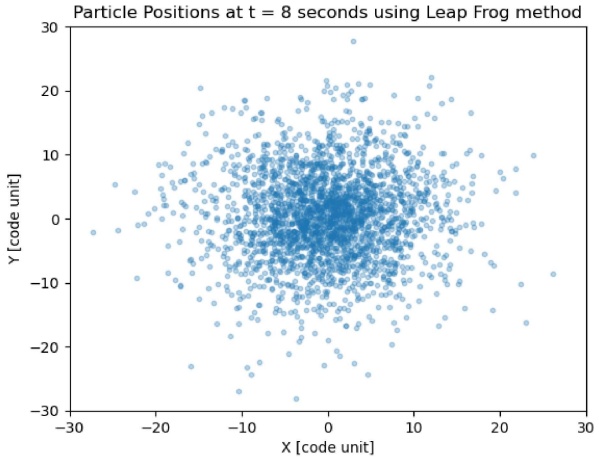

Fig. 11. particles project on x-y plane during t=8 (method:frog)

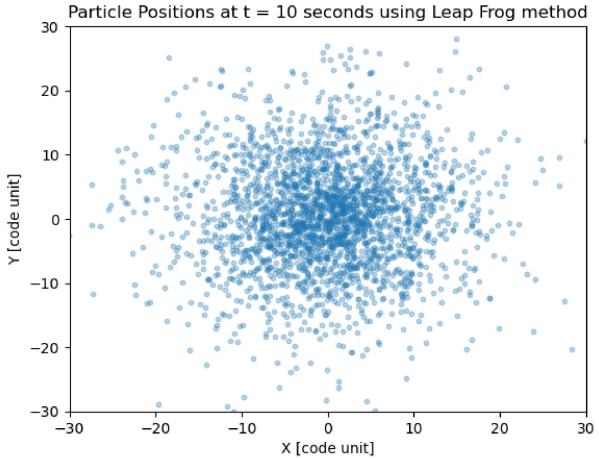

Fig. 12. particles project on x-y plane during t=10 (method:frog)

One can see that the result of problem 1 and 2 is pretty similar, as it should be.

VII. RESULT FOR PROBLEM 3

A. *for 3.(a) and 3.(b)*

The function of the total kinetic energy and potential energy is at particle.py, I will put the code fragment contains these 2 function here:

```
@njit(parallel=True)
def calculate_PE_kernel(nparticles,mass,
                  position,G=0.1,rsoft=0.01):
   pe = np.zeros(nparticles)
   for i in prange(nparticles):
      for j in prange(i+1,nparticles): #避
          免double counting，亦可算出全部粒子的位能
          再除以2
            rij = position[i,:] -
                position[j,:]
            r = np.sqrt(np.sum(rij**2) +
                rsoft**2)
            pe[i] = pe[i] - G * mass[i,0] *
                mass[j,0] / (r)
   return pe
#at Particle class:
   #計算動能，並直接帶入self.ke
   def calculate_KE(self):
      self.ke = np.zeros(self.nparticles)
      for i in range(self.nparticles):
         self.ke[i] =
             0.5*self.mass[i,0]*(np.sum(self.velocity

   #計算位能，並直接帶入self.pe
   def calculate_PE(self,G=0.1,rsoft=0.01):
      self.pe = np.zeros(self.nparticles)
      nparticles = self.nparticles
      mass = self.mass
      position = self.position
      self.pe =
          calculate_PE_kernel(nparticles,mass,positic
```

after all these, I can output the kinetic energy and potential energy for each particle, like down below:

```
def output(self, filename):
      mass = self.mass
      pos = self.position
      vel = self.velocity
      acc = self.acceleration
      tag = self.tag
      time = self.time
      self.calculate_KE()
      self.calculate_PE()
      header = """
            ------------------------------------------
            Data from a 3D direct N-body
                simulation.

            rows are i-particle;
            coumns are :mass, tag, x ,y, z,
                vx, vy, vz, ax, ay, az, ke, pe
            The data is for Homework 2.
            ------------------------------------------
            """
      header += "Time = {}".format(time)
      np.savetxt(filename,(tag[:],mass[:,0],pos[:,0],
                     pos[:,1],pos[:,2],
                     vel[:,0],vel[:,1],vel[:,2],
                     acc[:,0],acc[:,1],acc[:,2],
                     self.ke,self.pe),header=header)
```

```
25        return
```

finally, after all these, I can compute the total kinetic and potential energy at jupyter notebook:

```
particles = Particles(N = num_particles) #將設好的初始值丟入particles class
particles.mass = mass
particles.position = position
particles.velocity = velocity
particles.acceleration = accleration
particles.tag = tag

simulation = NBodySimulator(particles=particles)
simulation.setup(G=0.1, rsoft=0.01, method='rk2',outfreq=200) #因為dt=0.01,所以若要2秒一個數據則頻率為200*dt=2
simulation.evolve(dt = 0.01, tmax = 10.01)

file_paths = [
   'data_nbody\\nbody_000000.dat',
   'data_nbody\\nbody_000200.dat',
   'data_nbody\\nbody_000400.dat',
   'data_nbody\\nbody_000600.dat',
   'data_nbody\\nbody_000800.dat',
   'data_nbody\\nbody_001000.dat'
]

Total_Kinetic_Energy = np.zeros(len(file_paths))
Total_Potential_Energy = np.zeros(len(file_paths))
Total_Energy_RK2 = np.zeros(len(file_paths))

for idx, file_path in enumerate(file_paths):
   data = np.loadtxt(file_path)
   Total_Kinetic_Energy[idx] = np.sum(data[11, :])
   Total_Potential_Energy[idx] = np.sum(data[12, :])
   Total_Energy_RK2[idx] = Total_Kinetic_Energy[idx] + Total_Potential_Energy[idx]

time = np.arange(0, len(file_paths) * 2, 2)

plt.plot(time, Total_Kinetic_Energy, color='red', label='Kinetic Energy')
plt.plot(time, Total_Potential_Energy, color='black', label='Potential Energy')
plt.plot(time, Total_Energy_RK2, color='blue', label='Total Energy')
plt.xlabel('Time')
plt.ylabel('Energy')
plt.title('Time-Energy using rk2')
plt.legend()
plt.show()
print(Total_Kinetic_Energy)
print(Total_Potential_Energy)
print(Total_Energy_RK2)
```

Here I use RK2 method for example. For all these above, I output the kinetic energy and potential energy, also I cam get the total kinetic energy and total potential energy.

B. for 3.(c)

Finally, I need to get the total kinetic energy, potential energy and total energy using each method at t = 0,2,4,6,8,10. For each method, I will put the three kind of energy in one picture, and finally I will put all method's total energy on one figure.

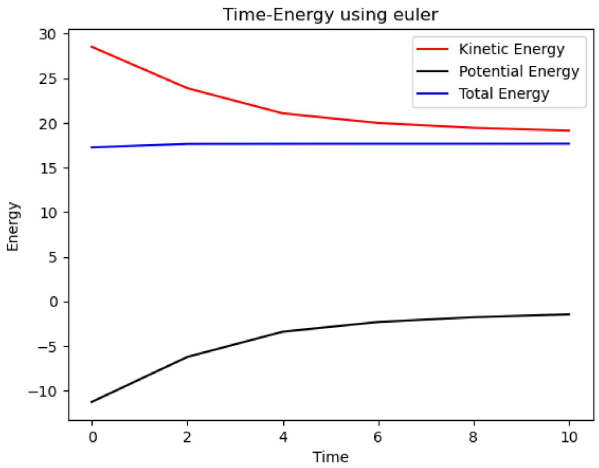

Fig. 13. Energy-time picture(method:euler)

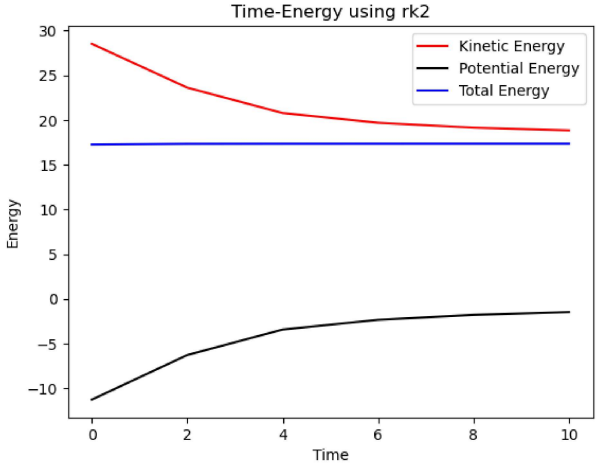

Fig. 14. Energy-time picture(method:RK2)

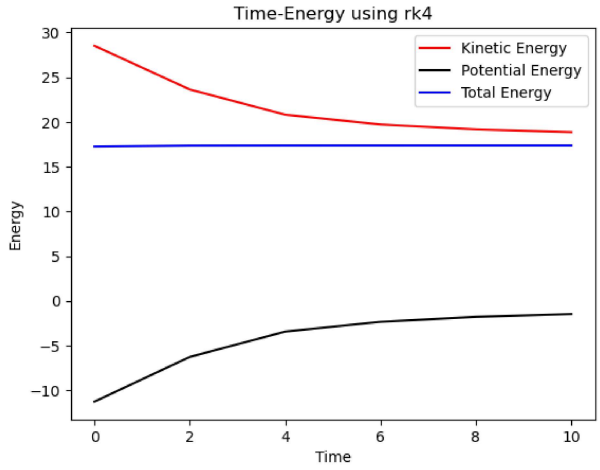

Fig. 15. Energy-time picture(method:RK4)

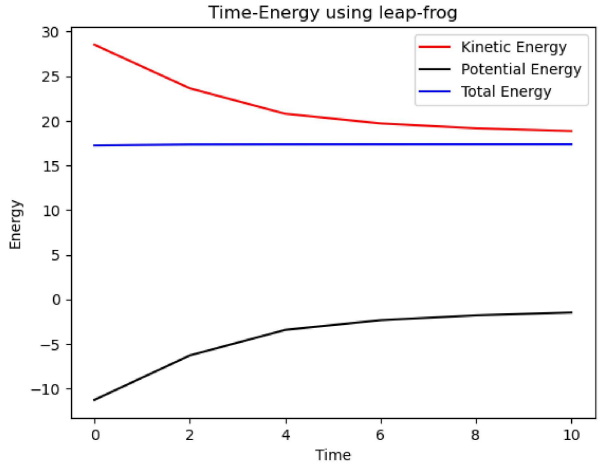

Fig. 16. Energy-time picture(method:leap-frog)

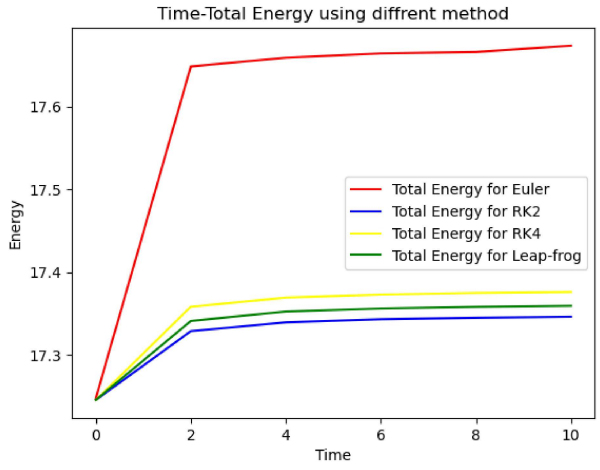

Fig. 17. Total Energy-time picture for each method

after comparing, I see that the Euler has the worst accuracy because the total energy shouldn't change while simulating. On the other hand, RK2, RK4 and leap-frog method all have similar accuracy.

C. for 3.(d)

For the leap-frog method, the order of accuracy is 2. It can be easily see by substituting the new v_i into new x_i, which we can see that new x_i contains second order of dt. The result we got also show that the leap-frog method is very similar to the second order accuracy—RK2.