TypeError TypeError ✕ •••

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


# Constants

#G = 6.67408e-11*(365.25*24*3600)**2/((1.496e8*1000)**3) # Gravitational c
G = 6.67430e-11
m1 = 5.9722e24   # Mass of Earth in kg
m2 =  1.989e30    # Mass of Sun in kg
omega = 1 # angular frequency of earth in yrs^-1
#ro = 1.48520121/(1.496e3)# distance in AU
ro = 1.4963e11
e = 0.0167
thetao = 0
mu = m1*m2/(m1+m2)


G
```

```
1.985201349180229e-29
```

Comienza a programar o generar con IA.

```
def accel(r):
  return -G*(m2)/r**2
#r = [ro, ro+omega*ro*1e-5 + 0.5*accel(ro)*1e-5**2]
#theta = [0, 0 + ro**2*omega*1e-5]
def stormer_verlet(r0, pr0, theta0, L0, n, h, to = 0):
  r = [r0]
  theta = [theta0]
  vt = [pr0]
  L = [L0]
  T = [0]
  w = omega
```

```
  for i in range(n):
    r_new = r[-1] + vt[-1]*h + 0.5*accel(r[-1])*h**2
    theta_new = theta[-1] + w*r[-1]**2*h
    vr_new = vt[-1] + 0.5*(accel(r[-1]) + accel(r_new))*h
    theta.append(theta_new)
    r.append(r_new)
    vt.append(vr_new)
    T.append(T[-1]+h)
    L.append(L0)
  return r, theta, vt, L, T
```

Comienza a programar o generar con IA.

```
1.1772582444345737e+20
```

Project 1

Two-Body Problem Simulation

The two body problem is a second order differential equation system governed by the following Hamiltonian wrt the center of mass of two bodies of mass m_a, m_b and positions $\overrightarrow{r_a}, \overrightarrow{r_b}$:

$$\hat{H} = \frac{\vec{p}_a^{\,2}}{2m_a} + \frac{\vec{p}_b^{\,2}}{2m_b} - \frac{Gm_am_b}{|\vec{r}_a - \vec{r}_b|}\hat{r}$$

In its center of mass frame, given $\vec{r} = \vec{r}_a - \vec{r}_b$ and $\mu = \frac{m_a m_b}{m_a+m_b}$ we get the equations of motion:

$$\dot{\vec{r}} = -\frac{\vec{p}}{\mu}$$

$$\dot{\vec{p}} = \frac{G\mu(m_a + m_b)}{\ldots}\hat{r}$$

Tú

Explica el error:

```
TypeError: 'float' object is not callable
```

Gemini

Hypothesis

The traceback shows a `TypeError: 'float' object is not callable` in the line `return -G*m2(m1+m2)/r**3` within the `accel` function. This suggests that `m2` is being treated as a function when it is likely intended to be a variable used for multiplication. This could be happening if `m2` is assigned a float value elsewhere in the code and is being called with parentheses, as if it were a function, in this line.

The code uses `m2(m1+m2)` which looks like calling a function `m2`. If `m2` is a float variable containing the value of the mass of object 2, the expression should be multiplication instead: `m2 * (m1 + m2)`.

Suggested Changes

```
def accel(r):
```

$$p = -\frac{\quad}{r^3} r$$

Given the consideration that $\vec{L} = \vec{r} \times \mu\vec{v}$ is conserved one can obtain that the movement of the system is constrained.

Also, the total energy $E = \frac{\vec{p}^2}{2\mu} - \frac{G\mu(m_a+m_b)}{r^3}$ is expected to remain constant.

Obtaining all the possible equations of motion we end up with:

$$\frac{dp_r}{dt} = -\frac{G\mu(m_a+m_b)}{r^3}$$
$$\frac{dp_\theta}{dt} = 0$$
$$\frac{dr}{dt} = \frac{p_r}{\mu}$$
$$\frac{d\theta}{dt} = \omega r^2$$

Proposition of solution

I'll be using the **Leapfrog Stormer-Verlet** method of integration for this system, since, because of the analysis done on "DynamicsTest", I know this method is best suited for consistent energy conservation simulations starting on low time signatures. So for long time signatures there will be a fairly stable simulation energy-wise. The leapfrog variation lends smoother, less chaotic simulations, since it automates corrections of energy (i.e. it "detects" when energy fluctuates, given a good enought timestep, and rectifies that).

Constants

The constants will be presented as follows:

```
def accel(r):
  """Calculates acceleration."""
  # Changed m2(m1+m2) to m2 * (m1 + m2) fo
  return -G * m2 * (m1 + m2) / r**3
```

Use code with caution

Califica esta respuesta

The constants will be presented as follows:

- $M_o = 3.33 \times 10^6 M_E$ where M_o is the mass of the Sun and M_e is the mass of the Earth
- $r_o = 1.48520121 \times 10^8\ m$ is the current distance from the Sun to Earth and will be used as the initial value for r (as of October 30, 2024) [2]
- $e = 0.0167$ [1]
- $G = 6.67 \times 10^{11} m^3 kg^{-1} s^{-2}$ [3]
- $\omega_E = \frac{1}{365.25}\ d^{-1}$ is the orbital frequency of Earth in 1/days.
- $v_o(E) = \frac{1.48520121 \times 10^8}{365.25}\ m/s$ is the velocity of Earth as of October 30, 2024, and will be used as the initial value to get p_o.
- $\theta_o(E) = 0°$ for convenience.

```
2.975552051337216e24/(1.495979e11)**3
```

```
8.887734251461273e-10
```

Attempting the solution on different step sizes

h = 1e-10

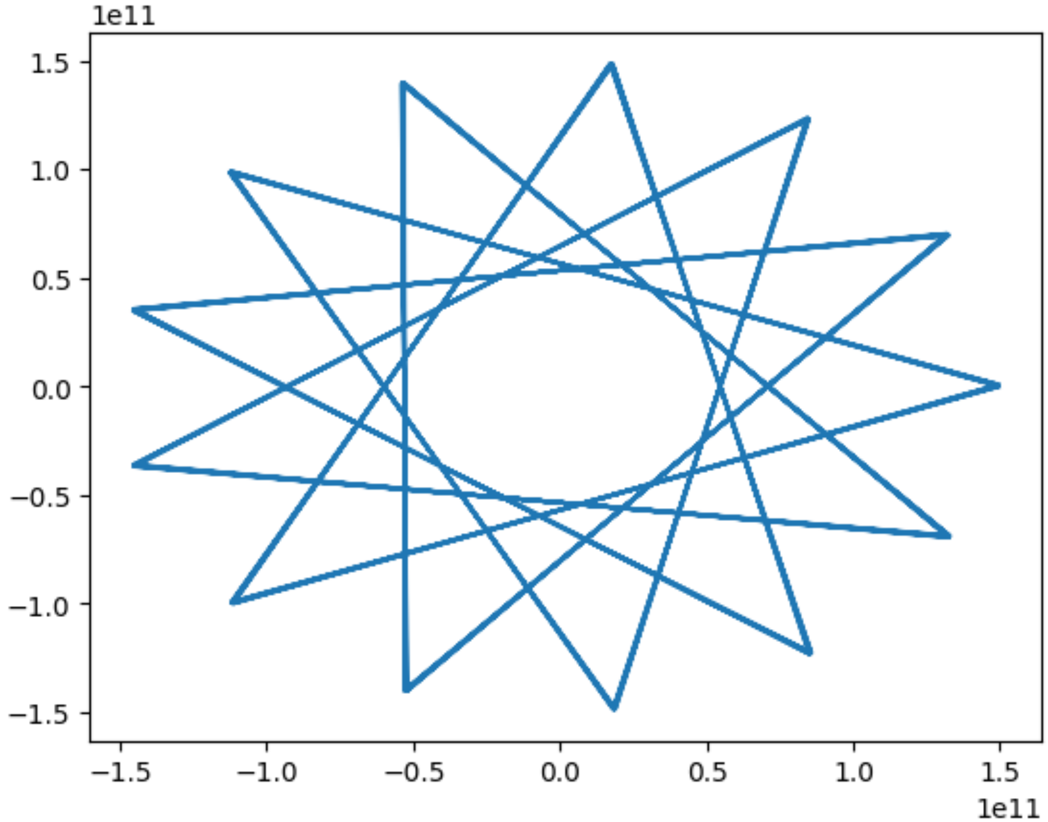
```
Comienza a programar o generar con IA.
```

```
vi = np.sqrt(G*m2/ro)
```

```
Comienza a programar o generar con IA.
```

```
# Rotation at 71n
sol = stormer_verlet(ro, ro/(24*3600*365.25), 0, omega, n=100, h=1e-10)
```

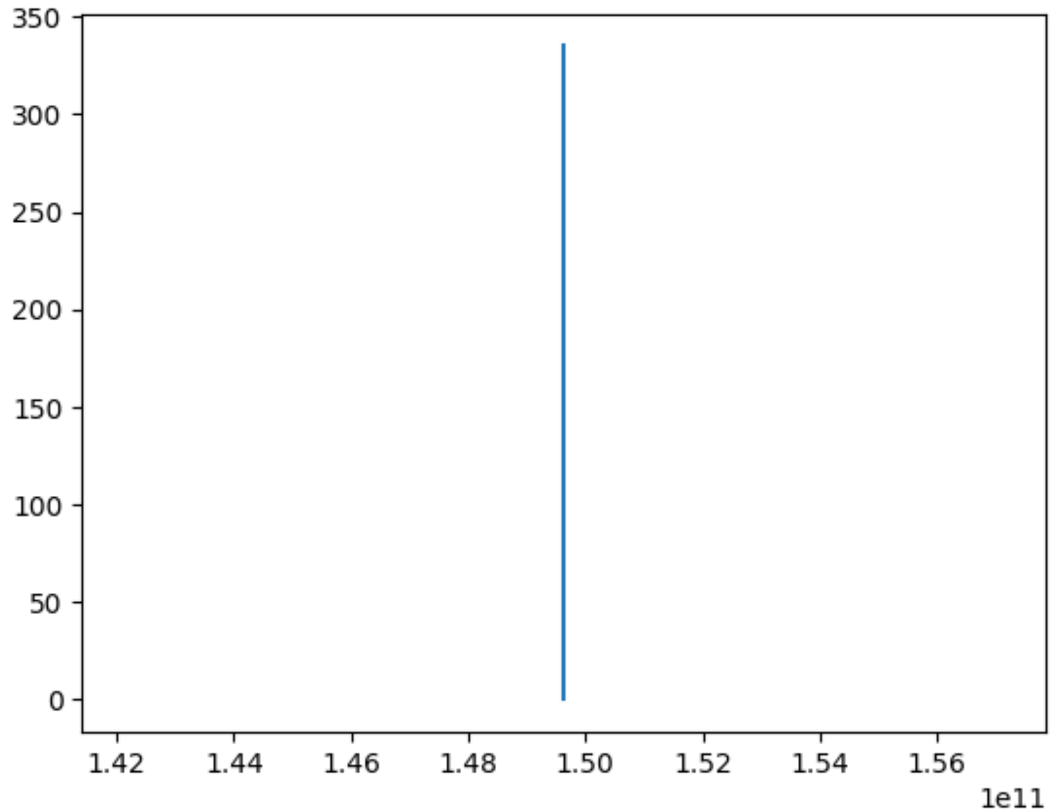
```
sol = stormer_verlet(ro, ro/(24*3600*365.25), 0, omega, n=100, h=1e-10)
x = [sol[0][i]*np.cos(sol[1][i]) for i in range(len(sol[0]))]
y = [sol[0][i]*np.sin(sol[1][i]) for i in range(len(sol[0]))]
plt.plot(x,y)
plt.show()
```



h = 1e-33

```
sol = stormer_verlet(ro, ro/(24*3600*365.25), 0, omega, n=100, h=1e-33)
x = [sol[0][i]*np.cos(sol[1][i]) for i in range(len(sol[0]))]
```

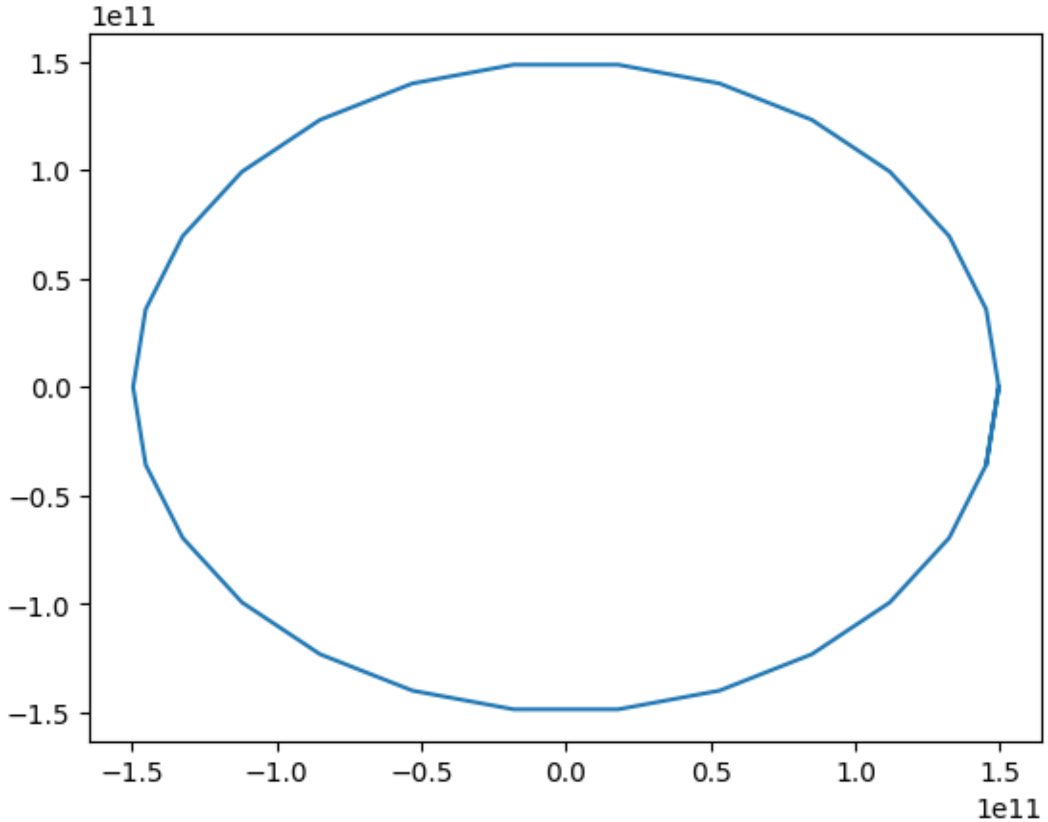
```
y = [sol[0][i]*np.sin(sol[1][i]) for i in range(len(sol[0]))]
plt.plot(x,y)
plt.show()
```



h = 1e-11

```
sol = stormer_verlet(ro, ro/(24*3600*365.25), 0, omega, n=27, h=1e-11)
x = [sol[0][i]*np.cos(sol[1][i]) for i in range(len(sol[0]))]
y = [sol[0][i]*np.sin(sol[1][i]) for i in range(len(sol[0]))]
plt.plot(x,y)
```

```
plt.plot(x,y)
plt.show()
```



Seems to work fine on h = 1e-33 I'll settle on that. I discovered it closes at n=27 as well.

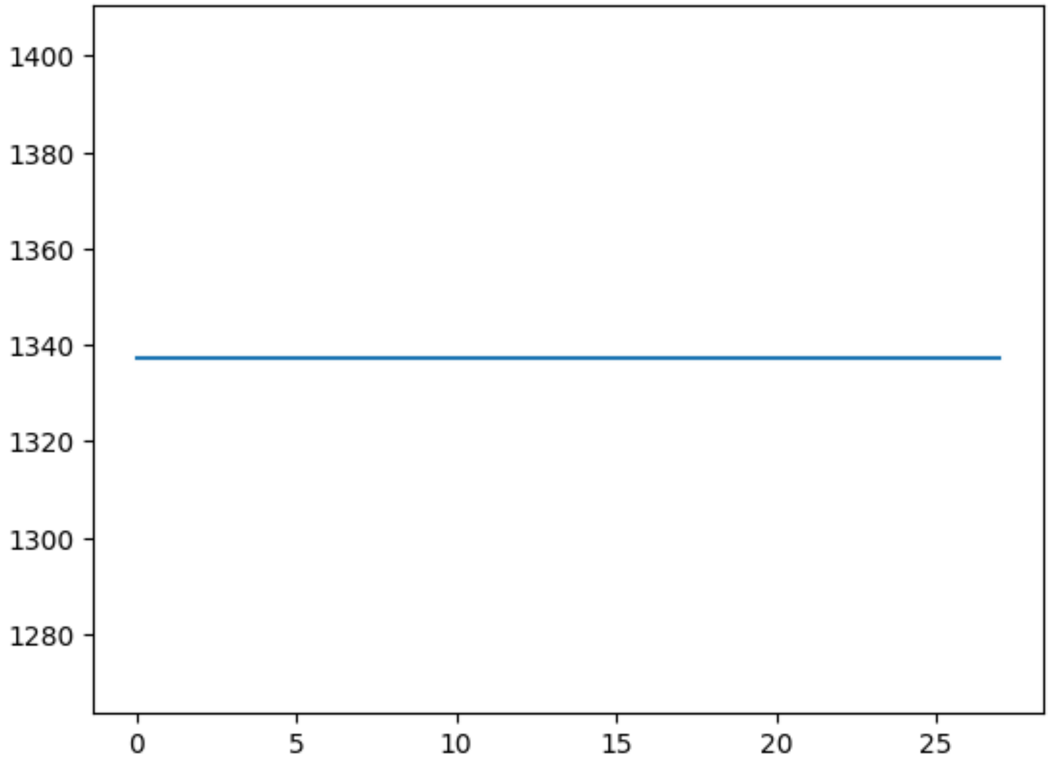
Checking on conserved quantitites

Using the definition of $L = r^2 \mu \omega$ will obtain L_i for each i in the mesh of the solution

solution.

```
L = [sol[0][i]**2*mu*omega/1e44 for i in range(len(sol[0]))]
plt.plot(L)
plt.plot(np.mean(L))
```

```
[<matplotlib.lines.Line2D at 0x78dce602f010>]
```



Calculating the mean of the array obtained by $\{L_i\}$ (adjusting for scale)

```
np.mean(L)
```

```
1337.120019086004
```

And it's standard deviation

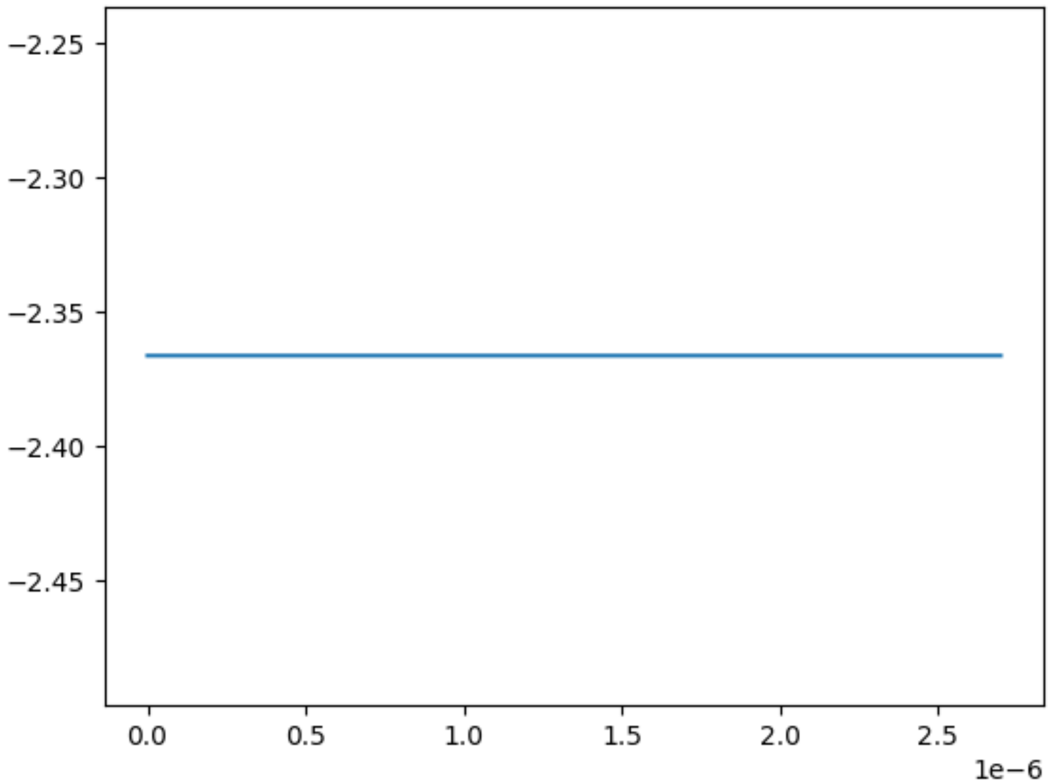
```
np.std(L)
```

```
6.661338147750939e-16
```

We calculate total energy

```
E = [(sol[2][i]**2/(2*mu)-(G*m1*m2)/(sol[0][i]**3))*1e-11 for i in range(ler
plt.plot(sol[-1], E)
```

```
[<matplotlib.lines.Line2D at 0x78dcdf8ab3d0>]
```



There seems to be a very little net total energy with respect to time. We obtain its mean

```
np.mean(E)/(np.mean(sol[2])**2/(2*mu)-(G*m1*m2)/np.mean(sol[0])**3)
```

```
0.9999999999999999
```

Considering the scales we use ($\approx 10^{10}$) the energy gained/lost is extremely small, but detectable. We get that the difference between the net energy arising from this system and the result of adding the mean kinetic energy and the mean potential energy is ≈ 0.00000000000001%. This is non trivial, and signals the fact that the change in energy is 1×10^{14}% of the total, so the energy barely changes.

Proving Kepler's Law

We must obtain the time frame $T_{a \to b}$, defined as a subset of the solution. And see if it's equal to its equivalent area, approximated using $A_c = \frac{\theta r^2}{2}$. For convenience, we shall start at $T = 0, \theta = 0, r = r_o$

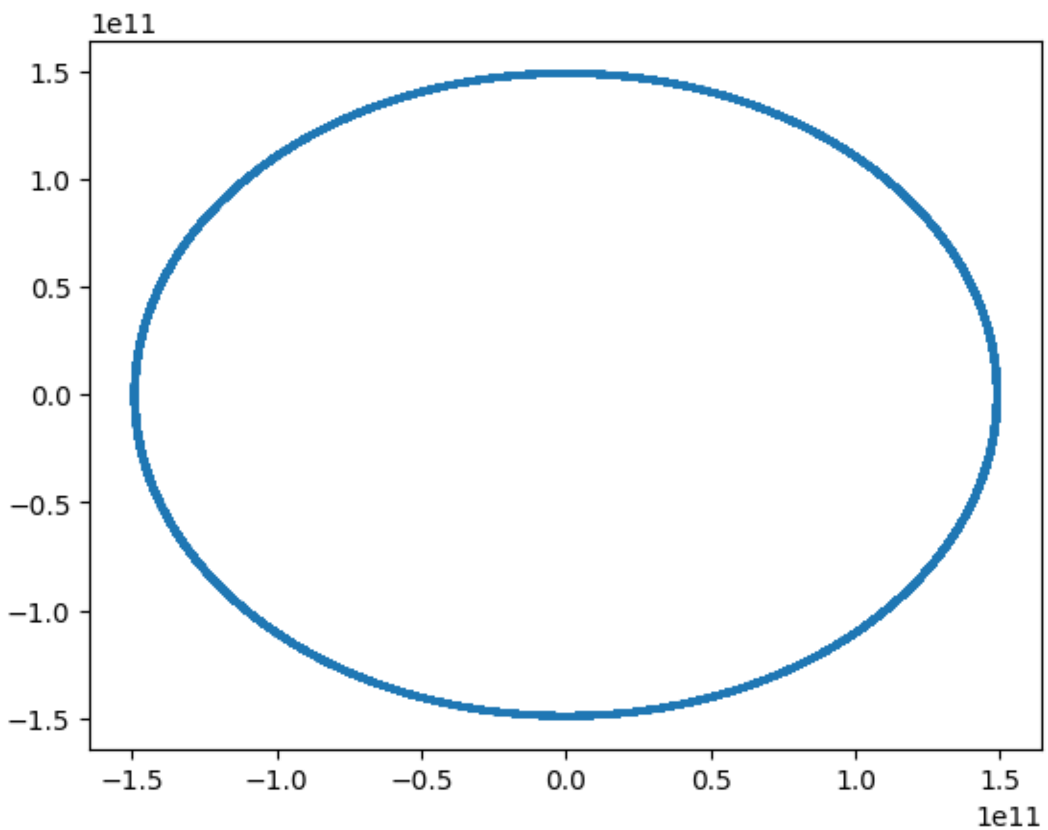
```
print((sol[1][5]*(sol[0][5])**2)/(2*(sol[-1][5])))
```

```
2.5063672556347083e+44
```

We get close to L. This is good news, since we already saw L is conserved, so second Law seems to work.

We now get the 10,000 cicles.

```
sol = stormer_verlet(ro, -G*m2/(ro**3), 0, omega*ro**2, n=270000, h=1e-11)
x = [sol[0][i]*np.cos(sol[1][i]) for i in range(len(sol[0]))]
y = [sol[0][i]*np.sin(sol[1][i]) for i in range(len(sol[0]))]
plt.plot(x,y)
plt.show()
```



Leapfrog gives good results on these big timescales because these systems depend on the positions of the gravitating masses, not so much on the velocities. So v is mainly used to correct energies for the accelerations present for r. Also, since $h < 2/\omega$we get good lower boundaries for the timestep, and it's easier for us to improve it without recurring to guesswork. I know this works because the only real qualm with this problem is the scale. These are big magnitudes so one needs to buffer them in some way, by chosing a nice timestep. Once that is taken care of, we can quickly obtain accelerations for a given r and a future r, so that we don't have to trust the system will get energy out of nowhere.

The timestep, I've checked that if it's not small enough, the simulation will very easily become chaotic and start getting weird results because the values for r, theta, v, etc will grow very fast.

Sources

1. NASA. (2024). *Earth fact sheet*. NASA. https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html
2. SunMoonTrack: Real Time Sun & Moon Tracking. *suncalc.org - real sun & moon tracking*. (2024). https://www.suncalc.org/sunmoontrack/
3. The NIST Reference on Constants, Units and Uncertainty. *Codata value: Newtonian constant of gravitation*. (2024). https://physics.nist.gov/cgi-bin/cuu/Value?bg