Computational Physics Lab

Homework 3

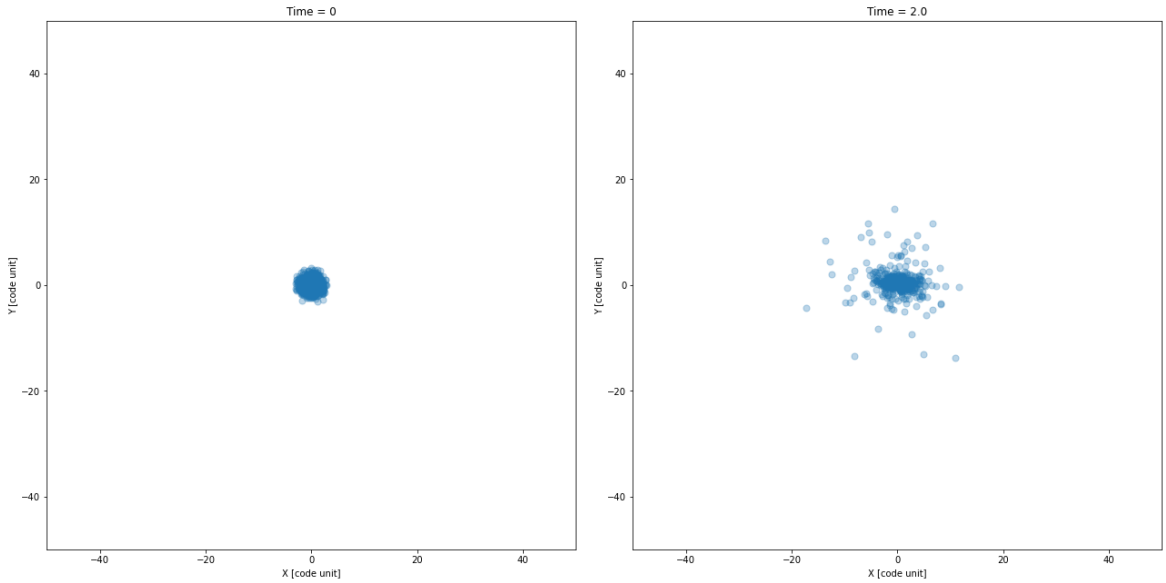
108000204
Yuan-Yen Peng
Dept. of Physics, NTHU
Hsinchu, Taiwan

December 7, 2022

1 Programming Assignments

1.1 (1) Normal Cloud

In this section, in the beginning, we create a cloud of particles ($N = 10^3$) distributed by a normal distribution with zero mean and one variance (or in other words, $s.d. = 1$) in 3D Cartesian coordinates. Additionally, we set the initial particle velocities, and accelerations are distributed by the "same" normal distribution and the system's total mass is 20. We, afterward, use our N body simulation code from t = 0 up to t = 10 with a constant time step $\Delta t = 0.01$ and a soften length $r_{soft} = 0.001$ (here, we adjust this soft length from 0.01 to 0.001 so as to avoid "through the mold"). We, meanwhile, implement `numba @njit` with 8 cores to accelerate the loop speed (`@jit (nopython=True)` have spent more time so I discard to use it). Besides, I, firstly, set the particle's number = 10^5 tried to use either `@jit` or `@njit` to accelerate the code; however, with `RK4` it takes no lowering than 800 minutes! Therefore, I use 10^3, instead (after discussing this problem with professor in Monday's lecture). Figure1 is the group of snap-shot of the normal cloud distribution with the RK4 method and with the extended boundary from -50 to +50. This is a time-consuming task, for orders of 4, RK4, it took 566 (for 10^4: 42015) [s]; for orders of two, RK2 and Leapfrog, they took 392 (for 10^4: 29506), and 321 (for 10^4: 18009) [s], respectively. The last order of one, Euler, took 322 (for 10^4: 17794) [s].



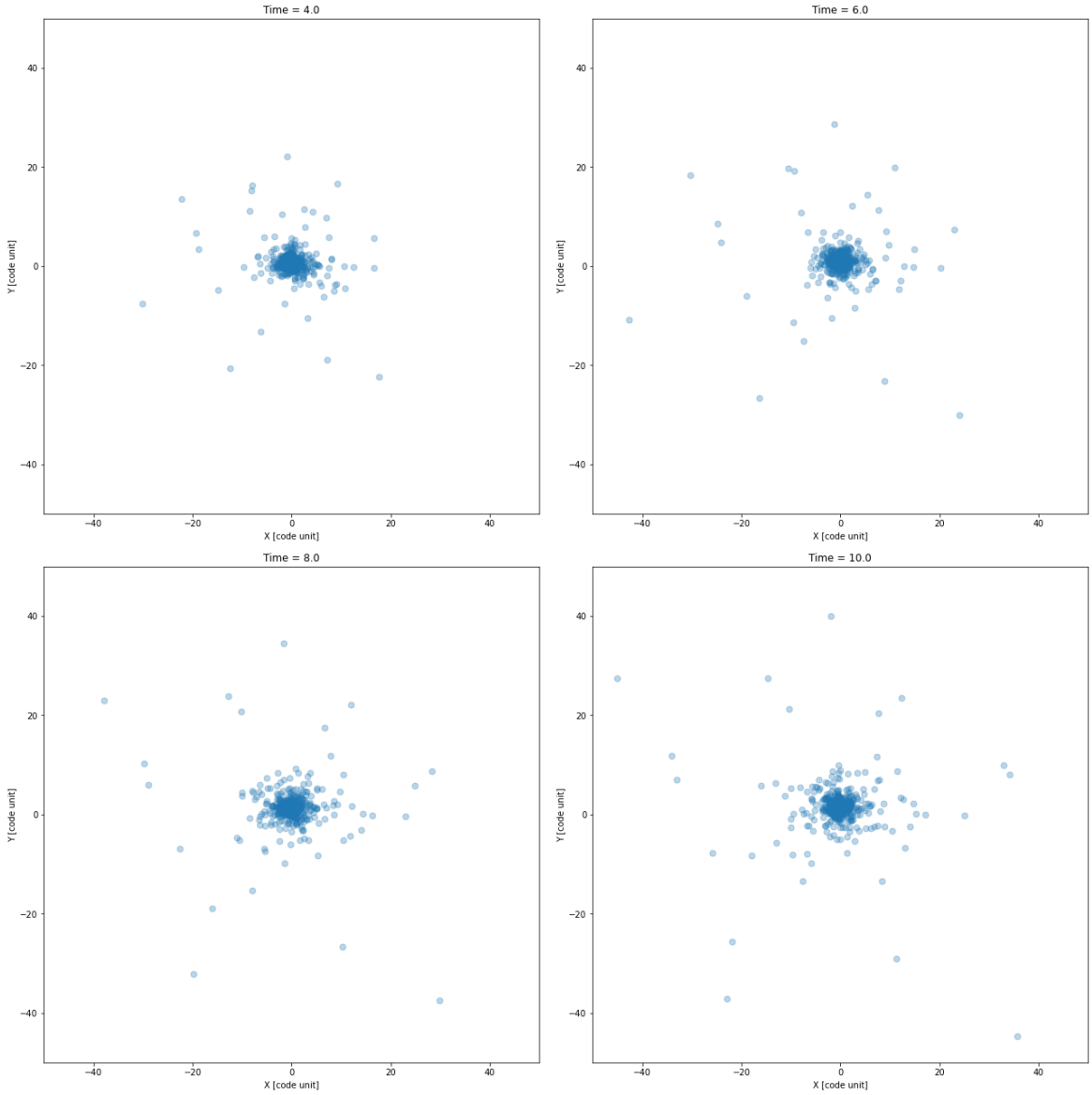


Figure 1: This the normal cloud with RK4, from $t = 0$ to $t = 10$ and $dt = 0.01$.

1.2 (2) Leapfrog method

The leapfrog method is a second-order method for solving an initial value problem. The main idea in this is divided into three parts, kick-drift-kick. For each time step dt, each particle receives a half-step kick,

$$v_{i+1/2} = v_i + a_i \frac{dt}{2}$$

and use the above half-step velocity, and followed by a full-step drift,

$$x_{i+1} = x_i + v_{i+1/2} dt$$

and lastly, use the above drift, and followed by another half-step kick,

$$v_{i+1} = v_{i+1/2} + a_{i+1} \frac{dt}{2}$$

In the following subsection, we will discuss and compare the Leapfrog, Euler, RK2, and RK4 as well.

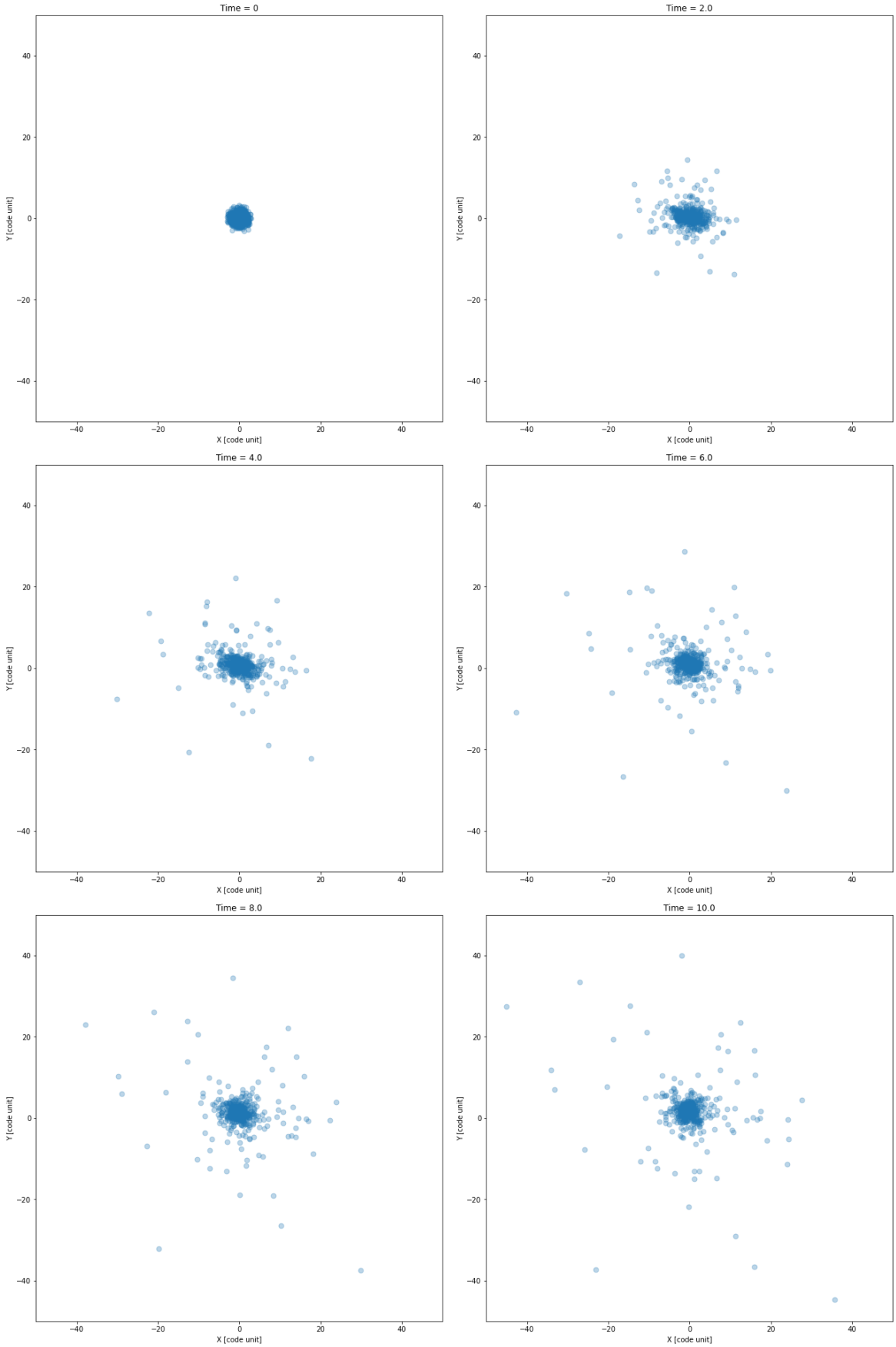


Figure 2: This the normal cloud with Leapfrog, from $t = 0$ to $t = 10$ and $dt = 0.01$.

1.3 (3) Energy comparison

In this section, we will discuss kinetic energy, potential energy, and total energy. The definitions are below:

$$U = -\frac{Gm_i m_j}{r + r_{soft}}$$

$$K = \frac{1}{2} m_i v_i^2$$

$$E = U + K$$

where the subscriptions indicate which particles; G is the gravitational constant, here we set it to 1; m is the mass of each particle; r and r_{soft} is the distance between two particles and the tolerance of distance, which can avoid the nominator equal to zero. After defining all the parameters, we use four different methods, Euler, RK2, RK4, and Leapfrog, and implement the initial conditions as in section 1 to rerun the codes.

Firstly, we use the "two-body" problem to check whether our `nbody` make sense or not. Therefore, we check with the energies in the solar system and two particles situation. In Figure 3, it is obvious that the total energy is conserved implying that our `nbody` package is "legitimate". On the other hand, in Figure 4, we use the same code but in total particle's number = 2, and the left is no collision situation, the total energy is conserved and the potential energy gradually increases with the kinetic energy decrease. However, the right also has the same conditions but with the "collision" which leads to a positive peak in the kinetic energy and also a negative peak in potential. Although physically, the total energy is conserved, this numerical simulation cannot handle the "through the mold" situation! Because the total energy is proportional to the first order of inverse r; the acceleration is proportional to the second order of inverse r (gravitational force) which directly influences velocities. Ergo, we can see that kinetic energy dominates the total energy and will have a manifest peak when collisions occur.

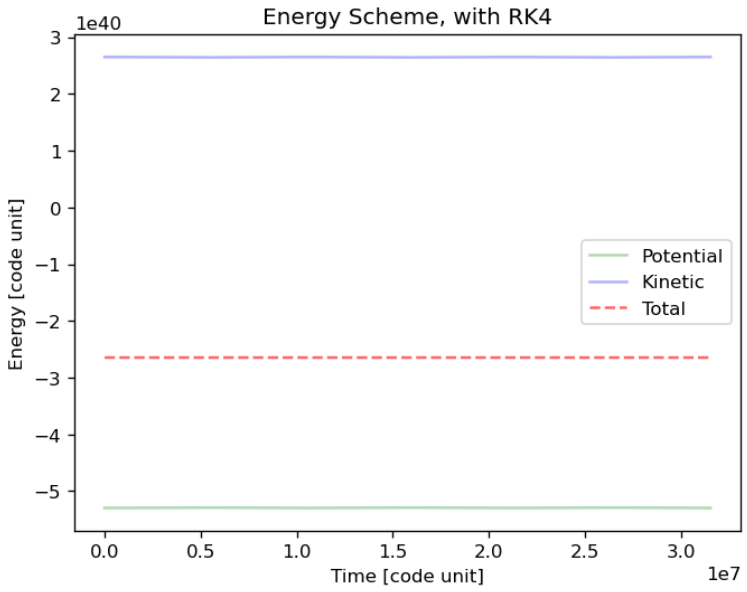


Figure 3: This the solar system simulation we use in the lecture, the source code I put it in section Code.

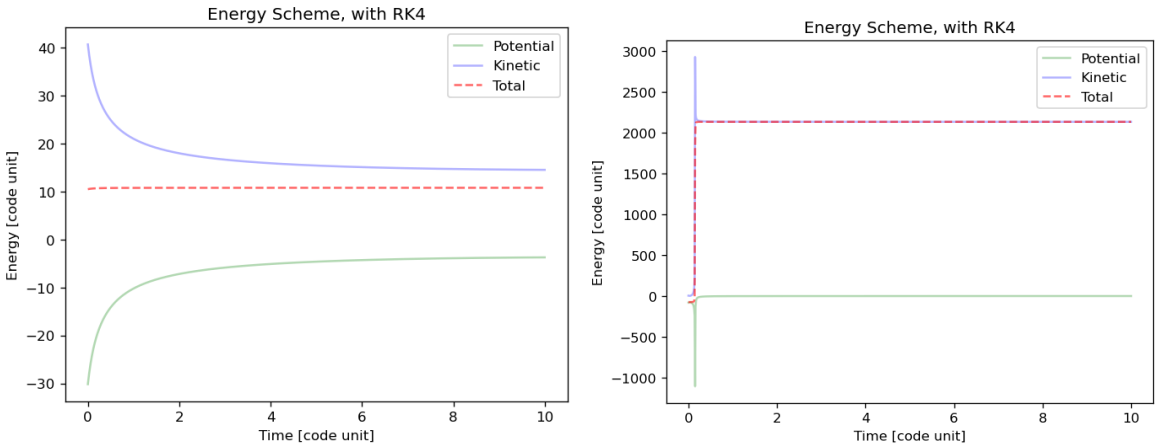


Figure 4: This the `nbody` simulation with two particles. The left is a non-collision situation; the right happened to collide.

See in Figure 5~8, we can find out that the potential energy is much smaller than the kinetic energy owing to the small masses of particles. On the other hand, the total energy is dominated by the kinetic part, which makes sense to us in this simulation (reasons have been elaborated above). In Figure 5~8, we can also see the total energy is not the "constant" because occur many collisions! In Figure 1, 2, we can find that no matter what time it is, there are some particles always crowded in the center; at the same time, they happen collisions! Besides, if we take the higher resolution (higher orders algorithm), the more severe oscillations the system behaves. These phenomena not merely can see in the RK4 but in other low resolutions as well.

It is obvious that Euler (first order) is the most appeased one yet we cannot conclude that it is the "precise" one. In the other words, it may be the most distorted one. While if we take the shape angle to the problem, for the second order, RK2 is more shapeless (fewer fluctuations) than Leapfrog. Therefore, we can sum up that if individuals want to calculate a large number of particles problem, and take shapes into account, they can choose the Leapfrog method because it has not merely high speed but also high sharpness. (detail of accuracy can see in the next section, the accuracy compared with the benchmark is within one order.)

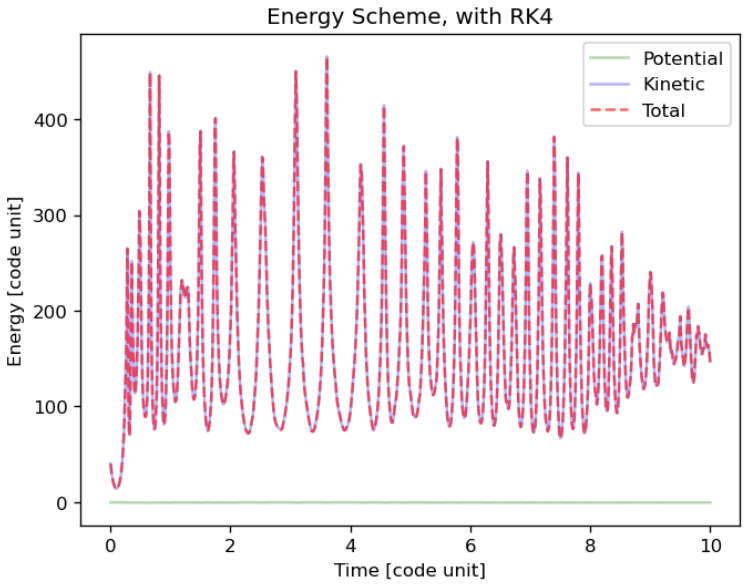


Figure 5: RK4, y-axis is Energy; x-axis is time.

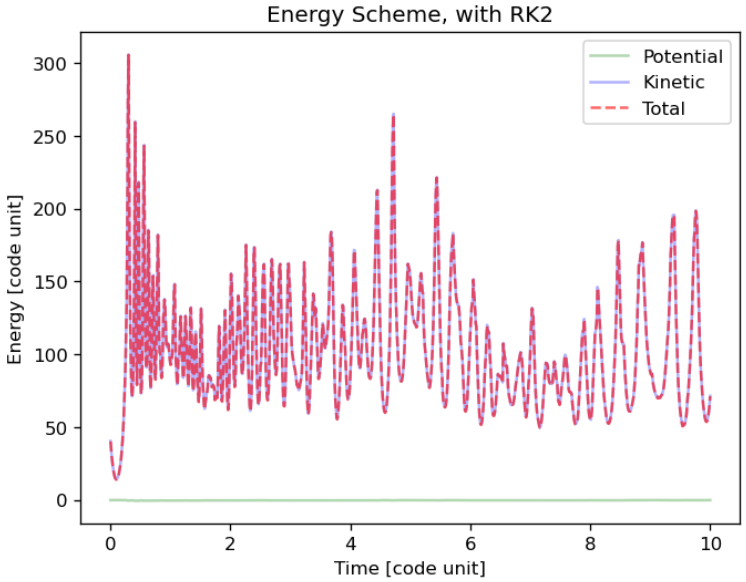


Figure 6: RK2, y-axis is Energy; x-axis is time.

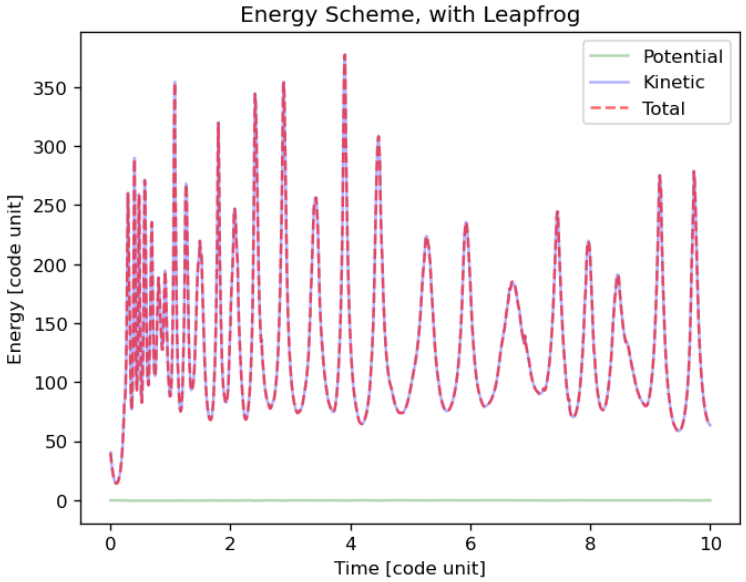


Figure 7: Leapfrog, y-axis is Energy; x-axis is time.

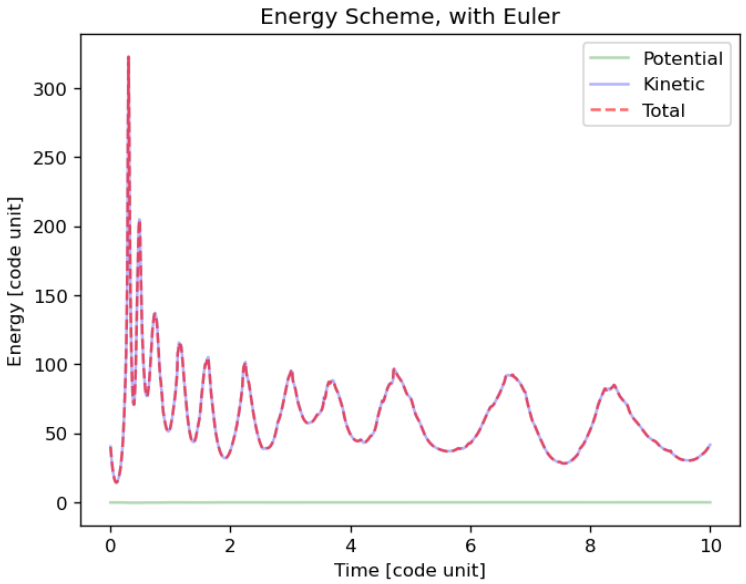


Figure 8: Euler, y-axis is Energy; x-axis is time.

We use RK4 (the highest preciseness algorithm in this simulation) as the theoretical benchmark even though it has acute fluctuations. So as to analyze the accuracy of the Leapfrog method, we set the RK4 (fourth order) as a different comparison benchmark. We, likewise, applied a logarithm of 10 to investigate the order of accuracy thereafter. In Figure 9, we can see the average accuracy comparisons between Leapfrog methods and the benchmark. Simultaneously, Rk2 and Euler are also in this benchmark comparison. We can see that the second-order has familiar accuracy which is better than the first-order algorithm.

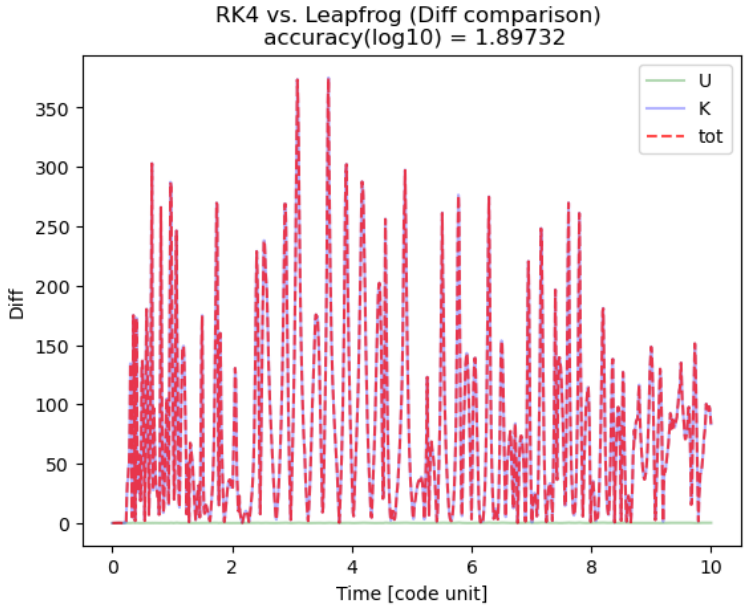


Figure 9: In the comparison of energy schemes between the RK4 and Leapfrog methods, the average accuracy is approximately 0.9 orders.

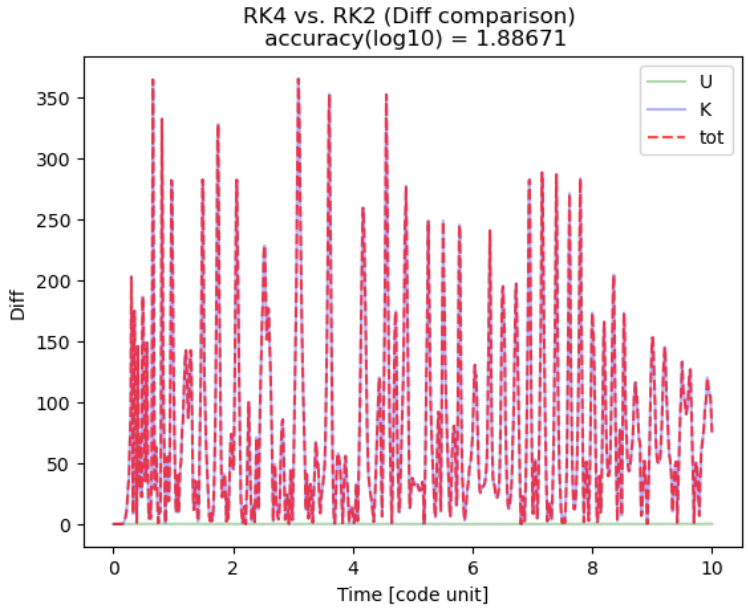


Figure 10: In the comparison of energy schemes between the RK4 and RK2 methods, the average accuracy is approximately 0.9 orders.

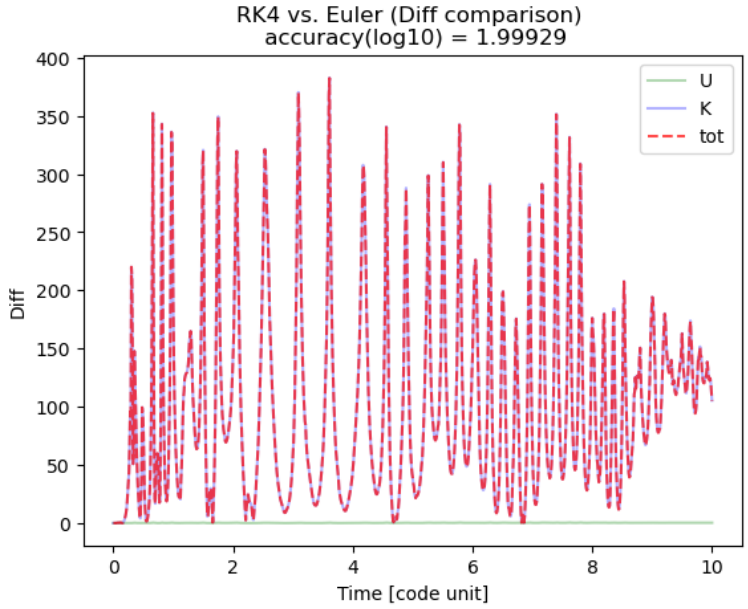


Figure 11: In the comparison of energy schemes between the RK4 and Euler methods, the average accuracy is approximately 0.9 orders.

2 Codes

All the codes are transferred from jupyterlab or python codes; hence, if you want to re-run them, please see the source code in the attached files or my GitHub repository:

<https://github.com/gary20000915/Comphyslab-HW3.git>

2.1 `particles.py`

```python
import numpy as np

class Particles:
    """

    The Particles class handles all particle properties

    for the N-body simulation.

    """
    def __init__(self, N:int = 100):
        """
        Prepare memories for N particles

        :param N: number of particles.

        By default: particle properties include:
               nparticles: int. number of particles
               _masses: (N,1) mass of each particle
               _positions: (N,3) x,y,z positions of each particle
               _velocities: (N,3) vx, vy, vz velocities of each particle
               _accelerations: (N,3) ax, ay, az accelerations of each partciel
               _tags: (N)  tag of each particle
               _time: float. the simulation time

        """
        self.nparticles = N
        self._time = 0 # initial time = 0
        self._masses = np.ones((N, 1))
        self._positions = np.zeros((N, 3))
        self._velocities = np.zeros((N, 3))
        self._accelerations = np.zeros((N, 3))
        self._tags = np.linspace(1, N, N)
        self._U = np.zeros((N, 1))
        self._K = np.zeros((N, 1))

        return


    def get_time(self):
        return self._time

    def get_masses(self):
        return self._masses

    def get_positions(self):
        return self._positions

    def get_velocities(self):
        return self._velocities

    def get_accelerations(self):
        return self._accelerations
```

```
54
55      def get_tags(self):
56          return self._tags
57
58      def get_time(self):
59          return self._time
60
61      def get_U(self):
62          return self._U
63
64      def get_K(self):
65          return self._K
66
67
68      def set_time(self, time):
69          self._time = time
70          return
71
72      def set_masses(self, mass):
73          self._masses = mass
74          return
75
76      def set_positions(self, pos):
77          self._positions = pos
78          return
79
80      def set_velocities(self, vel):
81          self._velocities = vel
82          return
83
84      def set_accelerations(self, acc):
85          self._accelerations = acc
86          return
87
88      def set_tags(self, IDs):
89          self._tags = IDs
90          return
91
92      def set_U(self, U):
93          self._U = U
94          return
95
96      def set_K(self, K):
97          self._K = K
98          return
99
100     def output(self, fn, time):
101         """
102         Write simulation data into a file named "fn"
103         """
104         mass = self._masses
105         pos  = self._positions
106         vel  = self._velocities
```

```
        acc = self._accelerations
        tag = self._tags
        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

               NTHU, Computational Physics Lab

               ------------------------------------------------------
               """
        header += "Time = {}".format(time)
        np.savetxt(fn,(tag[:],mass[:,0],pos[:,0],pos[:,1],pos[:,2],
                        vel[:,0],vel[:,1],vel[:,2],
                        acc[:,0],acc[:,1],acc[:,2]),header=header)

        return
```

2.2 simulation.py

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

"""

This program solve 3D direct N-particles simulations
under gravitational forces.

This file contains two classes:

1) Particles: describes the particle properties
2) NbodySimulation: describes the simulation

Usage:

    Step 1: import necessary classes

    from nbody import Particles, NbodySimulation

    Step 2: Write your own initialization function


        def initialize(particles:Particles):
            ....
            ....
            particles.set_masses(mass)
```

```
            particles.set_positions(pos)
            particles.set_velocities(vel)
            particles.set_accelerations(acc)

            return particles

    Step 3: Initialize your particles.

        particles = Particles(N=100)
        initialize(particles)


    Step 4: Initial, setup and start the simulation

        simulation = Simulation(particles)
        simulation.setip(...)
        simulation.evolve(dt=0.001, tmax=10)


Author: Yuan-Yen Peng (editted from Prof. Kuo-Chuan Pan, NTHU 2022.10.30)
Dept. of Physics, NTHU
Date: Npv. 28, 2022
For the course, computational physics lab

"""

set_num_threads(8)
@njit(parallel=True)
def ACC_njit(N, posx, posy, posz, G, mass, rsoft, U, K, vel_square):
    '''
    Acceleration with numba njit for loops (parallel)

    :param N: number of particles
    :param posx: position x
    :param posy: position y
    :param posz: position z
    :param G: gravitational constant
    :param mass: mass
    '''
    acc = np.zeros((N, 3))
    for i in prange(N):
        for j in prange(N):
            if j > i:
                x = posx[i] - posx[j]
                y = posy[i] - posy[j]
                z = posz[i] - posz[j]
                r = np.sqrt(x**2 + y**2 + z**2) + rsoft
                theta = np.arccos(z / r)
                phi = np.arctan2(y, x)
                F = - G * mass[i, 0] * mass[j, 0] / np.square(r)
                Fx = F * np.cos(phi) * np.sin(theta)
                Fy = F * np.sin(phi) * np.sin(theta)
                Fz = F * np.cos(theta)
```

```
                # Fz = 0

                acc[i, 0] += Fx / mass[i, 0]
                acc[j, 0] -= Fx / mass[j, 0]

                acc[i, 1] += Fy / mass[i, 0]
                acc[j, 1] -= Fy / mass[j, 0]

                acc[i, 2] += Fz / mass[i, 0]
                acc[j, 2] -= Fz / mass[j, 0]

                U[i] = - G * mass[i, 0] * mass[j, 0] / r
                K[i] = 0.5 * (mass[i, 0] * np.sum(vel_square[i, :])
                              + mass[j, 0] * np.sum(vel_square[j, :]))
    U = np.sum(U)
    K = np.sum(K)

    return acc, U, K

class NbodySimulation:
    """

    The N-body Simulation class.

    """

    def __init__(self,particles:Particles):
        """
        Initialize the N-body simulation with given Particles.

        :param particles: A Particles class.

        """

        # store the particle information
        self.nparticles = particles.nparticles
        self.particles = particles

        # Store physical information
        self.time = 0.0 # simulation time

        # Set the default numerical schemes and parameters
        self.setup()

        return

    def setup(self, G=1,
                  rsoft=0.01,
                  method="RK2",
                  io_freq=10,
                  io_title="particles",
                  io_screen=True,
                  visualized=False):
```

```
        """
        Customize the simulation enviroments.

        :param G: the graivtational constant
        :param rsoft: float, a soften length
        :param meothd: string, the numerical scheme
                       support "Euler", "RK2", and "RK4"

        :param io_freq: int, the frequency to outupt data.
                       io_freq <=0 for no output.
        :param io_title: the output header
        :param io_screen: print message on screen or not.
        :param visualized: on the fly visualization or not.

        """
        # TODO:
        self.G = G
        self.rsoft = rsoft
        self.method = method
        self.io_freq = io_freq
        self.io_title = io_title
        self.io_screen = io_screen
        self.visualized = visualized
        return

    def evolve(self, dt:float=0.01, tmax:float=1):
        """

        Start to evolve the system

        :param dt: time step
        :param tmax: the finial time

        """
        # TODO:
        method = self.method
        if method=="Euler":
            _update_particles = self._update_particles_euler
        elif method=="RK2":
            _update_particles = self._update_particles_rk2
        elif method=="RK4":
            _update_particles = self._update_particles_rk4
        elif method=="Leapfrog":
            _update_particles = self._update_particles_lf
        else:
            print("No such update meothd", method)
            quit()

        # prepare an output folder for lateron output
        io_folder = "data_"+self.io_title
        Path(io_folder).mkdir(parents=True, exist_ok=True)
        io_folder_fig = "fig_" + self.io_title
        Path(io_folder_fig).mkdir(parents=True, exist_ok=True)
```

```
190
191         # ====================================================
192         #
193         # The main loop of the simulation
194         #
195         # ====================================================
196
197         # TODO:
198         particles = self.particles # call the class: Particles
199         n = 0
200         t = self.particles.get_time()
201         t1 = time.time()
202         step = int(tmax / dt) + 1
203         UU = np.zeros((step, 1))
204         KK = np.zeros((step, 1))
205         EE = np.zeros((step, 1))
206         for i in range(step):
207             # update particles
208             particles = _update_particles(dt, particles)[0]
209             UU[i] = _update_particles(dt, particles)[1]
210             KK[i] = _update_particles(dt, particles)[2]
211             EE[i] = UU[i] + KK[i]
212             # update io
213             if (n % self.io_freq == 0):
214                 if self.io_screen:
215                     # print('n = ', n, 'time = ', t, 'dt = ', dt)
216                     # output
217                     fn = io_folder+"/data_"+self.io_title+"_"+str(n).zfill(5)+".txt"
218                     print(fn)
219                     self.particles.output(fn, t)
220
221                     # savefig
222                     scale = 50
223                     fig, ax = plt.subplots()
224                     fig.set_size_inches(10.5, 10.5, forward=True)
225                     fig.set_dpi(72)
226                     ax.set_xlim(-1*scale,1*scale)
227                     ax.set_ylim(-1*scale,1*scale)
228                     ax.set_aspect('equal')
229                     ax.set_xlabel('X [code unit]')
230                     ax.set_ylabel('Y [code unit]')
231                     pos = particles.get_positions()
232                     plt.title(f'Time = {np.round(t, 0)}')
233                     FIG = f'{io_folder_fig}/fig_{self.io_title}_{str(int(0.01 *
                            n)).zfill(1)}.png'
234                     ax.scatter(pos[:, 0], pos[:, 1], s = 50, alpha = .3)
235                     plt.savefig(FIG)
236                     plt.show()
237
238             # update time
239             if t + dt > tmax:
240                 dt = tmax - t
241             t += dt
```

```
            n += 1

        T = np.linspace(0, tmax, step)
        plt.figure(dpi=120)
        plt.plot(T, UU, 'g', alpha = .3, label = 'Potential')
        plt.plot(T, KK, 'b', alpha = .3, label = 'Kinetic')
        plt.plot(T, EE, '--r', alpha = .6, label = 'Total')
        plt.xlabel('Time [code unit]')
        plt.ylabel('Energy [code unit]')
        plt.title(f'Energy Scheme, with {method}')
        plt.legend()
        plt.show()
        t2 = time.time()
        print("Time diff: ", t2 - t1)
        print("Done!")

        return UU, KK

    def _calculate_acceleration(self, mass, pos):
        """
        Calculate the acceleration.
        """
        # TODO:
        particles = self.particles
        G = self.G
        rsoft = self.rsoft
        posx = pos[:, 0]
        posy = pos[:, 1]
        posz = pos[:, 2]
        N = self.nparticles
        U = particles.get_U()
        K = particles.get_K()
        vel = particles.get_velocities()
        vel_square = np.square(vel)

        arr = ACC_njit(N, posx, posy, posz, G, mass, rsoft, U, K, vel_square)

        return arr

    def _update_particles_euler(self, dt, particles:Particles):
        # TODO:
        mass = particles.get_masses()
        pos = particles.get_positions()
        vel = particles.get_velocities()
        acc = self._calculate_acceleration(mass, pos)[0]
        U = self._calculate_acceleration(mass, pos)[1]
        K = self._calculate_acceleration(mass, pos)[2]

        y0 = np.array([pos, vel])
        yder = np.array([vel, acc])

        y0 = np.add(y0, yder * dt)
        acc = self._calculate_acceleration(mass, y0[0])[0]
```

```
295
296         particles.set_positions(y0[0])
297         particles.set_velocities(y0[1])
298         particles.set_accelerations(acc)
299
300         return particles, U, K
301
302     def _update_particles_rk2(self, dt, particles:Particles):
303         # TODO:
304         mass = particles.get_masses()
305         pos = particles.get_positions()
306         vel = particles.get_velocities()
307         acc = self._calculate_acceleration(mass, pos)[0]
308         U = self._calculate_acceleration(mass, pos)[1]
309         K = self._calculate_acceleration(mass, pos)[2]
310
311         y0 = np.array([pos, vel])
312         yder = np.array([vel, acc])
313         k1 = yder
314         y_temp = y0 + dt * k1
315         acc = self._calculate_acceleration(mass, y_temp[0])[0]
316         k2 = np.array([y_temp[1], acc])
317         y0 = np.add(y0, (dt / 2) * (k1 + k2))
318         acc = self._calculate_acceleration(mass, y0[0])[0]
319
320         particles.set_positions(y0[0])
321         particles.set_velocities(y0[1])
322         particles.set_accelerations(acc)
323
324         return particles, U, K
325
326     def _update_particles_rk4(self, dt, particles:Particles):
327         # TODO:
328         mass = particles.get_masses()
329         pos = particles.get_positions()
330         vel = particles.get_velocities()
331         acc = self._calculate_acceleration(mass, pos)[0]
332         U = self._calculate_acceleration(mass, pos)[1]
333         K = self._calculate_acceleration(mass, pos)[2]
334
335         y0 = np.array([pos, vel])
336         yder = np.array([vel, acc])
337         k1 = yder
338         y_temp = y0 + 0.5 * dt * k1
339         acc = self._calculate_acceleration(mass, y_temp[0])[0]
340         k2 = np.array([y_temp[1], acc])
341         y_temp = y0 + 0.5 * dt * k2
342         acc = self._calculate_acceleration(mass, y_temp[0])[0]
343         k3 = np.array([y_temp[1], acc])
344         y_temp = y0 + dt * k3
345         acc = self._calculate_acceleration(mass, y_temp[0])[0]
346         k4 = np.array([y_temp[1], acc])
347
```

```
        y0 = np.add(y0, (1/6) * dt * (k1 + 2*k2 + 2*k3 + k4))
        acc = self._calculate_acceleration(mass, y0[0])[0]

        particles.set_positions(y0[0])
        particles.set_velocities(y0[1])
        particles.set_accelerations(acc)

        return particles, U, K

    def _update_particles_lf(self, dt, particles:Particles):
        # TODO:
        mass = particles.get_masses()
        pos = particles.get_positions()
        vel = particles.get_velocities()
        U = self._calculate_acceleration(mass, pos)[1]
        K = self._calculate_acceleration(mass, pos)[2]

        acc = self._calculate_acceleration(mass, pos)[0]
        vel_prime = vel + acc * 0.5 * dt
        pos = pos + vel_prime * dt
        acc = self._calculate_acceleration(mass, pos)[0]
        vel = vel_prime + acc * 0.5 * dt

        particles.set_positions(pos)
        particles.set_velocities(vel)
        particles.set_accelerations(acc)

        return particles, U, K


if __name__=='__main__':

    # test Particles() here
    particles = Particles(N=10)
    # test NbodySimulation(particles) here
    sim = NbodySimulation(particles=particles)
    sim.setup(G = 6.67e-8, io_freq=2, io_screen=True, io_title="test")
    sim.evolve(dt = 1, tmax = 10)
    print(sim.G)
    print("Done")
```

2.3 NormalCloud.ipynb

```
# %% [markdown]
# ## Homework 3
# ### Programming Assignment
# ### 111 Computational Physics Lab
#   >Author: Yuan-Yen Peng 108000204
#   >Email: garyphys0915@gapp.nthu.edu.com
#   >Date: Nov. 11, 2022
#   >LICENCE: MIT

```

```
# %%
import numpy as np
import glob
from numba import jit, njit, prange, set_num_threads
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
from nbody.particles import Particles
from nbody.simulation import NbodySimulation

# %%
problem_name = "NormalCloud"
Num = int(2)
tmax = 10
dt = 0.01
step = int(tmax / dt)
r_soft = 0.0001

radn = np.zeros((Num, 3))
for i in range(Num):
    mu, sigma = 0, 1 # mean = 0; variance = 1, i.e., standard deviation = sqrt(var)
        = 1
    radn[i, :] = np.random.normal(mu, sigma, 3)

# %%
set_num_threads(8)
@njit(parallel = True)
def generator(radn, N, positions, velocities, accelerations):
        for i in prange(N):
            positions[i, :] = radn[i]
            velocities[i, :] = radn[i]
            accelerations[i, :] = radn[i]

        return positions, velocities, accelerations

# %%
def initialRandomParticles(radn, N):
        """
        Initial particles

        """
        total_mass = 20
        particles = Particles(N = N)

        positions = particles.get_positions()
        velocities = particles.get_velocities()
        accelerations = particles.get_accelerations()
        masses = particles.get_masses() # ones array (size = N)
        mass = total_mass / particles.nparticles # single particel's mass

        particles.set_masses((masses * mass))
        particles.set_positions(generator(radn, N, positions, velocities,
            accelerations)[0])
```

```
61          particles.set_velocities(generator(radn, N, positions, velocities,
                accelerations)[1])
62          particles.set_accelerations(generator(radn, N, positions, velocities,
                accelerations)[2])
63
64          return particles
65
66  # %% [markdown]
67  # solve with t = 0 ~ 10 with dt = 0.01 and r_soft = 0.01.
68
69  # %%
70  # Initial particles here.
71  method = "RK4"
72  particles = initialRandomParticles(radn, N = Num)
73  # Run the n-body simulations
74  sim = NbodySimulation(particles)
75  sim.setup(G=1,method=method
76            ,io_freq=200
77            ,io_title=problem_name
78            ,io_screen=True
79            ,visualized=False)
80  Energy_RK4 = sim.evolve(dt=dt,tmax=tmax)
81
82  # %%
83  # Initial particles here.
84  method = "RK2"
85  particles = initialRandomParticles(radn, N = Num)
86  # Run the n-body simulations
87  sim = NbodySimulation(particles)
88  sim.setup(G=1,method=method
89            ,io_freq=200
90            ,io_title=problem_name
91            ,io_screen=True
92            ,visualized=False)
93  Energy_RK2 = sim.evolve(dt=dt,tmax=tmax)
94
95  # %%
96  # Initial particles here.
97  method = "Leapfrog"
98  particles = initialRandomParticles(radn, N = Num)
99  # Run the n-body simulations
100 sim = NbodySimulation(particles)
101 sim.setup(G=1,method=method
102           ,io_freq=200
103           ,io_title=problem_name
104           ,io_screen=True
105           ,visualized=False)
106 Energy_LF = sim.evolve(dt=dt,tmax=tmax)
107
108 # %%
109 # Initial particles here.
110 method = "Euler"
111 particles = initialRandomParticles(radn, N = Num)
```

```
# Run the n-body simulations
sim = NbodySimulation(particles)
sim.setup(G=1,method=method,
          io_freq=200,
          io_title=problem_name,
          io_screen=True,
          visualized=False)
Energy_Euler = sim.evolve(dt=dt,tmax=tmax)

# %%
TT = len(Energy_RK4[0])
Time = np.linspace(0, tmax, len(Energy_RK4[0]))
Diff_tot = np.abs((Energy_RK4[0] + Energy_RK4[1]) - (Energy_LF[0] + Energy_LF[1]))
Diff = np.abs(np.array(Energy_RK4) - np.array(Energy_LF))
avg = np.average(Diff_tot)
accuracy = np.log10(avg)
print("Average = ", avg)
print("Accuracy (log) = ", accuracy)

plt.plot(Time, Diff[0], "g", alpha = .3, label = "U")
plt.plot(Time, Diff[1], "b", alpha = .3, label = "K")
plt.plot(Time, Diff_tot, "--r", alpha = .7, label = "tot")
plt.xlabel("Time [code unit]")
plt.ylabel("Diff")
plt.title(f"RK4 vs. Leapfrog (Diff comparison) \n accuracy(log10) =
    {np.round(accuracy, 5)}")
plt.legend()
plt.show()

# %%
Energy_LF = Energy_RK2
TT = len(Energy_RK4[0])
Time = np.linspace(0, tmax, len(Energy_RK4[0]))
Diff_tot = np.abs((Energy_RK4[0] + Energy_RK4[1]) - (Energy_LF[0] + Energy_LF[1]))
Diff = np.abs(np.array(Energy_RK4) - np.array(Energy_LF))
avg = np.average(Diff_tot)
accuracy = np.log10(avg)
print("Average = ", avg)
print("Accuracy (log) = ", accuracy)

plt.plot(Time, Diff[0], "g", alpha = .3, label = "U")
plt.plot(Time, Diff[1], "b", alpha = .3, label = "K")
plt.plot(Time, Diff_tot, "--r", alpha = .7, label = "tot")
plt.xlabel("Time [code unit]")
plt.ylabel("Diff")
plt.title(f"RK4 vs. RK2 (Diff comparison) \n accuracy(log10) = {np.round(accuracy,
    5)}")
plt.legend()
plt.show()

# %%
Energy_LF = Energy_Euler
TT = len(Energy_RK4[0])
```

```
Time = np.linspace(0, tmax, len(Energy_RK4[0]))
Diff_tot = np.abs((Energy_RK4[0] + Energy_RK4[1]) - (Energy_LF[0] + Energy_LF[1]))
Diff = np.abs(np.array(Energy_RK4) - np.array(Energy_LF))
avg = np.average(Diff_tot)
accuracy = np.log10(avg)
print("Average = ", avg)
print("Accuracy (log) = ", accuracy)

plt.plot(Time, Diff[0], "g", alpha = .3, label = "U")
plt.plot(Time, Diff[1], "b", alpha = .3, label = "K")
plt.plot(Time, Diff_tot, "--r", alpha = .7, label = "tot")
plt.xlabel("Time [code unit]")
plt.ylabel("Diff")
plt.title(f"RK4 vs. Euler (Diff comparison) \n accuracy(log10) =
    {np.round(accuracy, 5)}")
plt.legend()
plt.show()
```

2.4 earth_sun.ipynb

```
# %% [markdown]
# # Sun-Earth System
#
# In this notebook, we will test our Solar System (Sun + Earth) simulation.\
# For convenice, that's define the `problem_name` here for handling data IO.

# %%
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from nbody.particles import Particles
from nbody.simulation import NbodySimulation

# %%
problem_name = "test"

# %% [markdown]
# Prepare physical constants

# %%
msun  = 1.989e33 # gram
mearth = 5.97219e27 # gram
au    = 1.496e13 # cm
day   = 86400    # sec
year  = 365*day  # sec
G     = 6.67e-8 # cgs

# %% [markdown]
# Re-implment the particle initialze condition of the Sun+Earth system.

# %%
def initialSolarSystem(particles:Particles):
```

```
33
34      num_part = 2
35      G = 6.67428e-8
36      AU = 1.49598e13
37      mass_earth = 5.97219e27
38      mass_sun = 1.989e33
39      d_earth = AU * (mass_sun / (mass_earth + mass_sun))
40      d_sun = - AU * (mass_earth / (mass_sun + mass_earth))
41      peroid = np.sqrt(4 * np.square(np.pi) * np.power(AU, 3) / (G * (mass_earth +
            mass_sun)))
42      vel_earth = -2 * np.pi * d_earth / peroid
43      vel_sun = 2 * np.pi * d_sun / peroid
44      acc_earth = -G * mass_earth / np.square(d_earth)
45      acc_sun = G * mass_sun / np.square(d_sun)
46
47      particles = Particles(N = num_part)
48
49      masses = particles.get_masses()
50      masses[0, 0] = mass_sun
51      masses[1, 0] = mass_earth
52
53      positions = particles.get_positions()
54      positions[0, 0] = d_sun
55      positions[1, 0] = d_earth
56
57      velocities = particles.get_velocities()
58      velocities[0, 1] = vel_sun
59      velocities[1, 1] = vel_earth
60
61      accelerations = particles.get_accelerations()
62      accelerations[0, 0] = acc_sun
63      accelerations[1, 0] = acc_earth
64
65      particles.set_masses(masses)
66      particles.set_positions(positions)
67      particles.set_velocities(velocities)
68      particles.set_accelerations(accelerations)
69
70      return particles
71
72  # %% [markdown]
73  # Once we initialize the particles, we could run our simulation by
74
75  # %%
76  particles = Particles(N=2)
77  particles = initialSolarSystem(particles)
78  sim = NbodySimulation(particles)
79  sim.setup(G=G,method="RK4",io_freq=30,io_title=problem_name,io_screen=True,visualized=False)
80  sim.evolve(dt=0.1*day,tmax=1*year)
81
82  # %% [markdown]
83  # # Load data and Visualization
84  #
```

```
# note: `conda install -c conda-forge ffmpeg` might be necessary

# %%
import glob

# %%
fns = "data_"+problem_name+"/"+"data_"+problem_name+"_[0-9][0-9][0-9][0-9][0-9].txt"
fns = glob.glob(fns)
fns.sort()
# print(fns)

# %%
scale = 2 * au

fig, ax =plt.subplots()
fig.set_size_inches(10.5, 10.5, forward=True)
fig.set_dpi(72)
ol,  = ax.plot([0,au],[0,0],'ro',alpha=0.3) # the initial conditions
line, = ax.plot([],[],'o')             # plots of particles

def init():
    ax.set_xlim(-scale,scale)
    ax.set_ylim(-scale,scale)
    ax.set_aspect('equal')
    ax.set_xlabel('X [code unit]')
    ax.set_ylabel('Y [code unit]')
    return line,

def updateParticles(frame):
    fn = fns[frame]
    m,t,x,y,z,vx,vy,vz,ax,ay,az = np.loadtxt(fn)
    # print("loadtxt done",fn)
    line.set_data(x,y)
    return line,

ani = animation.FuncAnimation(fig, updateParticles, frames=len(fns),init_func=init,
    blit=True)
ani.save('movie_'+problem_name+'.gif',fps=10)
```