Exploring Methods of Simulating of our Solar System

Ben Tate [redacted]

Submitted: 11/11/2022

A project was conducted to explore the process of accurately simulating the solar system, using the Runge-Kutta 4th order to solve 2nd order differential equations. Initially a program was written to solve the two-body case for the Sun and Earth. The orbital period of the Earth was calculated to be 365.28 days at a dt interval of 1000 seconds. A hypothetical Planet X had a calculated orbital period of 226.97 days. A second program was written to simulate the entire solar system along with bodies associated energies and momenta. Various orbits were found to be unstable for asteroids, due to the effect of Jupiter's gravity changing their Kinetic Energies.

1. Introduction

The aim of the project was to accurately simulate our solar system in 2-dimensions and explore various phenomena which can be observed, such as asteroid orbit and Kirkwood gaps.

The physics of solar system dynamics are highly driven by gravitational forces between bodies. As the space between bodies is near vacuum, frictional forces can be assumed to be zero, allowing us to simulate planetary motion using the **Differential Equations 1 – 4** below.

$$\frac{d^2x_1}{dt^2} = \frac{1}{m_1} \cdot \frac{Gm_1m_2(x_2-x_1)}{r_{12}^3} \qquad (1)$$

$$\frac{d^2y_1}{dt^2} = \frac{1}{m_1} \cdot \frac{Gm_1m_2(y_2-y_1)}{r_{12}^3} \qquad (2)$$

$$\frac{d^2x_2}{dt^2} = \frac{1}{m_2} \cdot \frac{Gm_1m_2(x_1-x_2)}{r_{12}^3} \qquad (3)$$

$$\frac{d^2x_2}{dt^2} = \frac{1}{m_2} \cdot \frac{Gm_1m_2(y_1-y_2)}{r_{12}^3} \qquad (4)$$

Where G is the Gravitational Constant ($G = 6.67\text{x}10^{-11}\ m^3kg^{-1}s^{-2}$), m_i = is the mass of the body, x_i and y_i are the x and y positions of the body, r_{ij} is the distance between bodies i and j.

To use these equations to simulate the Solar System they need to be split up into two first order differential equations. **Equation 1** is split up into **Equations 5** and **6** below. **Equations 2 – 4** are split up in the same way.

$$\frac{dx_1}{dt} = vx_1 \qquad (5)$$

$$\frac{dvx_1}{dt} = \frac{Gm_2(x_2-x_1)}{r_{12}^3} \qquad (6)$$

Where variables follow same convention as in **Equations 1 – 4**.

These are the equations which are used to simulate how each body interacts with every other body. They need to be calculated at every time step, dt. The value of dt essentially dictates how often the equations are calculated. Having a smaller value of dt will give more accurate results, with the drawback of requiring more calculations and therefore computing power.

These equations are simple to solve for a 2-body system, which is what was initially experimented with. Values for displacement and velocity could be easily calculated for the planet orbiting the fixed-position Sun at each time step. However, when more bodies are introduced, the problem becomes more complex as each body has an effect on every other body. This requires clever use of loops to efficiently calculate each time step.

One reason for wanting to simulate the solar system accurately is to enable us to forecast trajectories of objects such as asteroids and space craft. These need to be done numerically as there are no analytical solutions for an n-body system. Asteroid's trajectories are monitored constantly by NASA to give us early warning for any potential impacts with Earth.

Another reason for simulating solar system dynamics is to determine trajectories to send space craft in order to reach other planets, such as NASA's Cassini mission where the space probe was sent to Saturn and its moons. The spacecraft can make its own course corrections meaning that the simulation doesn't need to be perfectly precise (which would be impossible).

When running the simulation, it is important to find a compromise between accuracy and processing time. Solving the numerical solution for n-bodies becomes exponentially more difficult as the number of bodies increases. For this reason, a convergence method of determining a value for dt is used to give a simulation of acceptable accuracy, while keeping processing times to a minimum.

2. Method

There are various different analytical methods that could be used to solve our 2nd order differential equations. For this project Runge-Kutta 4th Order was primarily used, with some additional testing of Euler-Cromer.

To begin with, a simple 2-body system was simulated using the Euler-Cromer method. This was a relatively simple program however still proved very accurate. The sun was fixed at the origin (x=0, y=0) and the Earth started at x=0 and y=r, where r is the average distance of Earth from the Sun. Initial Planetary values can be found in **Appendix 1**.

The program consisted of a single for-loop where velocity and displacement are calculated using the new positions as dictated by the Euler-Cromer method. Code can be found in **Appendix 2**.

To determine the orbit period and therefore accuracy of the simulation an if-statement was included to check if the current x coordinate is positive and previous x coordinate is negative. This tells me when the program crosses the y axis and therefore has completed one complete revolution. Note: this only works as the sun is fixed at the origin (0, 0).

In the next stage of the project the code was rewritten to now use the Runge-Kutta 4th order method, which can be found in **Appendix 3**. Similarly, this program used a single for-loop which contains the Runge-Kutta 4th order method for calculating displacement and velocities. New variables had to be created to contain 1st, 2nd, 3rd, and 4th, values for the Runge-Kutta calculation. These variables had to be created for displacement and velocity in the x and y directions (i.e. 16 new variables). This substantially increases the memory used by the program but has little effect on computing times.

To determine whether Euler-Cromer or Runge-Kutta 4th Order was the best method, convergence was used to see which method was more efficient. This consists of picking a large and small dt value and repeatedly averaging them to converge on a single value of dt for which the simulation has an acceptable accuracy.

To determine the accuracy of the simulation, the total energy of the system was calculated at the start, and compared with the total energy of the system after each step. Total energy was calculated by combining the Kinetic Energy and the Gravitational Potential Energy. These can be found in **Equations 7** and **8** below. If the total energy after any step deviated from the original total energy by less than 0.1% over the course of 10 years simulation time, then the simulation was deemed to have an acceptable accuracy. Using this method, a value for dt was obtained which optimised computing power and accuracy.

$$Kinetic\ Energy = \frac{1}{2} \cdot m \cdot v^2 \quad (7)$$

$$G.P.Energy = -\frac{GM_{Sun}m_{Earth}}{r} \quad (8)$$

Once the ideal dt values were determined for each method, the program's run-time was calculated for each to confirm which was quickest at a given accuracy.

Using this method, it was determined that the Runge-Kutta 4th order was the superior method and this was used going forward. Code for optimising Euler-Cromer and Runge-Kutta dt values can be found in **Appendix 4** and **5** respectively.

The next stage of the project required us to simulate the orbits of more than 2 bodies. This adds considerable complexity due to each body having an effect on every other body. In order to create an accurate simulation, every body's Runge-Kutta value must be calculated before moving on to the next. This required multiple nested for-loops to solve, resulting in far more variables needed than in the 2-body simulation. Multiple arrays had to be created to store the variables in order to keep the code clean and efficient. Functions were also used when calculations such as 'acceleration' needed to be done multiple times. A print-out of the n-body code can be found in **Appendix 6**.

3. Results

The first results obtained were for the orbital period of Earth in a 2-body simulation with the Sun in a fixed position. This value was calculated for both Euler-Cromer and Runge-Kutta 4th Order methods of solving the differential equations. For both calculations A dt value of 10,000 seconds was used. This ensured a very accurate value for the orbital period would be calculated. The values can be found in **Table 1** below, with the Runge-Kutta plot in **Figure 1**.

Method	Orbital Period of Earth (days)
Euler-Cromer	365.27778
Runge-Kutta	365.27778

Table 1: Orbital periods for Earth in a fixed position Sun 2-body simulation

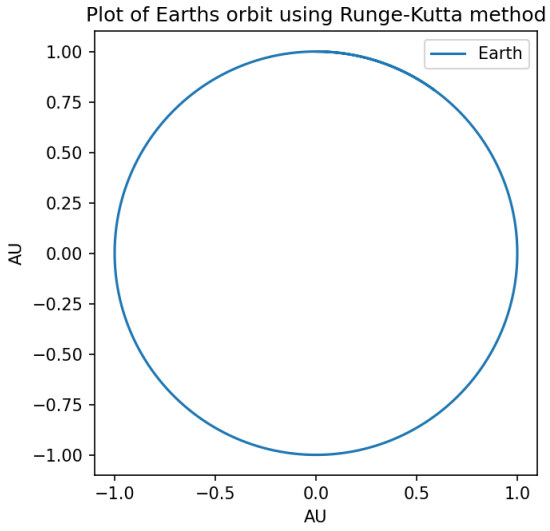

Figure 1: Plot of Earth's orbit in a fixed position Sun 2-body simulation using Runge-Kutta 4th Order

The values for both methods are exactly equal down to many decimal places. This is because the value of dt used was very small for both methods (10,000 seconds). This proves that both methods can be very accurate, however we will discuss in the analysis section why the Runge-Kutta is the better method.

The orbital period value is very slightly longer than the accepted value of 365.256 days. This could be because of a number of assumptions we have made in this simulation. For example; the sun is fixed, the Earth is not acted on by other solar bodies, there is zero friction, the simulation runs in a 2-dimensional frame, the Earth's orbit is assumed to be perfectly circular. All of these assumptions could be introducing minor errors which skew the orbital period slightly. The value calculated is however more than accurate enough to prove the simulation is working and we can proceed.

The next stage of the project involved simulating a hypothetical Planet X using the 2-body program. The planet's orbital radius r is given by **Equation 9** below.

$$r = 0.4 + \frac{8202}{25000} \qquad (9)$$

Where r is the planet's orbital radius and y_0 value, with units AU.

The Orbital period and plot for the planet X can be found in **Table 2** and **Figure 2** below.

Planet	Orbital Period of Earth (days)
Earth	365.27778
Planet X	226.96759

Table 2: Orbital periods for Earth and hypothetical Planet X in a fixed position Sun 2-body simulation using Runge-Kutta 4th Order

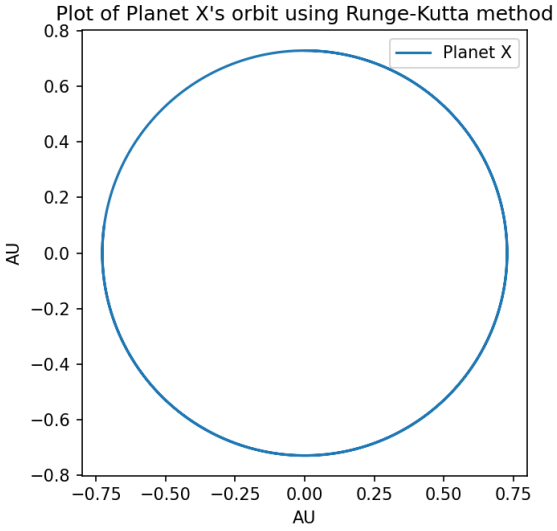

Figure 2: Plot of Planet X's orbit in a fixed position Sun 2-body simulation using Runge-Kutta 4th Order

The next step in the project required an n-body simulation. For reasons which will be covered in the Analysis section, Runge-Kutta 4th order was used for this instead of Euler-Cromer. See included plot in **Figure 3** for the orbits of every planet in the solar system. This was run over 250 simulated years to allow for Pluto to complete a full orbit. From running the program for such a long time frame, it was found that the inner planets, with short orbital periods, such as Mercury became unstable unless a

small dt value was used. This is because the errors can compound over longer time frames enough to send the planet on an outward trajectory. For this simulation a dt value of 100,000 seconds was used, which took the program 184.39 seconds to run.

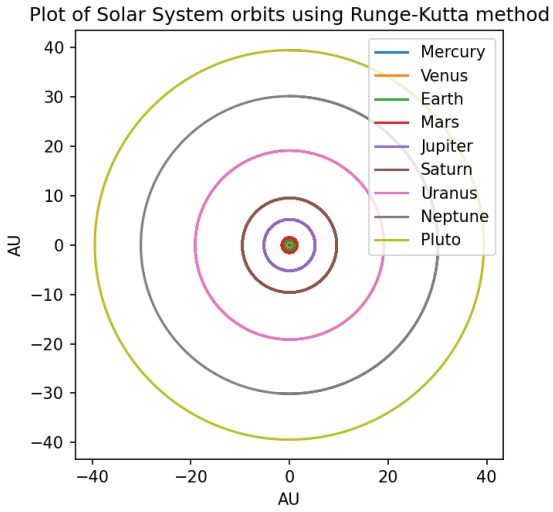

Figure 3: Plot of all planet's orbits in the solar system in a non-fixed Sun n-body simulation using Runge-Kutta 4th Order

In this simulation the Sun was also allowed to move, however this will be explored in the analysis section.

Now the n-body program is set up we can start experimenting with the simulation. An asteroid of mass 0.95×10^{21}kg was introduced into the simulation at various different orbits. Through experimentation, and research it was determined that some orbits are unstable. **Figure 4** on the right shows the asteroids orbit over 500 years with dt value of 500,000. The asteroid started with an orbital radius of 3.28AU. For visual ease only the asteroid has been plotted, however all of the planets are all still being simulated.

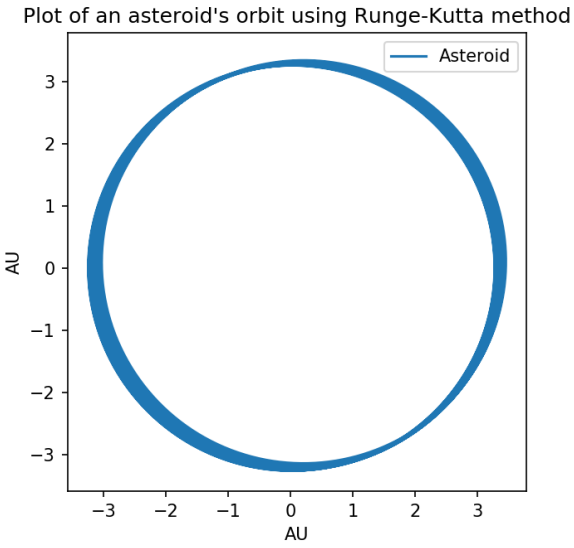

Figure 4: Plot of Asteroids orbit over 500 years using Runge-Kutta 4th Order

We can clearly see the orbit is shifting over time at this radius. This phenomenon only happens on very specific orbital radii, most other tested radii gave stable orbits with little to no movement over time. In the analysis section it will be explored as to what causes this behaviour and how best to visualise it.

The final part of this project will look at the potential devastating effects of a rogue planet. A rogue planet is an object of planetary mass, without a host planetary system. This is essentially a massive object moving through space, between planetary systems, with no regular orbit. Many astronomers agree that Rogue planets are likely to exist in abundance due to the way solar systems form. A worst-case scenario will be simulated and as such it will model the rogue planet with the upper limit to a planets mass. This is roughly 13 times the mass of the Jupiter, after which fusion starts to occur in the core and it becomes a brown dwarf. As such have introduced a rogue planet, with a mass 13 times that of Jupiter, to my simulation.

A view of the inner planets can be found in **Figure 5** on the next page. There is evidence of significant shift in Mars's orbit after just 100 years. Such a significant shift in orbit would have catastrophic effects on our solar system and could even lead to the ejection of a planet from our solar system.

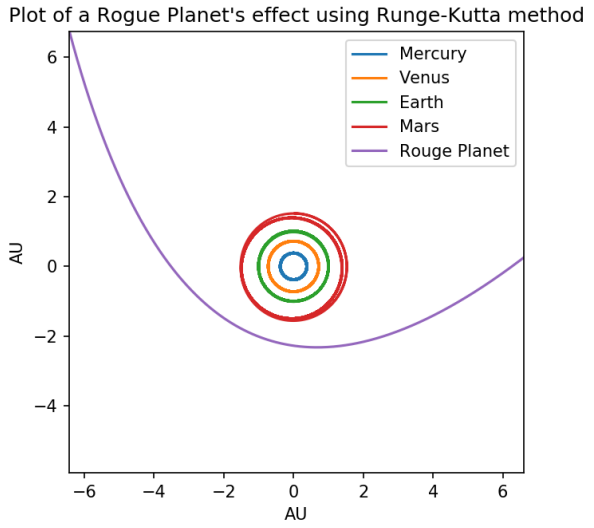

Figure 5: Plot of a rogue planet's trajectory and its effects on the Solar System using Runge-Kutta 4th Order

4. Analysis

The first stage of the project required me to choose which method to use, Euler-Cromer or Runge-Kutta. The method of convergence was used to see which method allowed for the largest dt value while still being accurate. A 10-year simulation was set up, where total initial energy of the system was compared with total energy after each step. The total energies will always differ slightly due to the nature of numerically solving differential equations, however by comparing energies after each step we are able to see the percentage error of the simulation at each step. A maximum error of 0.1% between initial and final total energy was decided and the program was run to determine which method was superior. The results can be found in **Table 3** below.

Method	Maximum timestep for <0.1% error (seconds)
Euler-Cromer	155,793
Runge-Kutta	1,131,186

Table 3: Maximum timestep (dt) in seconds for which calculated Total Energy deviates from Total Initial Energy over a period of 10 years, in a fixed position Sun 2-body simulation

It is evident that Runge-Kutta is a far more accurate method of numerically solving the differential equations, than Euler-Cromer. On its own, this information isn't all that useful, as the Runge-Kutta requires more computing power per timestep. If this extra computing power required for each timestep is large enough, it can negate the benefits of less timesteps being required. To be confident that Runge-Kutta was the best method, both methods simulations were run at their maximum timesteps calculated above, for a 5000-year span. Results showed a decisive win for the Runge-Kutta method which took just 1.04 seconds to complete, where Euler-Cromer took 2.62 seconds. This means that when both methods are simulating to the same accuracy, the Runge-Kutta method will complete the calculation over 2.5x quicker than Euler-Cromer.

In the 2-body simulation the sun was kept in a fixed position at (0,0). This worked well and gave accurate answers, however, to make the simulation more scientifically accurate the Sun was allowed to move. The initial velocity of the sun was calculated by setting the total linear momentum of the system to zero, following **Equation 10** below.

$$-m_{sun}v_{sun} = \sum_{i=1}^{n} m_i v_i \qquad (10)$$

Where m is mass, v is velocity, with momentum summed for all the planets

Using this method an initial value for the velocity of the sun was 15.98ms^{-1} in the negative x direction. Considering the speeds involved in our solar system, this appears to be very small, but when we consider that the mass of the sun accounts for 99.8% of the mass of the solar system this value starts to make more sense. Over a long enough time frame we would observe the orbits shifting to the left as the sun moves. In **Figure 6** a simulation was run where the suns initial velocity is multiplied by a factor of 100 to exaggerate the movement.

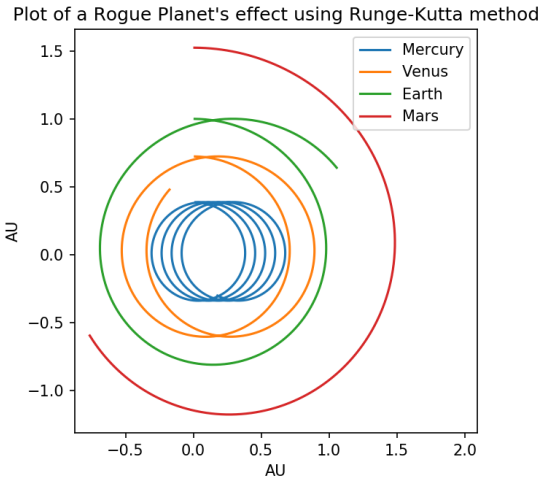

Figure 6: Plot of inner planets orbits when the sun moves 1,500ms^{-1} in the negative x direction using Runge-Kutta 4th Order

Possibly the most interesting findings from this project were the unstable orbits. In **Figure 4** we looked at an asteroid with an orbital radius of 3.28AU. There was an obvious shift in the orbit each rotation however it was difficult to see exactly what was happening. To get a better understanding, the Kinetic Energy of the asteroid was plotted as can be seen in **Figure 7** below.

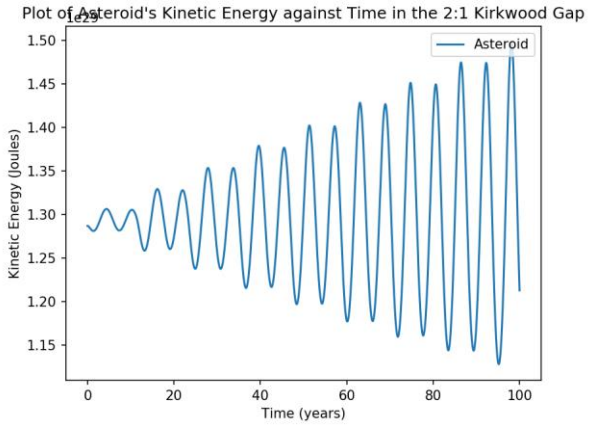

Figure 7: Plot of asteroids Kinetic Energy in the 2:1 (r=3.28AU) Kirkwood Gap using Runge-Kutta 4th Order

There is a clear increase in the Kinetic Energy over time, however what is more interesting is the way in which it increases. After every 2nd revolution there is a jump in Kinetic Energy. From further research, this specific orbit results in an orbital period of exactly twice that of Jupiter. This means the asteroid is in a 2:1 'resonance' with Jupiter and therefore will have the same relative position as Jupiter every other orbit. This results in Jupiter's gravity pulling on the asteroid every other orbit which leads to the increase in Kinetic Energy seen in **Figure 7**. Comparing this to an asteroid in a stable orbit as in **Figure 8**, we can see the clear difference in behaviour. The Kinetic Energy in the stable orbit does not increase like in the previous example.

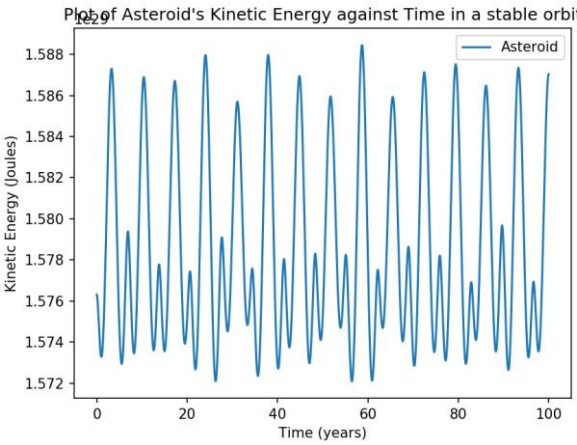

Figure 8: Plot of asteroids Kinetic Energy in a stable orbit (r=2.67AU) Kirkwood Gap using Runge-Kutta 4th Order

These orbits in resonance with Jupiter are known as Kirkwood Gaps. They arise at all orbital distances which form a simple ratio with Jupiter. The major Kirkwood Gaps occur at 2:1, 3:1, 5:2 and 7:3 orbital resonances.

5. Conclusions

A huge amount of knowledge was gained in this project in both computational and physical fields. Real world phenomena such as Kirkwood Gaps were uncovered using just a simple simulation, showing what an asset computers are to the field of Physics.

All in all, I would consider this project a success. The simulation created an accurate model and provided useful insight into our solar system. The simulation was run at acceptable accuracies on ordinary computers without taking large runtimes. This is a testament to how efficient the Runge-Kutta 4th Order method is for numerically solving differential equations. Given more time and computing power I would like to increase the accuracy by adding more variables and using smaller dt values to the simulation.

One thing I would change in the simulation is the way the orbit period is calculated. Currently it checks where the planet crosses from the negative x value to positive. This works well but only for a fixed sun simulation, with a small value for dt.

Another possible way to improve the program would involve animating the plots to get a real time view of how the planets are moving.

6. Appendix

Body	Mass (kg)	X_0 (m)	Y_0 (m)
Sun	1.98892×10^{30}	0	0
Mercury	3.30×10^{23}	0	5.79×10^{10}
Venus	4.87×10^{24}	0	1.082×10^{11}
Earth	5.9742×10^{24}	0	$1.495978707 \times 10^{11}$
Mars	6.42×10^{23}	0	2.280×10^{11}
Jupiter	1.8986×10^{27}	0	$7.78340821 \times 10^{11}$
Saturn	5.86×10^{26}	0	1.4320×10^{12}
Uranus	8.68×10^{25}	0	2.8670×10^{12}
Neptune	1.02×10^{26}	0	4.5150×10^{12}
Pluto	1.30×10^{22}	0	5.9064×10^{12}

Appendix 1: Initial planetary conditions

```
import numpy as np
import math
import matplotlib.pyplot as plt
import timeit

start = timeit.default_timer()

dt = 155793
dt_max = 86400 * 365.25 * 1.1
m_sun = 1.98892e30
r = 1.495978707e11
#r = 0.4 + (8202/25000)*1.495978707e11
G = 6.67408e-11
vx = ((G*m_sun)/r)**0.5
vy = 0
x = 0
y = r
x_array = []
y_array = []
x_array_au = []
y_array_au = []

for t in np.arange(dt, dt_max, dt):

    r = math.sqrt(x**2 + y**2)

    vx = vx - ((G*m_sun*x) / (r**3)) * dt
    vy = vy - ((G*m_sun*y) / (r**3)) * dt


    if x < 0 and x + vx * dt > 0:
        print('Orbit Period:', t/86400)

    x = x + vx * dt
    y = y + vy * dt

    x_array.append(x)
    y_array.append(y)


for i in range(len(x_array)):
    x_array_au.append(x_array[i]/r)
    y_array_au.append(y_array[i]/r)

plt.plot(x_array_au, y_array_au)
plt.legend(['Earth'],loc='upper right')
plt.axis('square')
plt.ylabel('AU')
plt.xlabel('AU')
plt.title('Plot of Earths orbit using Euler-Cromer method')
plt.show()

stop = timeit.default_timer()

print('Runtime:', '{:.2f}'.format(stop - start), 'seconds')
```

Appendix 2: Euler-Cromer 2-body fixed position Sun

```
import numpy as np
import math
import matplotlib.pyplot as plt
import timeit

start = timeit.default_timer()

dt = 1131186
dt_max = 86400 * 365.25 * 5000
m_sun = 1.98892e30
r = 1.495978707e11
#r = (0.4 + (8202/25000))*1.495978707e11
G = 6.67408e-11
vx = ((G*m_sun)/r)**0.5
vy = 0
x = 0
y = r
x_array = []
y_array = []
x_array_au = []
y_array_au = []

for t in np.arange(dt, dt_max, dt):

    r = math.sqrt(x**2 + y**2)

    a = vx
    a1 = (-(G*m_sun) / (r**3)) * x
    b = (vx + dt/2 * a1)
    b1 = (-(G*m_sun) / (r**3)) * (x + dt/2 * a)
    c = (vx + dt/2 * b1)
    c1 = (-(G*m_sun) / (r**3)) * (x + dt/2 * b)
    d = (vx + dt * c1)
    d1 = (-(G*m_sun) / (r**3)) * (x + dt * c)

    #if x < 0 and x + dt/6 * (a + 2*b + 2*c + d) > 0 :
    #    print(t/86400)

    x = x + dt/6 * (a + 2*b + 2*c + d)
    vx = vx + dt/6 * (a1 + 2*b1 + 2*c1 + d1)

    e = vy
    e1 = (-(G*m_sun) / (r**3)) * y
    f = (vy + dt/2 * e1)
    f1 = (-(G*m_sun) / (r**3)) * (y + dt/2 * e)
    g = (vy + dt/2 * f1)
    g1 = (-(G*m_sun) / (r**3)) * (y + dt/2 * f)
    h = (vy + dt * g1)
    h1 = (-(G*m_sun) / (r**3)) * (y + dt * g)

    y = y + dt/6 * (e + 2*f + 2*g + h)
    vy = vy + dt/6 * (e1 + 2*f1 + 2*g1 + h1)

    x_array.append(x)
    y_array.append(y)
```

```
for i in range(len(x_array)):
    x_array_au.append(x_array[i]/1.495978707e11)
    y_array_au.append(y_array[i]/1.495978707e11)

plt.plot(x_array_au, y_array_au)
plt.legend(['Planet X'],loc='upper right')
plt.axis('square')
plt.ylabel('AU')
plt.xlabel('AU')
plt.title("Plot of Planet X's orbit using Runge-Kutta method")
plt.show()

stop = timeit.default_timer()

print('Runtime:', '{:.2f}'.format(stop - start), 'seconds')
```

Appendix 3: Runge-Kutta 2-body fixed position Sun

```
import numpy as np
import math

dt = 0
dt_max = 86400 * 365.25 * 10
m_sun = 1.98892e30
m_earth = 5.9742e24
r = 1.495978707e11
G = 6.67408e-11


dt_high = 10000000
dt_low = 1

while dt_high - dt_low > 1:

    flag = 0
    dt = 0.5 * (dt_high + dt_low)

    vx = ((G*m_sun)/r)**0.5
    vy = 0
    x = 0
    y = 1.495978707e11
    x_array = []
    y_array = []

    KE_0 = 0.5 * m_earth * (vx**2 + vy**2)
    GPE_0 = -(G * m_sun * m_earth) / (x**2 + y**2)**0.5
    TE_0 = KE_0 + GPE_0

    for t in np.arange(dt, dt_max, dt):

        r = math.sqrt(x**2 + y**2)

        vx = vx - ((G*m_sun*x) / (r**3)) * dt
        vy = vy - ((G*m_sun*y) / (r**3)) * dt

        x = x + vx * dt
        y = y + vy * dt

        x_array.append(x)
        y_array.append(y)

        KE_1 = 0.5 * m_earth * (vx**2 + vy**2)
        GPE_1 = -(G * m_sun * m_earth) / (x**2 + y**2)**0.5
        TE_1 = KE_1 + GPE_1

        if abs((TE_0 - TE_1)/TE_1) >= 0.001:
            flag = 1
            break

    if flag == 1:
        dt_high = dt
    else:
        dt_low = dt
    print(dt_high, dt_low)
```

Appendix 4: Euler-Cromer 2-body fixed position Sun (Convergence for dt)

```
import numpy as np
import math

dt = 0
dt_max = 86400 * 365.25 * 10
m_sun = 1.98892e30
m_earth = 5.9742e24
r = 1.495978707e11
G = 6.67408e-11


dt_high = 10000000
dt_low = 1

while dt_high - dt_low > 1:

    flag = 0
    dt = 0.5 * (dt_high + dt_low)

    vx = ((G*m_sun)/r)**0.5
    vy = 0
    x = 0
    y = 1.495978707e11
    x_array = []
    y_array = []

    KE_0 = 0.5 * m_earth * (vx**2 + vy**2)
    GPE_0 = -(G * m_sun * m_earth) / (x**2 + y**2)**0.5
    TE_0 = KE_0 + GPE_0

    for t in np.arange(dt, dt_max, dt):

        r = math.sqrt(x**2 + y**2)

        a = vx
        a1 = (-(G*m_sun) / (r**3)) * x
        b = (vx + dt/2 * a1)
        b1 = (-(G*m_sun) / (r**3)) * (x + dt/2 * a)
        c = (vx + dt/2 * b1)
        c1 = (-(G*m_sun) / (r**3)) * (x + dt/2 * b)
        d = (vx + dt * c1)
        d1 = (-(G*m_sun) / (r**3)) * (x + dt * c)


        x = x + dt/6 * (a + 2*b + 2*c + d)
        vx = vx + dt/6 * (a1 + 2*b1 + 2*c1 + d1)

        e = vy
        e1 = (-(G*m_sun) / (r**3)) * y
        f = (vy + dt/2 * e1)
        f1 = (-(G*m_sun) / (r**3)) * (y + dt/2 * e)
        g = (vy + dt/2 * f1)
        g1 = (-(G*m_sun) / (r**3)) * (y + dt/2 * f)
        h = (vy + dt * g1)
        h1 = (-(G*m_sun) / (r**3)) * (y + dt * g)
```

```
        y = y + dt/6 * (e + 2*f + 2*g + h)
        vy = vy + dt/6 * (e1 + 2*f1 + 2*g1 + h1)

        x_array.append(x)
        y_array.append(y)

        KE_1 = 0.5 * m_earth * (vx**2 + vy**2)
        GPE_1 = -(G * m_sun * m_earth) / (x**2 + y**2)**0.5
        TE_1 = KE_1 + GPE_1

        if abs((TE_0 - TE_1)/TE_1) >= 0.001:
            flag = 1
            break

    if flag == 1:
        dt_high = dt
    else:
        dt_low = dt
    print(dt_high, dt_low)
```

Appendix 5: Runge-Kutta 2-body fixed position Sun (Convergence for dt)

```
import numpy as np
import matplotlib.pyplot as plt
import timeit

start = timeit.default_timer()

plt.rcParams['figure.dpi'] = 150
plt.rcParams['savefig.dpi'] = 150

N = 11
G = 6.67408e-11

dt = 50000
t_max = 86400 * 365.25 * 1

names = ['Sun', 'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Pluto', 'Asteroid']
r = [0, 5.79e10, 1.082e11, 1.495978707e11, 2.280e11, 7.78340821e11, 1.4320e12, 2.8670e12, 4.5150e12, 5.9064e12, 4.9e11]
m = [1.98892e30, 3.30e23, 4.87e24, 5.9742e24, 6.42e23, 1.8986e27, 5.68e26, 8.68e25, 1.02e26, 1.30e22, 0.95e21]
x = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
y = [0, 5.79e10, 1.082e11, 1.495978707e11, 2.280e11, 7.78340821e11, 1.4320e12, 2.8670e12, 4.5150e12, 5.9064e12, 4.9e11]
vx = [0,
      ((G*m[0])/r[1])**0.5,
      ((G*m[0])/r[2])**0.5,
      ((G*m[0])/r[3])**0.5,
      ((G*m[0])/r[4])**0.5,
      ((G*m[0])/r[5])**0.5,
      ((G*m[0])/r[6])**0.5,
      ((G*m[0])/r[7])**0.5,
      ((G*m[0])/r[8])**0.5,
      ((G*m[0])/r[9])**0.5,
      ((G*m[0])/r[10])**0.5]
vy = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

ax = np.zeros(N)
ay = np.zeros(N)
bx = np.zeros(N)
by = np.zeros(N)
cx = np.zeros(N)
cy = np.zeros(N)
dx = np.zeros(N)
dy = np.zeros(N)

avx = np.zeros(N)
avy = np.zeros(N)
bvx = np.zeros(N)
bvy = np.zeros(N)
cvx = np.zeros(N)
cvy = np.zeros(N)
dvx = np.zeros(N)
dvy = np.zeros(N)

#List of displacement values for each planet
x_array = [[],[],[],[],[],[],[],[],[],[],[]]
y_array = [[],[],[],[],[],[],[],[],[],[],[]]
x_array_au = [[],[],[],[],[],[],[],[],[],[],[]]
y_array_au = [[],[],[],[],[],[],[],[],[],[],[]]
```

```
#Kinetic Energy
KE = [[],[],[],[],[],[],[],[],[],[],[]]

#Gravitational Potential Energy
GPE = [[],[],[],[],[],[],[],[],[],[],[]]

#Total Energy
TE = [[],[],[],[],[],[],[],[],[],[],[]]

#Total Initial Linear Momentum of Planets
initial_mom = 0
for i in range(1, N):
        initial_mom += m[i] * (vx[i]**2 + vy[i]**2)**0.5

#print(initial_mom)
vx[0] = initial_mom / m[0]
print(vx[0])

def accnx(mj, xj, yj, xi, yi):
    xdiff = xj - xi
    ydiff = yj - yi
    r = (xdiff**2 +ydiff**2)**0.5
    return ((G*mj*(xdiff)) / (r**3))

def accny(mj, xj, yj, xi, yi):
    xdiff = xj-xi
    ydiff = yj-yi
    r = (xdiff**2 +ydiff**2)**0.5
    return ((G*mj*(ydiff)) / (r**3))


for t in np.arange(dt, t_max, dt):
    #RK 1
    for i in range(0, N):
        ax[i] = vx[i]
        ay[i] = vy[i]
        avx[i] = 0
        avy[i] = 0

        for j in range(0, N):
            if i != j:
                avx[i] += accnx(m[j], x[j], y[j], x[i], y[i])
                avy[i] += accny(m[j], x[j], y[j], x[i], y[i])
    #RK2
    for i in range(0, N):

        bx[i] = vx[i] + (dt/2)*(avx[i])
        by[i] = vy[i] + (dt/2)*(avy[i])
        bvx[i] = 0
        bvy[i] = 0

        for j in range(0, N):
            if i != j:
                bvx[i] += accnx(m[j], x[j]+(dt*ax[j]/2), y[j]+(dt*ay[j]/2), x[i]+(dt*ax[i]/2), y[i]+(dt*ay[i]/2))
                bvy[i] += accny(m[j], x[j]+(dt*ax[j]/2), y[j]+(dt*ay[j]/2), x[i]+(dt*ax[i]/2), y[i]+(dt*ay[i]/2))
```

```
    #RK4
    for i in range(0, N):

        dx[i] = vx[i] + (dt)*(cvx[i])
        dy[i] = vy[i] + (dt)*(cvy[i])
        dvx[i] = 0
        dvy[i] = 0

        for j in range(0, N):
            if i != j:
                dvx[i] += accnx(m[j], x[j]+(dt*cx[j]), y[j]+(dt*cy[j]), x[i]+(dt*cx[i]), y[i]+(dt*cy[i]))
                dvy[i] += accny(m[j], x[j]+(dt*cx[j]), y[j]+(dt*cy[j]), x[i]+(dt*cx[i]), y[i]+(dt*cy[i]))

    #if x[3] < 0 and x[3] + dt/6 * (ax[3] + 2*bx[3] + 2*cx[3] + dx[3]) > 0 :
        #print(t/86400)

    #RK Final
    for i in range(0, N):
        x[i] = x[i] + dt/6 * (ax[i] + bx[i]*2 + cx[i]*2 + dx[i])
        y[i] = y[i] + dt/6 * (ay[i] + by[i]*2 + cy[i]*2 + dy[i])

        vx[i] = vx[i] + dt/6 * (avx[i] + bvx[i]*2 + cvx[i]*2 + dvx[i])
        vy[i] = vy[i] + dt/6 * (avy[i] + bvy[i]*2 + cvy[i]*2 + dvy[i])

        x_array[i].append(x[i])
        y_array[i].append(y[i])
        KE[i].append(0.5 * m[i] * (vx[i]**2 + vy[i]**2))

    for i in range(1, N):
        GPE_temp = 0
        for j in range(0, N):
            if i != j:

                xdiff = x[j]-x[i]
                ydiff = y[j]-y[i]
                r = (xdiff**2 +ydiff**2)**0.5

                GPE_temp += ((G * m[i] * m[j])/ r)

        GPE[i].append(GPE_temp)
        TE[i].append(GPE[i][-1] + KE[i][-1])

for i in range(0,N):
    for j in range(len(x_array[i])):
        x_array_au[i].append(x_array[i][j]/1.495978707e11)
        y_array_au[i].append(y_array[i][j]/1.495978707e11)
```

```
#Plot
plt.plot(x_array_au[1], y_array_au[1],
         x_array_au[2], y_array_au[2],
         x_array_au[3], y_array_au[3],
         x_array_au[4], y_array_au[4],
         x_array_au[5], y_array_au[5],
         x_array_au[6], y_array_au[6],
         x_array_au[7], y_array_au[7],
         x_array_au[8], y_array_au[8],
         x_array_au[9], y_array_au[9],
         x_array_au[10], y_array_au[10]
         )
plt.legend([names[1],
            names[2],
            names[3],
            names[4],
            names[5],
            names[6],
            names[7],
            names[8],
            names[9],
            names[10]
            ],
            loc='upper right')
plt.axis('square')
plt.ylabel('AU')
plt.xlabel('AU')
plt.title("Plot of Solar System using Runge-Kutta method")
plt.show()

stop = timeit.default_timer()

print('Runtime:', '{:.2f}'.format(stop - start), 'seconds')
```

Appendix 6: Runge-Kutta n-body non-fixed position Sun. Minor variations of this code were used for plotting Kinetic Energy of asteroid, Rouge Planet, Sun's movement etc.