Search Medium Write

Member-only story

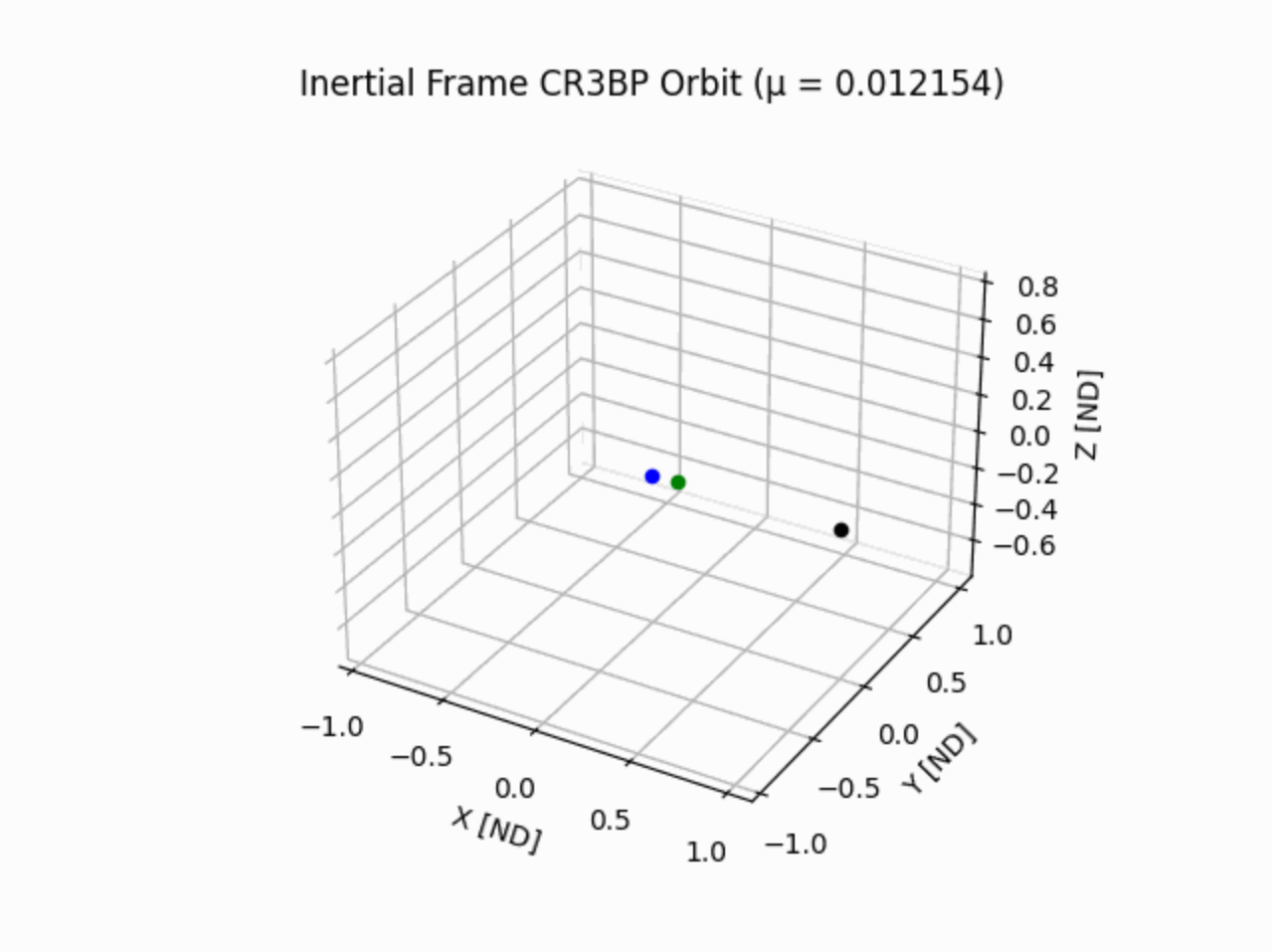
Use Python to Create Three-Body Orbits

Numerically integrate and visualize the trajectory of a satellite moving under the influence of two larger masses

Zack Fizell · Following

Published in Towards Data Science · 9 min read · May 30, 2022

170 1

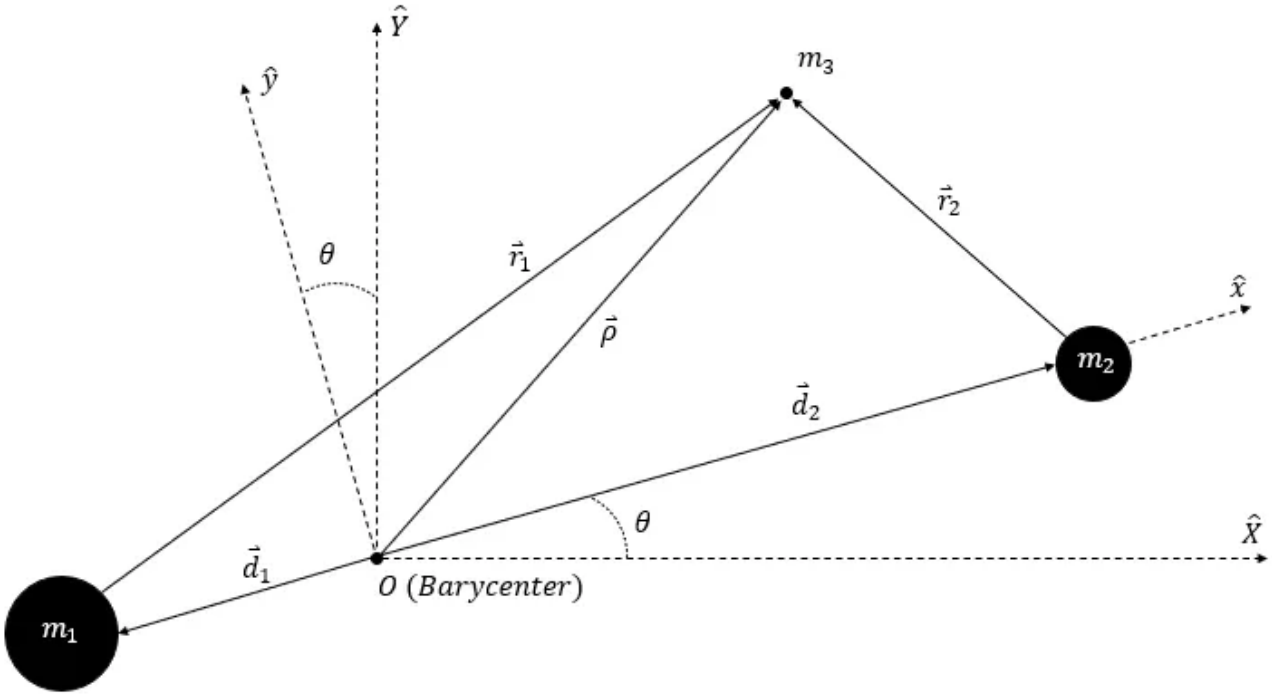


Animated CR3BP Orbit (Inertial Frame) [Created by Author]

In orbital mechanics, the three-body problem (3BP) is the study of how a negligible mass moves under the influence of two larger masses (planet, moon, star, etc.). The results are used to design spacecraft trajectories when the gravitational pull of two larger masses must be considered. For example, the James Webb telescope trajectory and current orbit were devised by utilizing the 3BP. Typically, when people think of orbits, they think of a negligible mass with an elliptic orbit around a single planet or star. With the three-body problem, an additional massive body is added to the system, which increases the difficulty in deriving the equations that describe the motion of the smaller body.

In order to create and visualize orbits in the 3BP, we must derive the equations of motion for the negligible mass. I skip the details of the

derivation here, but if you are interested, this article describes how to derive the equations for the three-body problem. I encourage you to explore that article to fully understand the variables and equations you are about to solve using Python. We will be studying the the circular-restricted three-body problem (CR3BP). In this restricted version of the broader three-body problem, the two larger masses (or primaries) orbit their respective center of mass in circular orbits. This simplifies the equations of motion for a spacecraft (or other negligible mass). The following diagram shows the setup for the CR3BP.



Three-Body Problem Diagram [Created by Author]

The CR3BP equations of motion are derived for m_3 (spacecraft, asteroid, etc.) in the rotating frame (*x, y, z-hat* frame):

$$\ddot{x} = x + 2\dot{y} - \frac{(1-\mu)(x+\mu)}{r_1^3} - \frac{\mu[x-(1-\mu)]}{r_2^3}$$
$$\ddot{y} = y - 2\dot{x} - \frac{(1-\mu)y}{r_1^3} - \frac{\mu y}{r_2^3}$$
$$\ddot{z} = -\frac{(1-\mu)z}{r_1^3} - \frac{\mu z}{r_2^3}$$

These equations are a little intimidating, but once you know what each variable is, they are not too bad. Here, ***x***, ***y***, and ***z*** (***ρ*** vector) are the position coordinates of m_3 in relation to the center of mass of the primaries, in a rotating frame (*x, y, z-hat* frame) that moves with the primaries. The dots over these terms indicate a time derivative, so one dot is a velocity term and two dots indicates an acceleration term. ***μ*** is a non-dimensional mass ratio of the primaries. r_1 and r_2 are the distances of m_3 from the respective primary.

The equations of motion above are derived using non-dimensional variables. The article mentioned above goes into details on how to use non-dimensional parameters, which is important if you are truly trying to learn about the three-body problem. Without going into too many details here, the parameters below are used to remove and add in dimensions (km, kg, s, etc.) as necessary. We can use ***M**** to manipulate mass units (kg), ***L**** for length units (km), and ***T**** for time units (s). This will be explained further in the coding section.

$$M^* \triangleq m_1 + m_2$$
$$L^* \triangleq a$$
$$G^* \triangleq 1 = G\,\frac{M^*(T^*)^2}{(L^*)^3}$$
$$T^* \triangleq \left[\frac{(L^*)^3}{GM^*}\right]^{1/2} \qquad (\text{chosen so } G^* = 1)$$

To solve the CR3BP, we will be using an ODE solver to numerically integrate the equations of motion for m_3. In order to do this, we need to define a state vector and a time-derivative state vector. For our problem, the state vector includes the position and velocity vectors in the rotational frame. The time-derivative state vector is simply the time-derivative of the state vector. These will be utilized using Python's `odeint` numerical integrator. Here, the double-dot x, y, and z are from the equations of motion defined above.

$$\text{State} = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \qquad \text{Time-Derivate State} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix}$$

We can convert a vector in the rotating frame (x, y, and z) to one in the inertial frame (X, Y, and Z) by using the following equation:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} x\cos(t) - y\sin(t) \\ x\sin(t) + y\cos(t) \\ z \end{bmatrix}$$

I know that was a brief overview of the problem, but you do not need to fully understand the CR3BP to be able to code a solution. I will be assuming that you know a little about numerical integration and how to implement it in Python. If you need some guidance on this, I have written another article that might help, and there are plenty of other resources on the internet. Without further ado, let's jump into the code!

. . .

Importing Packages

For this code, we will need to import a few packages. The following list briefly describes the packages used in this code:

- *NumPy* is used to create and manipulate arrays (defined as `np` for ease of calling)
- `odeint` from the *SciPy* library is used for numerical integration
- `pyplot` from *matplotlib* is used to visualize the results from numerical integration (defined as `plt` for ease of calling)

```
# Importing Packages
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
```

Creating Function for Numerical Integration

The next step is creating a user-defined Python function, `model_CR3BP`, that will be used in `odeint` to numerically integrate our state vector. The `odeint` function takes the current state vector to create the time-derivative state vector. Notice we pull the position and velocity components from the state vector to create the state vector time-derivative.

```
# CR3BP Model
def model_CR3BP(state, t):
    x = state[0]
    y = state[1]
    z = state[2]
    x_dot = state[3]
    y_dot = state[4]
    z_dot = state[5]
    x_ddot = x+2*y_dot-((1-mu)*(x+mu))/((x+mu)**2+y**2+z**2)**(3/2)\
```

```
            -(mu*(x-(1-mu)))/((x-(1-mu))**2+y**2+z**2)**(3/2)
    y_ddot = y-2*x_dot-((1-mu)*y)/((x+mu)**2+y**2+z**2)**(3/2)\
            -(mu*y)/((x-(1-mu))**2+y**2+z**2)**(3/2)
    z_ddot = -((1-mu)*z)/((x+mu)**2+y**2+z**2)**(3/2)\
            -(mu*z)/((x-(1-mu))**2+y**2+z**2)**(3/2)
    dstate_dt = [x_dot, y_dot, z_dot, x_ddot, y_ddot, z_ddot]
    return dstate_dt
```

Defining Non-Dimensional Parameters

Here, we define a few parameters for our system of choice. I chose the Earth-Moon system for this article, but if you wanted to choose another system (Pluto-Charon, Sun-Jupiter, etc.), you can adjust the masses and semi-major axis variables accordingly. Here, m_1 and m_2 are the masses of the Earth and Moon, respectively. a is the semi-major axis of the Moon's orbit (if they had a perfectly circular orbit). I used the average distance of the Moon's orbit for a since the Moon orbits the Earth in practically a circular orbit. Next, the non-dimensional parameters are defined using the equations from the beginning of the article.

```
# Defining ND Parameters
G = 6.67408E-20  # Univ. Gravitational Constant [km3 kg-1 s-2]
mEarth = 5.97219E+24  # Mass of the Earth [kg]
mMoon = 7.34767E+22  # Mass of the Moon [kg]
a = 3.844E+5  # Semi-major axis of Earth and Moon [km]
m1 = mEarth
m2 = mMoon
Mstar = m1+m2  # ND Mass Parameter
Lstar = a  # ND Length Parameter
Tstar = (Lstar**3/(G*Mstar))**(1/2)  # ND Time Parameter
mu = m2/Mstar
print('\u03BC = ' + str(mu))
```

Defining ODE Solver Inputs

Now, along with the user-defined model, `odeint` needs two more inputs: initial conditions and a time interval to integrate for. I defined an arbitrary

set of initial conditions to create the orbit you saw in the beginning. You can try your own initial conditions or use ephemeris data from this NASA JPL tool. Next, we define the initial state vector, `state_0`, and the time array of interest, `t`. Notice the dimensions are removed using the ND parameters from above. The kilometers and seconds units are removed by either dividing or multiplying by ***T**** and ***L****.

```
# Initial Conditions [Initial State Vector]
X_0 = 50000/Lstar  # ND x
Y_0 = 0            # ND y
Z_0 = 0            # ND z
VX_0 = 1.08*Tstar/Lstar  # ND x_dot
VY_0 = 3.18*Tstar/Lstar  # ND y_dot
VZ_0 = 0.68*Tstar/Lstar  # ND z_dot
state_0 = [X_0, Y_0, Z_0, VX_0, VY_0, VZ_0]  # ND ICs

# Time Array
t = np.linspace(0, 15, 1000)  # ND Time
```

Numerically Integrating Model

The next step is combining what we have already coded and using it all in the `odeint` function. Here, we use `model_CR3BP`, `state_0`, and `t`. The output of the numerical integration is a state vector for each step we defined in the time array. From the results, we can pull the x, y, and z coordinates (in the rotational frame) that define the position of m_3. Next, using the equations from the beginning of the article, we can convert the rotational state to an inertial state.

```
# Numerically Integrating
sol = odeint(model_CR3BP, state_0, t)

# Rotational Frame Position Time History
X_rot = sol[:, 0]
Y_rot = sol[:, 1]
Z_rot = sol[:, 2]
```

```
# Inertial Frame Position Time History
X_Iner = sol[:, 0]*np.cos(t) - sol[:, 1]*np.sin(t)
Y_Iner = sol[:, 0]*np.sin(t) + sol[:, 1]*np.cos(t)
Z_Iner = sol[:, 2]
```

Adding Primary Position Time History

This step is optional, but including the time history of the two primaries can be important to understanding the motion of m_3. The positions of the primaries are stationary in the rotating frame (for the CR3BP) and can be derived simply by using the center of mass of the primaries and converting to ND units. Next, using the same rotating frame to inertial frame conversion as before, we can determine the inertial motion time history of the primaries.

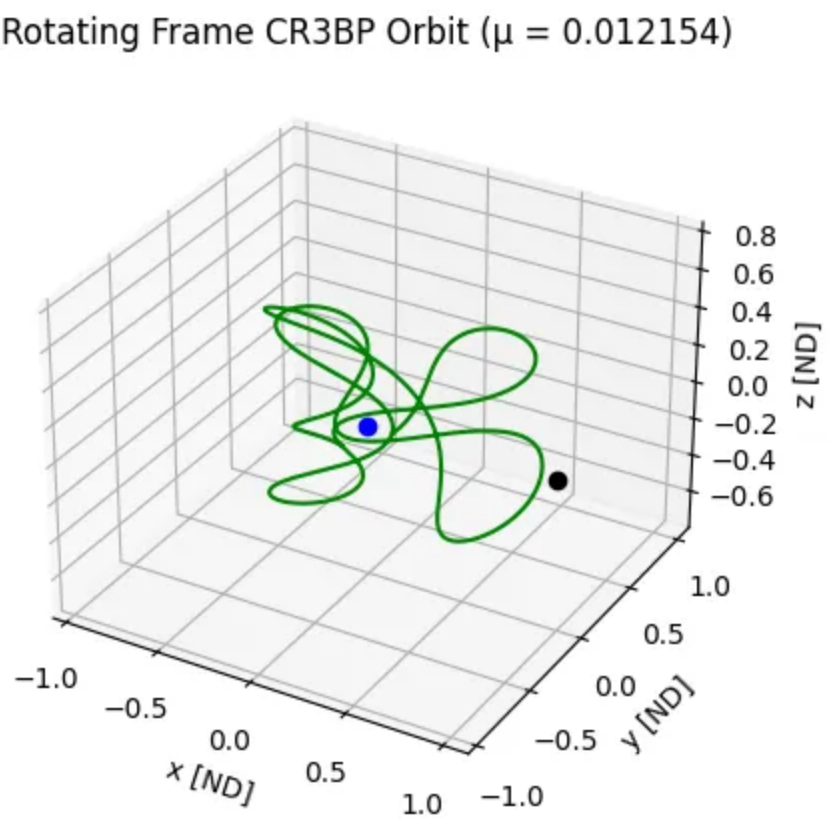
```
# Constant m1 and m2 Rotational Frame Locations for CR3BP Primaries
m1_loc = [-mu, 0, 0]
m2_loc = [(1-mu), 0, 0]

# Moving m1 and m2 Inertial Locations for CR3BP Primaries
X_m1 = m1_loc[0]*np.cos(t) - m1_loc[1]*np.sin(t)
Y_m1 = m1_loc[0]*np.sin(t) + m1_loc[1]*np.cos(t)
Z_m1 = m1_loc[2]*np.ones(len(t))
X_m2 = m2_loc[0]*np.cos(t) - m2_loc[1]*np.sin(t)
Y_m2 = m2_loc[0]*np.sin(t) + m2_loc[1]*np.cos(t)
Z_m2 = m2_loc[2]*np.ones(len(t))
```

Visualizing Data

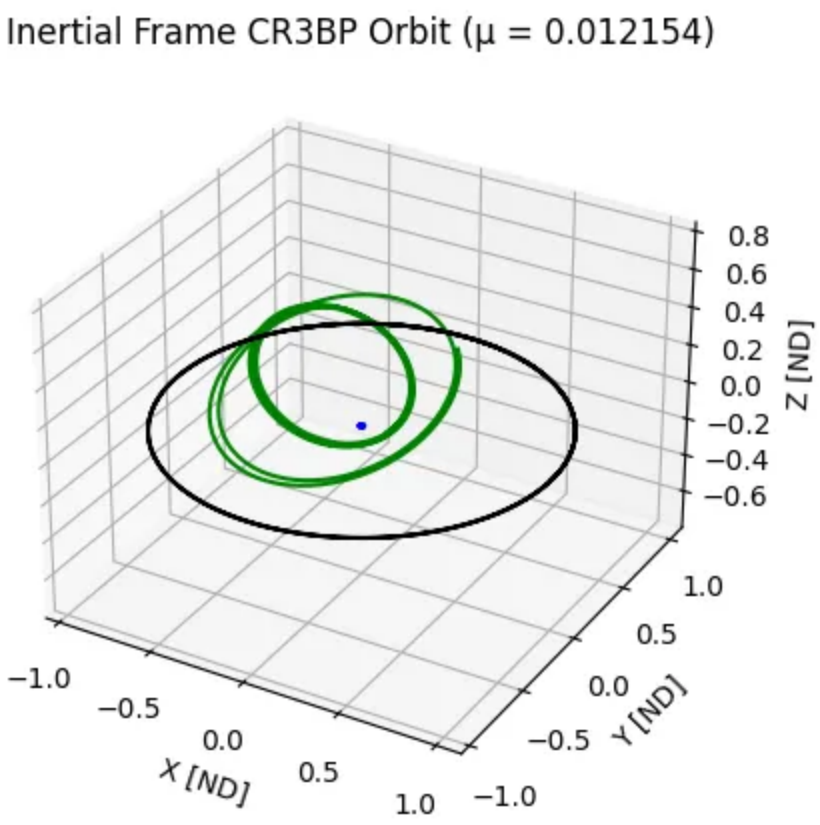
The final step is plotting the rotating and inertial frame motions of each mass. In order to differentiate the orbits, we will use green for m_3, black for the Moon, and blue for the Earth. We can also set the axes such that they display equal axis lengths to give a more accurate visual of the system.

```
# Rotating Frame Plot
fig = plt.figure()
ax = plt.axes(projection='3d')

# Adding Figure Title and Labels
ax.set_title('Rotating Frame CR3BP Orbit (\u03BC = ' + str(round(mu,
6)) + ')')
ax.set_xlabel('x [ND]')
ax.set_ylabel('y [ND]')
ax.set_zlabel('z [ND]')

# Plotting Rotating Frame Positions
ax.plot3D(X_rot, Y_rot, Z_rot, c='green')
ax.plot3D(m1_loc[0], m1_loc[1], m1_loc[2], c='blue', marker='o')
ax.plot3D(m2_loc[0], m2_loc[1], m2_loc[2], c='black', marker='o')

# Setting Axis Limits
xyzlim = np.array([ax.get_xlim3d(), ax.get_ylim3d(),
ax.get_zlim3d()]).T
XYZlim = np.asarray([min(xyzlim[0]), max(xyzlim[1])])
ax.set_xlim3d(XYZlim)
ax.set_ylim3d(XYZlim)
ax.set_zlim3d(XYZlim * 3 / 4)


# Inertial Frame Plot
fig = plt.figure()
ax = plt.axes(projection='3d')

# Adding Figure Title and Labels
ax.set_title('Inertial Frame CR3BP Orbit (\u03BC = ' + str(round(mu,
6)) + ')')
ax.set_xlabel('X [ND]')
ax.set_ylabel('Y [ND]')
ax.set_zlabel('Z [ND]')

# Plotting Inertial Frame Positions
ax.plot3D(X_Iner, Y_Iner, Z_Iner, c='green')
ax.plot3D(X_m1, Y_m1, Z_m1, c='blue')
ax.plot3D(X_m2, Y_m2, Z_m2, c='black')

# Setting Axis Limits
ax.set_xlim3d(XYZlim)
ax.set_ylim3d(XYZlim)
ax.set_zlim3d(XYZlim * 3 / 4)
plt.show()
```

Output of code:



CR3BP Orbit (Rotating Frame) [Created by Author]



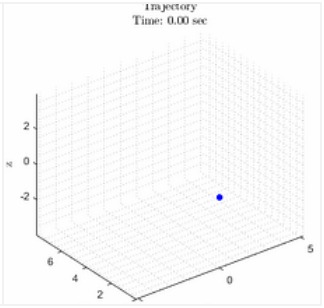
CR3BP Orbit (Inertial Frame) [Created by Author]

The plots above show the inertial and rotational frame results for the orbit of m_3 and the primary masses. As you can see, the spacecraft has a chaotic orbit for this set of initial conditions. If you adjust the initial conditions, you can create various different orbits (some even periodic). I encourage you to mess with the initial conditions and experiment on your own. In order to keep the article and the code as short as possible, I left out how to animate to the plot. If you are interested in adding it, check out the step-by-step guide below:

How to Animate Plots in MATLAB

A simple method to animate data to create dynamic visuals

towardsdatascience.com



. . .

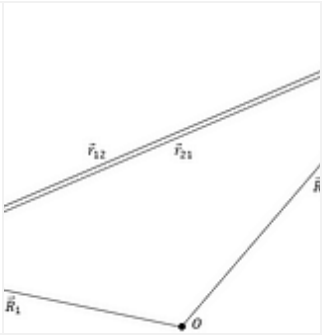
Thank you for reading the article! This was a brief overview of the circular-restricted three-body problem and how to numerically integrate the equations of motion. This is an advanced orbital mechanics topic, so it can be challenging to learn. If you have any questions, feel free to reach out here or on LinkedIn. Give me a follow for weekly articles on orbital mechanics, coding, and machine learning!

If you found this article interesting, you might find the following interesting as well:

How to Solve the Two-Body Problem

Learn the fundamentals of orbital mechanics by deriving the equations of motion for a two-body system

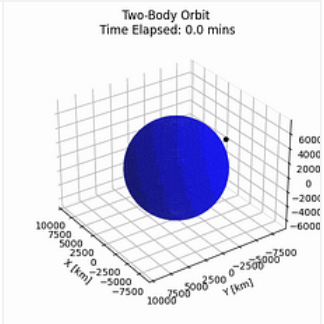
medium.com



Use Python to Create Two-Body Orbits

Learn how to use Python to determine the motion of a spacecraft under the influence of a gravity of a larger body

towardsdatascience.com



Python Physics Space Science Programming

More from the list: "Orbital Mechanics"

Curated by Zack Fizell

Zack ... in Towards Data ...

How to Use MATLAB to Create Two-Body Orbits

6 min read · Mar 25, 2022

Zack Fiz... in ILLUMINATI...

Maneuvering Spacecraft on Orbit

5 min read · Aug 5

Zack Fiz... in ILLUMI

Deriving the Effect o Solar Radiation Pres

7 min read · Jul 4, 2

View list

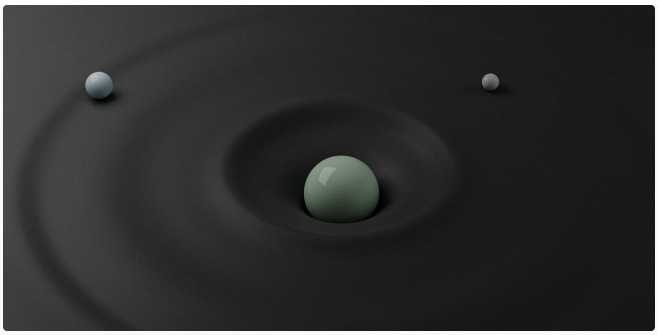
Written by Zack Fizell

Following

1.4K Followers · Writer for Towards Data Science

M.S. in Aeronautics and Astronautics — Articles on Orbital Mechanics| Machine Learning| Coding — https://medium.com/@zackfizell10/membership

More from Zack Fizell and Towards Data Science



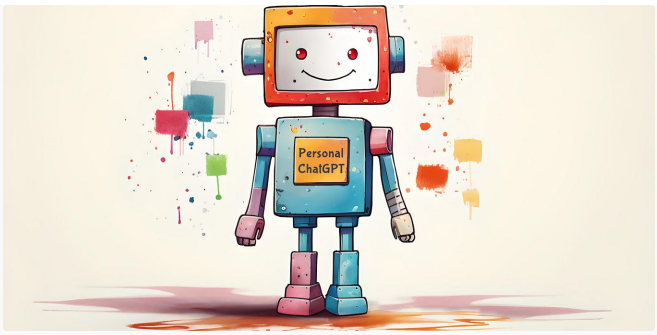
Zack Fizell in Intuition

What Are Gravity Wells?

How do we escape the gravitational pull of a planet?

✦ · 5 min read · May 2, 2022

99 4



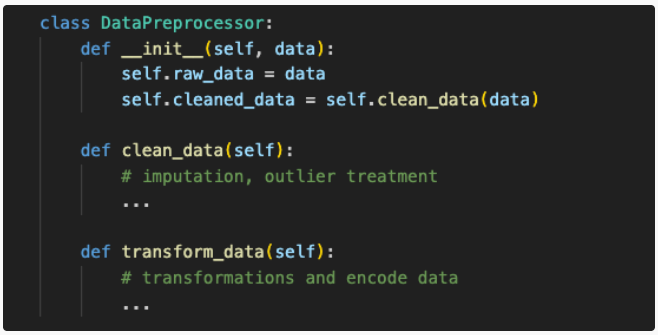
Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

✦ · 15 min read · Sep 7

564 7



```
class DataPreprocessor:
    def __init__(self, data):
        self.raw_data = data
        self.cleaned_data = self.clean_data(data)

    def clean_data(self):
        # imputation, outlier treatment
        ...

    def transform_data(self):
        # transformations and encode data
        ...
```

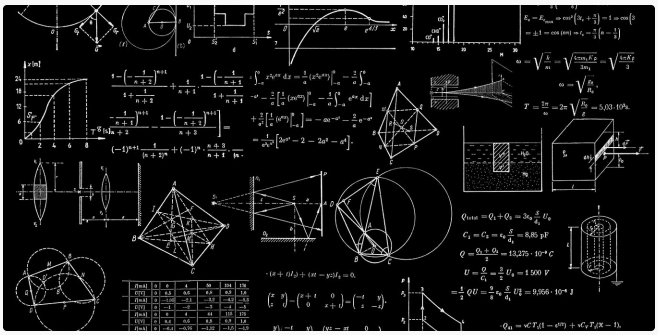
Molly Ruby in Towards Data Science

Object Oriented Data Science: Refactoring Code

Elevating machine learning models and data science products with efficient code and...

✦ · 7 min read · Aug 24

243 4



Zack Fizell in ILLUMINATION

Are you Smart Enough to Solve this 300-Year-Old Problem?

A look into the problem that gave birth to modern optimal control theory

✦ · 5 min read · Aug 1, 2022

294 1

See all from Zack Fizell | See all from Towards Data Science

Recommended from Medium

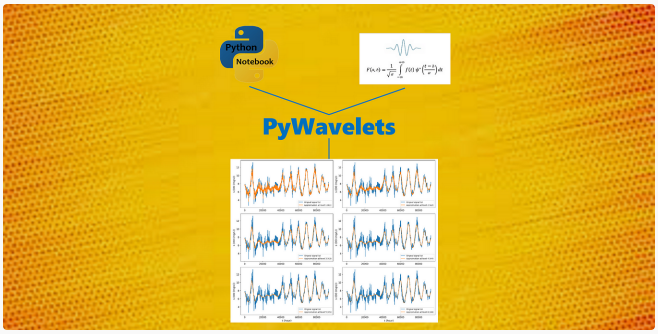
alienmummy

Alien Mummies, Nazca Desert, Peru

Interview 2023 with Prof. Zuniga Aviles Roger University of San Luis Gonzaga de Ica, Peru

8 min read · Jul 13

10



Dr. Shouke Wei

Multilevel Discrete Wavelet Transform and Noise Reduction o...

How to make multilevel Discrete Wavelet Transform and noise reduction of 1D Time...

✦ · 8 min read · Mar 21

110 3

Lists

Coding & Development

11 stories · 181 saves

It's never too late or early to start something

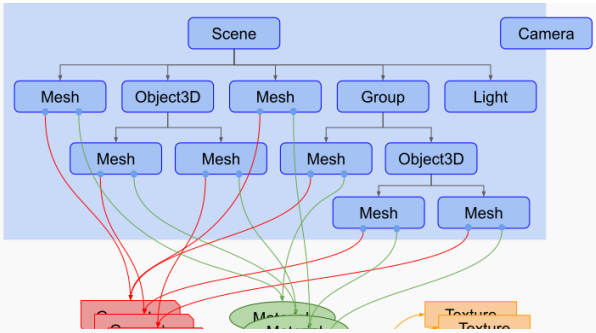
15 stories · 128 saves

General Coding Knowledge

20 stories · 353 saves

ChatGPT

21 stories · 156 saves



Rabin Lamichhane

Three.js

Three.js is a popular open-source JavaScript library used for creating and displaying 3D...

4 min read · Apr 27

15

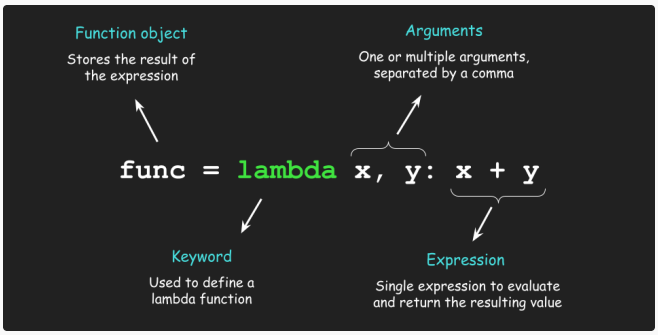
Cris Velasquez, MSc.

Riding the Waves of Stock Prices with Wavelet Transform Signals in...

Towards Unlocking Market Signals for Clearer Trading Insights

· 9 min read · Sep 11

240 3



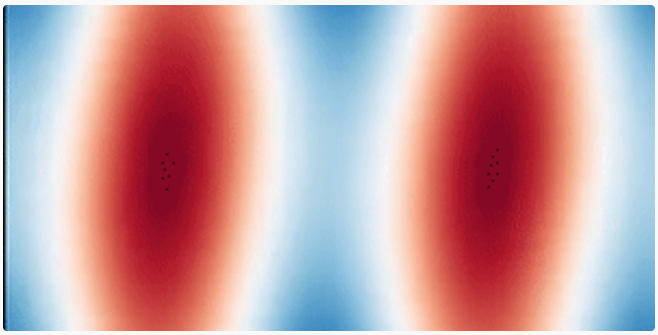
Ernest Asena

Lambda Functions in Python: Unleashing the Magic of Concise...

Welcome to a magical journey through the world of Python lambda functions, where...

4 min read · Aug 23

8



Philip Mocz in Level Up Coding

Create Your Own Navier-Stokes Spectral Method Fluid Simulation...

For today's recreational coding exercise, we solve the Navier-Stokes equations for an...

· 7 min read · Aug 3

634 3

See more recommendations