n-Body Orbital Model using Euler's Method

Frank J. Vilimek

November 2023

1 Introduction

Isaac Newton developed his Law of Universal Gravitation in the 17th century, a revolutionary concept that propelled science forward into a new age. It has been able to describe the motion of the moons, planets, and stars, and even required the invention of calculus. Calculus, however, utilizes infinitesimally small changes in time, space, etc., something that is impossible to describe with computers – you cannot tell a computer to make a time step of size $1/\infty$ seconds, it is just not possible. Thankfully, Leonhard Euler, a very famous mathematician, came up with his method for solving ordinary differential equations, Euler's Method. Armed with Newton's Law of Universal Gravitation and Euler's Method, simulating orbital mechanics with a computer will be relatively easy. We will first show the basics of gravity and the several relations between position, velocity, and acceleration, and then the basics of Euler's Method and how its accuracy depends on the step size. We will then apply these to classical orbital mechanics and first model a system of two celestial bodies, then to n celestial bodies.

2 Newton's Law of Universal Gravitation

Newton's Law of Universal Gravitation is a law that describes the attractive force exerted on two bodies as a result of them having mass. All objects with mass exert some amount of gravitational force on each other. The equation for the force of gravity $\vec{F}_g$ is

$$\vec{F}_g = -\frac{GM_1M_2}{r^2}\hat{r} \tag{1}$$

where G is the gravitational constant ($G = 6.67430 \times 10^{-11}$ m^3 kg^{-1} s^{-2}), M_1 is the mass of the first body, M_2 is the mass of the second body, r is the radius between the centers of mass of both bodies, and $\hat{r}$ is the radial vector that determines the direction of force.

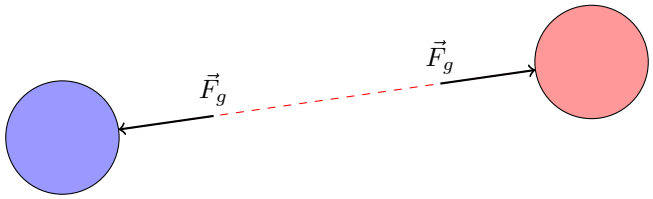

Figure 1: Two massive bodies exerting a gravitational force on each other.

Notice that this equation only accounts for the mass of two bodies – that is because gravity only acts between two objects. That is not to say that bodies cannot be affected by many massive bodies at the same time, however the force of gravity itself needs to be determined between all individual pairs of bodies in a system before each of their contributions affects the system as a whole.

With eq. (1), we can use Newton's laws of motion to derive more equations for orbital motion. Let us start with finding the velocity of a body due to gravity – we first start with the famous $\vec{F} = m\vec{a}$

$$\begin{aligned} \vec{F} &= m\vec{a} \\ \vec{F} &= m\frac{d\vec{v}}{dt} \\ \frac{d\vec{v}}{dt} &= \frac{\vec{F}}{m} \end{aligned}$$

We now have an ordinary differential equation (ODE), which we can solve through separation of variables

$$\begin{aligned} \int_{v_0}^{\vec{v}(t)} d\vec{v} &= \int_0^t \frac{\vec{F}}{m} dt \\ \vec{v}(t) &= \frac{\vec{F}}{m}t + v_0 \end{aligned} \tag{2}$$

We can substitute eq. (1) into eq. (2) to yield

$$\vec{v}(t) = -\frac{GM_1M_2t}{mr^2}\hat{r} + v_0 \tag{3}$$

If we set $m = M_1$ we get

$$\vec{v}(t) = -\frac{GM_2t}{r^2}\hat{r} + v_0 \tag{4}$$

With this equation, we can find the velocity of the first body as a result of gravity. The velocity we get with this equation is the total velocity of the body; though this is useful, we need to describe it using some coordinate system in order to actually carry out calculations. For our model we will use Cartesian coordinates.

Consider a body that has velocity $\vec{v}$ at some angle θ from the horizontal x-axis. The velocity, a vector quantity, has an x-component v_x and a y-component v_y.

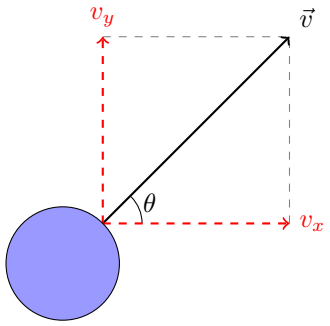

Figure 2: A body with velocity $\vec{v}$ and velocity components v_x and v_y.

The velocity is thus a sum of the x- and y-components: $\vec{v} = v_x + v_y$. Using trigonometry, the components are thus

$$v_x = \vec{v}\cos(\theta) \tag{5}$$
$$v_y = \vec{v}\sin(\theta) \tag{6}$$

The directional vector $\hat{r}$ points outwards from the body we are analyzing. Since gravity is attractive, however, we want the vector to point towards the body, hence the negative sign in front of the force of gravity. $\hat{r}$ can also be broken up into x- and y-components, and can be described by comparing the position of the body we are analyzing to the position of the other body

$$\hat{r}_x = x_{body\ 1} - x_{body\ 2} \tag{7}$$
$$\hat{r}_y = y_{body\ 1} - y_{body\ 2} \tag{8}$$

Since $\hat{r}$ is purely directional, it has magnitude 1; we need to divide the components by the magnitude of that component

$$\hat{r}_x = \frac{x_{body\ 1} - x_{body\ 2}}{|x_{body\ 1} - x_{body\ 2}|} \tag{9}$$
$$\hat{r}_y = \frac{y_{body\ 1} - y_{body\ 2}}{|y_{body\ 1} - y_{body\ 2}|} \tag{10}$$

With this, we have the x- and y-velocities *almost* fully described

$$v_x = \left(-\frac{GM_2 t}{r^2}\hat{r}_x + v_0\right)\cos(\theta) \tag{11}$$
$$v_y = \left(-\frac{GM_2 t}{r^2}\hat{r}_y + v_0\right)\sin(\theta) \tag{12}$$

The reason why the velocities are *almost* fully described is because we have not dealt with the r in the denominator. I might be missing something in the equations I used, but to me it seems like accounting for r analytically would end up with infinite recursions. Let me explain: r is the distance between the centers of mass of both bodies, and can be found using the Pythagorean Theorem

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{13}$$

But remember that the x- and y-positions are functions of time. As an example, we can use eq. (11) and solve the ODE

$$v_x = \left(-\frac{GM_2 t}{r^2}\hat{r}_x + v_0\right)\cos(\theta)$$
$$\frac{dx}{dt} = \left(-\frac{GM_2 t}{r^2}\hat{r}_x + v_0\right)\cos(\theta)$$
$$\int_{x_0}^{x(t)} dx = \int_0^t \left(-\frac{GM_2 t}{r^2}\hat{r}_x + v_0\right)\cos(\theta)dt$$

But now we have an integral with r in the denominator again, while trying to figure out what it is in the first place. I know there is a way to solve this problem (as the two-body problem has been solved by

Newton himself), but without doing more math, we can instead use a method for numerically solving ODEs: Euler's Method.

3 Euler's Method

Euler's Method is a useful tool when dealing with ODEs that are time-consuming or impossible to solve. Being a numerical solver, the solution is *not* 100% accurate, but depending on how you set it up, Euler's Method can get arbitrarily close to being 100% accurate. The method begins by assuming a function $f(x)$ is locally linear, meaning that when you zoom in, any curves can be approximated as a tangent line.

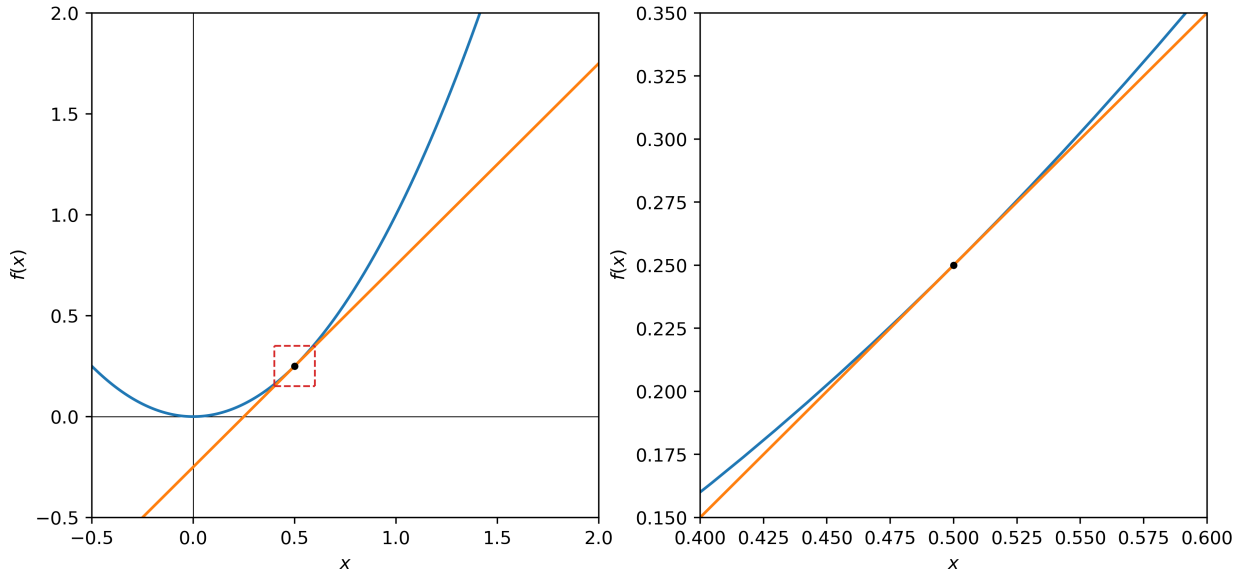

Figure 3: When zoomed in, a function can be approximated by a tangent line.

Thus, at the point $x = x_0$

$$f(x_0) \approx \left.\frac{df(x)}{dx}\right|_{x_0} x_0 + y_{01} \tag{14}$$

If you move some Δx away from x_0, then you can re-evaluate the linear approximation

$$f(x_x + \Delta x) \approx \left.\frac{df(x)}{dx}\right|_{x_0+\Delta x} (x_0 + \Delta x) + y_{02} \tag{15}$$

By this logic, the function can be approximated, point-by-point, using the formula

$$f_{n+1} \approx \left.\frac{df(x)}{dx}\right|_{x_n} \Delta x + f_n \tag{16}$$

Using an example, let us use Euler's Method to solve the ODE: $df(x)/dx = 2x$ with a step size of $\Delta x = 0.2$ and initial condition $f(0) = 0$. The slope at $f(0)$ is

$$\left.\frac{df(x)}{dx}\right|_{x=0} = 2(0) = 0 \tag{17}$$

Thus, our first point f_{n+1} is given by

$$f_{n+1} \approx (0)(0.2) + 0 = 0 \tag{18}$$

Our next point f_{n+1} is built off of this previous f_{n+1} (now f_n). The slope at the new f_{n+1} is

$$\left.\frac{df(x)}{dx}\right|_{x=0.2} = 2(0.2) = 0.4 \tag{19}$$

Thus, our second point is given by

$$f_{n+1} \approx (0.4)(0.2) + 0 = 0.08 \tag{20}$$

Continuing this pattern develops our numerical solution. For Euler's Method, the size of Δx greatly impacts the accuracy of the solution, as shown in fig. (4).

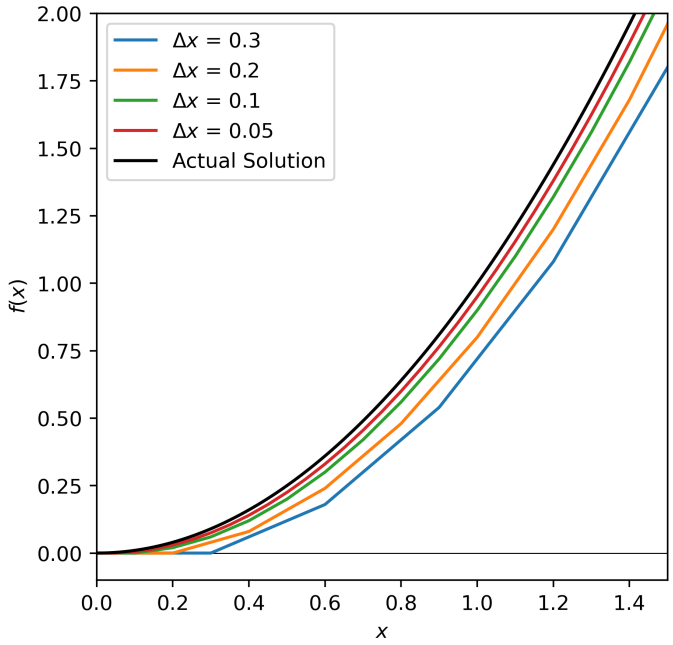

Figure 4: Approximating the solution to $df(x)/dx = 2x$ using Euler's Method. Notice that smaller values of Δx yield more accurate approximations.

Euler's method is especially useful if you know how a system changes, but not necessarily what the system is. Tying this in with our orbital model, we do not need to know the actual function for v_x or v_y while accounting for changes in r, we can just numerically solve the system with Euler's Method using a small time step and get a relatively accurate approximation.

4 Making the Two-Body Model

With Newton's law of universal gravitation and Euler's Method, we can begin coming up with our model. To transform our problem into one Euler's Method can solve, we need to go back to eq. (11), (12), and (13).

Remember that velocity is the derivative of position in time, so

$$\frac{dx}{dt} = \left(-\frac{GM_2 t}{r^2}\hat{r}_x + v_0\right)\cos(\theta)$$
$$\frac{dy}{dt} = \left(-\frac{GM_2 t}{r^2}\hat{r}_y + v_0\right)\sin(\theta)$$

Transforming into Euler's Method form, the equations turn into

$$v_{x,n+1} = -\frac{GM_2\Delta t}{r_n^2}\hat{r}_{x,n}\cos(\theta_n) + v_{x,n} \tag{21}$$
$$v_{y,n+1} = -\frac{GM_2\Delta t}{r_n^2}\hat{r}_{y,n}\sin(\theta_n) + v_{y,n} \tag{22}$$

This transformation added a lot of n's everywhere, meaning that we need to calculate each nth parameter for every time step. With these component velocities, the new x- and y-positions can be calculated using the equations

$$x_{n+1} = v_{x,n+1}\Delta t + x_n \tag{23}$$
$$y_{n+1} = v_{y,n+1}\Delta t + y_n \tag{24}$$

We can now begin coding our model. We begin by important libraries – `numpy` and `matplotlib` are really all we need. We can also specify a simulation time, time step, and time array for plotting.

```
# Import libraries
import numpy               as np
import matplotlib.pyplot as plt

# Simulation time
Time   = 50
t      = 0
deltat = 0.1
```

We can also set the properties of our two bodies, such as initial position, velocity components, mass, and lists to append the calculated positions to.

```
# Planet properties
Planet1Position = [0, 0]
Planet1Velocity = [0, 2]
Planet1Mass     = 1E12
P1x             = []
P1y             = []

Planet2Position = [10, 0]
Planet2Velocity = [1, -1]
Planet2Mass     = 7E11
P2x             = []
P2y             = []
```

We can now make a `Planet` class that accepts a mass, the current body's position, the other body's position, and the current velocity of the current body. The class will use this information with Euler's Method to calculate the new velocity components and positions.

```
class Planet:
    def __init__(self, PlanetMass, xyPosition, OtherxyPosition, \
                 CurrentVelocity):
    # Gravitational constant
    G = 6.6743E-11

    # Current position and other body's position
    x0Position, y0Position   = xyPosition
    xPosition, yPosition     = OtherxyPosition
    CurrentXVel, CurrentYVel = CurrentVelocity

    # Direction vectors
    x = x0Position - xPosition
    y = y0Position - yPosition
    r = np.sqrt(x**2 + y**2)

    # Avoid division by zero
    if x == 0:
        xVector = 0
    else:
        xVector = x / abs(x)
```

Continued onto next page

```
        if y == 0:
            yVector = 0
        else:
            yVector = y / abs(y)

        # Angle scaling
        theta  = np.arctan(y / x)
        xScale = np.cos(theta)
        yScale = np.sin(abs(theta))

        # New velocity
        vx = (-G * PlanetMass / r**2 * deltat) * (xVector * xScale)
        vy = (-G * PlanetMass / r**2 * deltat) * (yVector * yScale)

        self.xVelocity = vx + CurrentXVel
        self.yVelocity = vy + CurrentYVel

        # New position
        xNew = vx * deltat + x0Position
        yNew = vy * deltat + y0Position

        self.xPosition = xNew
        self.yPosition = yNew
```

Now we can start looping over each time step to calculate the new velocities and positions for both bodies. I opted to use a `while` loop over a `for` loop because, down the line, it would make it easier to break out of a loop once certain conditions are met. Of course, we can also plot the results.

```
while t < Time:
    # Initialize each planet
    Planet1 = Planet(Planet2Mass, Planet1Position, Planet2Position, \
                     Planet1Velocity)
    Planet2 = Planet(Planet1Mass, Planet2Position, Planet1Position, \
                     Planet2Velocity)

    # Append data to position lists
    P1x.append(Planet1Position[0])
    P1y.append(Planet1Position[1])
```

Continued onto next page

```
        P2x.append(Planet2Position[0])
        P2y.append(Planet2Position[1])

        # New positions and velocities
        Planet1Position = [Planet1.xPosition, Planet1.yPosition]
        Planet2Position = [Planet2.xPosition, Planet2.yPosition]

        Planet1Velocity = [Planet1.xVelocity, Planet1.yVelocity]
        Planet2Velocity = [Planet2.xVelocity, Planet2.yVelocity]

        # Increment time step
        t += deltat

plt.figure(figsize = (7, 7))
plt.xlabel(r"$x$, meters")
plt.ylabel(r"$y$, meters")
plt.plot(P1x, P1y, color = "tab:blue",   label = "Body 1")
plt.plot(P2x, P2y, color = "tab:orange", label = "Body 2")
plt.legend()
plt.show()
```

This exact code will yield the plot

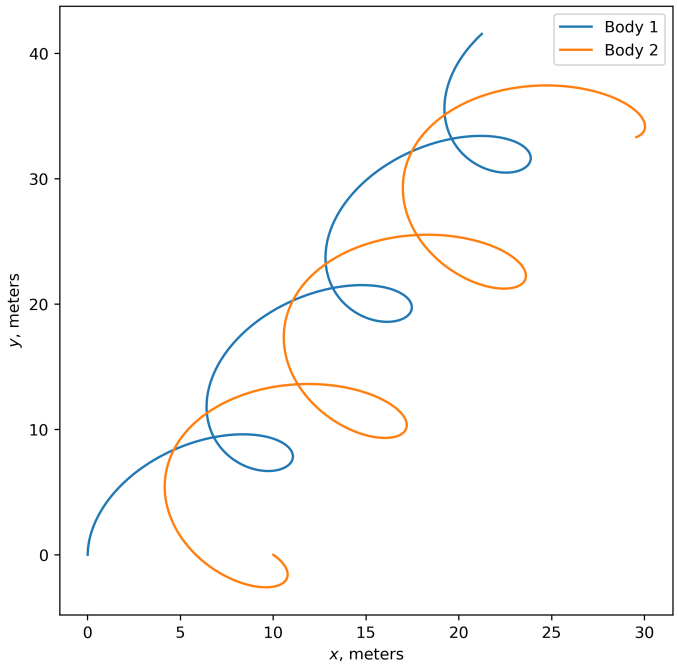

Figure 5: Two massive bodies affected by each other's force of gravity.

The resulting plot looks promising as it seems that the bodies are indeed being pulled in the right directions, creating interesting spirals over time. To confirm if the model is accurate, however, we can use some real-life data and see how close we get to expected results – we will model the orbit of the Earth around the Sun. We only need to change some parameters, namely total modeling time, time step size, initial positions, initial velocities, and body masses.

```
Time    = 31536000
t       = 0
deltat  = 0.8
tplot   = np.arange(0, Time, deltat)

Planet1Position = [0, 0]
Planet1Velocity = [0, 29.78E3]
Planet1Mass     = 1.9891E30
P1x             = []
P1y             = []
Planet2Position = [1.495978707E11, 0]
Planet2Velocity = [0, 0]
Planet2Mass     = 5.972E24
P2x             = []
P2y             = []
```

This yields the plot

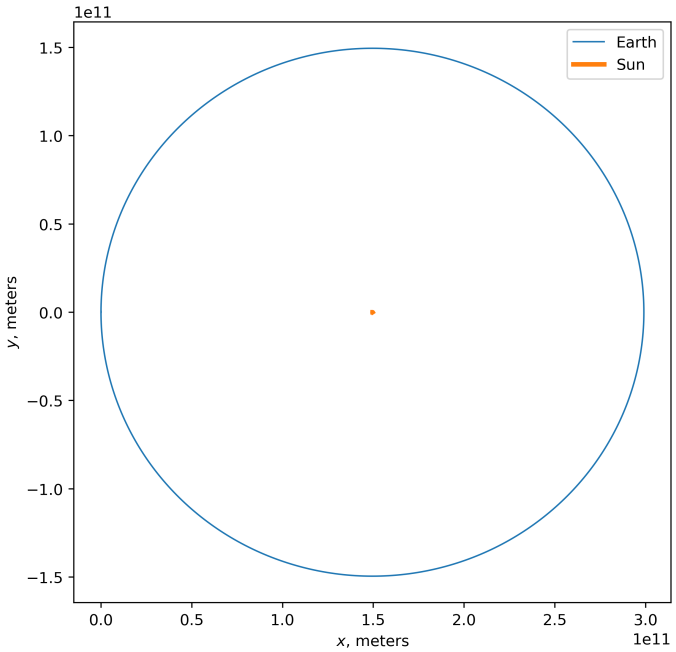

Figure 6: Model of Earth orbiting around the Sun.

This plot shows an almost perfect orbit of the Earth around the Sun, which confirms the validity of our model. There is a little bit of overlap near the $x = 0$ portion, but that is to be somewhat expected because: (1) the actual orbit of the Earth around the Sun is not perfectly circular, and has an eccentricity of around 0.01671; (2) a year is not exactly 365 days; and (3) our time steps were not as small as they could be, however any smaller and Google Colaboratory (what I used to create this plot) crashes before it can produce a plot.

We can confirm the validity of our model again by modelling the orbit of Mercury, the planet with the most elliptical orbit with an eccentricity of about 0.2056, and see how close we can get. We need to change the time, velocity, etc. values again.

```
Time    = 7603200
t       = 0
deltat = 0.3
tplot   = np.arange(0, Time, deltat)

Planet1Position = [0, 0]
Planet1Velocity = [0, 58.97E3]
Planet1Mass     = 1.9891E30
P1x             = []
P1y             = []
Planet2Position = [46E9, 0]
Planet2Velocity = [0, 0]
Planet2Mass     = 0.33010E24
P2x             = []
P2y             = []
```

This yields the plot on the next page

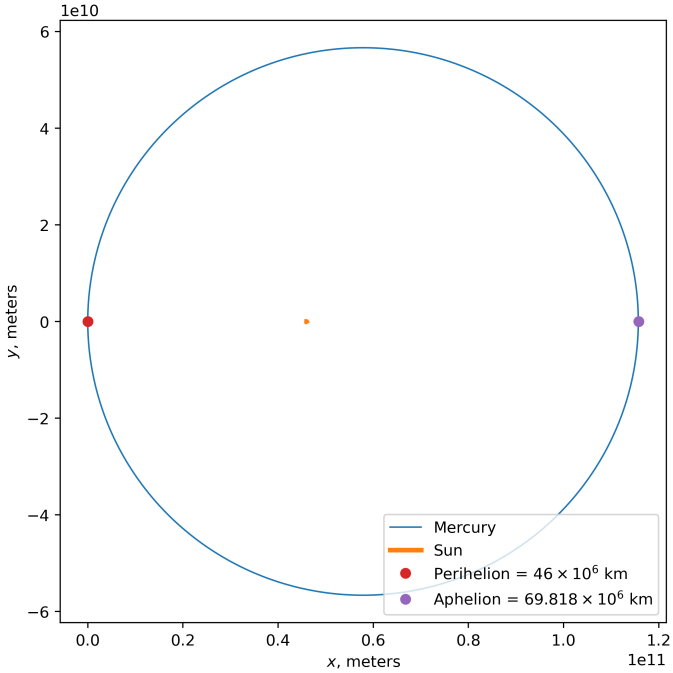

Figure 7: Model of Mercury orbiting around the Sun. The red and purple points are the actual Perihelion and Aphelion values for Mercury, respectively.

We can see that our model is indeed very close to the actual orbit of Mercury. The aphelion in our model is a little bit smaller than the actual value, however this can be attributed to slightly inaccurate data and, again, a relatively large step size.

5 Making the n-Body Model

Now that we have confirmed that our two-body model can be quite accurate, let us extend it to a system with n-bodies. In math and physics, this is called the *n-Body Problem*, and it is unsolved. Thus, anything we model cannot be confirmed with analytical equations describing a body's orbit. The n-Body model is particularly useful when modeling systems like binary star systems with planets, comet orbits that are affected by several planets, or if you need an accurate model for a solar system where all planets are affected by every other planet, among others.

Since the underlying math is the same between the two-body and n-body models, we can reuse most of our old code and just add modifications to how we calculate the new velocities and positions. As stated earlier, the force of gravity is calculated only between two bodies at once. This means that the total force of gravity exerted on the ith body is the sum of all forces of gravity between it and all $j \neq i$ bodies

$$\begin{aligned} \vec{F}_{tot} &= \vec{F}_{i,j} + \vec{F}_{i,j+1} + ... + \vec{F}_{i,n-1} + \vec{F}_{i,n} \\ &= \sum_{j}^{n} \vec{F}_{i,j} \end{aligned} \tag{25}$$

Implementing this into our code would increase our computational complexity to $\mathcal{O}(n^2)$, since all n bodies

need to have their total force calculated between all $n-1$ other bodies. As such, if computation time is a constraint, we are either limited to a system with fewer bodies and a longer modelling time, or a system with more bodies and a shorter modelling time.

Since we do have to add adjustments to various places, let us just rewrite the entire program instead of poking and prodding our old one. We again begin by importing our libraries and initializing our simulation and body properties, but this time specifying the number of bodies with `Bodies` for designating `for` loop iteration, list length, etc. We also store our planet properties in a dictionary for cleaner organization.

```
# Import libraries
import numpy                as np
import matplotlib.pyplot as plt
import sys

# Simulation parameters
Time   = 200
t      = 0
deltat = 0.05
Bodies = 3

# Planet properties
PlanetProperties = {
    "InitialPosition" : [[-70, -50],
                         [20, 0],
                         [0, 60]],
    "InitialVelocity" : [[1, 0.1],
                         [0.1, 0.2],
                         [-1, 0]],
    "PlanetMass"      : [1E12,
                         7E11,
                         1E13],
}
```

Since our program depends heavily on the number of iterations in calculations (need to calculate the force between all pairs of bodies), we need a check to make sure that the value of `Bodies` matches the length of parameters in the dictionary `PlanetProperties` – if it does not, then it should raise an error without executing further.

```
# Check if number of bodies equals number of initial conditions
if Bodies != len(PlanetProperties["InitialPosition"]) or \
   Bodies != len(PlanetProperties["InitialVelocity"]) or \
   Bodies != len(PlanetProperties["PlanetMass"]):
     raise ValueError("The specified number of bodies (%i) does not match \
----------------------the length of at least one Planet property."%Bodies)
```

The `Planet` class stays mostly the same, however we get rid of the `CurrentVelocity` argument, as we will calculate those outside of the class. We also get rid of the `self.xPosition` and `self.yPosition`, as we will also calculate those outside of the class.

```
# Planet class
class Planet:
    def __init__(self, PlanetMass, xyPosition, OtherxyPosition):
        # Gravitational constant and planet mass
        G = 6.6743E-11

        # Current position and other body's position
        x0Position, y0Position   = xyPosition
        xPosition, yPosition     = OtherxyPosition

        # Direction vectors
        x = x0Position - xPosition
        y = y0Position - yPosition
        r = np.sqrt(x**2 + y**2)

        # Avoid division by zero
        if x == 0:
          xVector = 0
          x       = sys.float_info.min
        else:
          xVector = x / abs(x)

        if y == 0:
          yVector = 0
        else:
          yVector = y / abs(y)
```

Continued onto next page

```
        # Angle scaling
        theta  = np.arctan(y / x)
        xScale = np.cos(theta)
        yScale = np.sin(abs(theta))

        # New velocity
        vx = (-G * PlanetMass / r**2 * deltat) * (xVector * xScale)
        vy = (-G * PlanetMass / r**2 * deltat) * (yVector * yScale)

        self.xVelocity = vx
        self.yVelocity = vy
```

We now need to initialize lists for each body's x- and y-positions and velocities. To make this scalable with whatever value of `Bodies` you choose, we can include a `for` loop within a list. We can also specify initial conditions within these loops

```
# Initialize Planet objects and position/velocity arrays
PlanetXPosition = [[PlanetProperties["InitialPosition"][body][0]] \
                   for body in range(Bodies)]
PlanetYPosition = [[PlanetProperties["InitialPosition"][body][1]] \
                   for body in range(Bodies)]

PlanetXVelocity = [[PlanetProperties["InitialVelocity"][body][0]] \
                   for body in range(Bodies)]
PlanetYVelocity = [[PlanetProperties["InitialVelocity"][body][1]] \
                   for body in range(Bodies)]
```

We now need to make our `while` loop, and within it a nested `for` loop. Nested `for` loops are not ideal for efficiency, however since we do need to calculate all forces between all bodies, it is necessary for accuracy. We need to make sure that we are not calculating artificial forces that a planet exerts on itself, hence the: `if j != i:` statement.

```
while t < Time:
    for i in range(Bodies):
        # Initialize velocity vectors for ith body
        xVelocityVector = PlanetXVelocity[i][-1]
        yVelocityVector = PlanetYVelocity[i][-1]
```

Continued onto next page

```
        for j in range(Bodies):
            if j != i:
                # Planet instance
                PlanetMass      = PlanetProperties["PlanetMass"][j]
                xyPosition      = (PlanetXPosition[i][-1], \
                                   PlanetYPosition[i][-1])
                OtherxyPosition = (PlanetXPosition[j][-1], \
                                   PlanetYPosition[j][-1])
                CurrentVelocity = (PlanetXVelocity[i][-1], \
                                   PlanetYVelocity[i][-1])
                Planet_i        = Planet(PlanetMass, xyPosition, \
                                         OtherxyPosition)

                # New velocity vector from Planet ij pair
                xVelocity_ij = Planet_i.xVelocity
                yVelocity_ij = Planet_i.yVelocity

                xVelocityVector += xVelocity_ij
                yVelocityVector += yVelocity_ij

        # Append velocities and positions
        PlanetXVelocity[i].append(xVelocityVector)
        PlanetYVelocity[i].append(yVelocityVector)

        PlanetXPosition[i].append(xVelocityVector * deltat + \
                                  PlanetXPosition[i][-1])
        PlanetYPosition[i].append(yVelocityVector * deltat + \
                                  PlanetYPosition[i][-1])
    t += deltat
```

Instead of adding all `CurrentVelocity`'s to the new velocity as we did in the `Planet` class for the two-body model, we instead add them to the final x- and y-velocity vectors, because if we added all of them within the class then we would essentially be adding it n times. Finally, we can plot our results.

```
# Plot results
plt.figure(figsize = (7, 7))
for body in range(Bodies):
    plt.plot(PlanetXPosition[body], PlanetYPosition[body], \
             label = "Planet {}".format(body + 1), linewidth = 1)
plt.xlabel(r"$x$, meters")
plt.ylabel(r"$y$, meters")
plt.legend()
plt.show()
```

This exact code yields the plot

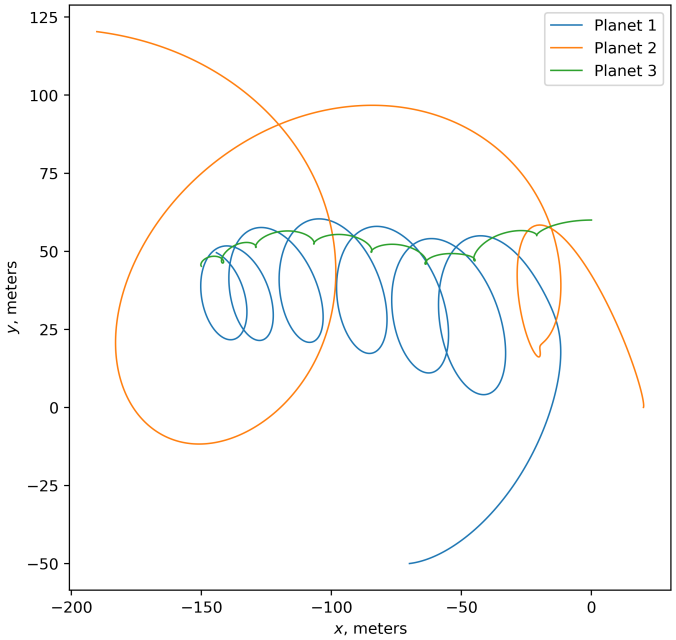

Figure 8: Three massive bodies affected by each other's force of gravity.

The good thing about this n-body model is that no matter how many bodies you want in the simulation, the only things you need to do is specify `Bodies` and their initial positions and velocities. Of course, to confirm the accuracy of our model, we can recreate three-body systems that we know – we can model the Sun, Earth, and the Moon. Each body's parameters are listed in the code snippet below.

The model shows that the Moon orbits the Earth about 12 times for every one orbit of the Earth around the Sun, which we would expect (in reality, the Moon orbits the Earth every 27 days). There is some overlap for the Moon's orbit on the right side of the plot, which could account for the "missing" days that make the Moon orbit every month. As for the code, it took a very long time to produce this plot; I did not time it, but it felt like 15-20 minutes.

```
# Simulation parameters
Time    = 31536000
t       = 0
deltat = 0.8
Bodies = 3

# Planet properties
PlanetProperties = {
    "InitialPosition" : [[0, 0],                  # Sun
                         [147.1E9, 0],            # Earth (perihelion)
                         [147.1E9, 0.3633E9]],    # Moon  (perigee)

    "InitialVelocity" : [[0, 0],                  # Sun
                         [0, 29.78E3],            # Earth (perihelion)
                         [-1.082E3, 29.78E3]],    # Moon  (perigee)

    "PlanetMass"      : [1.9891E30,               # Sun
                         5.9724E24,               # Earth
                         0.07346E24],             # Moon
}
```

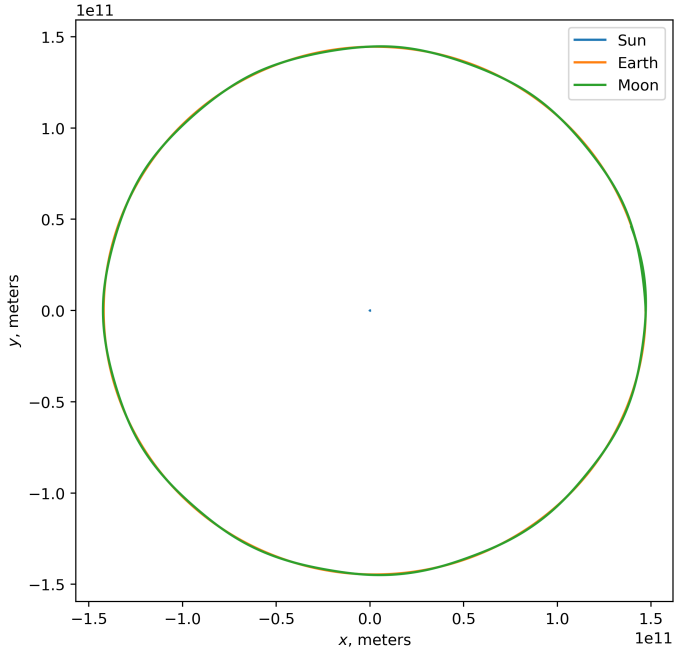

Figure 9: Model of the Earth orbiting the sun, and the Moon orbiting the Earth.

Now that we confirmed the accuracy of the model, let us verify that the code does indeed model any number of bodies that we choose. Let us make some cool plots.

```
PlanetProperties = {
    "InitialPosition" : [[10, 10],
                         [-50, 0],
                         [60, -70],
                         [0, 85]],
    "InitialVelocity" : [[0.61, -0.1],
                         [-0.5, -0.5],
                         [-0.2, -0.2],
                         [0, 0]],
    "PlanetMass"      : [0.5E11,
                         2E11,
                         3E11,
                         4E11],
}
```

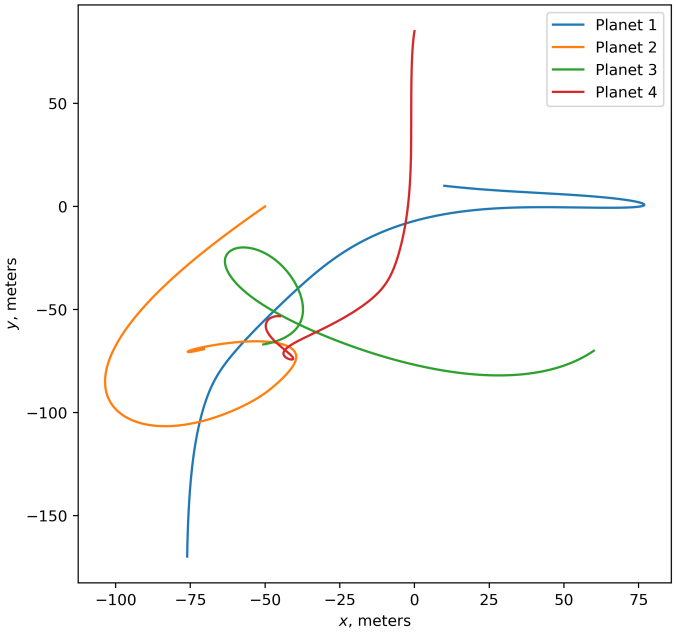

Figure 10: Four massive bodies affected by each other's force of gravity. Time = 470.

```
PlanetProperties = {
    "InitialPosition" : [[10, 10],
                         [-50, 0],
                         [60, -70],
                         [0, 85],
                         [85, 0]],
    "InitialVelocity" : [[0.5, -0.1],
                         [-0.5, -0.5],
                         [-0.2, -0.2],
                         [0, 0],
                         [0.3, 00.9]],
    "PlanetMass"      : [0.5E11,
                         2E11,
                         3E11,
                         4E11,
                         3E11],
}
```

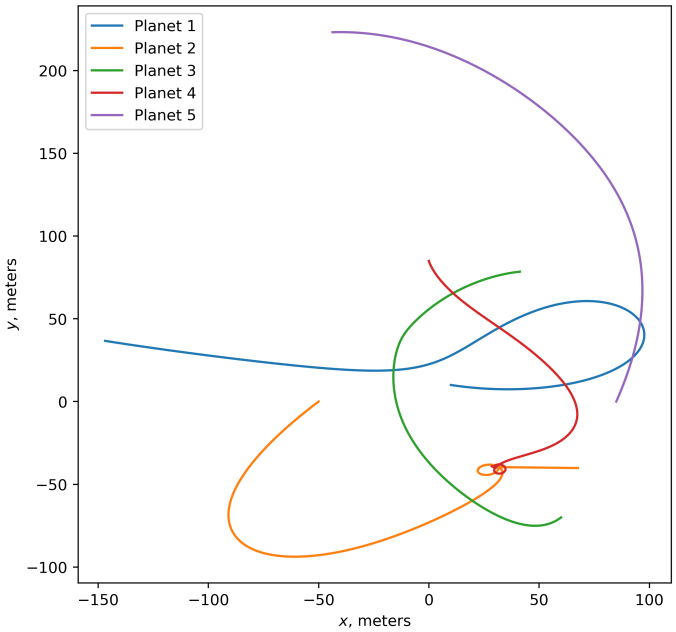

Figure 11: Five massive bodies affected by each other's force of gravity. Time = 527.

```
PlanetProperties = {
    "InitialPosition" : [[-110, 70], [-60, 75], [-1, 44], [50, 40],
                         [100, 50], [-100, -50], [-50, -40], [0, -50],
                         [50, -60], [100, -70],],
    "InitialVelocity" : [[-0.2, 1], [-0.1, 0.2], [0.4, 0.22], [0.01, -0.2],
                         [0.2, 0.2], [-1, 1], [0.6, -0.2], [-0.6, -1],
                         [-2, -0.2], [0.5, -0.2],],
    "PlanetMass"      : [11E10, 41E10, 11E10, 21E10, 11E10, 0.51E10,
                         11E10, 11E10, 61E10, 11E10],
}
```

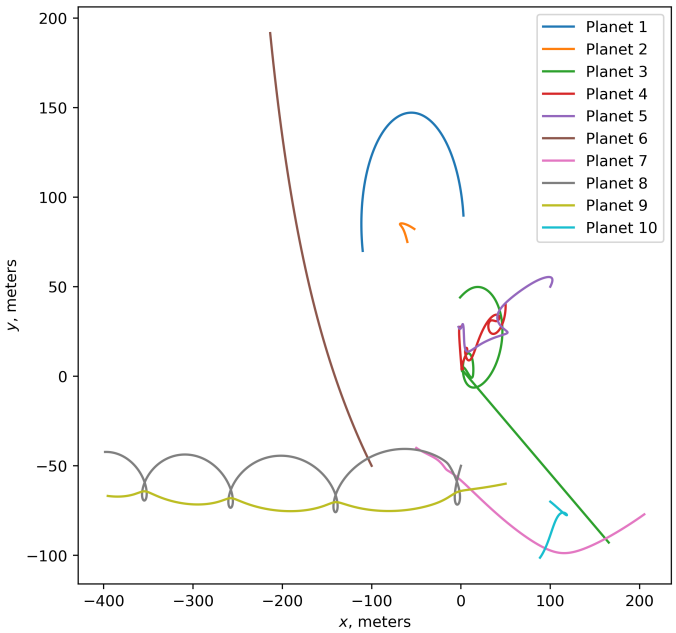

Figure 12: Ten massive bodies affected by each other's force of gravity. Time = 260. I discovered that it is very difficult to make ten bodies have relatively stable paths.

6 Conclusion

In this document we built a foundation of Newton's Law of Universal Gravitation, a groundbreaking advancement in physics at the time that allowed people to describe the motion of celestial bodies, and Euler's Method, a simple yet effective way of numerically solving first-order differential equations. We then developed a program to model two-body systems specifically, then extended that program to model n-body systems. The accuracy of the two- and n-body models was verified by modeling known bodies, such as the Earth orbiting the Sun, and the Moon orbiting the Earth orbiting the Sun. For future developments, it would be useful to consider other popular ODE solvers, such as the fourth order Runge-Kutta method, which may be more accurate than Euler's Method even with larger step sizes.