Workbook 3 Hand-in

Nicolas THIERRY

May 4, 2024

Contents

1 Ex 11: The stability of spin motion for an irregular asteroid

To study the stability of spin motion for an irregular shape, we need a mass model to represent it. In the case of this asteroid, we are going to use the following:

z = 1		z = 0		z = -1	
x	y	x	y	x	y
-1	4	-3	3	-2	2
0	3	-3	1	-1	5
0	-5	-2	4	-1	3
1	4	-2	2	-1	1
1	-2	-2	0	-1	-5
		-2	-4	0	6
		-1	5	0	4
		-1	3	0	2
		-1	1	0	0
		-1	-1	0	2
		-1	-3	0	4

z = 1		z = 0		z = -1	
		-1	-5	0	6
		-1	-7	1	5
		0	6	1	3
		0	4	1	1
		0	2	1	-3
		0	0	1	-5
		0	-2	2	4
		0	-4		
		0	-6		
		1	5		
		1	3		
		1	1		
		1	-1		
		1	-3		
		1	-5		
		2	4		
		2	2		
		2	0		
		2	-2		
		3	1		

[1]:
```
import numpy as np

SCALE = 300
masslumps=np.array([[-1,4,1],
[0,3,1],
[0,-5,1],
[1,4,1],
[1,-2,1],
[-3,3,0],
[-3,1,0],
[-2,4,0],
[-2,2,0],
[-2,0,0],
[-2,-4,0],
[-1,5,0],
[-1,3,0],
[-1,1,0],
[-1,-1,0],
[-1,-3,0],
[-1,-5,0],
[-1,-7,0],
[0,6,0],
[0,4,0],
[0,2,0],
```

```
[0,0,0],
[0,-2,0],
[0,-4,0],
[0,-6,0],
[1,5,0],
[1,3,0],
[1,1,0],
[1,-1,0],
[1,-3,0],
[1,-5,0],
[2,4,0],
[2,2,0],
[2,0,0],
[2,-2,0],
[3,1,0],
[-2,2,-1],
[-1,5,-1],
[-1,3,-1],
[-1,1,-1],
[-1,-5,-1],
[0,6,-1],
[0,4,-1],
[0,2,-1],
[0,0,-1],
[0,2,-1],
[0,4,-1],
[0,6,-1],
[1,5,-1],
[1,3,-1],
[1,1,-1],
[1,-3,-1],
[1,-5,-1],
[2,4,-1]])

unit_com = np.mean(masslumps, axis=0)
center_of_mass = unit_com*SCALE
points_remap = masslumps*SCALE - center_of_mass
list(unit_com) # Center of mass coordinates in the form [x, y, z] in the unit
 ↪cell
```

[1]: [-0.037037037037037035, 0.7962962962962963, -0.24074074074074073]

[2]:
```
list(center_of_mass) # Center of mass coordinates in the form [x, y, z] in
 ↪physical space
```

[2]: [-11.11111111111111, 238.88888888888889, -72.22222222222221]

To be sure that the new coordinate system is centered around the center of mass, we can calculate its center of mass and expect a point centered at the origin. The result will not be perfect due to computer float number precision.

[3]:
```
list(np.mean(points_remap, axis=0)) # Should be [0, 0, 0]
```

[3]:
```
[1.0947621410229691e-13, 6.736997790910579e-14, -3.0527021240063567e-14]
```

Now, knowing that the total mass of the asteroid is $5.10^{13}kg$, it is possible to deduce the mass of each lump.

[4]:
```
total_mass = 5e13 # Total mass of the system (in kg)
mass_per_point = total_mass / len(masslumps)
mass_per_point # Mass of each point (in kg)
```

[4]:
```
925925925925.9259
```

Now, it is possible to calculate the inertia matrix as well as the three principal moments of inertia with the formula:

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix} = \begin{pmatrix} \sum m(y^2+z^2) & -\sum mxy & -\sum mxz \\ -\sum mxy & \sum m(x^2+z^2) & -\sum myz \\ -\sum mxz & -\sum myz & \sum m(x^2+y^2) \end{pmatrix}$$

For principal moments of inertia, it is the Eigen values of this matrix.

[5]:
```
def inertia_moment(mass, x, y):
    return mass * (x**2 + y**2)
def inertia_product(mass, x, y):
    return mass * x * y

inertia = np.array([[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]], dtype=float)

m = mass_per_point
for [x, y, z] in points_remap:
    inertia += np.array([[inertia_moment(m, y, z), -inertia_product(m, x, y),␣
 ↪-inertia_product(m, x, z)],
                         [-inertia_product(m, x, y), inertia_moment(m, x, z),␣
 ↪-inertia_product(m, y, z)],
                         [-inertia_product(m, x, z), -inertia_product(m, y, z),␣
 ↪inertia_moment(m, x, y)]])

inertia
```

[5]:
```
array([[ 5.73858025e+19, -2.16049383e+17,  4.01234568e+16],
       [-2.16049383e+17,  9.31635802e+18,  1.72067901e+18],
       [ 4.01234568e+16,  1.72067901e+18,  6.33904321e+19]])
```

[6]:
```
w,_=np.linalg.eig(inertia)
print('E-value:', w) # Eigenvalues of the inertia tensor (in kg m^2) in the
 ↪form [Ixx, Iyy, Izz]
```

```
E-value: [9.26067960e+18 5.73866005e+19 6.34453125e+19]
```

1.1 Spin kinetic energy

Knowing that the spin kinetic energy is given by:

$$E = \sum_{k=1}^{3} \frac{1}{2} I_k \Omega_k^2$$

By differentiating the expression with respect to time:

$$\frac{dE}{dt} = \sum_{k=1}^{3} I_k \Omega_k \frac{d\Omega_k}{dt}$$

Or, according to Euler's equations:

$$\frac{d\Omega_x}{dt} + \frac{(I_z - I_y)\Omega_y\Omega_z}{I_x} = \frac{Q_x}{I_x}$$

$$\frac{d\Omega_y}{dt} + \frac{(I_x - I_z)\Omega_z\Omega_x}{I_y} = \frac{Q_y}{I_y}$$

$$\frac{d\Omega_z}{dt} + \frac{(I_y - I_x)\Omega_x\Omega_y}{I_z} = \frac{Q_z}{I_z}$$

Without external torque $Q = 0$.

$$\frac{d\Omega_x}{dt} = -\frac{(I_z - I_y)\Omega_y\Omega_z}{I_x}$$

$$\frac{d\Omega_y}{dt} = -\frac{(I_x - I_z)\Omega_z\Omega_x}{I_y}$$

$$\frac{d\Omega_z}{dt} = -\frac{(I_y - I_x)\Omega_x\Omega_y}{I_z}$$

Thus, combining this into the differentiation of the first equation:

$$\frac{dE}{dt} = \left(I_y - I_z + I_z - I_x + I_x - I_y\right) \prod_{k=1}^{3} \Omega_k$$

This means that $\frac{dE}{dt} = 0$. Therefore, E is constant if any external torque is apply on the system.

2 Ex 12: The Spin Ellipsoid

Knowing that the spin kinetic energy is:

$$E = \sum_{k=1}^{3} \frac{1}{2} I_k \Omega_k^2$$

It can be written as:

$$1 = \sum_{k=1}^{3} \frac{1}{2E} \frac{1}{I_k^{-1}} \Omega_k^2$$

$$1 = \sum_{k=1}^{3} \frac{\Omega_k^2}{2\frac{E}{I_k}}$$

This is an ellipsoid equation as $(\frac{x}{a})^2 + (\frac{y}{b})^2 + (\frac{z}{c})^2 = 1$. If the ellipse coefficient a, b and c are all equals to 1, therefore, the ellipse equation transform to $x^2 + y^2 + z^2 = 1$, which is the equation for a sphere of radius 1. In our case it would be:$\sqrt{2\frac{E}{I_x}} = \sqrt{2\frac{E}{I_y}} = \sqrt{2\frac{E}{I_z}} = 1$. Which can also be written as $2\frac{E}{I_x} = 2\frac{E}{I_y} = 2\frac{E}{I_z} = 1$

[7]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint

SCALE = 300
masslumps=np.array([[-1,4,1],
[0,3,1],
[0,-5,1],
[1,4,1],
[1,-2,1],
[-3,3,0],
[-3,1,0],
[-2,4,0],
[-2,2,0],
[-2,0,0],
[-2,-4,0],
[-1,5,0],
[-1,3,0],
[-1,1,0],
[-1,-1,0],
[-1,-3,0],
[-1,-5,0],
[-1,-7,0],
[0,6,0],
[0,4,0],
[0,2,0],
[0,0,0],
[0,-2,0],
[0,-4,0],
```

```python
[0,-6,0],
[1,5,0],
[1,3,0],
[1,1,0],
[1,-1,0],
[1,-3,0],
[1,-5,0],
[2,4,0],
[2,2,0],
[2,0,0],
[2,-2,0],
[3,1,0],
[-2,2,-1],
[-1,5,-1],
[-1,3,-1],
[-1,1,-1],
[-1,-5,-1],
[0,6,-1],
[0,4,-1],
[0,2,-1],
[0,0,-1],
[0,2,-1],
[0,4,-1],
[0,6,-1],
[1,5,-1],
[1,3,-1],
[1,1,-1],
[1,-3,-1],
[1,-5,-1],
[2,4,-1]])

center_of_mass = np.mean(masslumps, axis=0)*SCALE
points_remap = masslumps*SCALE - center_of_mass
total_mass = 5e13 # Total mass of the system (in kg)
mass_per_point = total_mass / len(masslumps)

def inertia_moment(mass, x, y):
    return mass * (x**2 + y**2)
def inertia_product(mass, x, y):
    return mass * x * y

inertia = np.array([[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]], dtype=float)

m = mass_per_point
for [x, y, z] in points_remap:
```

```
    inertia += np.array([[inertia_moment(m, y, z), -inertia_product(m, x, y),␣
↪-inertia_product(m, x, z)],
                         [-inertia_product(m, x, y), inertia_moment(m, x, z),␣
↪-inertia_product(m, y, z)],
                         [-inertia_product(m, x, z), -inertia_product(m, y, z),␣
↪inertia_moment(m, x, y)]])

w,_=np.linalg.eig(inertia)
w
```

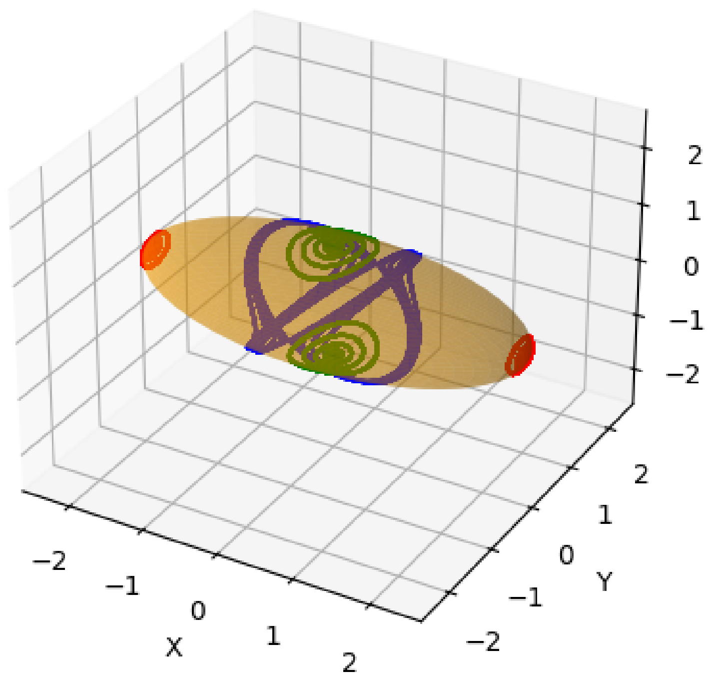
[7]:
```
array([9.26067960e+18, 5.73866005e+19, 6.34453125e+19])
```

To draw the spin energy ellipsoid representing the possible values of rotation velocities for a specific energy, we need to fix an energy. Then, to simulate trajectories that also lives on that shape, it will be required to evaluate that ellipsis to have the starting coordinates for the solver, in that way, it ensures that the trajectory will correspond to the right energy.

[8]:
```
target_energy = 3e19
principal_intertia = np.array(w)

# Define the system of differential equations
def eulereqns(t, angvel):
    return (-np.subtract(*principal_intertia[2:0:-1]) * np.multiply(*angvel[1:
↪]) / principal_intertia[0],
            -np.subtract(*principal_intertia[::2]) * np.multiply(*angvel[::-2])␣
↪/ principal_intertia[1],
            -np.subtract(*principal_intertia[1::-1]) * np.multiply(*angvel[:2])␣
↪/ principal_intertia[2])

# Define the coefficients of the ellipsoid
rx, ry, rz =np.sqrt(2*target_energy/principal_intertia)
# Define the 3 central angles of the ellipsoid (one for each axis) [x, y, z]
uv = [(0, np.pi/2, 'red'), (np.pi/2, np.pi/2, 'blue'), (0, 0, 'green'),
      (np.pi, np.pi/2, 'red'), (3*np.pi/2, np.pi/2, 'blue'), (0, np.pi,␣
↪'green')]
tfinal = 100
number_of_trajectories = 6

angvels = []
for u, v, color in uv:
    # Then we add a small variation to the central angles to see multiple␣
↪trajectories around that central point
    for var in np.linspace(0, np.pi/24, number_of_trajectories):
        v += var/2
        u += var/2
        x = rx * np.cos(u) * np.sin(v)
        y = ry * np.sin(u) * np.sin(v)
```

```python
        z = rz * np.cos(v)
        angvel0 = np.array([x, y, z])
        inenergy = 0.5 * np.sum(principal_intertia * angvel0 ** 2)
        t = np.linspace(0, tfinal, 10000)
        angvel = odeint(eulereqns, angvel0, t, tfirst=True)
        tend = len(angvel) - 1
        finenergy = 0.5 * np.sum(principal_intertia * angvel[tend] ** 2)
        accuracy = abs((finenergy - inenergy) / inenergy)
        angvels.append((accuracy, t, angvel, color))

print(f'Max Accuracy (worst) = {max([acc for acc, _, _, _ in angvels]): .4%}')

# Set of all spherical angles:
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

# Cartesian coordinates that correspond to the spherical angles:
# (this is the equation of an ellipsoid):
x = rx * np.outer(np.cos(u), np.sin(v))
y = ry * np.outer(np.sin(u), np.sin(v))
z = rz * np.outer(np.ones_like(u), np.cos(v))

# Plot:
fig = plt.figure(figsize=plt.figaspect(1))  # Square figure
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.plot_surface(x, y, z, color='orange', alpha=0.4)
for _, _, angvel, color in angvels:
    ax.plot(angvel[:, 0], angvel[:, 1], angvel[:, 2], color=color) # Draw the
 ↪trajectories

# Adjustment of the axes, so that they all have the same span:
max_radius = max(rx, ry, rz)
for axis in 'xyz':
    getattr(ax, 'set_{}lim'.format(axis))((-max_radius, max_radius))

plt.show()
```

```
Max Accuracy (worst) =  0.0005%
```



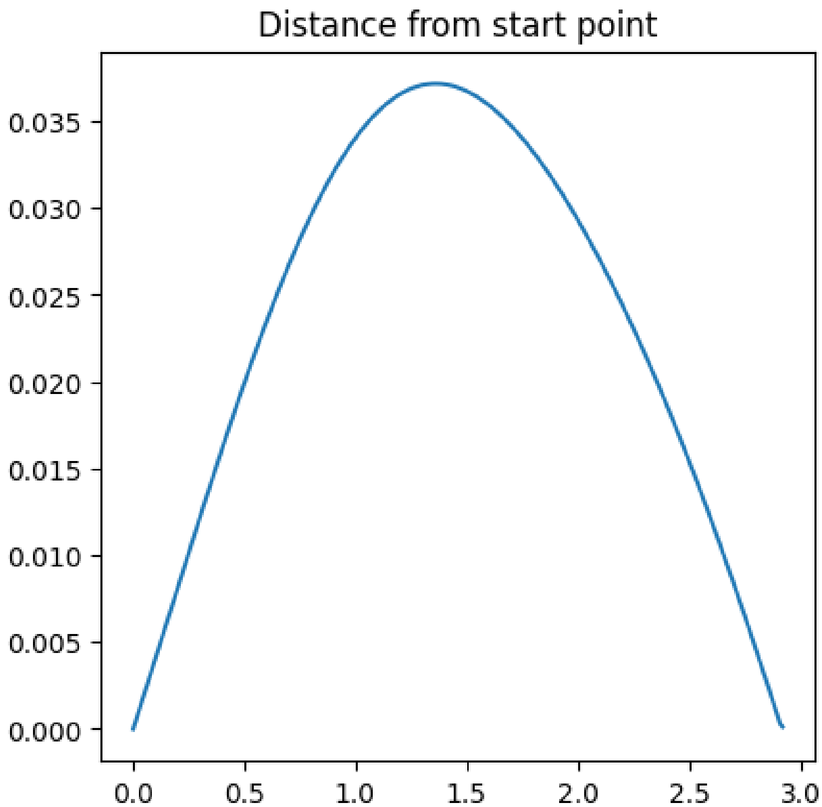
From the previous digram, it is clear that the most stable axis is the X axis and the less stable axis is the Y axis. Even really small rotation velocity around this axis will end up on high amplitude motion.

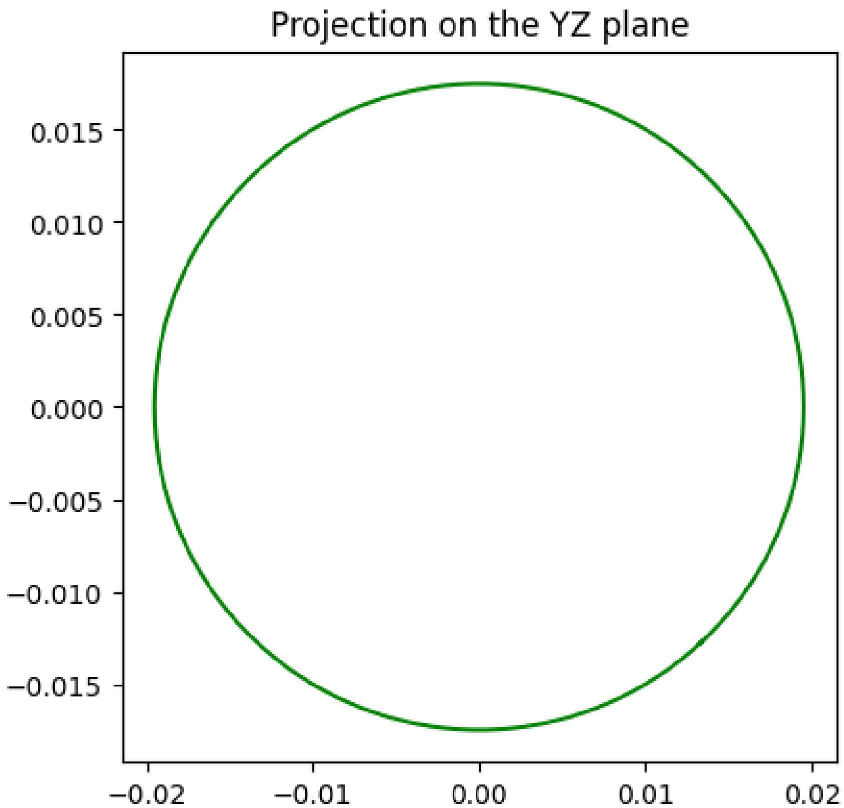
2.1 Period of motion

[9]:
```
from scipy.signal import argrelextrema
from numpy import less
_, t, av, _ = angvels[1]
distance_from_start = np.sqrt((av[:, 0]-av[0, 0])**2 + (av[:, 1]-av[0, 1])**2 +
    ↪(av[:, 2]-av[0, 2])**2)
idx = argrelextrema(distance_from_start, less)
first_loop = idx[0][0]
print(f"Period of motion: {t[first_loop]}") # Period of motion in seconds

plt.figure(figsize=plt.figaspect(1))  # Square figure
plt.title('Distance from start point')
plt.plot(t[:first_loop+1], distance_from_start[:first_loop+1])
plt.show()
plt.figure(figsize=plt.figaspect(1))  # Square figure
plt.title('Projection on the YZ plane')
plt.plot(av[:first_loop+1, 1], av[:first_loop+1, 2], color=color) # Draw the
    ↪trajectories
```

```
plt.show()
```

```
Period of motion: 2.9202920292029204
```





[10]:
```
# Farther from the center point
_, t, av, _ = angvels[5]
distance_from_start = np.sqrt((av[:, 0]-av[0, 0])**2 + (av[:, 1]-av[0, 1])**2 +
 ↪(av[:, 2]-av[0, 2])**2)
idx = argrelextrema(distance_from_start, less)
first_loop = idx[0][0]
print(f"Period of motion: {t[first_loop]}") # Period of motion in seconds
```

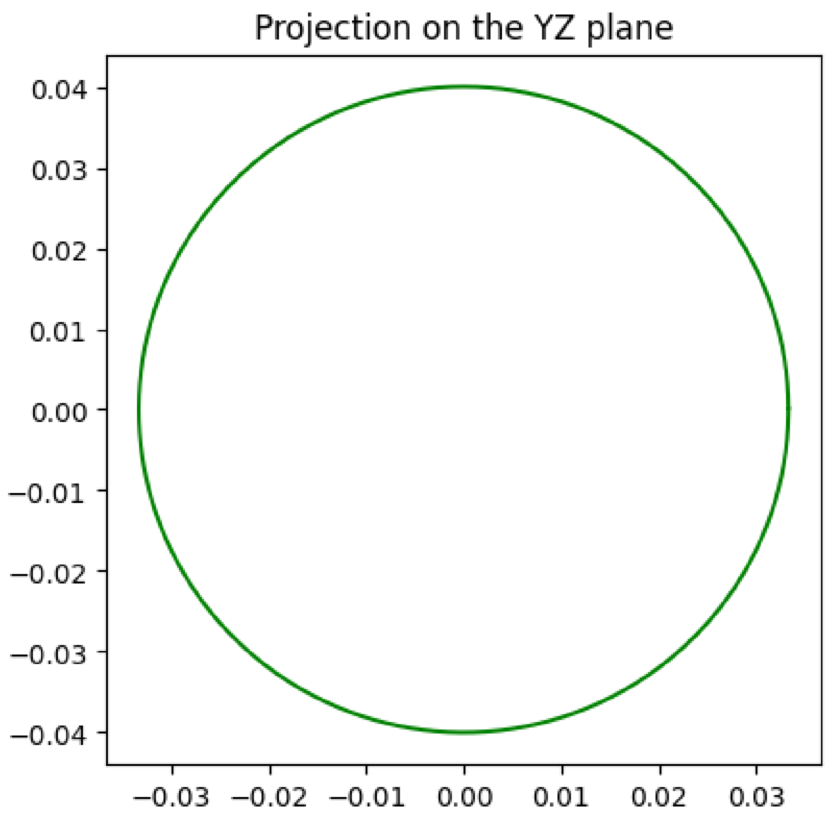
```
Period of motion: 3.0303030303030303
```

Here is the period of motion when relatively close to the center point of motion along the X axis. As we repeat this calculation farther from the axis, the period increase. This can be repeated for the other stable axis Z.

[11]:
```
_, t, av, _ = angvels[number_of_trajectories*2+1]
distance_from_start = np.sqrt((av[:, 0]-av[0, 0])**2 + (av[:, 1]-av[0, 1])**2 +
 ↪(av[:, 2]-av[0, 2])**2)
idx = argrelextrema(distance_from_start, less)
first_loop = idx[0][0]
print(f"Period of motion: {t[first_loop]}") # Period of motion in seconds
```

```
plt.figure(figsize=plt.figaspect(1))  # Square figure
plt.title('Projection on the YZ plane')
plt.plot(av[:first_loop+1, 0], av[:first_loop+1, 1], color=color) # Draw the␣
 ↪trajectories
plt.show()
```

```
Period of motion: 8.22082208220822
```



The period of motion for the Y axis is much greater than the X axis. This is due to the shape being more flat on that axis meaning that the distance is greater to travel for the oscillator.

2.2 Euler's equations period

Assuming the following Euler's equation:

$$\frac{d\Omega_x}{dt} + \frac{I_z - I_y}{I_x}\Omega_y\Omega_z = 0$$
$$\frac{d\Omega_y}{dt} + \frac{I_x - I_z}{I_y}\Omega_x\Omega_z = 0$$
$$\frac{d\Omega_z}{dt} + \frac{I_y - I_x}{I_z}\Omega_y\Omega_x = 0$$

It is possible to derive the period of motion from these.

2.2.1 Axis X

For a motion closer as possible from this axis. It can be assumed that $\frac{\Omega_x}{dt} = 0$. From that we differentiate the equations for Y and Z axis.

$$\begin{cases} \frac{d^2\Omega_y}{dt^2} + \frac{I_x - I_z}{I_y}\Omega_x \frac{d\Omega_z}{dt} = 0 \\ \frac{d^2\Omega_z}{dt^2} + \frac{I_y - I_x}{I_z}\Omega_x \frac{d\Omega_y}{dt} = 0 \end{cases}$$

$$\begin{cases} \frac{d^2\Omega_y}{dt^2} - \frac{I_x - I_z}{I_y}\Omega_x^2 \frac{I_y - I_x}{I_z}\Omega_y = 0 \\ \frac{d^2\Omega_z}{dt^2} - \frac{I_y - I_x}{I_z}\Omega_x^2 \frac{I_x - I_z}{I_y}\Omega_z = 0 \end{cases}$$

Now, this is clear that Ω_y and Ω_z follow a harmonic oscillator equation of type $\frac{d^2x}{dt^2} + \frac{k}{m}x = 0$. Then, the period can be defined as $T = \frac{2\pi}{\sqrt{\frac{k}{m}}}$.

In this case, $\frac{k}{m} = -\frac{(I_y - I_x)(I_x - I_z)}{I_y I_z}\Omega_x^2$. Hence,

$$T = \frac{2\pi}{\sqrt{-\frac{(I_y - I_x)(I_x - I_z)}{I_y I_z}\Omega_x^2}}$$

[12]:
```python
x0 = rx
km = -x0**2/(w[1]*w[2])*(w[0]-w[2])*(w[1]-w[0])
period = 2*np.pi/np.sqrt(km)
print(period) # Period of motion in seconds
```

```
2.916776136151734
```

As expected the period is really close to the period previously calculate with the simulated motion.

2.2.2 Axis Z

Now, we are doing the same calculation with the Z axis. In this case, it can be assumed that $\frac{\Omega_z}{dt} = 0$. From that, we differentiate the equations for X and Y axis.

$$\begin{cases} \frac{d^2\Omega_x}{dt^2} + \frac{I_z - I_y}{I_x}\Omega_z \frac{d\Omega_y}{dt} = 0 \\ \frac{d^2\Omega_y}{dt^2} + \frac{I_x - I_z}{I_y}\Omega_z \frac{d\Omega_x}{dt} = 0 \end{cases}$$

$$\begin{cases} \frac{d^2\Omega_x}{dt^2} - \frac{I_z - I_y}{I_x}\Omega_z^2 \frac{I_x - I_z}{I_y}\Omega_x = 0 \\ \frac{d^2\Omega_y}{dt^2} + \frac{I_x - I_z}{I_y}\Omega_z^2 \frac{I_z - I_y}{I_x}\Omega_y = 0 \end{cases}$$

Now, this is clear that Ω_y and Ω_z follow a harmonic oscillator equation of type $\frac{d^2x}{dt^2} + \frac{k}{m}x = 0$. Then, the period can be defined as $T = \frac{2\pi}{\sqrt{\frac{k}{m}}}$.

In this case, $\frac{k}{m} = -\frac{(I_x - I_z)(I_z - I_y)}{I_y I_x}\Omega_z^2$. Hence,

$$T = \frac{2\pi}{\sqrt{-\frac{(I_x - I_z)(I_z - I_y)}{I_y I_x}\Omega_z^2}}$$

[13]:
```python
z0 = rz
km = -z0**2/(w[1]*w[0])*(w[0]-w[2])*(w[2]-w[1])
period = 2*np.pi/np.sqrt(km)
print(period) # Period of motion in seconds
```

```
8.22058023422136
```

Finally, for the Z axis the results are also similar to what we should expect.

3 Ex 13: Kuiper belt object interception

We are focusing on intercepting $1994GV_9$. We are assuming that the asteroid is evolving on the same plane as the earth (the eccliptic) and orbiting the sun at 43.6AU in a circular motion.

[14]:
```python
import matplotlib.pyplot as plt

from space_base import GravBody, Probe
import numpy as np

# Define constants
G = 6.67e-11  # Gravitational constant
g0 = 9.80665
sun = GravBody(name="Sun", mass=1_988_500e24, radius=695_700e3)  # Sun as an
 ↪object with mass and radius

# Define conversion function
def AU_to_meters(UA):
    return UA * 1.496e11
def meters_to_AU(meters):
    return meters / 1.496e11
```

3.1 Long duration burn simulation

To be more realistic with our ion rocket, we are not going to simulate an impulsive burn but a continuous burn. For that, the differential function use to simulate the probe needs some changes.

Z and vz will be used to store the mass of the probe as space_base do not support 4D inputs.

[15]:
```python
mass_lost_rate = 10e-6 # kg/s
dry_mass = 300 # dry mass of the probe
def probeqns_rocket(_, posvelmass):
    Isp = 3400 # in seconds
```

```
    if posvelmass[2] <= dry_mass:
        posvelmass[5] = 0.0
    else:
        posvelmass[5] = -mass_lost_rate

    r = np.sqrt(posvelmass[0] ** 2 + posvelmass[1] ** 2)
    f = -G * sun.mass / r ** 3
    gravity_force = f * posvelmass[0:2]
    axy = gravity_force + posvelmass[3:5]*np.abs(posvelmass[5])*g0*Isp/
 ↪(posvelmass[2]*np.linalg.norm(posvelmass[3:5]))

    return posvelmass[3], posvelmass[4], posvelmass[5], axy[0], axy[1], 0.0
```

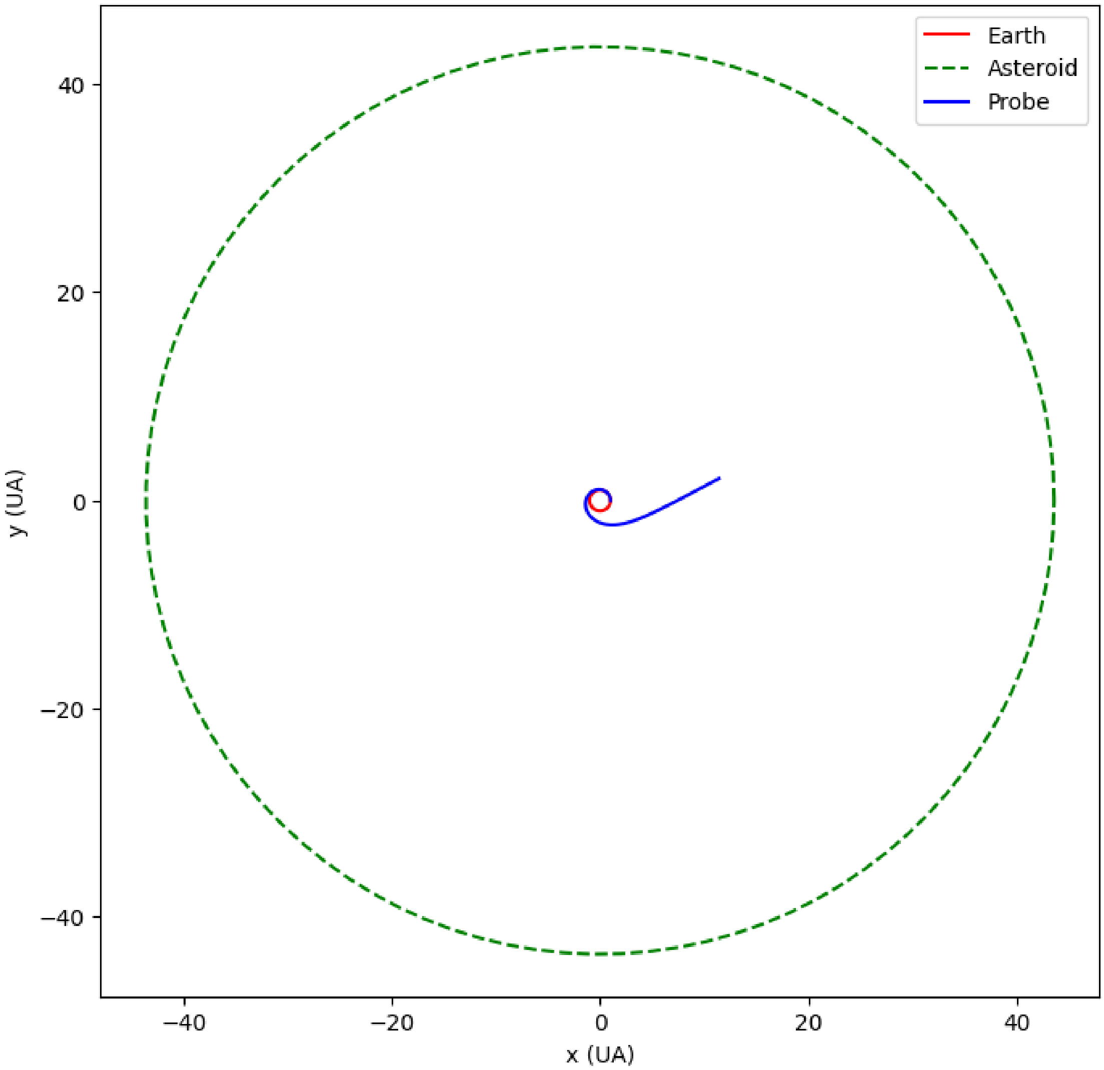
We then initialize our probe at Earth's orbit around the sun as the L4 point is on this orbit.

[16]:
```
fuel_mass = 1000 # kg
v0 = np.sqrt(G * sun.mass / AU_to_meters(1))  # initial speed
xymass0 = [AU_to_meters(1), 0, dry_mass + fuel_mass]  # start position
vxy0 = [0, v0]  # start vertical speed
tf = fuel_mass/mass_lost_rate  # Max burn time

probe = Probe(probeqns_rocket, tf, tf/3600, x0=xymass0[0], vx0=vxy0[0],
            y0=xymass0[1], vy0=vxy0[1], z0=xymass0[2], vz0=mass_lost_rate) #␣
 ↪probe as an object
t, posvel = probe.odesolve() # solve the differential equations
```

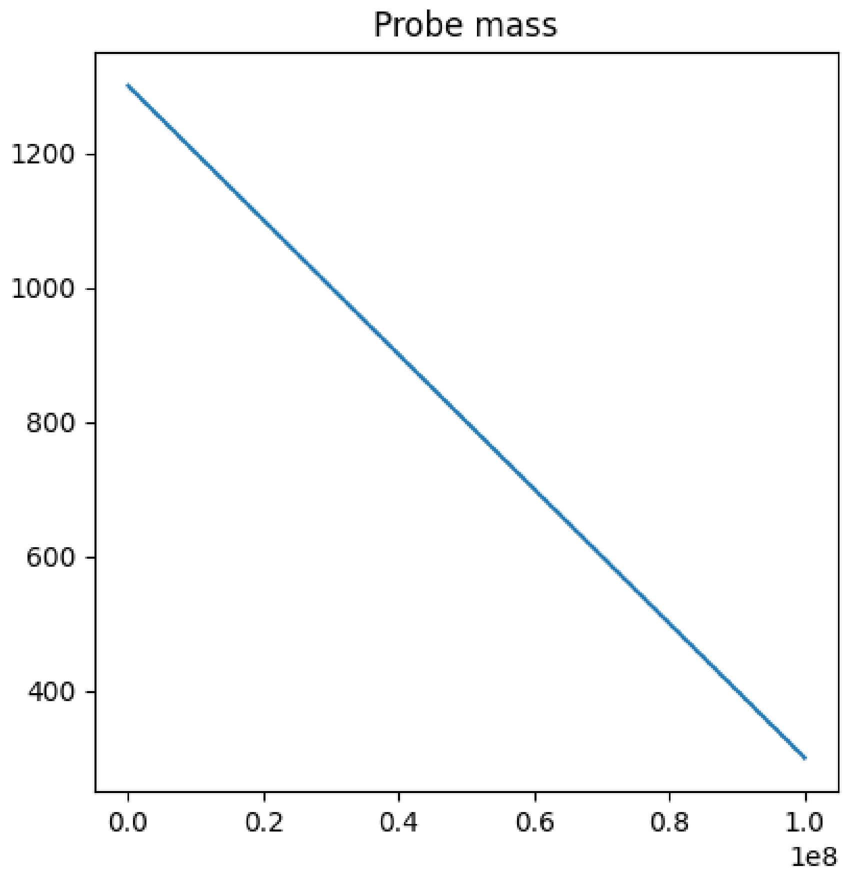
This is the probe trajectory at the end of the burn:

[17]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

# Plotting Earth's orbit
uang = np.linspace(0, 2 * np.pi, 100)
x = np.cos(uang)
y = np.sin(uang)
plt.plot(x, y, color='red', label='Earth')
# Plotting Asteroid's orbit
uang = np.linspace(0, 2 * np.pi, 100)
x = 43.6 * np.cos(uang)
y = 43.6 * np.sin(uang)
plt.plot(x, y, color='green', linestyle="--", label='Asteroid')

plt.plot(meters_to_AU(posvel[:, 0]), meters_to_AU(posvel[:, 1]), color='blue',␣
 ↪label="Probe") # plot the probe's orbit

plt.xlabel('x (UA)')
plt.ylabel('y (UA)')
plt.axis('equal')
plt.legend()
```

```
plt.show() # make plot appear
```




```
[18]: plt.figure(figsize=(5, 5))  # create figure, figsize can be changed as preferred
plt.title("Probe mass")
plt.plot(t, posvel[:, 2])
plt.show()
```



3.2 Fuel calculation

As the starting amount of fuel will determine the final orbit the probe will reach, it is important to tune this parameter so that our probe reach the desired orbit. For that, we will explore two possible solutions. In these two case, we will use a loop that will find the right starting fuel mass to reach the desire aphelion by interactively interpolate between previous know solution to try converging faster.

3.2.1 Minimum fuel mass

The first solution could be to try making the smallest burn possible so that after it the probe will settle in a ecliptical orbit with an apoapsis matching the asteroid's orbit. It is the burn that will use the less fuel because we only burn to raise our orbit high enough to reach the asteroid but not more.

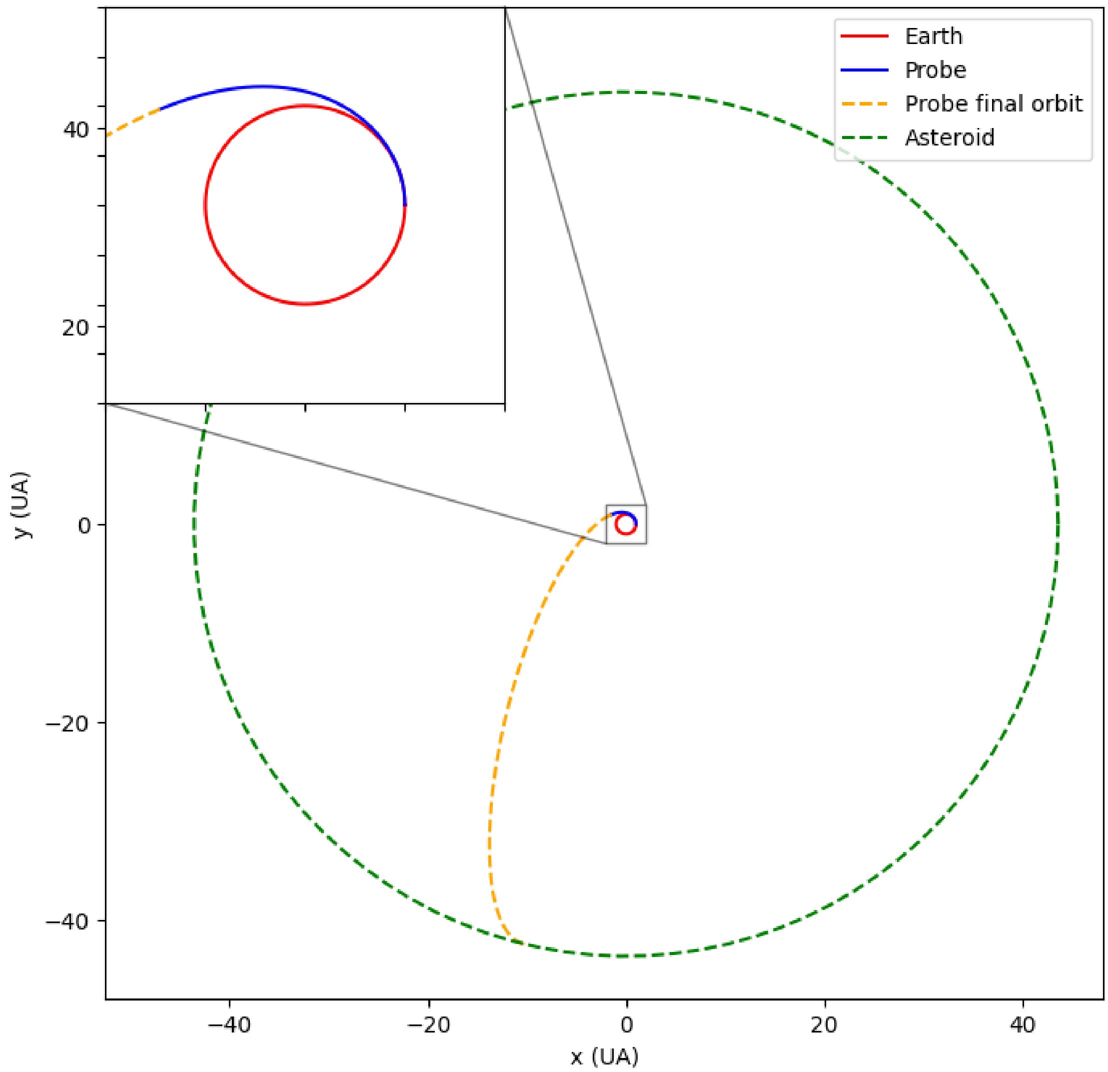
[19]:
```
import pandas as pd

def probeqns(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2)
    f = -G * sun.mass / r ** 3
```

```
    gravity_force = f * posvel[0:2]
    axy = gravity_force

    return posvel[2], posvel[3], axy[0], axy[1]

dist_to_sun = 1
fuel_mass_dist_cache = pd.DataFrame({'d': [1.0, 500.0], 'dry_mass': [0, 500]})
fuel_mass_dist_cache.set_index('d', inplace=True)
while np.abs(dist_to_sun - 43.6) >= 0.01:
    v0 = np.sqrt(G * sun.mass / AU_to_meters(1))  # initial speed
    fuel_mass = np.interp(43.6, fuel_mass_dist_cache.index,␣
 ↪fuel_mass_dist_cache["dry_mass"])
    xymass0 = [AU_to_meters(1), 0, dry_mass + fuel_mass]  # start position
    vxy0 = [0, v0]  # start vertical speed
    tf = fuel_mass/mass_lost_rate  # Max burn time

    probe = Probe(probeqns_rocket, tf, tf/3600, x0=xymass0[0], vx0=vxy0[0],
                  y0=xymass0[1], vy0=vxy0[1], z0=xymass0[2], vz0=mass_lost_rate)␣
 ↪# probe as an object
    t, posvel = probe.odesolve() # solve the differential equations

    last_v = np.linalg.norm(posvel[-1, 3:5])
    last_r = np.linalg.norm(posvel[-1, 0:2])
    a = np.abs(G * sun.mass / (last_v ** 2 - 2*G*sun.mass/last_r))

    period = np.sqrt(4 * np.pi**2 * np.abs(a)**3 / (G * sun.mass))/2  # Orbital␣
 ↪period
    probe = Probe(probeqns, period, period/3600, x0=posvel[-1, 0],␣
 ↪vx0=posvel[-1, 3],
                  y0=posvel[-1, 1], vy0=posvel[-1, 4]) # probe as an object
    t_after, posvel_after = probe.odesolve() # solve the differential equations

    dist_to_sun = meters_to_AU(max(np.linalg.norm(posvel_after[:, 0:2],␣
 ↪axis=1)))
    new_data = {'d': dist_to_sun, 'dry_mass': dry_mass}
    fuel_mass_dist_cache.loc[dist_to_sun] = fuel_mass
    fuel_mass_dist_cache = fuel_mass_dist_cache.sort_index()

fuel_mass # Fuel mass of the probe to reach the asteroid in kg
```

[19]: 147.06252817521252

[20]:
```
fig = plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as␣
 ↪preferred
ax = fig.add_subplot(111)

# Plotting Earth's orbit
```

```python
uang = np.linspace(0, 2 * np.pi, 100)
x = np.cos(uang)
y = np.sin(uang)
ax.plot(x, y, color='red', label='Earth')

ax.plot(meters_to_AU(posvel[:, 0]), meters_to_AU(posvel[:, 1]), color='blue', label="Probe") # plot the probe's orbit

# Plot probe's final orbit
ax.plot(meters_to_AU(posvel_after[:, 0]), meters_to_AU(posvel_after[:, 1]), color='orange', label='Probe final orbit', linestyle="--")

# inset axes....
box_size = 2
x1, x2, y1, y2 = -box_size, box_size, -box_size, box_size  # subregion of the original image
axins = ax.inset_axes(
    [0, 0.6, 0.4, 0.4],
    xlim=(x1, x2), ylim=(y1, y2), xticklabels=[], yticklabels=[])
axins.plot(x, y, color='red', label='Earth')
axins.plot(meters_to_AU(posvel[:, 0]), meters_to_AU(posvel[:, 1]), color='blue', label="Probe") # plot the probe's orbit
axins.plot(meters_to_AU(posvel_after[:, 0]), meters_to_AU(posvel_after[:, 1]), color='orange', label='Probe final orbit', linestyle="--")

# Plotting Asteroid's orbit
uang = np.linspace(0, 2 * np.pi, 100)
x = 43.6 * np.cos(uang)
y = 43.6 * np.sin(uang)
ax.plot(x, y, color='green', linestyle="--", label='Asteroid')

ax.indicate_inset_zoom(axins, edgecolor="black")
ax.set_xlabel('x (UA)')
ax.set_ylabel('y (UA)')
ax.axis('equal')
ax.legend(loc="upper right")
plt.show() # make plot appear
```



[21]:
```
dist_to_sun = np.linalg.norm(posvel_after[:, 0:2], axis=1)
r_aphelion = np.max(dist_to_sun)
meters_to_AU(r_aphelion) # Aphelion distance in UA
```

[21]: 43.59192394818552

[22]:
```
a = (1.0 + r_aphelion) / 2
meters_to_AU(a) # semi-major axis of the probe's orbit in UA
```

[22]: 21.7959619740961

[23]:
```
e = (r_aphelion - 1.0) / (r_aphelion + 1.0)
e # eccentricity of the probe's orbit
```

[23]: 0.9999999999996934

[24]:
```
(t_after[-1] + t[-1]) / (365*24*3600) # Total time to reach the asteroid in␣
↪years
```

[24]: 53.48736974572454

This type of burn (almost like an Hohmann transfer) is indeed really cheap on fuel mass. The fuel needed is lower than the probe dry mass. However, it is extremely slow.

3.2.2 Fastest journey

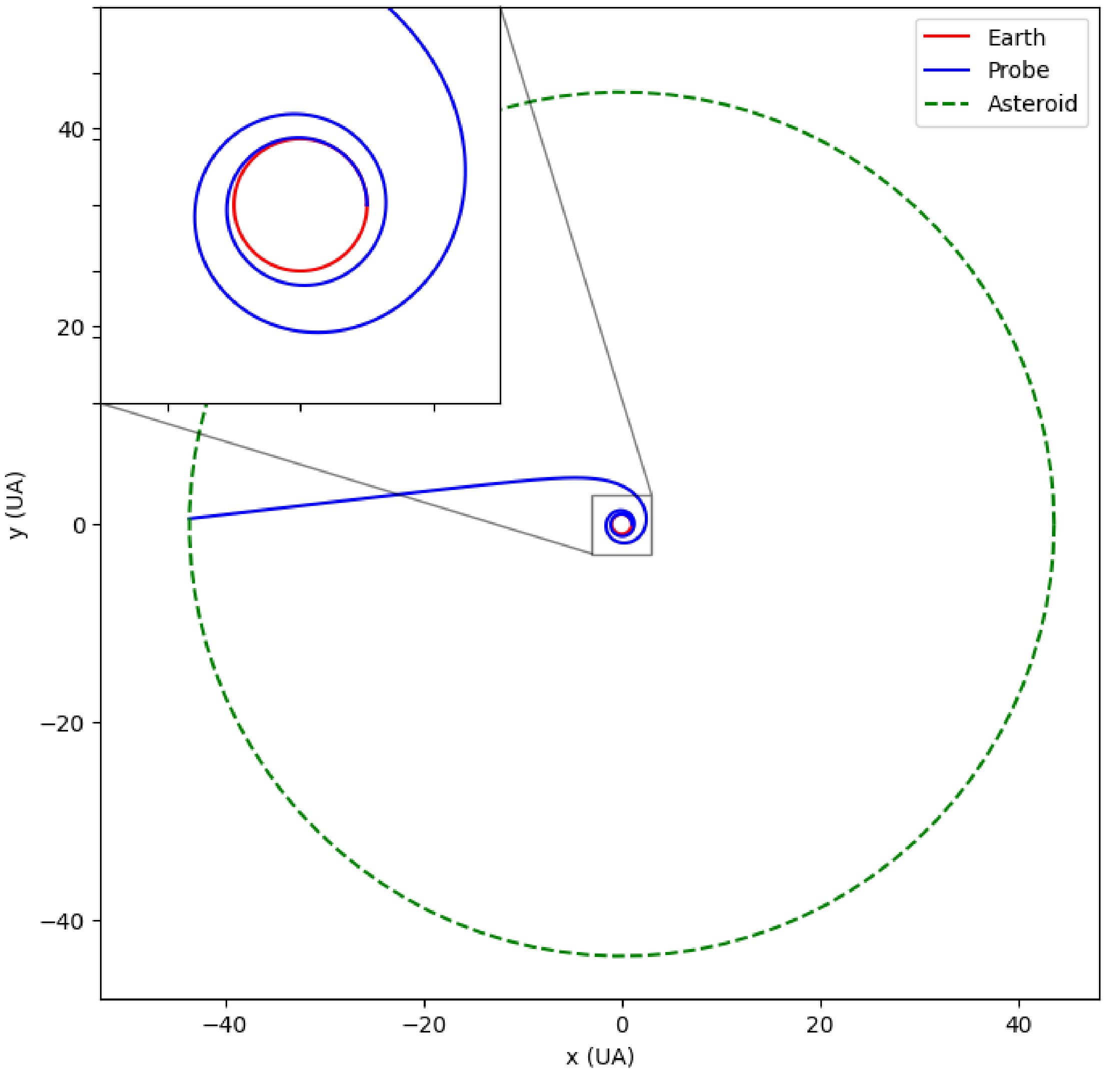
To address the travel time problem with the previous burn, it is possible to try another approach and end the burn only when the probe reaches the asteroid's orbit.

[25]:
```
dist_to_sun = 1
fuel_mass_dist_cache = pd.DataFrame({'d': [1.0, 500.0], 'dry_mass': [0,␣
↪100000]})
fuel_mass_dist_cache.set_index('d', inplace=True)
while np.abs(dist_to_sun - 43.6) >= 0.01:
    v0 = np.sqrt(G * sun.mass / AU_to_meters(1))  # initial speed
    fuel_mass = np.interp(43.6, fuel_mass_dist_cache.index,␣
↪fuel_mass_dist_cache["dry_mass"])
    xymass0 = [AU_to_meters(1), 0, dry_mass + fuel_mass]  # start position
    vxy0 = [0, v0]  # start vertical speed
    tf = fuel_mass/mass_lost_rate  # Max burn time

    probe = Probe(probeqns_rocket, tf, tf/3600, x0=xymass0[0], vx0=vxy0[0],
                  y0=xymass0[1], vy0=vxy0[1], z0=xymass0[2], vz0=mass_lost_rate)␣
↪# probe as an object
    t, posvel = probe.odesolve() # solve the differential equations

    dist_to_sun = meters_to_AU(max(np.linalg.norm(posvel[:, 0:2], axis=1)))
    new_data = {'d': dist_to_sun, 'dry_mass': dry_mass}
    fuel_mass_dist_cache.loc[dist_to_sun] = fuel_mass
    fuel_mass_dist_cache = fuel_mass_dist_cache.sort_index()

fuel_mass # Fuel mass of the probe to reach the asteroid in kg
```

[25]: 3399.801033121058

[26]:
```
fig = plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as␣
↪preferred
ax = fig.add_subplot(111)

# Plotting Earth's orbit
uang = np.linspace(0, 2 * np.pi, 100)
x = np.cos(uang)
```

```
y = np.sin(uang)
ax.plot(x, y, color='red', label='Earth')

ax.plot(meters_to_AU(posvel[:, 0]), meters_to_AU(posvel[:, 1]), color='blue',␣
 ↪label="Probe") # plot the probe's orbit

# inset axes....
box_size = 3
x1, x2, y1, y2 = -box_size, box_size, -box_size, box_size  # subregion of the␣
 ↪original image
axins = ax.inset_axes(
    [0, 0.6, 0.4, 0.4],
    xlim=(x1, x2), ylim=(y1, y2), xticklabels=[], yticklabels=[])
axins.plot(x, y, color='red', label='Earth')
axins.plot(meters_to_AU(posvel[:, 0]), meters_to_AU(posvel[:, 1]),␣
 ↪color='blue', label="Probe") # plot the probe's orbit

# Plotting Asteroid's orbit
uang = np.linspace(0, 2 * np.pi, 100)
x = 43.6 * np.cos(uang)
y = 43.6 * np.sin(uang)
ax.plot(x, y, color='green', linestyle="--", label='Asteroid')

ax.indicate_inset_zoom(axins, edgecolor="black")
ax.set_xlabel('x (UA)')
ax.set_ylabel('y (UA)')
ax.axis('equal')
ax.legend(loc="upper right")
plt.show() # make plot appear
```



The resulting orbit as an interesting shape. As the fuel mass decrease the acceleration gain by the probe is higher. This is why the trajectory forms a spiral, as times pass the mass decrease and the acceleration increase.

To determine the final orbit characteristic we can use the energy formula to find the semi-major axis:

$$E = \frac{1}{2}mV^2 - \frac{GM_{sun}m}{r} = -\frac{GM_{sun}m}{2a}$$

$$a = \frac{rGM_{sun}}{2GM_{sun} - rV^2}$$

[27]:
```
v_final = np.linalg.norm(posvel[-1, 2:4])
v_final / 1e3 # Final speed of the probe in km/s (after burning all fuel)
```

[27]: 75.14672173003704

[28]:
```
dist_to_sun = np.linalg.norm(posvel[:, 0:2], axis=1)
a = dist_to_sun[-1]*G*sun.mass / (2*G*sun.mass - dist_to_sun[-1]*v_final**2)
meters_to_AU(a) # semi-major axis of the probe's orbit in UA
```

[28]: -0.1581389242707825

To find the Eccentricity we can use the norm of the Eccentricity vector:

$$e = \frac{V \times (r \times V)}{GM_{sun}} - \frac{r}{|r|}$$

[29]:
```
pos3d = np.hstack([posvel[-1, 0:2], [0]])
vel3d = np.hstack([posvel[-1, 2:4], [0]])
e = np.linalg.norm(np.cross(vel3d, np.cross(pos3d, vel3d))/(G*sun.mass) - pos3d/
 ↪dist_to_sun[-1])
e
```

[29]: 276.6707886577895

The semi-major axis is negative and the eccentricity is above 1 meaning that the trajectory is hyperbolic. This hyperbolic trajectory suggests that the probe has a velocity greater than the escape velocity which can be verified.

[30]:
```
np.sqrt(2*G*sun.mass / dist_to_sun[-1]) / 1e3 # Escape speed at the probe␣
 ↪position in km/s
```

[30]: 6.377211973108335

[31]:
```
t[-1] / (365*24*3600) # Total time to reach the asteroid in years
```

[31]: 10.780698354645669

Even if the fuel consumption is way greater, the travel time has been drastically improved. But the two values does not evolve linearly, indeed, the time have been cut-off be almost 5 but the fuel mass has been multiplied by 20. Furthermore, the probe has also accumulated a great velocity, therefore, to stay into orbit at arrival the slowdown burn would be really expensive.

One better solution could be to average these two previous solution and also use a slingshot manoeuvre from Jupiter or Mars to help the probe gain velocity without using too much fuel. This also allows to reduce the travel time.

4 Ex 14: Fast track to the Moon

Our goal is to transfer from a parking orbit to the moon. It is possible to use a Hohmann transfer orbit, however, even it is often used because it requires the least amount of impulse, it is also the slowest. To improve the transfer time, another arbitrary point (not the periapsis of the future orbit) could be used as impulse point. In this case, the orbit after the impulse is characterised by the polar coordinates of the burn and its angle.

[32]:
```
from space_base import GravBody, Probe
import matplotlib.pyplot as plt
import numpy as np

# Constants
G = 6.67e-11  # Gravitational constant
earth = GravBody.earth()  # Earth as an object with mass and radius
```

The initial given conditions are:

- Speed after burn $V_0 = 10.85kms^{-1}$
- Altitude $z_0 = 300km$
- Angle $\psi_0 = 6°$ (angle between $\vec{V}$ and $\hat{\theta}$ ($\hat{\theta} \perp \vec{r}$))

[33]:
```
z0 = 300e3  # Initial altitude
r0 = earth.radius + z0  # Initial distance from center of Earth
v0 = 10.85e3  # Initial velocity
psi0 = np.deg2rad(6) # Initial angle
r_moon = 384_400e3

# Initial position and velocity vectors
xy0 = [-r0, 0] # Start at left of the graph
vxy0 = [-v0*np.sin(psi0), -v0*np.cos(psi0)]
```

Knowing this, it is possible to calculate the specific energy ϵ using:

$$\epsilon = -\frac{GM_{earth}}{2a} = \frac{1}{2}V^2 - \frac{GM_{earth}}{r}$$

And the specific angular momentum h,

$$h = r^2\dot{\theta} = r_0 v_0 \cos(\psi_0)$$

[34]:
```
energy0 = 0.5*v0**2-(G*earth.mass)/r0
energy0 # Initial energy (should be constant throughout the simulation)
```

[34]: -851797.5191125795

[35]:
```
h0 = r0*v0*np.cos(psi0)
h0 / 1e6 # Initial angular momentum (km^2/s)
```

[35]: 71983.84286941901

Then, the semi-major axis can be calculated from the previous energy equation.

$$a = -\frac{GM_{earth}}{2\epsilon}$$

[36]:
```
a = -G*earth.mass/(2*energy0)
a / 1e3 # Semi-major axis (km)
```

[36]: 233826.54390388748

Finally, it is possible to use the polar equation of the orbit to find the eccentricity:

$$r = \frac{a(1-e^2)}{1+e\cos(\theta)}$$
$$ae^2 + er\cos(\theta) + r - a = 0$$

Hence,

$$\Delta = (r\cos(\theta))^2 - 4a(r-a)$$
$$e = \frac{-r\cos(\theta) \pm \sqrt{\Delta}}{2a}$$

[37]:
```
discriminant = r0**2 - 4*a*(r0-a)
em = (-r0+np.sqrt(discriminant))/(2*a)
ep = (-r0-np.sqrt(discriminant))/(2*a)
e = max(em, ep) # Eccentricity
e
```

[37]: 0.971470304916528

Finally, the time of traveling from a point (r_0, θ_0) to another point (r_0, θ_1) is given using the flight time formula:

$$t(r) = \frac{GM_{earth}}{(-2\epsilon)^{3/2}} \left[\sin^-1 (\left\{ \frac{GM_{earth} + 2\epsilon r}{\sqrt{(GM_{earth})^2 + 2\epsilon h^2}} \right\}) - \left\{ \frac{\sqrt{2\epsilon(h^2 - 2GM_{earth}r - 2\epsilon r^2)}}{GM_{earth}} \right\} \right]$$

Hence, the transfer time should be $t(r_1) - t(r_0)$

Additionally, because we want that the angle increase as the distance r increase we need to add some modification on the sin^-1 block. First, result of the funcion is expected to be on the range $[-\frac{\pi}{2}; \frac{\pi}{2}]$, it is needed to switch it to $[0; \pi]$. Secondly, $\frac{GM_{earth}+2\epsilon r}{\sqrt{(GM_{earth})^2+2\epsilon h^2}}$ is decreasing as r increase. Therefore, it is needed to invert it so that the inverted sin function will increase as r increase.

[38]:
```
def flight_time(r):
    A = np.arcsin(-(G*earth.mass+2*energy0*r)/np.sqrt((G*earth.
↪mass)**2+2*energy0*h0**2)) + np.pi/2
    B = np.sqrt(2*energy0*(h0**2-2*G*earth.mass*r - 2*energy0*r**2))/(G*earth.
↪mass)
    return (G*earth.mass/(-2*energy0)**1.5) * (A - B)

# Time of flight
travel_time = abs(flight_time(r_moon) - flight_time(r0))
```

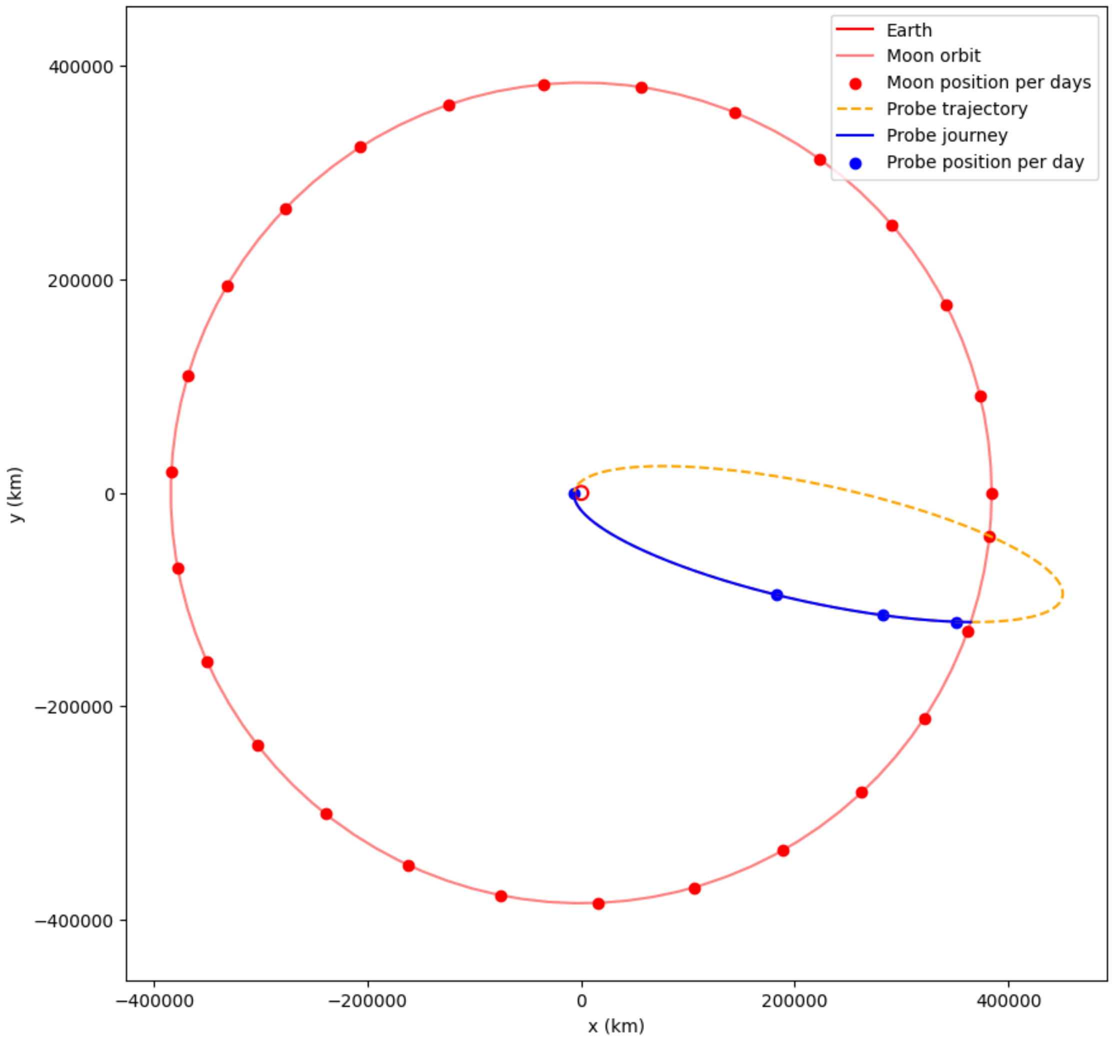
```
travel_time / (24*3600) # Time of flight in days
```

[38]: 3.2483424316374734

[39]:
```
def probeqns(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2)
    f = -G * earth.mass / r ** 3
    gravity_force = f * posvel[0:2]
    axy = gravity_force

    return posvel[2], posvel[3], axy[0], axy[1]

probe = Probe(probeqns, travel_time, travel_time / 60, x0=xy0[0], vx0=vxy0[0],
             y0=xy0[1], vy0=vxy0[1]) # probe as an object
t, posvel = probe.odesolve() # solve the differential equations
```

[40]:
```
# Plot the trajectory
plt.figure(figsize=(10,10))

# Plotting Earth
uang = np.linspace(0, 2 * np.pi, 100)
x = (earth.radius / 1e3) * np.cos(uang)
y = (earth.radius / 1e3) * np.sin(uang)
plt.plot(x, y, color='red', label='Earth')
# Moon orbit
x = (r_moon / 1e3) * np.cos(uang)
y = (r_moon / 1e3) * np.sin(uang)
plt.plot(x, y, color="red", label='Moon orbit', alpha=0.5)
moon_day_period = np.sqrt(4*np.pi**2*r_moon**3/(G*earth.mass)) / (24*3600)
day_angle = 2*np.pi / moon_day_period
moon_days_round = np.floor(moon_day_period)
uang = np.linspace(0, moon_days_round * day_angle, int(moon_days_round))
x = (r_moon / 1e3) * np.cos(uang)
y = (r_moon / 1e3) * np.sin(uang)
plt.scatter(x, y, color="red", label='Moon position per days')

# Plotting entire orbit
probe = Probe(probeqns, travel_time*4, travel_time*4 / 60, x0=posvel[-1, 0],␣
↪vx0=posvel[-1, 2],
             y0=posvel[-1, 1], vy0=posvel[-1, 3]) # probe as an object
t_after, posvel_after = probe.odesolve() # solve the differential equations
plt.plot(posvel_after[:, 0] / 1e3, posvel_after[:, 1] / 1e3, color='orange',␣
↪linestyle="--", label="Probe trajectory") # plot the probe's orbit

plt.plot(posvel[0:, 0] / 1e3, posvel[0:, 1] / 1e3, color='blue', label="Probe␣
↪journey") # plot the probe's orbit
```

```
plt.scatter(posvel[0::60*24, 0] / 1e3, posvel[0::60*24, 1] / 1e3, color='blue',␣
↪label="Probe position per day") # plot the probe's orbit

plt.xlabel('x (km)')
plt.ylabel('y (km)')
plt.axis('equal')
plt.legend()
plt.show() # make plot appear
```



[41]:
```
r = np.sqrt(posvel_after[:, 0] ** 2 + posvel_after[:, 1] ** 2) # distance from␣
↪the center of the Earth
r_per = np.min(r) # perigee
r_ap = np.max(r) # apogee
r_per / 1e3, r_ap / 1e3 # in km
```

[41]: (6599.827350109479, 461056.0972096128)

After the simulation we can check that real values match previous calculation.

[42]:
```
a_real = (r_per + r_ap) / 2 # real semi-major axis
a_real / 1e3, (a - a_real) / 1e3 # in km
```

[42]: (233827.96227986112, -1.4183759736418724)

[43]:
```
e_real = (r_ap - r_per) / (r_ap + r_per) # real eccentricity
e_real, e - e_real
```

[43]: (0.9717748583798103, -0.0003045534632822866)

4.1 Comparing to Hohmann transfer

To compare our previous result, we will assume a Hohmann transfer starting at an altitude 300km and with an apoapsis matching moon orbit altitude.

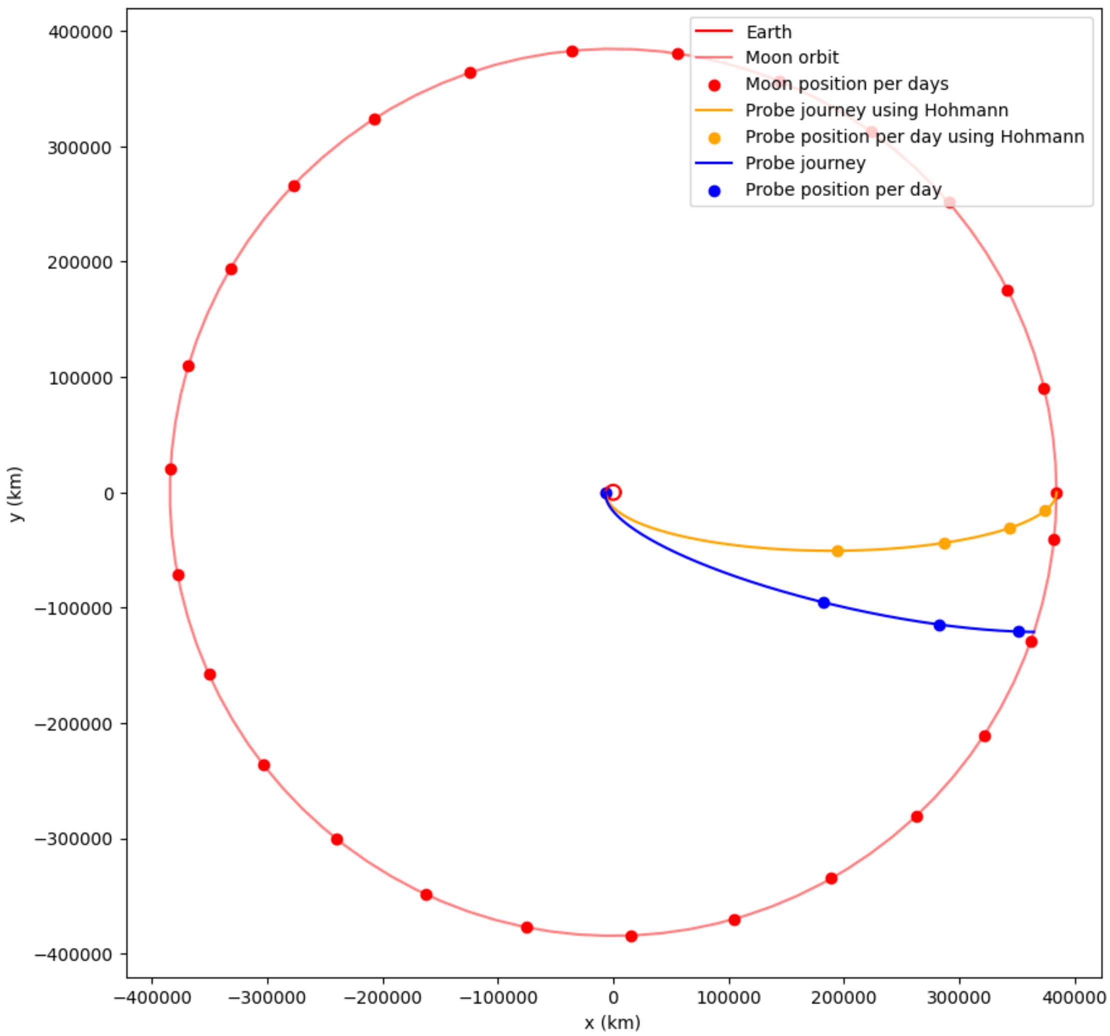
[44]:
```
r_per = r0
r_ap = r_moon
a = (r_per + r_ap) / 2
e = (r_ap - r_per) / (r_ap + r_per)
journey_time = np.sqrt(4 * np.pi**2 * a**3 / (G * earth.mass)) / 2
journey_time / (24*3600) # Time of flight in days
```

[44]: 4.981320278873557

The transfer time using an Hohmann transfer is greater than the previous technique. To better understand the difference, it can be interesting to plot the two trajectories.

[45]:
```
xy0 = [-r0, 0] # Start at left of the graph
v0 = np.sqrt(G * earth.mass * (2 / r_per - 1 / a))
vxy0 = [0, -v0]

probe = Probe(probeqns, journey_time, journey_time / 60, x0=xy0[0], vx0=vxy0[0],
             y0=xy0[1], vy0=vxy0[1]) # probe as an object
t_hohmann, posvel_hohmann = probe.odesolve() # solve the differential equations

# Plot the trajectory
plt.figure(figsize=(10,10))

# Plotting Earth
uang = np.linspace(0, 2 * np.pi, 100)
x = (earth.radius / 1e3) * np.cos(uang)
y = (earth.radius / 1e3) * np.sin(uang)
plt.plot(x, y, color='red', label='Earth')
# Moon orbit
x = (r_moon / 1e3) * np.cos(uang)
```

```
y = (r_moon / 1e3) * np.sin(uang)
plt.plot(x, y, color="red", label='Moon orbit', alpha=0.5)
moon_day_period = np.sqrt(4*np.pi**2*r_moon**3/(G*earth.mass)) / (24*3600)
day_angle = 2*np.pi / moon_day_period
moon_days_round = np.floor(moon_day_period)
uang = np.linspace(0, moon_days_round * day_angle, int(moon_days_round))
x = (r_moon / 1e3) * np.cos(uang)
y = (r_moon / 1e3) * np.sin(uang)
plt.scatter(x, y, color="red", label='Moon position per days')

plt.plot(posvel_hohmann[0:, 0] / 1e3, posvel_hohmann[0:, 1] / 1e3,␣
 ↪color='orange', label="Probe journey using Hohmann") # plot the probe's orbit
plt.scatter(posvel_hohmann[0::60*24, 0] / 1e3, posvel_hohmann[0::60*24, 1] /␣
 ↪1e3, color='orange', label="Probe position per day using Hohmann") # plot␣
 ↪the probe's orbit

plt.plot(posvel[0:, 0] / 1e3, posvel[0:, 1] / 1e3, color='blue', label="Probe␣
 ↪journey") # plot the probe's orbit
plt.scatter(posvel[0::60*24, 0] / 1e3, posvel[0::60*24, 1] / 1e3, color='blue',␣
 ↪label="Probe position per day") # plot the probe's orbit

plt.xlabel('x (km)')
plt.ylabel('y (km)')
plt.axis('equal')
plt.legend(loc='upper right')
plt.show() # make plot appear
```



Earth
Moon orbit
Moon position per days
Probe journey using Hohmann
Probe position per day using Hohmann
Probe journey
Probe position per day
400000
300000
200000
100000
0
−100000
−200000
−300000
−400000
y (km)
−400000
−300000
−200000
−100000
0
100000
200000
300000
400000
x (km)