PHYS 349: Advanced Computational Physics

Final Project

Kamalesh Reddy Paluru
Omer Sipra

April 25, 2023

Statements

Omer

- Worked primarily on documentation and theory for the N-body problem.
- Picked and allocated implementations of the code and where it belonged, i.e., validation and application.
- Contributed to the designing the RK4 integration method.
- Contributed to the discussion about the second (and better) version of the project.

Kamalesh

- Implemented the theory using Python.
- Created a class to simulate a system of bodies using Euler's method (first version).
- Contributed to the designing of the RK4 integration method (helped transition into modelling the second version of the repository).
- Modularized classes to support Euler's method and RK4 (or any other algorithm for that matter).

Abstract

The N-body problem is one of the most famous problems in classical physics. Dating all the way back to Isaac Newton himself. The general problem involves predicting the motion of a finite number of celestial bodies that adhere to a gravitational law. Although, in its modern renditions, it involves dynamics of any number of and type of body; regardless of physical scale or the forces they are subject to.

The 2-body problem was first solved by Isaac Newton in the 17th century, and then a general solution was proposed by Johann Bernoulli later. However, no such solution is possible in cases with 3 or more bodies in a system. Apart from a few special cases that allow for certain restrictions. We hence, for all other cases, we use numerical integration to solve the system(s).

This report will focus on developing methods for numerically solving any generic case of this problem. But first, we need to explore the theory that underlies it.

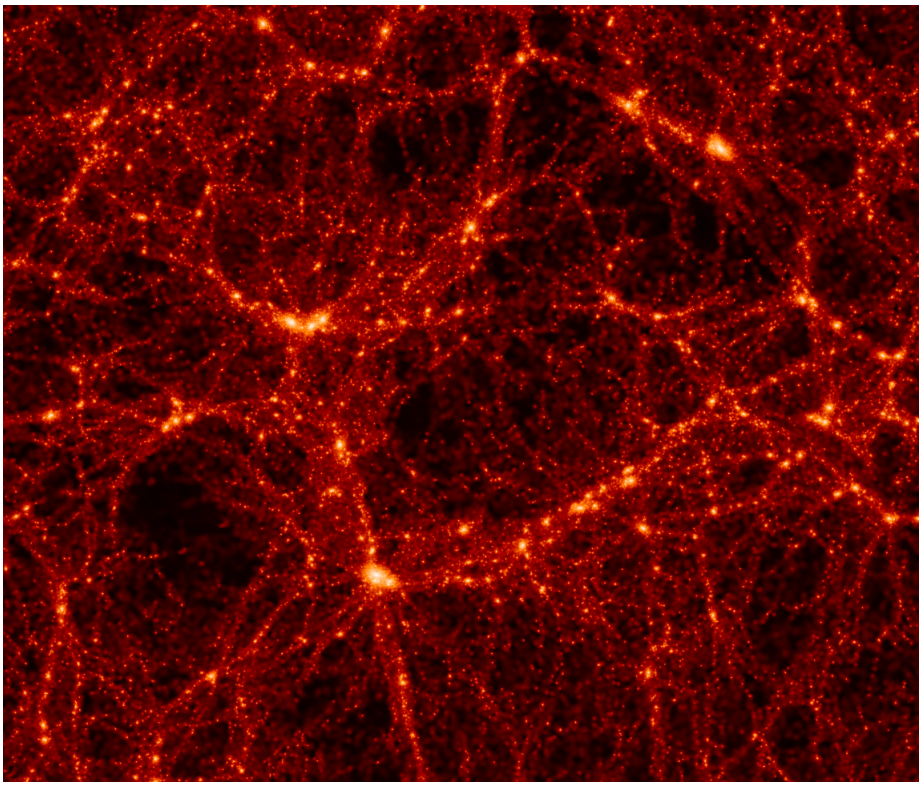

Figure 1: Complex structures that are a result of N-body simulations, the universe is observed to have a similar web-like structure

Theory

- We will contextualize the problem using the gravitational case. Treating the inputs as masses, positions and velocities of each body.

- The output would typically be the predictive trajectories as defined by solved functions of position and velocity. Assuming Newtonian physics.

- Suppose we have n particles with masses m_i and positions $\mathbf{r}_i(t)$. Then, particle i has a gravitational force exerted on it by particle j. This is given by:

$$\mathbf{f}_{i,j}(t) = -\frac{Gm_i m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t))$$

- The total force on particle i is then:

$$\mathbf{F}_i(t) = \sum_{j=0, j\neq i}^{n-1} \mathbf{f}_{i,j}(t)$$

  Which can be written as:

$$\mathbf{F}_i(t) = -Gm_i \sum_{j=0, j\neq i}^{n-1} \frac{m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t))$$

  We know from Newton's law that $\mathbf{F} = m\mathbf{a}$, or in this context:

$$\mathbf{F}_i(t) = m_i \ddot{\mathbf{r}}_i(t)$$

  We can now substitute these relations to obtain a second order differential equation system:

$$\ddot{\mathbf{r}}_i(t) = -G \sum_{j=0, j\neq i}^{n-1} \frac{m_j}{||\mathbf{r}_i(t) - \mathbf{r}_j(t)||^3}(\mathbf{r}_i(t) - \mathbf{r}_j(t))$$

- This project, will hence focus on solving this exact problem. To make life easier, we will make some mathematical assumptions. Starting with the assumption that all the functions we will be dealing with, are functions of real numbers.

Numerics

Initial Approach

- We know certain pieces of information, such as the conditions:

$$\mathbf{r}_i(0) = \mathbf{r}_0,\ \dot{\mathbf{r}}_i(0) = \mathbf{v}_0$$

- Thus, we have an *initial*-value problem.
- If the conditions are of the form:

$$\mathbf{r}_i(0) = \mathbf{r}_0,\ \dot{\mathbf{r}}_i(t) = \mathbf{r}_f$$

  we call it a *boundary*-value problem.
- These problems are different enough that their numerical solutions are distinct.
- We will now explore methods of how we can solve these problems in different contexts. A naive approach to numerical integration for solving our obtained differential equations would follow the steps:

```
Given input data (Initial Conditions)
    For each timestep Δt:
        For each particle i:
            Compute F_net(t)
            Update r_i(t) and ṙ_i(t) = v(t) for each particle i
            Return new positions and velocities
```

- This is called Euler's method! The way it was initially implemented was not ideal. It was a "brute-force" approach where the algorithm above was followed as is.
- This was a problem because when we tried to introduce a new algorithm (**RK4**), code had to be changed in a lot of places. [1]

Euler's Method

Background

- Euler's Method is designed to solve IVPs for first-order ordinary differential equations of the form:

$$y' = f(x, y), \quad y(x_0) = y_0$$

- Where it can be used the approximate numerically the values of the solution of the ODE:

$$x_i = x_0 + ih, \quad i = 0, 1, \ldots, n,$$

  Where:

$$h = \frac{b - x_0}{n}.$$

[1] This was the route that we went and failed at miserably. The code was cryptic and very hard to understand.

- We do this by starting with the tangent to an integral curve, given by:

$$y = y(x_i) + f(x_i, y(x_i))(x - x_i)$$

- Setting the step $(x - x_i)$ to h, and using the IC $y(x_0) = y_0$ we can give a general expression for all solutions to $y(x)$ as:

$$y_{i+1} = y_i + hf(x_i, y_i), \quad 0 \leq i \leq n - 1.$$

- However, the problem we intend to solve, always involves second-order ODEs, as shown before. Luckily, there is a way to convert any such DE into two coupled first-order ODEs. Consider a generalized form of the n-body ODE we defined earlier:

$$\ddot{\mathbf{r}}(t) = -\mathbf{r}(t)$$

- We can convert this to a system of $2n$ first order ODEs:

$$\dot{\mathbf{r}}_\mathbf{i}(t) = \mathbf{v}_i(t)$$

$$\dot{\mathbf{v}}_\mathbf{i}(t) = \frac{1}{m_i}\mathbf{F}_i(t)$$

  Under the initial conditions:

$$\mathbf{r}_i(0) = \mathbf{r}_i, 0, \ \mathbf{v}_i(0) = \mathbf{v}_i, 0$$

  Allowing us to numerically solve any system.

`Python`

- After failing at morphing the previous adaptation to swap out algorithms, it was clear that there was a need for a revamp.
- First, we seperated out the algorithm, instead it being embedded in the `System` class, we pulled it out into a (here) `EulerUtils` class, this was then passed to `System`.
- The main takeaway was: **Each Algorithm can spit out an updated vector; it needs to know how to "differentiate" the initial vector and the initial vector itself.**
- Here, the algorithm itself doesn't care what "differentiate" means, it's an implementation detail for all it cares. What it means for us is that we show it a way of getting (for example) dv given v (conveniently, dv can be inferred from the acceleration, $\frac{dv}{dt}$).
- The algorithm looks like:

```
1 def Euler_algorithm(f, t_i, y_i, dt):
2     y_i_plus_1 = y_i + dt*f(t_i, y_i)
3     return y_i_plus_1
4
```

Listing 1: Euler's method

- Here, `y_i` is the vector which we want to calculate the next step of; `t_i` is the initial time; `f` calculates the derivative (for a given `t` and `y`); and `dt` is the time differential.
- Now let's look at what `System` looks like:

```
class System:
    def __init__(self,
                 dt,
                 algorithm,
                 bodies=[],
                 law=None):
        self.dt = dt
        self.algorithm = algorithm
        self.bodies = np.array(bodies)
        self.law = law

    # Convinience method to get the latest vector for the algorithm(s) to use.
    def latest_vector(self):
        vec = np.array([self.bodies[0].position[-1], self.bodies[0].velocity[-1]])
        for body in self.bodies[1:]:
            vec = np.append(vec, [body.position[-1], body.velocity[-1]], axis=0)
        return vec

    def initial_vector(self):
        vec = np.array([self.bodies[0].position[0], self.bodies[0].velocity[0]])
        for body in self.bodies[1:]:
            vec = np.append(vec, [body.position[0], body.velocity[0]], axis=0)
        return vec

    # Builds a list of masses of the bodies that make up the system.
    def masses(self):
        return np.array(list(map(lambda body: body.mass, self.bodies)))

    def derivator(self, t, y):
        G = coconst.G.value
        # Here, we need to tell RK4 how to differentiate y, and then return it.
        y_prime = np.zeros_like(y)
        # Even indices of `y_prime` are just equal to the next index in y.
        for i, _y in enumerate(y):
            if i % 2 == 0:
                y_prime[i] = y[i + 1]

        for j, _y2 in enumerate(y):
            if j % 2 != 0:
                for other_body in np.delete(self.bodies, int((j - 1)/2)):
                    y_prime[j] += -G*other_body.mass*(y[j - 1] - other_body.
    position[-1])/(np.linalg.norm(y[j - 1] - other_body.position[-1]))**3


        # Now that we filled in all the blanks, we can return `y_prime`.
        return y_prime

    def simulate(self, until=0.0):
        t = 0.0
        while t < until:
            # This is the next step of our special vector.
            step = self.algorithm(f=self.derivator,
                                  t_i=0.,
                                  y_i=self.latest_vector(),
                                  dt=self.dt)

            for i, body in enumerate(self.bodies):
                body.position = np.append(body.position, [step[i*2]], axis=0)
                body.velocity = np.append(body.velocity, [step[i*2 + 1]], axis=0)
            t += self.dt

```

Listing 2: System

- Okay, that's a lot to unpack. Most of the methods are self explanatory, the only methods that really concern us are `derivator` and `simulate`:
  - `derivator`: Recall how we needed to tell the algorithm how to calculate the derivative of a given `y` vector. This essentially maps y to y':

    $$\begin{pmatrix} x_i \\ \dot{x}_i \end{pmatrix} \to \begin{pmatrix} \dot{x}_i \\ \ddot{x}_i \end{pmatrix}$$

    Notice that one of the vector entries is simply copied over to a different index ($\dot{x}_i$). This is exactly what's happening in line 36 of Listing 2. For the second entry, $\ddot{x}_i$, we make use of the law of the system and calculate it explicitly:

    $$a_i = \ddot{x}_i = \sum_j \frac{-Gm_j(\vec{x_i} - \vec{x_j})}{||\vec{x_i} - \vec{x_j}||^3}$$

    We (the algorithm rather) now have the differentiated vector!
  - `simulate`: This is where we call the algorithm and pass the parameters it needs: The "derivator" that we implemented earlier and the initial conditions which can be found with the convenience methods `latest_vector` and `initial_vector`.
- That's all!

RK4 Integration

Background

- Consider the same IVP as before:

  $$y' = f(x, y), \quad y(x_0) = y_0$$

  We pick a step-size $h > 0$ and define:

  $$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h,$$

  $$t_{n+1} = t_n + h, \ n = 0, 1, 2, 3, \ldots, n$$

  Where:

  $$k_1 = hf\left(t_n, y_n\right)$$

  $$k_2 = hf\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_1\right)$$

  $$k_3 = hf\left(t_n + \frac{h}{2}, y_n + \frac{1}{2}k_2\right)$$

  $$k_4 = hf\left(t_n + h, y_n + k_3\right)$$

- However, as we reduce a second order ODE to 2 couples first order ODEs:

  $$\dot{\mathbf{r}}_\mathbf{i}(t) = \mathbf{v}_i(t)$$

  $$\dot{\mathbf{v}}_\mathbf{i}(t) = \frac{1}{m_i}\mathbf{F}_i(t)$$

  The k parameters of the RK4 solutions change to two sets, one for approximations of position, and one for velocity. Given as follows:

  $$k_1 = h\mathbf{v}\left(t_i, x_i, v_i\right)$$

$$k_2 = h\mathbf{v}\left(t_i + \frac{h}{2}, x_i + \frac{hk_1}{2}, v_i + \frac{hl_1}{2}\right)$$

$$k_3 = h\mathbf{v}\left(t_i + \frac{h}{2}, x_i + \frac{hk_2}{2}, v_i + \frac{hl_2}{2}\right)$$

$$k_4 = h\mathbf{v}\left(t_i + h, x_i + hk_3, v_i + hl_3\right)$$

$$x_{i+1} = x_i + \frac{h}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

And:

$$l_1 = \frac{h}{m_i}\mathbf{F}\left(t_i, x_i, v_i\right)$$

$$k_2 = \frac{h}{m_i}\mathbf{F}\left(t_i + \frac{h}{2}, x_i + \frac{hk_1}{2}, v_i + \frac{hl_1}{2}\right)$$

$$k_3 = \frac{h}{m_i}\mathbf{F}\left(t_i + \frac{h}{2}, x_i + \frac{hk_2}{2}, v_i + \frac{hl_2}{2}\right)$$

$$k_4 = \frac{h}{m_i}\mathbf{F}\left(t_i + h, x_i + hk_3, v_i + hl_3\right)$$

$$v_{i+1} = v_i + \frac{h}{6}\left(l_1 + 2l_2 + 2l_3 + l_4\right)$$

`Python`

- The algorithm above was implemented in its own file (as was done for Euler's method in the previous section):

```python
def RK4_algorithm(f, t_i, y_i, dt):
    '''
    For steps involved, refer to: https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods.
    '''
    # Know k_1y:
    k_1y = f(t_i, y_i)

    # Find k_2y:
    k_2y = f(t_i + dt/2, y_i + dt*k_1y/2)

    # Find k_3y:
    k_3y = f(t_i + dt/2, y_i + dt*k_2y/2)

    # Find k_4y:
    k_4y = f(t_i + dt, y_i + dt*k_3y)

    # Find y_{i + 1}:
    y_i_plus_1 = y_i + (dt/6)*(k_1y + 2*k_2y + 2*k_3y + k_4y)
    return y_i_plus_1

```

Listing 3: RK4 integration

- Something extremely important to notice here is that **this has the exact same inputs and outputs of `Euler_algorithm` in Listing 1!**
- Because of the way we abstracted the implementation detail of the algorithm from `System` in this approach, this meant that all we had to do was pass this algorithm to the constructor of `System`.

Tests

Now, let's actually use the classes and algorithm that we built to simulate specific systems.

Earth-Sun

Classical 2-Body Problem

- We will use the famous Kepler's problem as the first test to gauge how well our numerical methods work.
- In classical mechanics, the Kepler problem is essentially a special case of the 2-body problem in which bodies interact a radial inverse square potential. The problem is to find their position/speeds as a time series, given their masses, and initial positions, velocities.
- The central force between two objects is defined as:

  $$\mathbf{F} = \frac{k}{r^2}(\hat{\mathbf{r}})$$

  Which corresponds to the scalar potential:

  $$V(r) = \frac{k}{r}$$

  The equation of motion of a point mass in a particle is given by:

  $$m\frac{d^2r}{dt^2} - mr\omega^2 = m\frac{d^2r}{dt^2} - \frac{L^2}{mr^3} = -\frac{dV}{dr}$$

  Where $w = \frac{d\theta}{dt}$ and $L = mr^2\omega$ (angular momentum) is conserved.

- This DE is of the same form as our general n-body ODE. Under the condition that L be non-zero (which it is), we can convert this DE in polar coordinates, with r being a $r(\theta)$ instead of $r(t)$:

  $$\frac{d}{dt} = \frac{L}{mr^2}\frac{d}{d\theta}$$

- Making the DE independent of time:

  $$\frac{L}{r^2}\frac{d}{d\theta}\left(\frac{L}{mr^2}\frac{dr}{d\theta}\right) - \frac{L^2}{mr^3} = -\frac{dV}{dr}$$

- Finally we use change of variable $u = \frac{1}{r}$ to make the ODE quasi-linear:

  $$\frac{d^2u}{d\theta^2} + u = -\frac{m}{L^2}\frac{d}{du}V\left(\frac{1}{u}\right) = -\frac{km}{L^2}$$

- Which can be analytically solved to give us:

  $$u \equiv \frac{1}{r} = -\frac{km}{L^2}\left[1 + e\cos(\theta - \theta_0)\right]$$

- Here, e represents the eccentricity of the orbit, with it ranging between 0 and 1. The lower bound corresponding to perfect circular motion, and the upper bound corresponding to an unbound parabola. It relates to energy as:

$$e = \sqrt{1 + \frac{2EL^2}{k^2 m}}$$

Note that the eccentricity and angular momentum of the Earth are known to be:

$$e = 0.0167$$

$$L = 2.7 \times 10^{40} \text{kg} \frac{\text{m}^2}{\text{s}}$$

We can use this solution to model the Earth-Sun system.

Euler's Method

- The first system that is the easiest to test is the Earth-Sun system, since we know exactly what to expect in this case: The Earth revolves around the Sun at 1AU.
- We first create the objects `earth` and `sun` using a `Body` class, all this does is arrange the meta-data associated with a body in a neat way, using:

```
class Body:
    def __init__(self,
                 name,
                 mass=0.0,
                 position=[[0, 0]],
                 velocity=[[0, 0]],
                 net_force=[0.0, 0.0]):
        self.name = name
        self.mass = mass
        self.position = np.array(position)
        self.velocity = np.array(velocity)
        self.net_force = np.array(net_force)

```

Listing 4: `Body` class

- We can hence create instances of this class as:

```
earth = Body(name="Earth",
             mass=aconst.M_earth.value,
             position=[[aconst.au.value, 0]],
             velocity=[[0, 29784.8]])

halley = Body(name="Halley",
             mass=2.2e14,
             position=[[0.5871*aconst.au.value, 0]],
             velocity=[[0, 53545]])


sun = Body(name="Sun",
           mass=aconst.M_sun.value)

jwst = Body(name="JWST",
                 mass=6500,
                 position=[L1],
                 velocity=[[0, 0]])

```

Listing 5: `Body` objects

- We create an instance of `System` and pass a list of the `Body` instances that we want in the system. Let's first try an Earth-Sun system: [2]

```
bodies = [earth, sun]
system = System(bodies=bodies,
                dt=100000,
                algorithm=eu.Euler_algorithm,
                # Not used
                law=lambda m1, m2, x1, x2: ((coconst.G.value*m1*m2)/((lin.norm(x2 - x1))**3)) * (x2 - x1))

```

Listing 6: Earth-Sun system

- Now, we can use the `simulate` method discussed earlier to mutate the positions and velocities of `bodies` (for 1 year):

```
system.simulate(until=3.154e7)

```

Listing 7: Simulate the system for a year

- There is also a `PlotUtils` class which handles plotting for this project. Upon plotting the trajectories of `earth` and `sun` using Euler integration, we get:

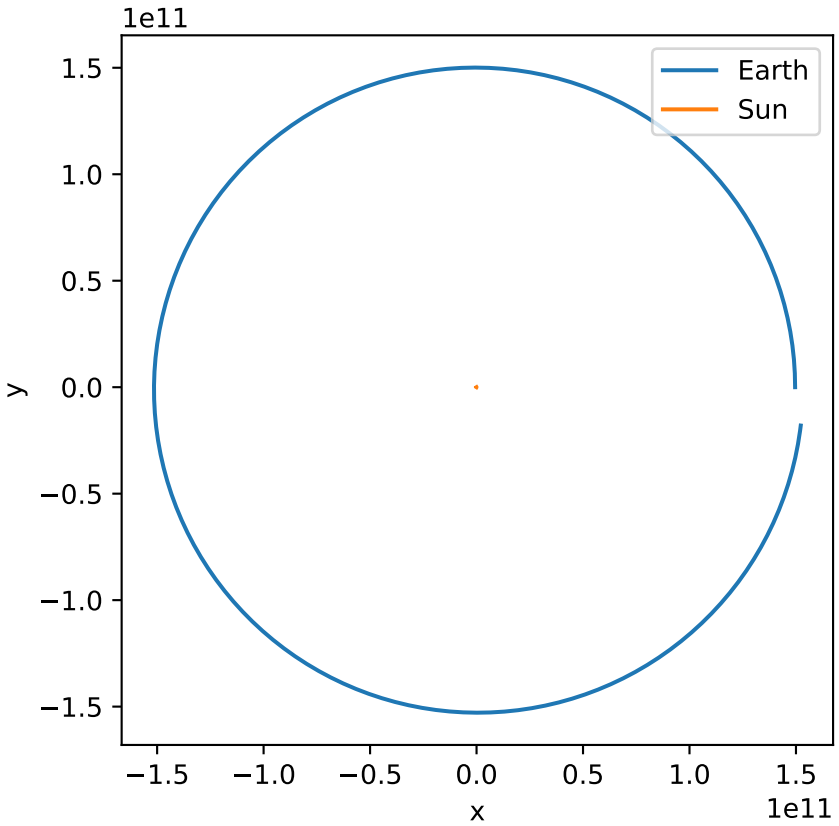

Figure 2: Earth-Sun simulation with Euler's method (1 year)

[2] Here, the `law` parameter is not used, this is beyond the scope of this project where we want might consider a system governed by a law that might not be Newtonian. It is however, still a good idea to generalize this.

- Notice that the Earth doesn't quite make it to the initial spot and the radius of its orbit increases.
- This suggests that it "loses" energy. The loss can be attributed to the fact that Euler's method linearly interpolates the differential trajectories. Since the radius has been increasing over the "circle", it doesn't make it to the initial position and drifts away.
- This is more apparent if we run the simulation for say, 20 years (with the same time step):

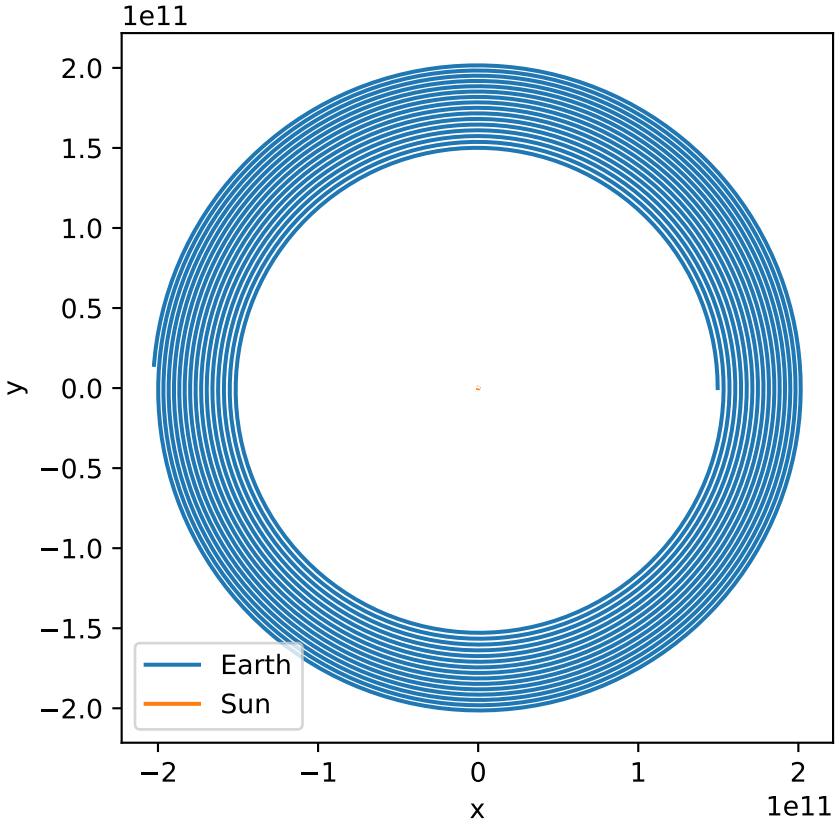

Figure 3: Earth-Sun simulation with Euler's method (20 years)

- This drift is a result of the integration method used. The way we fix it is to introduce a better algorithm, **RK4**!

RK4 Integration

- Let's run the previous simulation (1 year, using the same time step, $dt = 10000$s) using RK4[3]. We get the following plot:

[3]Recall that all we need to do is swap the algorithm that we pass to `Sysstem`.

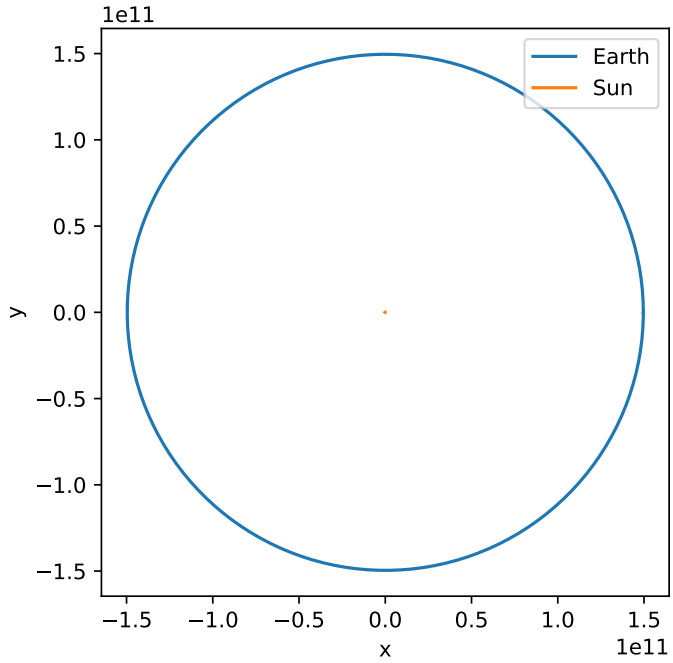

Figure 4: Earth-Sun simulation with RK4 integration (1 year)

- Because of how the RK4 method averages over smaller steps in dt, we see immediately from Figure 4 that this is a much better result, earth seems to return to nearly the same initial position after 1 year (as we would expect).
- Now, let's run this for 50 years to re-affirm our analysis.

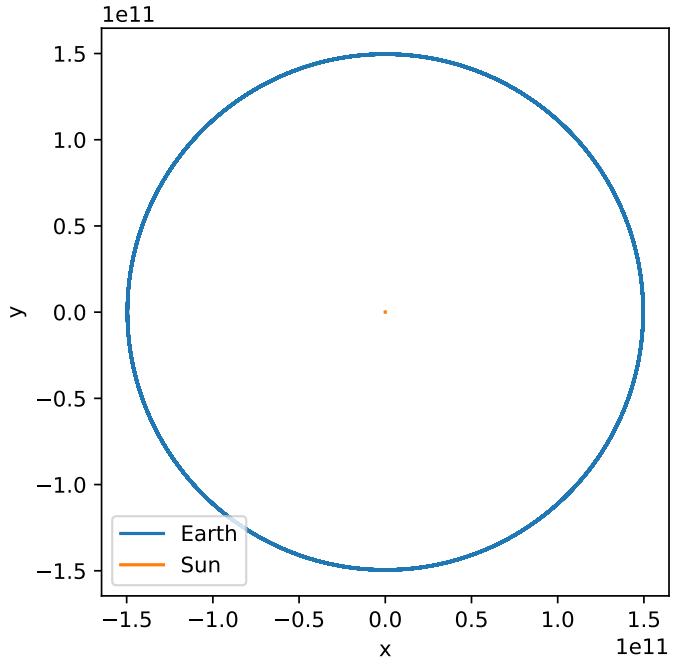

Figure 5: Earth-Sun simulation with RK4 integration (**50 years**!)

- And we don't drift away, even after 50 years (the orbits overlap)![4]

[4]This ran for longer than Euler's method, since the time complexity is $\approx O(n^2)$. But humanity survives!

Halley-Earth-Sun

- As another sanity check, we can use RK4 on the Earth-Sun system along with Halley's comet. the period of Halley's comet is known to be approximately 76 years. We hence add the `Body` instance of this comet, `halley`, to the list of bodies and pass it to the `system`:

```
earth = Body(name="Earth",
             mass=aconst.M_earth.value,
             position=[[aconst.au.value, 0]],
             velocity=[[0, 29784.8]])

halley = Body(name="Halley",
              mass=2.2e14,
              position=[[0.5871*aconst.au.value, 0]],
              velocity=[[0, 53545]])

sun = Body(name="Sun",
           mass=aconst.M_sun.value)

bodies = [halley, earth, sun]

system = System(bodies=bodies,
                dt=100000,
                algorithm=rk.RK4_algorithm,
                law=lambda m1, m2, x1, x2: ((coconst.G.value*m1*m2)/((lin.norm
(x2 - x1))**3)) * (x2 - x1))

# 76 Years
system.simulate(until=3.154e7*76)

```

Listing 8: Simulate the system for a year

- Notice that we increased the time step so we can generate the plot in a reasonable amount of time. Upon plotting the trajectories of the bodies using RK4, we get:

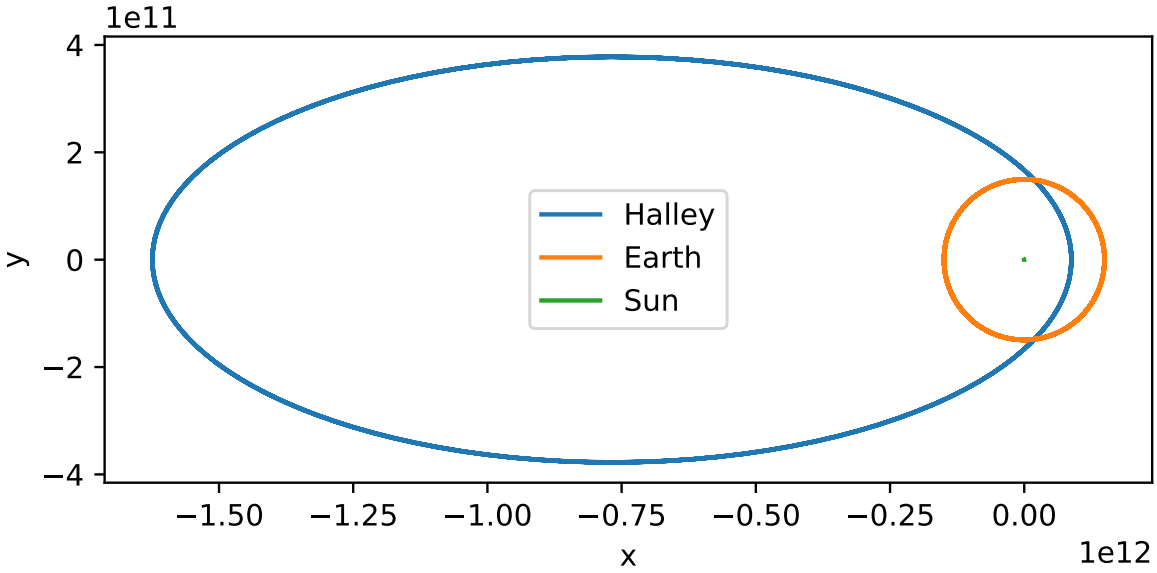

Figure 6: Halley-Earth-Sun simulation with RK4 integration (**75 years**!)

- Halley's comet returns to near-Earth so humanity can observe it once in a lifetime![5]

[5]The Aphelion and Perhelion of the comet also appear to be accurate.

Studies