Modelling the Three Body Problem in Classical Mechanics using Python

An overview of the fundamentals of gravitation, the odeint solver in Scipy and 3D plotting in Matplotlib

Gaurav Deshmukh
Jul 3, 2019 · 12 min read ★

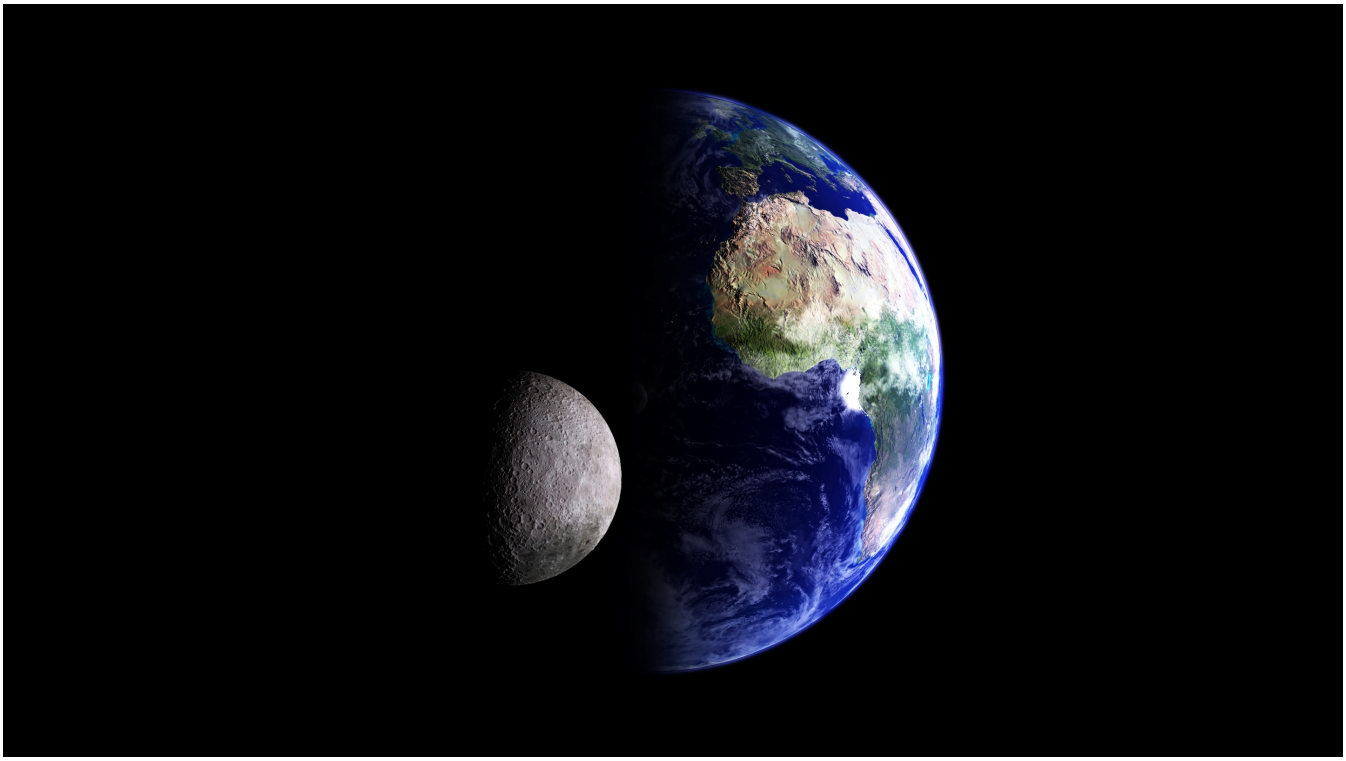


Image by Kevin Gill on Flickr

1. Introduction

I recently read *The Three Body Problem*, a sci-fi book by Chinese author Liu Cixin. In it, he describes a fictitious alien civilization living on a planet called Trisolaris that is surrounded by three stars. How different do you imagine their existence would be from ours due to the presence of three stars? Blinding sunshine? Persistent summers? As it turns out, something much worse.

and the system still remains stable. It has, what we call, an analytical solution — that is, we can solve the equations describing it and get a function that gives the time evolution of the system from 1 second to a million years accurately.

However, when you add a third body, something extraordinary happens. The system becomes chaotic and highly unpredictable. It has no analytical solution (except for a few special cases) and its equations can only be solved numerically on a computer. They can turn abruptly from stable to unstable and vice versa. The Trisolarans living in such a chaotic world developed the ability to "dehydrate" themselves and hibernate during the "Chaotic Eras" and awake and live peacefully during the "Stable Eras".

The intriguing visualization of the star system in the book inspired me to read up on the n-body class of problems in gravitation and the numerical methods used to solve them. This article touches upon a few core concepts of gravitation required to understand the problem and the numerical methods required to solve the equations describing the systems.

Through this article, you will get to read about the implementation of the following tools and concepts:

- Solving differential equations in Python using the **odeint** function in the **Scipy** module.
- Non-dimensionalizing an equation
- Making 3D plots in **Matplotlib**

2. Basics of Gravitation

2.1 Newton's Law of Gravitation

Newton's Law of Gravitation says that any two point masses have an attractive force between them (called the gravitational force), the magnitude of which is **directly proportional to the product of their masses** and **inversely proportional to the square of the distance between them.** The equation below represents this law in vector form.

Here, G is the universal gravitational constant, m_1 and m_2 are the masses of the two objects and r is the distance between them. The unit vector points away from the body m_1 towards m_2 and the force too acts in the same direction.

2.2 Equation of Motion

According to **Newton's second law of motion**, the net force on an object produces a net change in momentum of the object — in simple terms, **force is mass times acceleration**. So, applying the above equation to the body having mass m_1, we get the following differential equation of motion for the body.

Note here that we have unraveled the unit vector as the vector r divided by its magnitude $|r|$, thus increasing the power of the r term in the denominator to 3.

Now, we have a **second-order differential equation** that describes the interaction between two bodies due to gravity. To simplify its solution, we can break it down into **two first order differential equations.**

The acceleration of an object is the change in velocity of the object with time so the **second order differential of position** can be replaced with a **first order differential of velocity**. Similarly, the **velocity** can be expressed as a **first order differential of the position.**

system, we will be solving two sets of these two equations.

2.3 Centre of Mass

Another useful concept to keep in mind is the centre of mass of a system. The centre of mass is a point where the the **sum of the all the mass moments of the system is zero** — in simple terms, you can imagine it as the point where the whole mass of the system is balanced.

There is a simple formula to find the centre of mass of a system and its velocity. It involves taking mass-weighted averages of the position and velocity vectors.

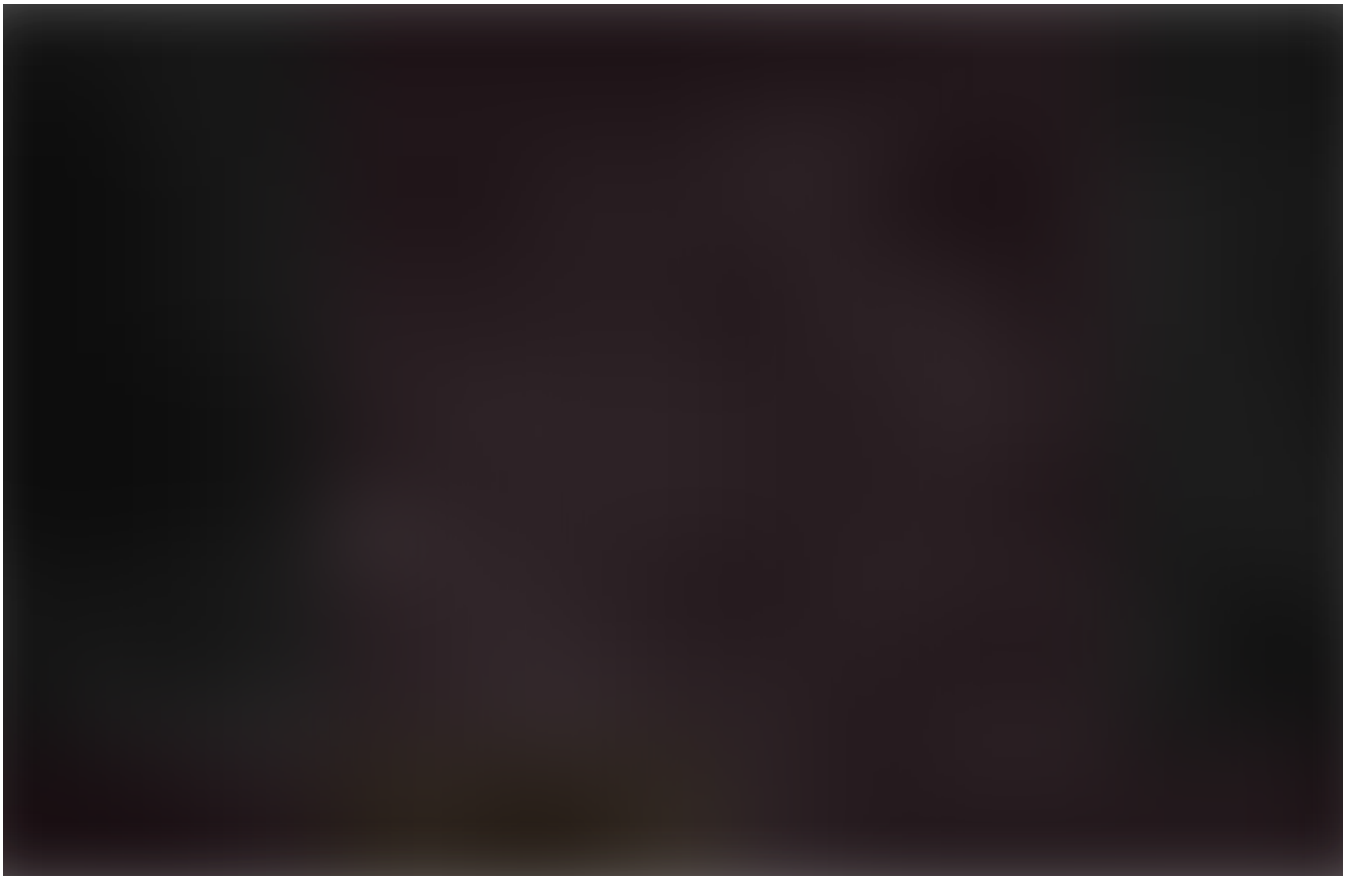
Before modelling a three-body system, let us first model a two-body system, observe its behaviour and then extend the code to work for three bodies.

3. The Two-Body Model

3.1 Alpha Centauri Star System

A famous real-world example of a two-body system is perhaps the **Alpha Centauri** star system. It contains three stars — **Alpha Centauri A, Alpha Centauri B** and **Alpha Centauri C** (commonly called Proxima Centauri). However, since Proxima Centauri has a mass that is negligibly small as compared to the other two stars, Alpha Centauri is considered to be a **binary star system**. An important point to note here is that the

Moon do not significantly influence the Sun's path.



The Alpha Centauri binary star system captured from the Paranal Observatory in Chile by John Colosimo

3.2 Non-Dimensionalization

Before we start solving these equations, we have to first non-dimensionalize them. What does that mean? We convert all the quantities in the equation (like **position, velocity, mass** and so on) that have dimensions (like **m, m/s, kg** respectively) to non-dimensional quantities that have magnitudes close to unity. The reasons for doing so are:

- In the differential equation, different terms may have different orders of magnitude (from 0.1 to 10^{30}). Such a **huge disparity may lead to slow convergence of numerical methods.**
- If the magnitude of all terms becomes close to unity, all the **calculations will become computationally cheaper** than if the magnitudes were asymmetrically large or small.

or large on the cosmic scale. However, if I say 2 times the mass of the sun, you will easily be able to grasp the significance of that quantity.

To non-dimensionalize the equations, **divide each quantity by a fixed reference quantity**. For example, divide the mass terms by the mass of the sun, position (or distance) terms with the distance between the two stars in the Alpha Centauri system, time term with the orbital period of Alpha Centauri and velocity term with the relative velocity of the earth around the sun.

When you divide each term by the reference quantity, you will also need to multiply it to avoid changing the equation. All these terms along with G can be clubbed into a constant, say K_1 for equation 1 and K_2 for equation 2. Thus, the non-dimensionalized equations are as follows:

The bar over the terms indicates that the terms are non-dimensional. So these are the final equations that we'll be using in our simulation.

3.3 Code

Let us begin by importing all the required modules for the simulation.

```
#Import scipy
import scipy as sci

#Import matplotlib and associated modules for 3D and animations
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
```

```
#Define universal gravitation constant
G=6.67408e-11 #N-m2/kg2

#Reference quantities
m_nd=1.989e+30 #kg #mass of the sun
r_nd=5.326e+12 #m #distance between stars in Alpha Centauri
v_nd=30000 #m/s #relative velocity of earth around the sun
t_nd=79.91*365*24*3600*0.51 #s #orbital period of Alpha Centauri

#Net constants
K1=G*t_nd*m_nd/(r_nd**2*v_nd)
K2=v_nd*t_nd/r_nd
```

It is time to define some parameters that define the two stars that we are trying to simulate — their **masses**, **initial positions** and **initial velocities**. Note that these parameters are non-dimensional, so the mass of Alpha Centauri A is defined as 1.1 (indicating 1.1 times the mass of the sun, which is our reference quantity). The velocities are arbitrarily defined in a way such that none of the bodies escape each others' gravitational pull.

```
#Define masses
m1=1.1 #Alpha Centauri A
m2=0.907 #Alpha Centauri B

#Define initial position vectors
r1=[-0.5,0,0] #m
r2=[0.5,0,0] #m

#Convert pos vectors to arrays
r1=sci.array(r1,dtype="float64")
r2=sci.array(r2,dtype="float64")

#Find Centre of Mass
r_com=(m1*r1+m2*r2)/(m1+m2)

#Define initial velocities
v1=[0.01,0.01,0] #m/s
v2=[-0.05,0,-0.1] #m/s

#Convert velocity vectors to arrays
v1=sci.array(v1,dtype="float64")
v2=sci.array(v2,dtype="float64")
```

We have now defined most of the major quantities required for our simulation. We can now move on to preparing the **odeint** solver in scipy to solve our system of equations.

To solve any ODE, you require the **equations** (of course!), **a set of initial conditions** and the **time span** for which the equations are to be solved. The **odeint** solver also requires these primary three things. The equations are defined through the means of a function. The function takes in an array containing all the dependent variables (here position and velocity) and an array containing all the independent variables (here time) in that order. It returns the values of all the differentials in an array.

```
#A function defining the equations of motion
def TwoBodyEquations(w,t,G,m1,m2):
    r1=w[:3]
    r2=w[3:6]
    v1=w[6:9]
    v2=w[9:12]

    r=sci.linalg.norm(r2-r1) #Calculate magnitude or norm of vector

    dv1bydt=K1*m2*(r2-r1)/r**3
    dv2bydt=K1*m1*(r1-r2)/r**3
    dr1bydt=K2*v1
    dr2bydt=K2*v2

    r_derivs=sci.concatenate((dr1bydt,dr2bydt))
    derivs=sci.concatenate((r_derivs,dv1bydt,dv2bydt))
    return derivs
```

From the code snippet, you may be able to identify the differential equations quite easily. What are the other odds and ends? Remember that we are solving the equation for 3 dimensions, so each position and velocity vector will have 3 components. Now, if you consider the two vector differential equations given in the previous section, they need to be solved for all the 3 components of the vectors. So, for a single body you need to solve 6 scalar differential equations. For two bodies, you got it, 12 scalar differential equations. So we make an array *w* having size 12 that stores the position and velocity coordinates of the two bodies in question.

At the end of the function, we concatenate or join all the different derivatives and return an array *derivs* of size 12.

```
#Package initial parameters
init_params=sci.array([r1,r2,v1,v2]) #create array of initial params
init_params=init_params.flatten() #flatten array to make it 1D
time_span=sci.linspace(0,8,500) #8 orbital periods and 500 points

#Run the ODE solver
import scipy.integrate

two_body_sol=sci.integrate.odeint(TwoBodyEquations,init_params,time_
span,args=(G,m1,m2))
```

The variable *two_body_sol* contains all the information about the two-body system including the position vectors and velocity vectors. To create our plots and animations, we only need the position vectors so let us extract them to two different variables.

```
r1_sol=two_body_sol[:,:3]
r2_sol=two_body_sol[:,3:6]
```

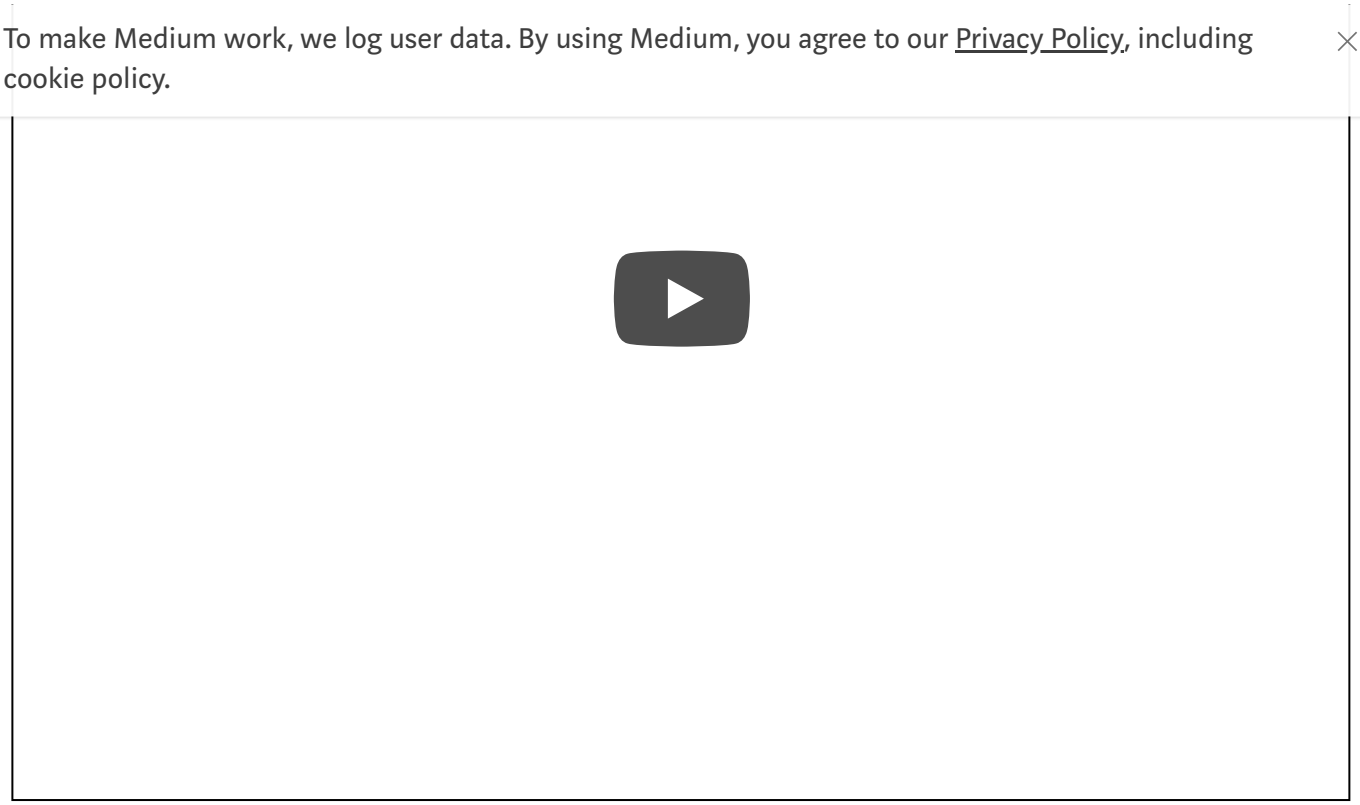
It's time to plot! This is where we will utilize the 3D plotting capabilities of Matplotlib.

```
#Create figure
fig=plt.figure(figsize=(15,15))

#Create 3D axes
ax=fig.add_subplot(111,projection="3d")

#Plot the orbits
ax.plot(r1_sol[:,0],r1_sol[:,1],r1_sol[:,2],color="darkblue")
ax.plot(r2_sol[:,0],r2_sol[:,1],r2_sol[:,2],color="tab:red")

#Plot the final positions of the stars
ax.scatter(r1_sol[-1,0],r1_sol[-1,1],r1_sol[-1,2],color="darkblue",m
arker="o",s=100,label="Alpha Centauri A")
ax.scatter(r2_sol[-1,0],r2_sol[-1,1],r2_sol[-1,2],color="tab:red",ma
rker="o",s=100,label="Alpha Centauri B")

#Add a few more bells and whistles
ax.set_xlabel("x-coordinate",fontsize=14)
ax.set_ylabel("y-coordinate",fontsize=14)
ax.set_zlabel("z-coordinate",fontsize=14)
ax.set_title("Visualization of orbits of stars in a two-body
```

The final plot makes it pretty clear that the orbits follow a predictable pattern, as is expected of a solution of a two-body problem.

A Matplotlib plot showing the time evolution of the orbits of the two stars

Here's an animation that shows the step-by-step evolution of the orbits.



An animation made in Matplotlib that shows the time evolution step-by-step (code not given in article)

There is one more visualization that we can make, and that is from the frame of reference of the centre of mass. The above visualization is from some arbitrary stationary point in space but if we observe the motion of the two bodies from the centre of mass of the system, we will see an even more visible pattern.

So, first let us find the position of the centre of mass at every time step and then subtract that vector from the position vectors of the two bodies to find their locations relative to the centre of mass.

```
#Find location of COM
rcom_sol=(m1*r1_sol+m2*r2_sol)/(m1+m2)

#Find location of Alpha Centauri A w.r.t COM
r1com_sol=r1_sol-rcom_sol

#Find location of Alpha Centauri B w.r.t COM
r2com_sol=r2_sol-rcom_sol
```

Finally, we can use the code utilized for plotting the previous visual with a change in variables to plot the following visual.

A Matplotlib plot showing the time evolution of the orbits of the two stars as seen from the COM

If you were to sit at the COM and observe the two bodies, you would see the above orbits. It is not clear from this simulation since the timescale is very small but even these orbits keep rotating ever so slightly.

It is quite clear now that they follow very predictable paths and that you can use a function — perhaps the equation of an ellipsoid — to describe their motion in space, as expected of a two-body system.

4. The Three-Body Model

4.1 Code

Now to extend our previous code to a three-body system, we have to make a few additions to the parameters — add mass, position and velocity vectors of the third

```
#Mass of the Third Star
m3=1.0 #Third Star

#Position of the Third Star
r3=[0,1,0] #m
r3=sci.array(r3,dtype="float64")

#Velocity of the Third Star
v3=[0,-0.01,0]
v3=sci.array(v3,dtype="float64")
```

We need to update the formulas of centre of mass and velocity of the centre of mass in the code.

```
#Update COM formula
r_com=(m1*r1+m2*r2+m3*r3)/(m1+m2+m3)

#Update velocity of COM formula
v_com=(m1*v1+m2*v2+m3*v3)/(m1+m2+m3)
```

For a three-body system, we will need to modify the equations of motion to include the extra gravitational force exerted by the presence of another body. Thus, we need to add a force term on the RHS for every other body exerting a force on the body in question. In the case of a three-body system, one body will be affected by the forces exerted by the two remaining bodies and hence **two force terms will appear on the RHS**. It can be represented mathematically as.

To reflect these changes in the code, we'll need to create a new function to supply to the **odeint** solver.

```
    r2=w[3:6]
    r3=w[6:9]
    v1=w[9:12]
    v2=w[12:15]
    v3=w[15:18]

    r12=sci.linalg.norm(r2-r1)
    r13=sci.linalg.norm(r3-r1)
    r23=sci.linalg.norm(r3-r2)

    dv1bydt=K1*m2*(r2-r1)/r12**3+K1*m3*(r3-r1)/r13**3
    dv2bydt=K1*m1*(r1-r2)/r12**3+K1*m3*(r3-r2)/r23**3
    dv3bydt=K1*m1*(r1-r3)/r13**3+K1*m2*(r2-r3)/r23**3
    dr1bydt=K2*v1
    dr2bydt=K2*v2
    dr3bydt=K2*v3

    r12_derivs=sci.concatenate((dr1bydt,dr2bydt))
    r_derivs=sci.concatenate((r12_derivs,dr3bydt))
    v12_derivs=sci.concatenate((dv1bydt,dv2bydt))
    v_derivs=sci.concatenate((v12_derivs,dv3bydt))
    derivs=sci.concatenate((r_derivs,v_derivs))
    return derivs
```

Finally, we need to call the **odeint** function and supply the above function as well as initial conditions to it.

```
#Package initial parameters
init_params=sci.array([r1,r2,r3,v1,v2,v3]) #Initial parameters
init_params=init_params.flatten() #Flatten to make 1D array
time_span=sci.linspace(0,20,500) #20 orbital periods and 500 points

#Run the ODE solver
import scipy.integrate

three_body_sol=sci.integrate.odeint(ThreeBodyEquations,init_params,t
ime_span,args=(G,m1,m2,m3))
```

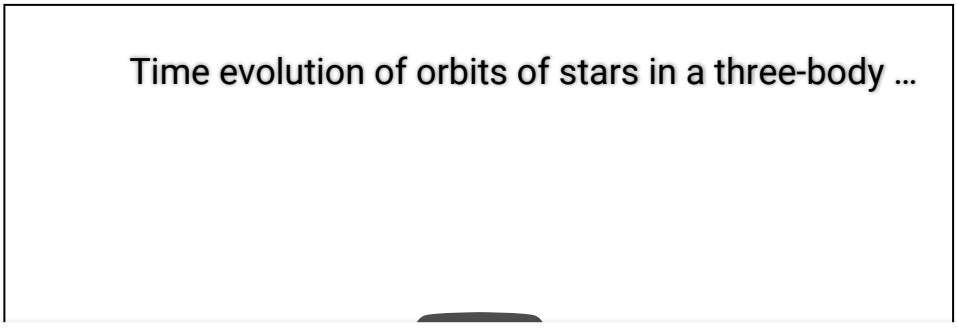
As with the two-body simulation, we need to extract the position coordinates for all the three bodies for plotting.

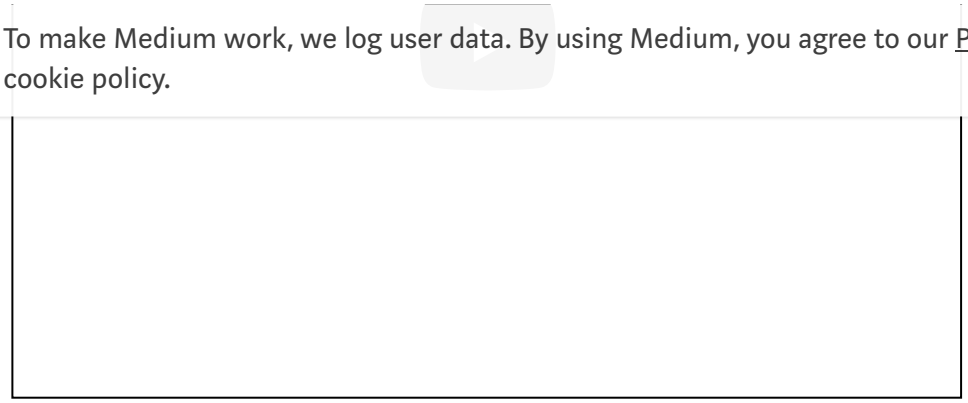
```
r1_sol=three_body_sol[:,:3]
r2_sol=three_body_sol[:,3:6]
r3_sol=three_body_sol[:,6:9]
```

plot below.

A Matplotlib plot showing the time evolution of the orbits of the three stars

An animation will make it easier to make sense of the messy plot.





An animation made in Matplotlib that shows the time evolution step-by-step (code not given in article)

Here's a solution to another initial configuration in which you can observe that the solution seems to be stable initially but then abruptly becomes unstable.

An animation made in Matplotlib that shows the time evolution step-by-step (code not given in article)

In recent years, many interesting solutions to the three-body problem have been discovered due to the availability of greater computational power, some of which appear to be periodic — like the figure-8 solution in which all the three bodies move in a planar figure-8 path.

. . .

Some references for further reading:

1. A small essay on the mathematical description of the three-body problem.
2. A thesis on the study of planar restricted three-body problem solutions (includes figure-8 and Hill's solutions).

I haven't included the code for the animations in this article. If you want to know more, you can email me or contact me on Twitter.

Science Programming Simulation Mathematics Towards Data Science