# Space Dynamics Workbook

Nicolas THIERRY

May 4, 2024

## Contents

## 0.1 Ex 1: Uniform Gravity

[1]:
```
import matplotlib.pyplot as plt
from space_base import GravBody, Probe


def projectile(_, posvel):
    current_gravity = gravity
    return posvel[1], -current_gravity


# Constants
G = 6.67e-11  # Gravitational constant
earth = GravBody.earth()  # Earth as an object with mass and radius
gravity = 9.81  # simple gravity

# Initial Conditions
x0 = 0  # start position
vx0 = 850  # start vertical speed
t_num = 2_000  # number of steps in trajectory
```

To compute the time of the flight we can just solve the SUVAT equation like follow:

$$v = u - gt$$

$$s = ut - \frac{1}{2}gt^2$$

$$0 = ut - \frac{1}{2}gt^2$$

$$t * (u - \frac{1}{2}gt) = 0$$

So, $t = 0$ and $t = \frac{2*u}{g}$

[2]:
```
t_final = (2*vx0) / gravity  # time of trajectory given
t_final # Time of flight in seconds
```

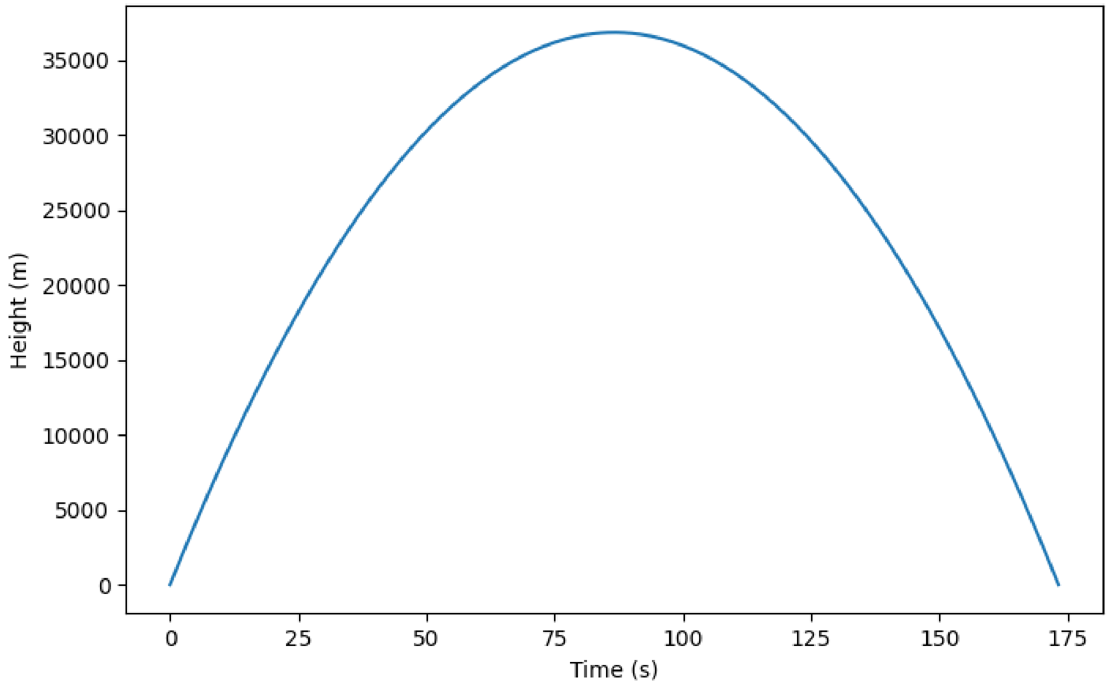
[2]:
```
173.2925586136595
```

[3]:
```
# Running Solver
probe = Probe(projectile, t_final, t_num, x0=x0, vx0=vx0, event=0)  # probe as␣
 ↪an object
in_energy = 0.5 * probe.mass * probe.posvel0[1] ** 2 + probe.mass * gravity *␣
 ↪x0  # initial energy
t, posvel = probe.odesolve()  # solve the differential equations

# Solver Results
t_end = len(t) - 1  # last value of array
```

```
fin_energy = 0.5 * probe.mass * posvel[t_end][1] ** 2 + probe.mass * gravity *␣
 ↪posvel[t_end][0]  # final energy
accuracy = (fin_energy - in_energy) / in_energy  # accuracy of solver
accuracy
```

```
[3]: 4.833854193505944e-16
```

```
[4]: # Plotting
plt.figure(figsize=(8, 5))  # create figure, figsize can be changed as preferred
plt.plot(t, posvel[:, 0])  # plot time against height
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()  # make plot appear
```



The final graph is a parabola. This is because the altitude (z) follows a second order equation.

### 0.2 Ex 2: Realistic Gravity

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

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

```
gravity = 9.81  # simple gravity

# Initial Conditions
x0 = 0  # start position
vx0 = 850  # start vertical speed
t_num = 500_000  # number of steps in trajectory
```

First, we start by running the previous simulation with uniform gravity and then do the same simulation but with a more realistic approach.

[6]:
```
def projectile_uniform_gravity(_, posvel):
    current_gravity = gravity
    return posvel[1], -current_gravity

t_final = (2*vx0) / gravity  # time of trajectory given
# Running Solver
probe = Probe(projectile_uniform_gravity, t_final, t_num, x0=x0, vx0=vx0,␣
 ↪event=0)  # probe as an object
t, posvel = probe.odesolve()  # solve the differential equations
max_height_uniform_gravity = np.max(posvel, axis=0)[0]
```

Current gravity can be computed as follows:

$$g = \frac{GM}{(R+z)^2}$$

Where, $M$ is the earth mass, $R$ its radius and $z$ is the current altitude. Of course, we also need to update the energy as it is now defined as:

$$E = \frac{1}{2}m * v^2 - \frac{GMm}{R+z}$$

[7]:
```
def projectile_with_gravity(t, posvel):
    current_gravity = G * earth.mass / (earth.radius + posvel[0])**2
    return posvel[1], -current_gravity

t_final_temp = 200
# Running Solver
probe = Probe(projectile_with_gravity, t_final_temp, t_num, x0=x0, vx0=vx0,␣
 ↪event=0)  # probe as an object
t, posvel = probe.odesolve()  # solve the differential equations

# Solver Results
max_height_realistic_gravity = np.max(posvel, axis=0)[0]
t_end = len(t) - 2
t_final_realistic_gravity = t[t_end]
```

With the new way of computing gravity we found that the maximum altitude if higher by:

[8]:
```
max_height_realistic_gravity - max_height_uniform_gravity
```

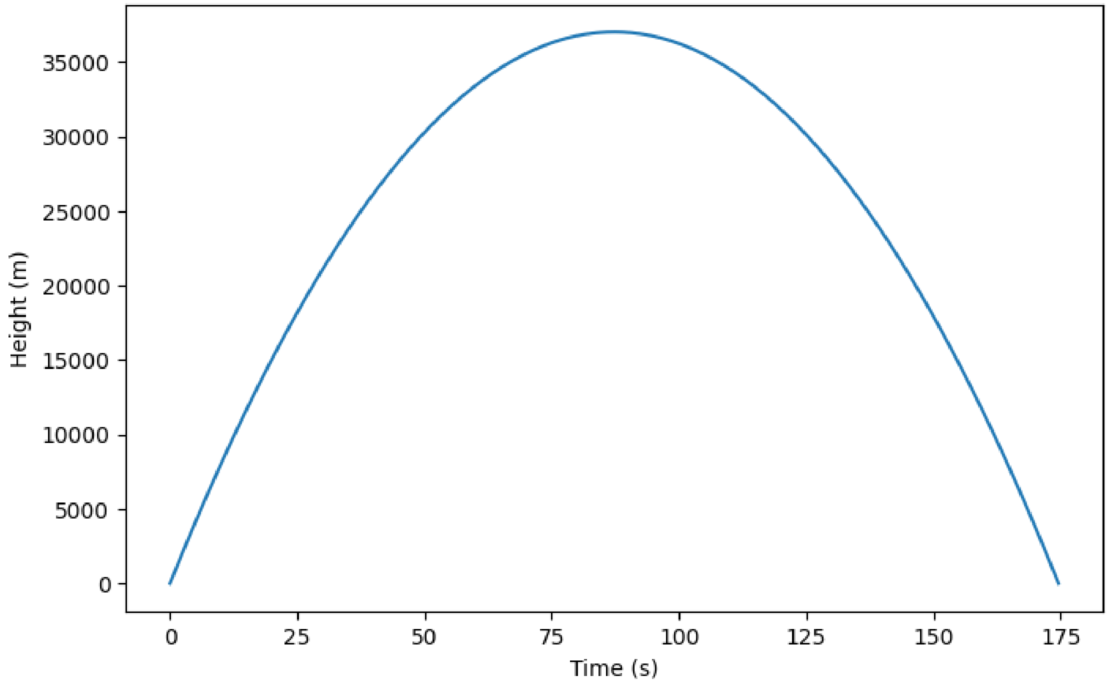
```
[8]: 198.99484451519675
```

As the trajectory is modified, we can assume that time of flight will also be longer.

```
[9]: t_final_realistic_gravity - t_final
```

```
[9]: 1.273390518238756
```

```
[10]: # Plotting
      plt.figure(figsize=(8, 5))  # create figure, figsize can be changed as preferred
      plt.plot(t, posvel[:, 0])  # plot time against height
      plt.xlabel('Time (s)')
      plt.ylabel('Height (m)')
      plt.show()  # make plot appear
```



We can check that the penultimate value is indeed close to zero (below 1m).

```
[11]: posvel[t_end][0] # height at the end of the trajectory (in m)
```

```
[11]: 0.13790983261424117
```

We can now check if the energy is well conserved.

```
[12]: in_energy = 0.5 * probe.mass * posvel[0][1] ** 2 - G * earth.mass * probe.mass ␣
        ↪/ (earth.radius + posvel[0][0])  # initial energy
```

```
fin_energy = 0.5 * probe.mass * posvel[t_end][1] ** 2 - G * earth.mass * probe.
↪mass  / (earth.radius + posvel[t_end][0])  # final energy
accuracy = (fin_energy - in_energy) / in_energy  # accuracy of solver
accuracy
```

[12]:
```
-7.135652621342857e-12
```

We can also express the accuracy as a percentage that is close to 100% if the energy at the beginning and the end of the simulation are the same.

[13]:
```
accuracy = 100 * in_energy / fin_energy  # accuracy of solver
accuracy
```

[13]:
```
100.00000000071356
```

Or, in contrary, make an error computation in percentage (we will then want to keep this percentage as low as possible):

[14]:
```
error_percentage = 100 * abs((fin_energy - in_energy) / in_energy)
error_percentage
```

[14]:
```
7.135652621342857e-10
```

## 0.3 Ex 3: Drag in a uniform atmosphere

[15]:
```
import matplotlib.pyplot as plt
from space_base import GravBody, Probe
import numpy as np

# Constants
G = 6.67e-11  # Gravitational constant
earth = GravBody.earth()  # Earth as an object with mass and radius
gravity = 9.81  # simple gravity

# Initial Conditions
x0 = 0  # start position
vx0 = 850  # start vertical speed
t_num = 100_000  # number of steps in trajectory
```

We are going to implement the drag force in our simulation as a force that oppose to the speed vector. And expressed as follows:

$$F_{drag} = -\frac{C_D}{2}\rho A V^2 \hat{V}$$

Where, $\hat{V}$ is a unit vector in the direction of motion.

This is the maximum height achieve with drag in meters:

[16]:
```
def projectile(_, posvel):
    cd=1.0
    A=0.01
    mass=1.0
    Density=1.217

    current_gravity = gravity
    drag_force = -0.5 * cd * A * Density * abs(posvel[1]) * posvel[1]

    return posvel[1], -current_gravity + drag_force / mass

# Running Solver
t_final = 200  # time of trajectory given
probe = Probe(projectile, t_final, t_num, x0=x0, vx0=vx0, event=0)  # probe as␣
 ↪an object
t, posvel = probe.odesolve()  # solve the differential equations
t_end = len(t) - 2
np.max(posvel, axis=0)[0] # maximum height (in m)
```

[16]:
```
501.83817944712223
```

This is the time of flight in seconds. We immediately saw that it is way more faster than previous flights and that the apoapsis is way lower. This tells us that in dense atmosphere, drag plays an important part of motion.

[17]:
```
t[t_end] # Time of flight (in s)
```

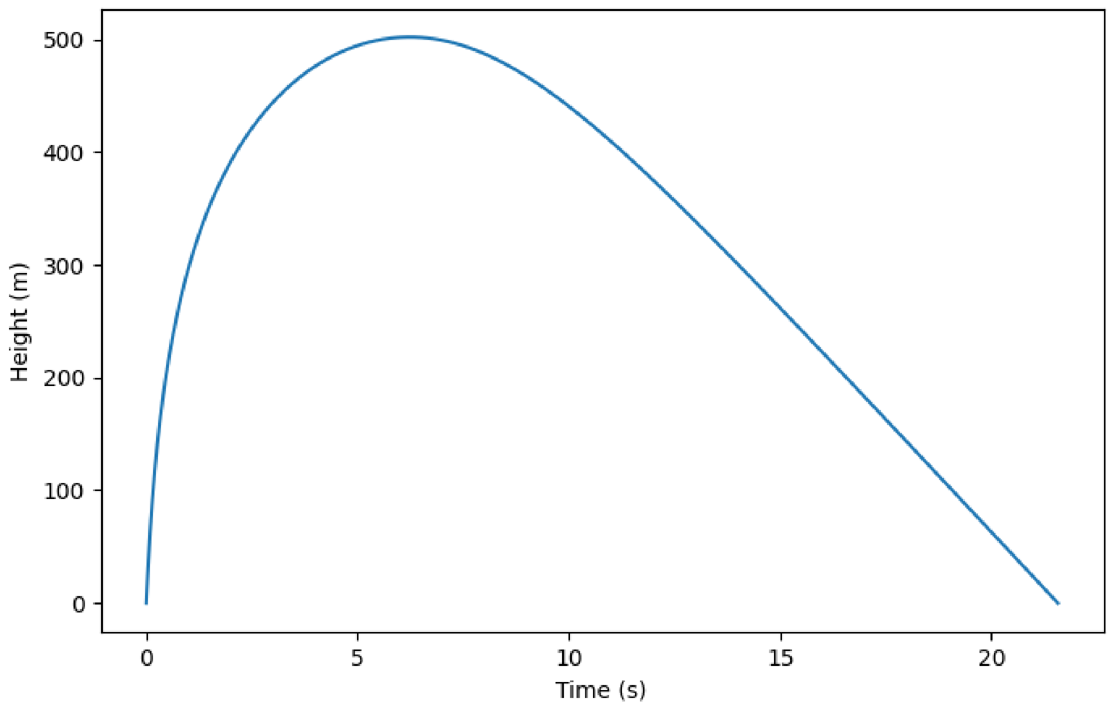
[17]:
```
21.56821568215682
```

To validate our result, we can check that we are close to the ground at the end of the simulation.

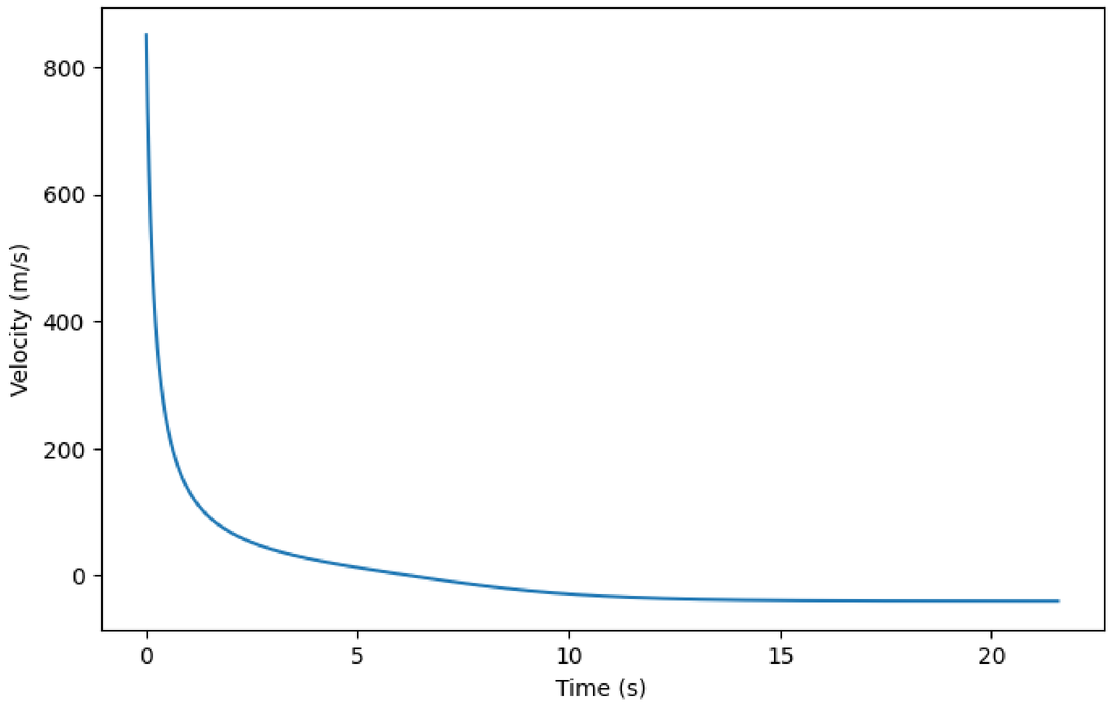
[18]:
```
posvel[t_end][0] # Altitude at the end (in m)
```

[18]:
```
0.04153531472651384
```

[19]:
```
# Plotting
plt.figure(figsize=(8, 5))  # create figure, figsize can be changed as preferred
plt.plot(t, posvel[:, 0])  # plot time against height
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()  # make plot appear
```



[20]:
```
# Plotting
plt.figure(figsize=(8, 5))  # create figure, figsize can be changed as preferred
plt.plot(t, posvel[:, 1])  # plot time against height
plt.xlabel('Time (s)')
plt.ylabel('Velocity (m/s)')
plt.show()  # make plot appear
```



We clearly see that the projectile is slowed very quickly. Then, after reaching its apoapsis it enters a free fall state where the falling velocity will converge to what we can wall the terminate velocity of the projectile. This velocity is the maximum velocity that the projectile can reach in free fall because of the drag that equilibrate with the gravity at some speed.

[21]:
```
in_energy = 0.5 * probe.mass * posvel[0][1] ** 2 + probe.mass * gravity *␣
  ↪posvel[0][0]  # initial energy
fin_energy = 0.5 * probe.mass * posvel[t_end][1] ** 2 + probe.mass * gravity *␣
  ↪posvel[t_end][0]  # final energy
error_percentage = 100 * abs((fin_energy - in_energy) / in_energy)
error_percentage
```

[21]:
```
99.77724777069092
```

Here, we see that energy is no longer conserved. This is because energy is lost due to the drag force. In fact, this force consume energy by for example convert it to heat. This is why at huge speed drag can cause major heat problematics.

## 0.4 Ex 4: An isothermal atmosphere model

[22]:
```
import matplotlib.pyplot as plt
from space_base import GravBody, Probe
import numpy as np

# Constants
```

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

# Initial Conditions
x0 = 0  # start position
vx0 = 850  # start vertical speed
t_num = 100_000  # number of steps in trajectory
```

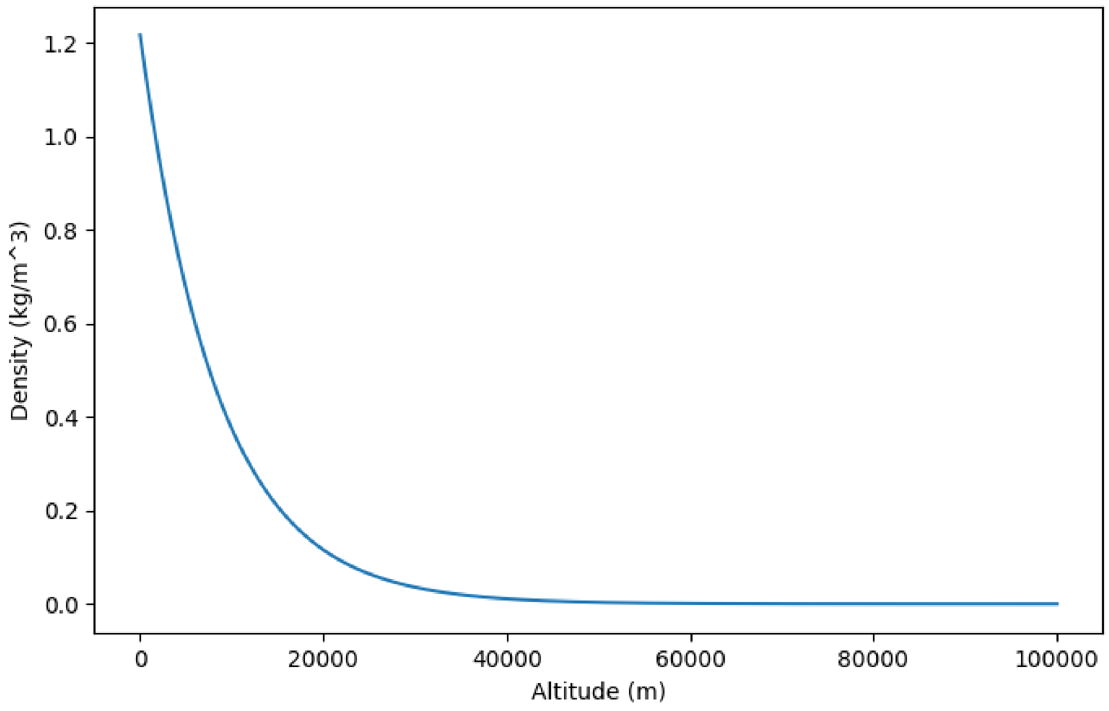
We are going to compute density as function of altitude as follows:

$$\rho(h) = \rho_{surface} * exp(-h/H)$$

Where, $\rho_{surface}$ and $H$ are respectively, the density at sea-level and the Earth's scale-height.

[23]:
```
def atmosphere(h):
    surfacedens=1.217
    scaleheight=8500
    return surfacedens*np.exp(-h/scaleheight)
```

[24]:
```
h = np.linspace(0, 100_000, 5000)  # linearly separated time steps
rho = atmosphere(h)

plt.figure(figsize=(8, 5))  # create figure, figsize can be changed as preferred
plt.plot(h, rho)  # plot time against height
plt.xlabel('Altitude (m)')
plt.ylabel('Density (kg/m^3)')
plt.show()  # make plot appear
```



[25]:
```
def projectile(_, posvel):
    cd=1.0
    A=0.01
    mass=1.0

    current_gravity = G * earth.mass / (earth.radius + posvel[0])**2
    drag_force = -0.5 * cd * A * atmosphere(posvel[0]) * abs(posvel[1]) *␣
 ↪posvel[1]

    return posvel[1], -current_gravity + drag_force / mass

# Running Solver
t_final = 50  # time of trajectory given
probe = Probe(projectile, t_final, t_num, x0=x0, vx0=vx0, event=0)  # probe as␣
 ↪an object
t, posvel = probe.odesolve()  # solve the differential equations
t_end = len(t) - 2
np.max(posvel, axis=0)[0] # maximum height (in m)
```

[25]:
```
512.8527377153681
```

[26]:
```
t[t_end] # Time of flight (in s)
```

[26]:
```
21.80271802718027
```

As we can see, with not uniform density our projectile is going a bit higher than previously. This is because at high altitude (apoapsis) the drag is reduced due to a lower air density. Of course, this effect should be more important with higher apoapsis. If the flare was launch with a higher initial vertical speed. Because, as show above the density curve is not straight and above 40km the density of the atmosphere is nearly zero creating very limited to zero drag force allowing the flare to go higher and to achieve faster speed at return.

Finally, to validate our result, we can check that we are close to the ground at the end of the simulation.

[27]:
```
posvel[t_end][0] # Last altitude (in m)
```

[27]:
```
0.01016905735843121
```

## 0.5 Ex 5: Probe goes haywire

[28]:
```
import matplotlib.pyplot as plt
from space_base import GravBody, Probe
import numpy as np

# Constants
```

```
G = 6.67e-11  # Gravitational constant
moon = GravBody(name="Moon", mass=7.34767309e22, radius=1.7371e6)  # Moon as an␣
↪object with mass and radius
```

Now that we want to use 3D coordinates, we will need to adapt the previous formulas. Of course, as we are now reaching high altitude, we will also compute realistic gravity instead of uniform. So, our state vectors will follow these equations:

$$\frac{dx}{dt} = V_x$$
$$\frac{dy}{dt} = V_y$$
$$\frac{dz}{dt} = V_z$$

$$\frac{dV_x}{dt} = -\frac{GM}{r^3}x$$
$$\frac{dV_y}{dt} = -\frac{GM}{r^3}y$$
$$\frac{dV_z}{dt} = -\frac{GM}{r^3}z$$

[29]:
```
def probeqnsmoon(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2 + posvel[2] ** 2)
    f = -G * moon.mass / r ** 3
    axyz = f * posvel[0], f * posvel[1], f * posvel[2]
    return posvel[3], posvel[4], posvel[5], axyz[0], axyz[1], axyz[2]
```

To solve this problem, we need to compute the initial velocity so that $Q = \frac{V^2 R_{moon}}{GM_{moon}}$ where $Q = 1$. Knowing that we have $\|\vec{V}\| = \sqrt{V^2}$, we can express the velocity with:

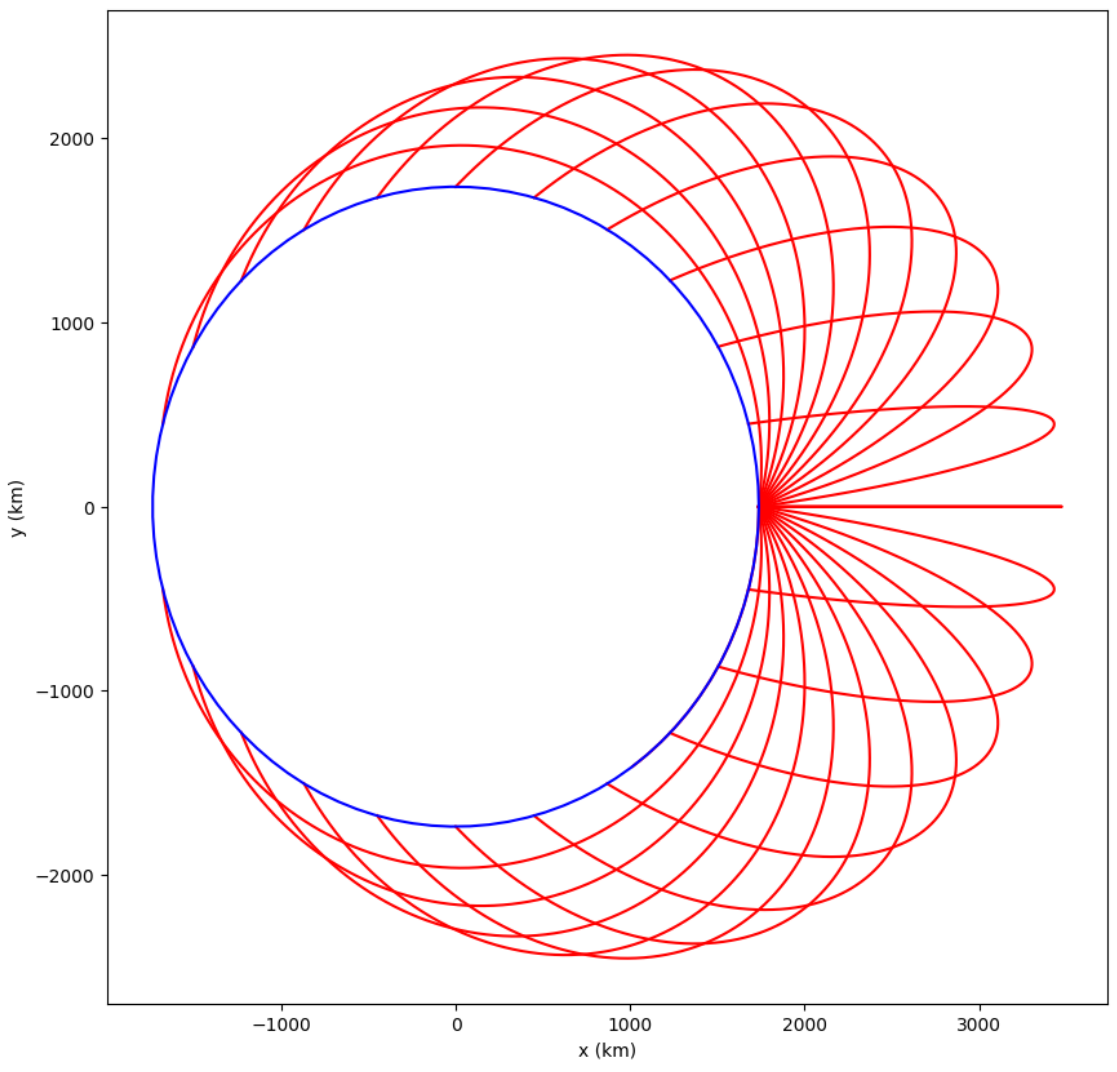
$$\|\vec{V}\| = \sqrt{\frac{QGM_{moon}}{R_{moon}}}$$

[30]:
```
Q = 1
v = np.sqrt(Q * G * moon.mass / moon.radius)  # Velocity of the probe
v / 1e3  # Velocity in km/s
```

[30]:
```
1.6796756238936446
```

[31]:
```
fig = plt.figure(figsize=(10, 10)) # create figure, figsize can be changed as␣
↪preferred
ax = fig.add_subplot(111)
init_angles = np.arange(-np.pi/2, np.pi/2, np.pi/24)  # Angle of the probe

# Initial Conditions
```

```
t_final = 3600 * 12 # determined trajectory time
t_num = t_final # number of steps in trajectory
xyz0 = [moon.radius, 0, 0]  # start position
for angle in init_angles:
    vxyz0 = [v * np.cos(angle), v * np.sin(angle), 0]  # start vertical speed

    probe = Probe(probeqnsmoon, t_final, t_num, x0=xyz0[0], vx0=vxyz0[0],
                  y0=xyz0[1], vy0=vxyz0[1], z0=xyz0[2], vz0=vxyz0[2], event=moon.
↪radius) # probe as an object
    t, posvel = probe.odesolve() # solve the differential equations
    ax.plot(posvel[:, 0] / 1e3, posvel[:, 1] / 1e3, color='red')

# Plotting Moon
uang = np.linspace(0, 2 * np.pi, 100)
x = (moon.radius / 1e3) * np.cos(uang)
y = (moon.radius / 1e3) * np.sin(uang)
ax.plot(x, y, color='blue')
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')
plt.show() # make plot appear
```



From what we saw with this simulation, their is no safest place on the moon to be when the probe launcher goes haywire. Even the exact opposite side of the planet (in our case x=-moon.radius, y=0, z=0). Because when Q = 1, our speed if equal to $\sqrt{\frac{GM_{moon}}{R_{moon}}}$ which is, according to Newton's second law, the speed of a perfect circular orbit for our altitude. This means that, in a perfect case like ours where the planet is perfectly circular, the probe could reach an orbit with an altitude of 0m for an initial angle of $\frac{\pi}{2}$ (or $-\frac{\pi}{2}$). So, it will reach every single point of the moon.

## 0.6 Ex 6: Drag for 3D motion

[32]:
```
from space_base import GravBody
import numpy as np

# Constants
G = 6.67e-11  # Gravitational constant
```

```
earth = GravBody.earth()  # Earth as an object with mass and radius
```

To create an isothermal atmosphere around the earth, we can reuse our atmosphere function to compute air density.

[33]:
```
def atmosphere(h):
    surfacedens=1.217
    scaleheight=8500
    return surfacedens*np.exp(-h/scaleheight)
```

Then, we need to adapt the previous equation by using the 3 dimensions.

[34]:
```
def probeqns(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2 + posvel[2] ** 2)
    f = -G * earth.mass / r ** 3
    gravity_force = f * posvel[0], f * posvel[1], f * posvel[2]

    cd=1.0
    A=0.01
    v2 = posvel[3] ** 2 + posvel[4] ** 2 + posvel[5] ** 2
    unit_v = posvel[3:6] / np.sqrt(v2)
    drag_force = -0.5 * cd * A * atmosphere(r - earth.radius) * v2 * unit_v
    axyz = drag_force + gravity_force

    return posvel[3], posvel[4], posvel[5], axyz[0], axyz[1], axyz[2]
```

## 0.7 Ex 7: Aerobreaking

[35]:
```
import matplotlib.pyplot as plt
from space_base import GravBody, Probe
import numpy as np

# Constants
G = 6.67e-11  # Gravitational constant
mars = GravBody(name="Mars", mass=0.64169e24, radius=3389.5e3)  # Mars as an␣
 ↪object with mass and radius
```

We start by taking the equations of motion writes in the previous exercice.

[36]:
```
def atmosphere(h):
    surfacedens=0.020
    scaleheight=11.1e3
    return surfacedens*np.exp(-h/scaleheight)
```

Then, we need to adapt the previous equation by using the 3 dimensions.

[37]:
```
def probeqns(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2 + posvel[2] ** 2)
    f = -G * mars.mass / r ** 3
    gravity_force = f * posvel[0], f * posvel[1], f * posvel[2]

    cd=1.0
    A=0.01
    v2 = posvel[3] ** 2 + posvel[4] ** 2 + posvel[5] ** 2
    unit_v = posvel[3:6] / np.sqrt(v2)
    drag_force = -0.5 * cd * A * atmosphere(r - mars.radius) * v2 * unit_v
    axyz = drag_force + gravity_force

    return posvel[3], posvel[4], posvel[5], axyz[0], axyz[1], axyz[2]
```

Then, we need to compute the semi-major axis of the initial orbit knowing that $r_p + r_a = 2 * a$, so $a = \frac{r_p + r_a}{2}$

[38]:
```
r_p = mars.radius + 100e3
r_a = 47972e3
a_initial = (r_p + r_a) / 2
```

We will now use the energy equation to compute the initial velocity assuming that on first orbit the energy is conserved.

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

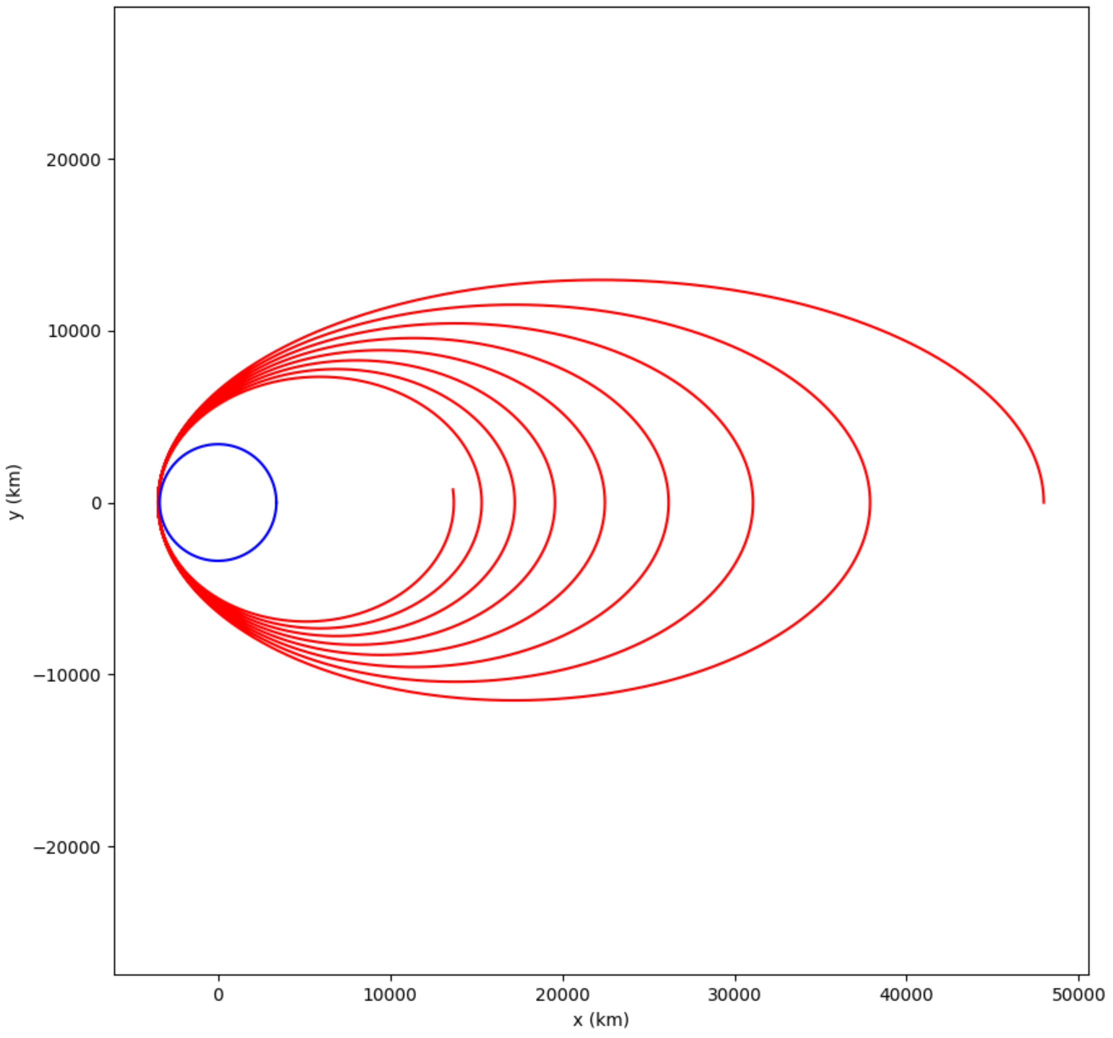
$$V = \sqrt{2GM_{planet}(\frac{1}{r} - \frac{1}{2a})}$$

[39]:
```
v = np.sqrt(2*G*mars.mass*(1/r_a - 1/(2*a_initial)))
v # Initial velocity in m/s
```

[39]:
```
347.84600419637707
```

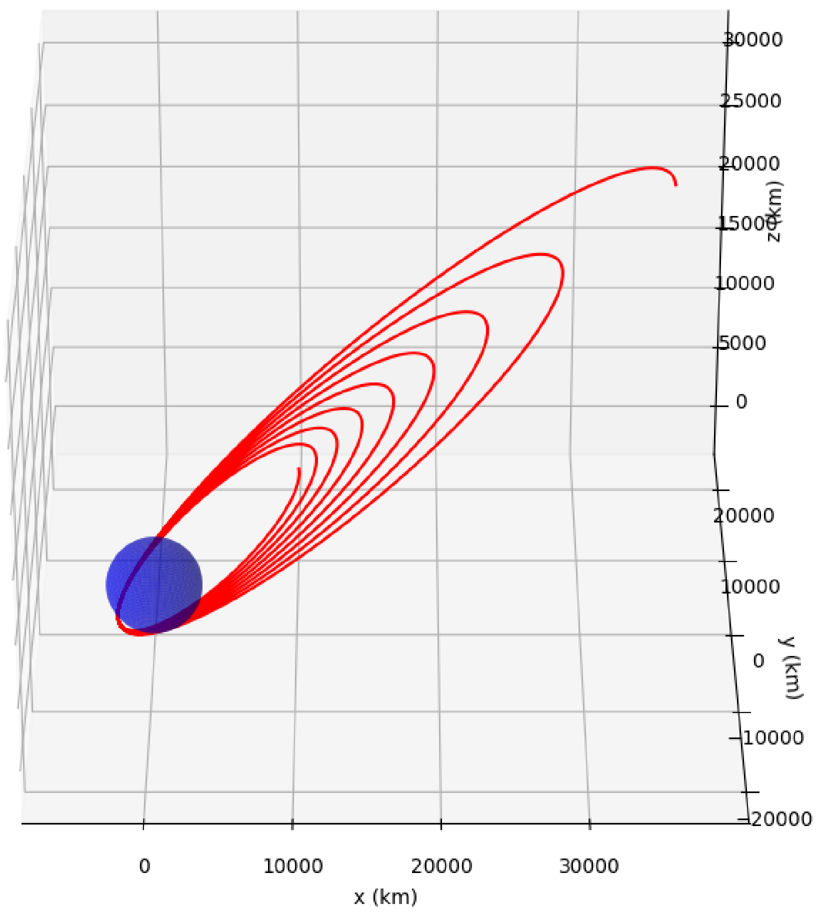
[40]:
```
# Initial Conditions
t_final = 3600 * 24 * 5 # determined trajectory time
t_num = t_final # number of steps in trajectory
xyz0 = [r_a*np.cos(30 * np.pi / 130), 0, r_a*np.sin(30 * np.pi / 130)]  # start␣
 ↪position
vxyz0 = [0, v, 0]  # start vertical speed

probe = Probe(probeqns, t_final, t_num, x0=xyz0[0], vx0=vxyz0[0],
            y0=xyz0[1], vy0=vxyz0[1], z0=xyz0[2], vz0=vxyz0[2], event=mars.
 ↪radius) # probe as an object
t, posvel = probe.odesolve() # solve the differential equations

# Plotting 2D trajectory in orbit plane
fig = plt.figure(figsize=(10, 10)) # create figure, figsize can be changed as␣
 ↪preferred
ax = fig.add_subplot(111)
```

```
ax.plot(posvel[:, 0] /(np.cos(30 * np.pi / 130) * 1e3), posvel[:, 1] / 1e3,␣
 ↪color='red')

# Plotting Mars
uang = np.linspace(0, 2 * np.pi, 100)
x = (mars.radius / 1e3) * np.cos(uang)
y = (mars.radius / 1e3) * np.sin(uang)
ax.plot(x, y, color='blue')
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')
ax.axis('equal')
plt.show() # make plot appear
```



By running the simulation for 5 days, we clearly see that the probe is losing energy each time it is close to Mars as the result of drag. Of course, drag parameters, like, coefficient of drag, probe surface and velocity will change drastically the speed of energy loss and the number of orbits required to slow down but this will increase heating.

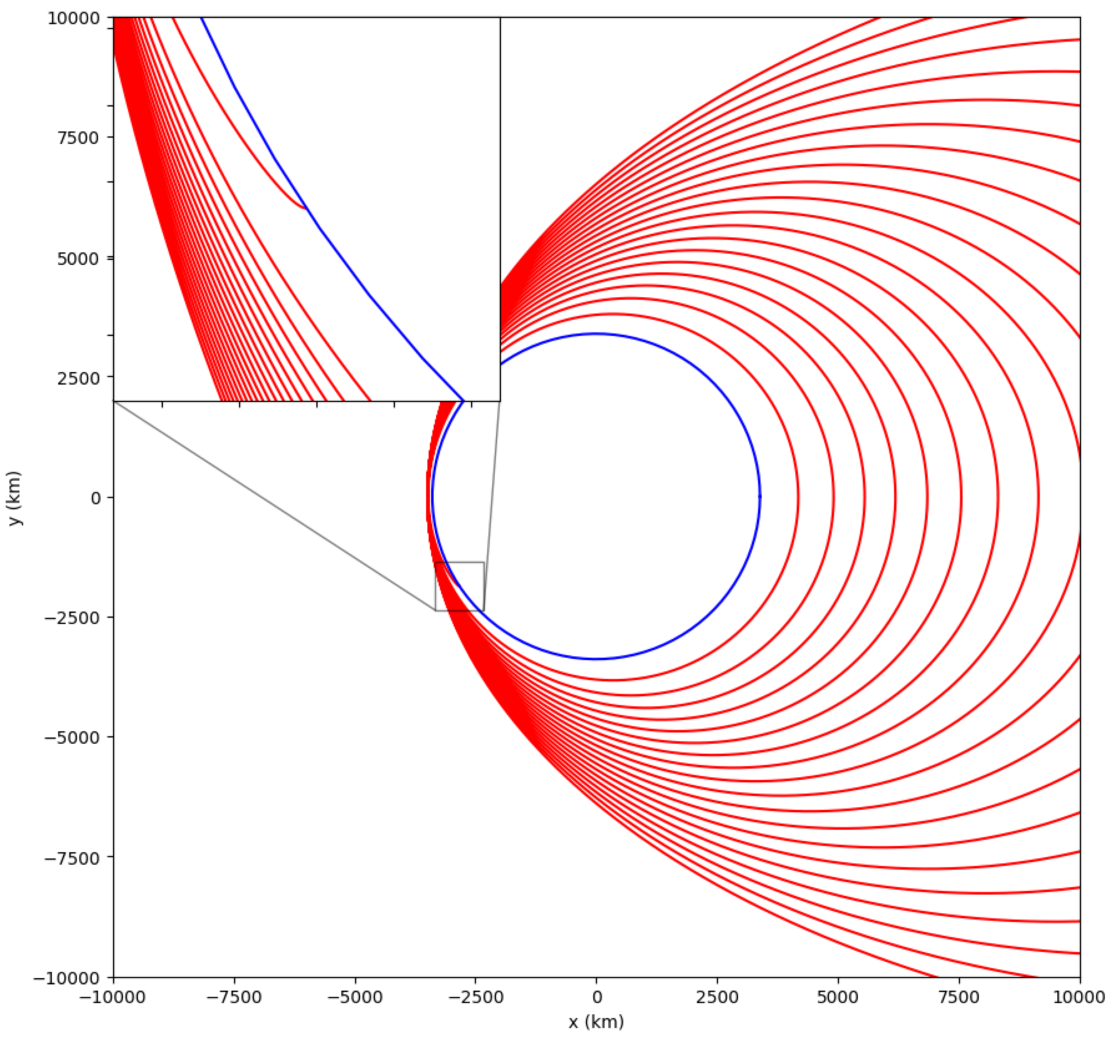
[41]:
```
fig = plt.figure(figsize=(15, 10)) # create figure, figsize can be changed as␣
↪preferred
ax = fig.add_subplot(111, projection='3d')
ax.plot(posvel[:, 0] / 1e3, posvel[:, 1] / 1e3, posvel[:, 2] / 1e3, color='red')

# Plotting Mars
uang = np.linspace(0, 2 * np.pi, 100)
vang = np.linspace(0, np.pi, 100)
x = mars.radius / 1e3 * np.outer(np.cos(uang), np.sin(vang))
y = mars.radius / 1e3 * np.outer(np.sin(uang), np.sin(vang))
z = mars.radius / 1e3 * np.outer(np.ones(np.size(uang)), np.cos(vang))
ax.plot_surface(x, y, z, color='blue', alpha=0.5)
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')
ax.set_zlabel('z (km)')
ax.axis('equal')
ax.azim = -90

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



With this 3D graph of the orbit we can clearly see the inclination of 30°.

[42]:
```
# Initial Conditions
t_final = 3600 * 24 * 8 # determined trajectory time
t_num = t_final # number of steps in trajectory
xyz0 = [r_a*np.cos(30 * np.pi / 130), 0, r_a*np.sin(30 * np.pi / 130)]  # start␣
 ↪position
vxyz0 = [0, v, 0]  # start vertical speed

probe = Probe(probeqns, t_final, t_num, x0=xyz0[0], vx0=vxyz0[0],
            y0=xyz0[1], vy0=vxyz0[1], z0=xyz0[2], vz0=vxyz0[2], event=mars.
 ↪radius) # probe as an object
t, posvel = probe.odesolve() # solve the differential equations
```

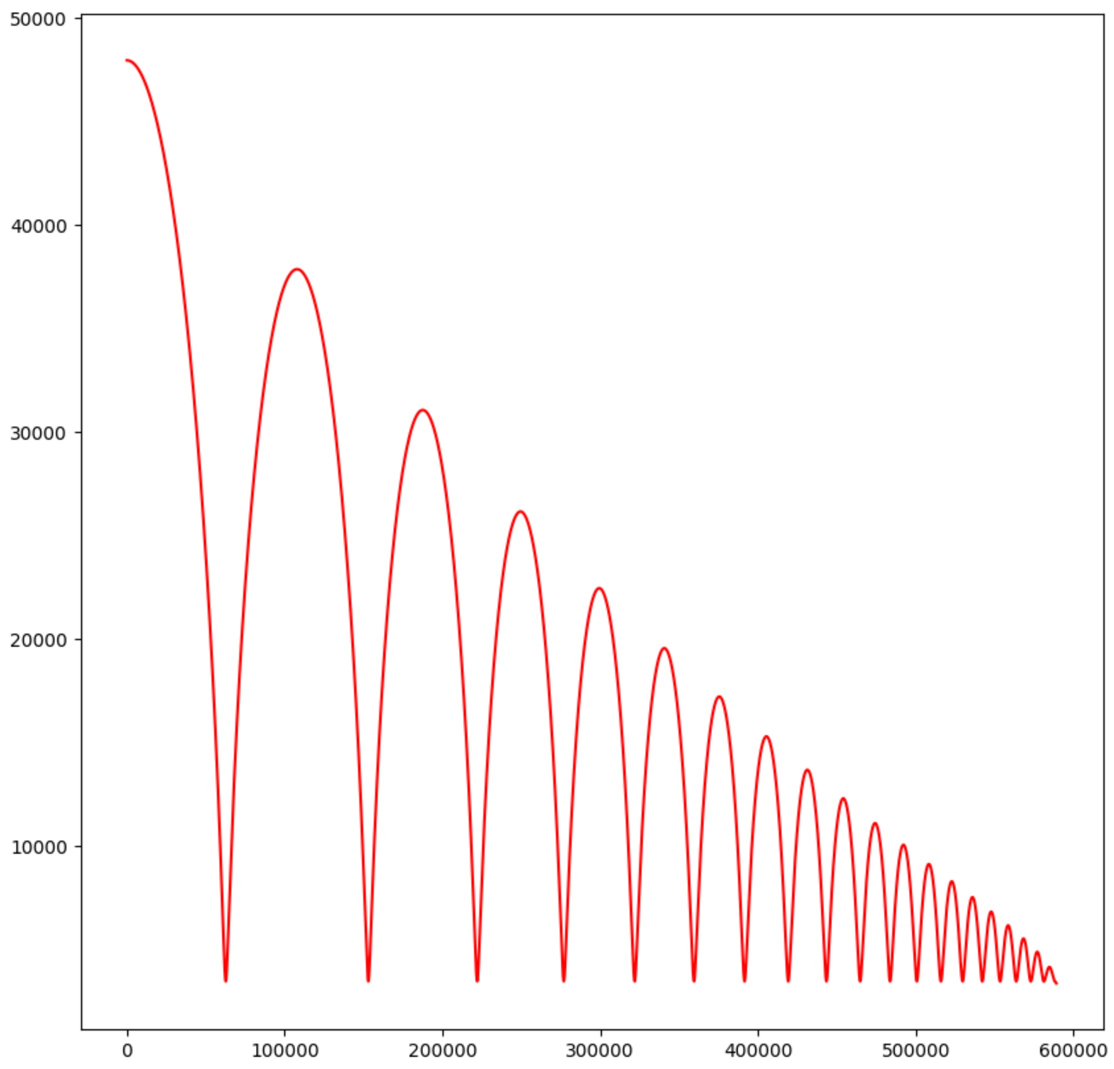
```
t_end = len(t) - 2
altitude = np.sqrt(posvel[t_end, 0] ** 2 + posvel[t_end, 1] ** 2 +␣
 ↪posvel[t_end, 2] ** 2) - mars.radius

# Plotting 2D trajectory in orbit plane
fig = plt.figure(figsize=(10, 10)) # create figure, figsize can be changed as␣
 ↪preferred
ax = fig.add_subplot(111)
ax.set_xlim(-10_000, 10_000)
ax.set_ylim(-10_000, 10_000)
ax.plot(posvel[:, 0] /(np.cos(30 * np.pi / 130) * 1e3), posvel[:, 1] / 1e3,␣
 ↪color='red')

# Plotting Mars
uang = np.linspace(0, 2 * np.pi, 100)
x = (mars.radius / 1e3) * np.cos(uang)
y = (mars.radius / 1e3) * np.sin(uang)
ax.plot(x, y, color='blue')
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')

# inset axes....
window_size = 1_000_000
projected_x = posvel[t_end, 0] / np.cos(30 * np.pi / 130)
x1, x2, y1, y2 = projected_x - 1_000_000/2, projected_x + 1_000_000/2,␣
 ↪posvel[t_end, 1] - 1_000_000/2, posvel[t_end, 1] + 1_000_000/2  # subregion␣
 ↪of the original image
axins = ax.inset_axes(
    [0, 0.6, 0.4, 0.4],
    xlim=(x1 / 1e3, x2 / 1e3), ylim=(y1 / 1e3, y2 / 1e3), xticklabels=[],␣
 ↪yticklabels=[])
axins.plot(posvel[:, 0] /(np.cos(30 * np.pi / 130) * 1e3), posvel[:, 1] / 1e3,␣
 ↪color='red')
axins.plot(x, y, color='blue')

ax.indicate_inset_zoom(axins, edgecolor="black")

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



With the above graph we can view the probe colliding with the surface of Mars. To get the semi-major axis and the eccentricity of the last complete orbit, we can measure the apoapsis on the graph.

To be more precise, here, we are going to take another approach by detecting the last apoapsis and using this value for our final measure.

[43]:
```
x, y = posvel[:, 0] / np.cos(30 * np.pi / 130), posvel[:, 1]
distance = np.sqrt(x ** 2 + y ** 2)

plt.figure(figsize=(10, 10))
plt.plot(t, distance / 1e3, color='red')
plt.show()
```



We verify that we are really close to the ground at the end of the simulation to ensure that our results are consistent.

```
[44]: (distance[t_end] - mars.radius) # final altitude in m
```

```
[44]: 139.51565441163257
```

```
[45]: from scipy.signal import argrelextrema
      from numpy import greater

      idx = argrelextrema(distance, greater)
      apoapsis_of_all_orbits = distance[idx[0]]
      last_apoapsis = apoapsis_of_all_orbits[-1]
      (last_apoapsis - mars.radius) / 1e3 # apoapsis of the last orbit in km
```

```
[45]: 793.4118939975207
```

Knowing that the periapsis is still at 100km of altitude we have the following values:

```
[46]: final_rp = 100e3 + mars.radius
      final_ra = last_apoapsis

      final_a = (final_rp + final_ra) / 2
      final_a / 1e3 # semi-major axis in km
```

```
[46]: 3836.2059469987603
```

```
[47]: e = (final_ra - final_rp) / (final_ra + final_rp)
      e # eccentricity
```

```
[47]: 0.09037730293651318
```

The eccentricity of the final orbit is close to zero as our orbit is way more circular (0 means a perfectly circular orbit).

## 1 Ex 8: Transfer Orbit

```
[48]: import matplotlib.pyplot as plt
      from math import pi

      from space_base import GravBody, Probe
      import numpy as np

      # Define constants
      G = 6.67e-11  # Gravitational constant
      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
```

We must first calculate the Homman transfer orbit by using the perihelion and the aphelion constraint we have. Using the following equations:

$$a = \frac{r_{perihelion} + r_{aphelion}}{2}$$

$$e = \frac{r_{perihelion} - r_{aphelion}}{r_{perihelion} + r_{aphelion}}$$

[49]:
```
r_per = AU_to_meters(1)  # Perihelion distance = Earth-Sun distance
r_aph = AU_to_meters(1.524)  # Aphelion distance = Mars-Sun distance

a = (r_per + r_aph) / 2  # Semi-major axis
a / 1.496e11  # Semi-major axis in UA
```

[49]:
```
1.262
```

[50]:
```
e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
e # Eccentricity
```

[50]:
```
0.2076069730586371
```

We could calculate the velocity at a given coordinate using the orbital energy formula:

$$-\frac{GM_{sun}}{2a} = \frac{1}{2}V^2 - \frac{GM_{sun}}{r}$$
$$V = \sqrt{GM_{sun}(\frac{2}{r} - \frac{1}{a})}$$

[51]:
```
v_per = np.sqrt(G * sun.mass * (2 / r_per - 1 / a))  # Velocity at perihelion
v_per / 1e3  # Velocity at perihelion in km/s
```

[51]:
```
32.72071038681002
```

The full period of an elliptical orbit can be deduced from Kepler's third law of planetary motion, which gives us this:

$$P = \sqrt{\frac{4\pi^2}{GM_{sun}}a^3}$$

[52]:
```
period = np.sqrt(4 * pi**2 * a**3 / (G * sun.mass)) / 2  # Orbital period
period / (24 * 3600)  # Half orbital period in days
```

[52]:
```
258.9982551297217
```

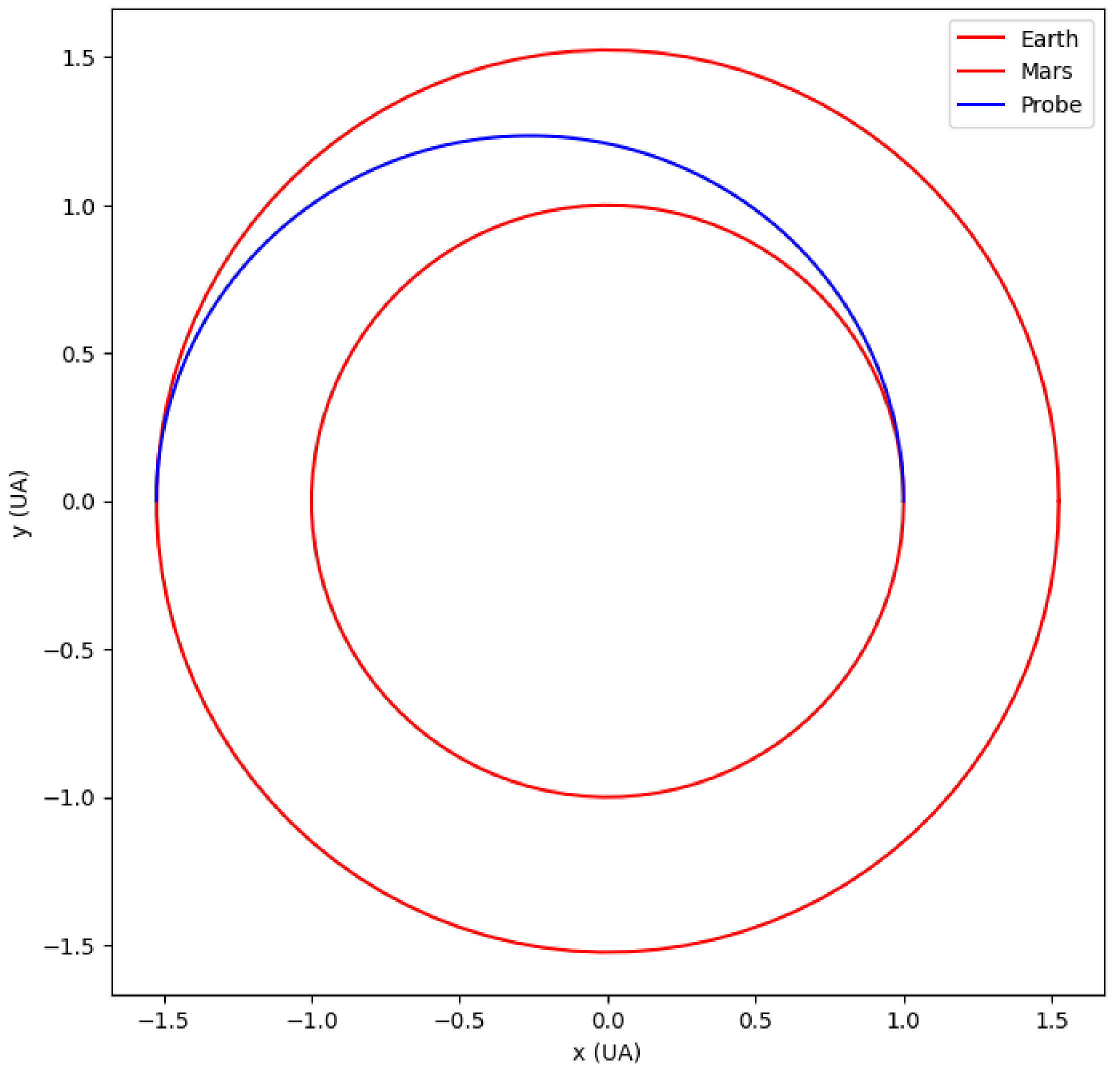
Here, we calculate the trajectory of the spacecraft with the Sun as the primary interaction body.

[53]:
```
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]
```

We can then simulate the probe as usual.

```
[54]: xy0 = [r_per, 0]  # start position
      vxy0 = [0, v_per]  # start vertical speed

      probe = Probe(probeqns, period, period / 3600, x0=xy0[0], vx0=vxy0[0],
                  y0=xy0[1], vy0=vxy0[1], event=r_aph, eventflip=True) # probe as an␣
       ↪object
      t, posvel = probe.odesolve() # solve the differential equations
```

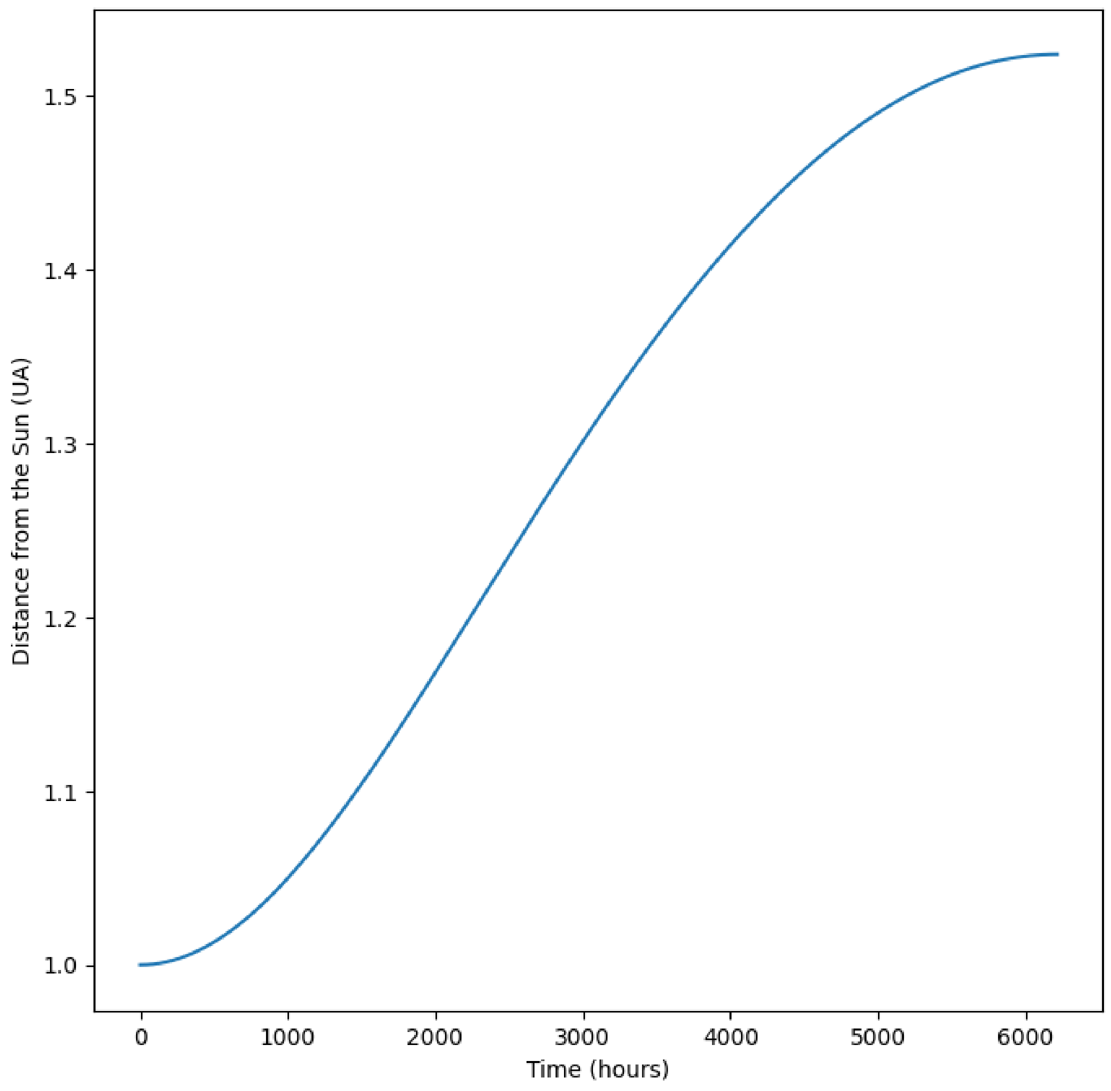
```
[55]: plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

      # Plotting Earth's and Mars' orbit
      for dist in [1, 1.524]:
          uang = np.linspace(0, 2 * pi, 100)
          x = dist * np.cos(uang)
          y = dist * np.sin(uang)
          plt.plot(x, y, color='red')

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

      plt.xlabel('x (UA)')
      plt.ylabel('y (UA)')
      plt.axis('equal')
      plt.legend(['Earth', 'Mars', 'Probe'])
      plt.show() # make plot appear
```



We check that the constraints are met, pherihelion and aphelion equal to 1UA and 1.524UA respectively.

```
[56]: plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred
      r = meters_to_AU(np.sqrt(posvel[:, 0] ** 2 + posvel[:, 1] ** 2)) # distance␣
        ↪from the Sun in UA
      print(f"Minimum distance : {min(r)}UA & Maximum distance : {max(r)}UA")
      plt.plot(t / 3600, r)
      plt.xlabel('Time (hours)')
      plt.ylabel('Distance from the Sun (UA)')
      plt.show()
```

```
Minimum distance : 1.0UA & Maximum distance : 1.5240000532591178UA
```

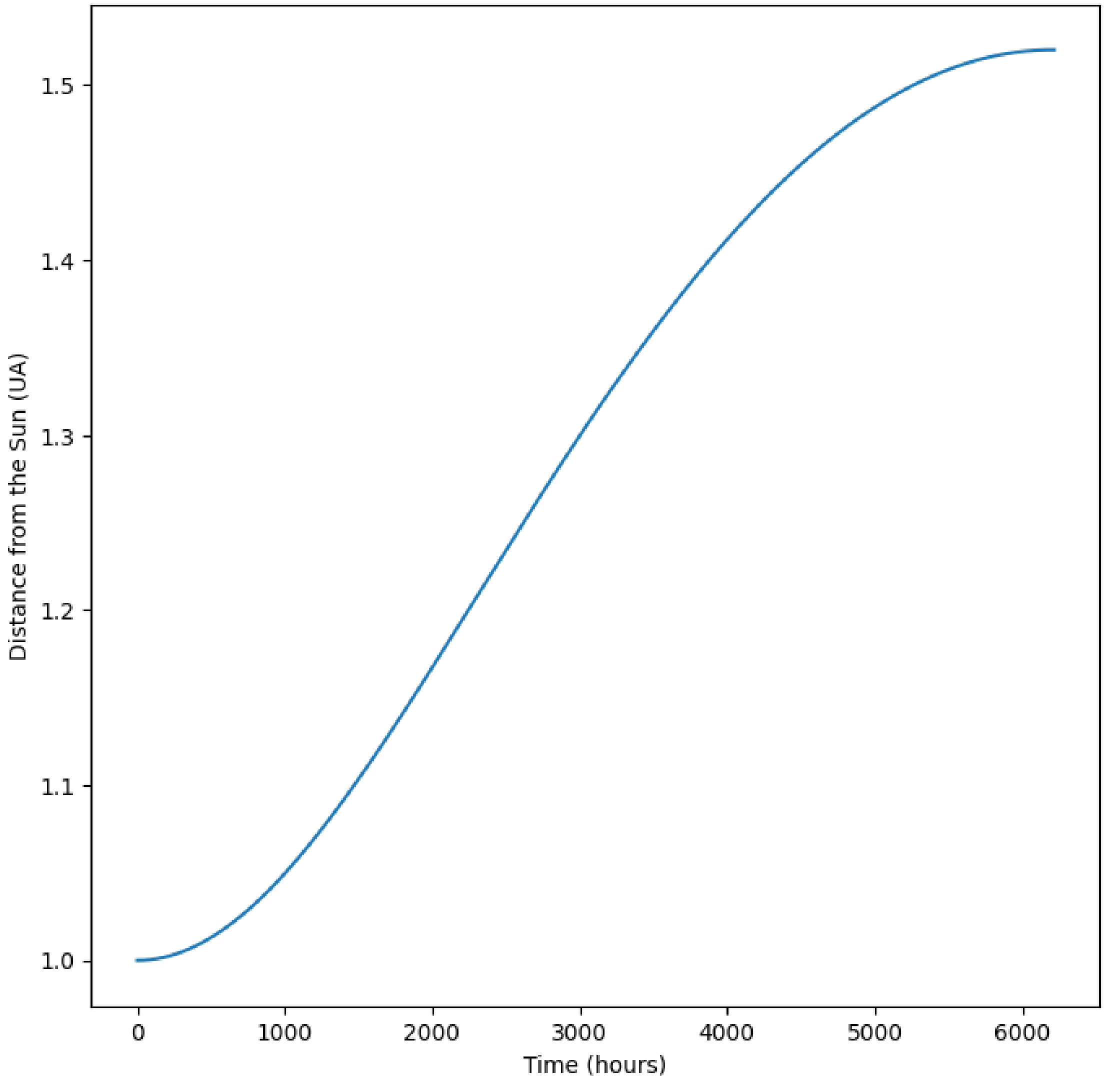


By reducing the initial velocity to 99.95% of the required velocity to reach Mars, we are quite far from the planet in absolute terms.

```
[57]: xy0 = [r_per, 0]  # start position
vxy0 = [0, v_per * 0.9995]  # start vertical speed

probe = Probe(probeqns, period, period / 3600, x0=xy0[0], vx0=vxy0[0],
             y0=xy0[1], vy0=vxy0[1], event=r_per) # probe as an object
t, posvel = probe.odesolve() # solve the differential equations

plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred
r = np.sqrt(posvel[:, 0] ** 2 + posvel[:, 1] ** 2) / 1.496e11 # distance from␣
  ↪the Sun in UA
print(f"Minimum distance : {min(r)}UA & Maximum distance : {max(r)}UA")
```

```
plt.plot(t / 3600, r)
plt.xlabel('Time (hours)')
plt.ylabel('Distance from the Sun (UA)')
plt.show()
```

```
Minimum distance : 1.0UA & Maximum distance : 1.5201603801952317UA
```



[58]:
```
AU_to_meters(1.524 - max(r)) / 1e3 # Distance from Mars in km
```

[58]:
```
574407.1227933413
```

### 1.1 Asteroid $2013LA_2$ analyze

Now we will study the transfer from Earth to asteroid $2013LA_2$.

[59]:
```
ast_e = 0.4656 # Eccentricity of asteroid
ast_a = AU_to_meters(5.6841) # Semi-major axis of asteroid

ast_per = ast_a * (1 - ast_e) # Perihelion distance of asteroid
ast_aph = ast_a * (1 + ast_e) # Aphelion distance of asteroid
```

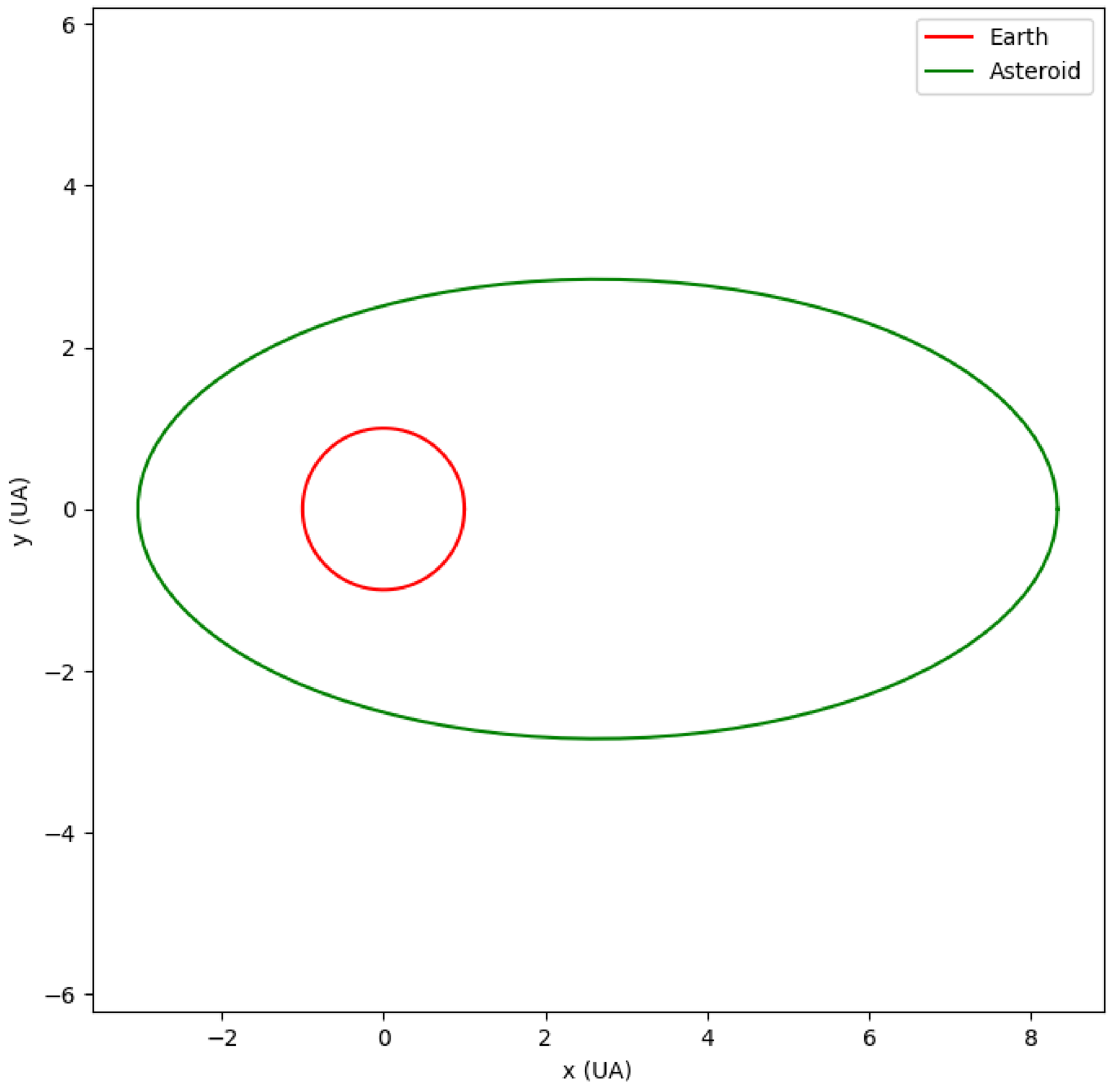
[60]:
```
ast_per / 1e3 # Perihelion distance of asteroid in km
```

[60]:
```
454422422.784
```

[61]:
```
ast_aph / 1e3 # Aphelion distance of asteroid in km
```

[61]:
```
1246260297.216
```

[62]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

# Plotting Earth's orbit
uang = np.linspace(0, 2 * pi, 100)
x = np.cos(uang)
y = np.sin(uang)
plt.plot(x, y, color='red')

# Plotting Asteroid's orbit
u=meters_to_AU(ast_a - ast_per)
v=0
a=meters_to_AU(ast_a)
b=a / 2
t = np.linspace(0, 2*pi, 100)
x = u+a*np.cos(t)
y = v+b*np.sin(t)
plt.plot(x, y, color='green')

plt.xlabel('x (UA)')
plt.ylabel('y (UA)')
plt.axis('equal')
plt.legend(['Earth', 'Asteroid'])
plt.show() # make plot appear
```

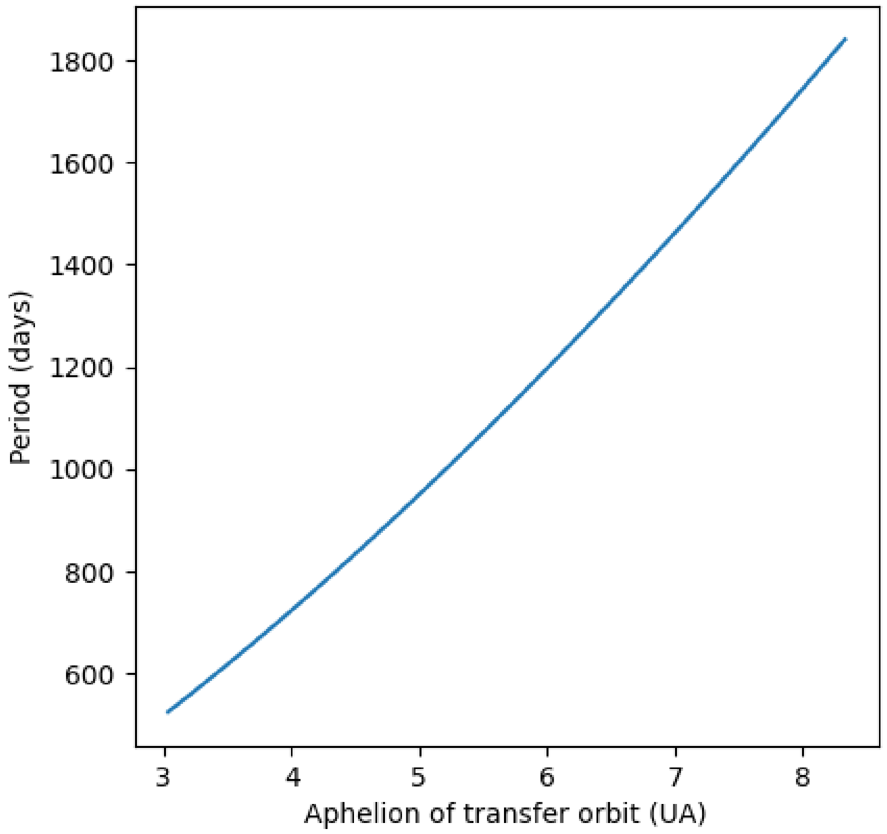


### 1.1.1 Least transit time

First, let's find the transfer that will take the least time. Looking at the graph above, we can think that this orbit will be from the Earth's orbit to the asteroid's perihelion, as this is the closest point the asteroid can get.

We can confirm this by analysing the period formula: $P = \sqrt{\frac{4\pi^2}{GM_{sun}} a^3}$ with $a = \frac{r_{perihelion} + r_{aphelion}}{2}$

[63]:
```
ast_alts = np.linspace(ast_per, ast_aph, 10000)
a = (ast_alts + AU_to_meters(1)) / 2

plt.figure(figsize=(5, 5))  # create figure, figsize can be changed as preferred
p = np.sqrt((4 * pi**2 * a**3)/(G * sun.mass))/2
plt.plot(ast_alts / 1.496e11, p / (24*3600))
plt.xlabel('Aphelion of transfer orbit (UA)')
plt.ylabel('Period (days)')
plt.show()
```



As we can see the further the aphelion the longer the orbital period, so for this transfer to be the fastest we will set our perihelion to the radius of the Earth's orbit and the aphelion to the asteroid's perihelion.

[64]:
```
r_per = AU_to_meters(1)  # Perihelion distance = Earth-Sun distance
r_aph = ast_per  # Aphelion distance = Asteroid Perihelion distance

a = (r_per + r_aph) / 2  # Semi-major axis
a / 1.496e11  # Semi-major axis in UA
```

[64]:
```
2.01879152
```

[65]:
```
e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
e # Eccentricity
```

[65]:
```
0.5046541507168606
```

We can check what are the values of the period and the differences in speed at arrival.

```
[66]: period = np.sqrt((4 * pi**2 * a**3)/(G * sun.mass))/2
      period / (24 * 3600)  # Transit time in days
```

```
[66]: 524.0165621549162
```

```
[67]: ast_vel = np.sqrt(G * sun.mass * (2 / r_aph - 1 / ast_a))  # Velocity of␣
       ↪asteroid
      vel = np.sqrt(G * sun.mass * (2 / r_aph - 1 / a))  # Velocity of probe
      abs(vel + ast_vel) / 1e3 # Difference in velocity in km/s (-vel is negative␣
       ↪because it is in the opposite direction)
```

```
[67]: 32.70655681694997
```

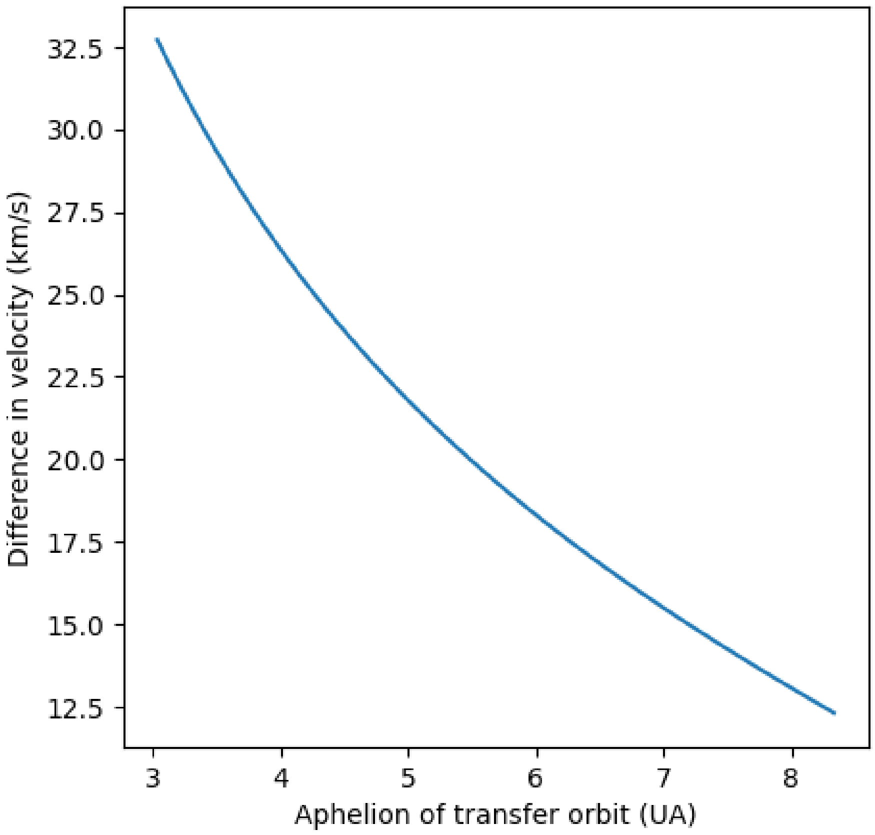
```
[68]: xy0 = [r_per, 0]  # start position
      vxy0 = [0, np.sqrt(G * sun.mass * (2 / r_per - 1 / a))]  # start vertical speed

      probe = Probe(probeqns, period, period / 3600, x0=xy0[0], vx0=vxy0[0],
                   y0=xy0[1], vy0=vxy0[1], event=r_aph, eventflip=True) # probe as an␣
       ↪object
      t, quickest_posvel = probe.odesolve() # solve the differential equations
```

### 1.1.2 Smallest burn on arrival

For the second orbit, we are looking for the smallest burn on arrival, which means we want to have the smallest difference between the velocity of the asteroid and that of the spacecraft.

Firstly, we can imagine that we should target the asteroid's aphelion, as this is where its velocity is lowest. To confirm this, we could plot the difference in velocity between the two using the energy formula.

```
[69]: ast_alts = np.linspace(ast_per, ast_aph, 10000)
      ast_vel = np.sqrt(G * sun.mass * (2 / ast_alts - 1 / ast_a))  # Velocity of␣
       ↪asteroid
      a = (ast_alts + AU_to_meters(1)) / 2
      vel = np.sqrt(G * sun.mass * (2 / ast_alts - 1 / a))  # Velocity of probe
      diff = abs(vel + ast_vel)

      plt.figure(figsize=(5, 5))  # create figure, figsize can be changed as preferred
      plt.plot(ast_alts / 1.496e11, diff / 1e3)
      plt.xlabel('Aphelion of transfer orbit (UA)')
      plt.ylabel('Difference in velocity (km/s)')
      plt.show()
```



We can clearly see that the further away we are from the asteroid, the smaller the difference in speed. So for this orbit we will set our perihelion to the distance between the Sun and the Earth and our aphelion to the asteroid's aphelion.

```
[70]: r_per = AU_to_meters(1)  # Perihelion distance = Earth-Sun distance
      r_aph = ast_aph  # Aphelion distance = Asteroid Perihelion distance

      a = (r_per + r_aph) / 2  # Semi-major axis
      a / 1.496e11  # Semi-major axis in UA
```

```
[70]: 4.66530848
```

```
[71]: e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
      e # Eccentricity
```

```
[71]: 0.7856519018437983
```

We can use the graph to check that the final value is what we expected.

```
[72]: ast_vel = np.sqrt(G * sun.mass * (2 / r_aph - 1 / ast_a))  # Velocity of␣
       ↪asteroid
```

```
vel = np.sqrt(G * sun.mass * (2 / r_aph - 1 / a))  # Velocity of probe
abs(vel + ast_vel) / 1e3 # Difference in velocity in km/s
```

```
[72]: 12.31762997981529
```

```
[73]: period = np.sqrt((4 * pi**2 * a**3)/(G * sun.mass))/2
      period / (24 * 3600)  # Transit time in days
```

```
[73]: 1840.892185350244
```

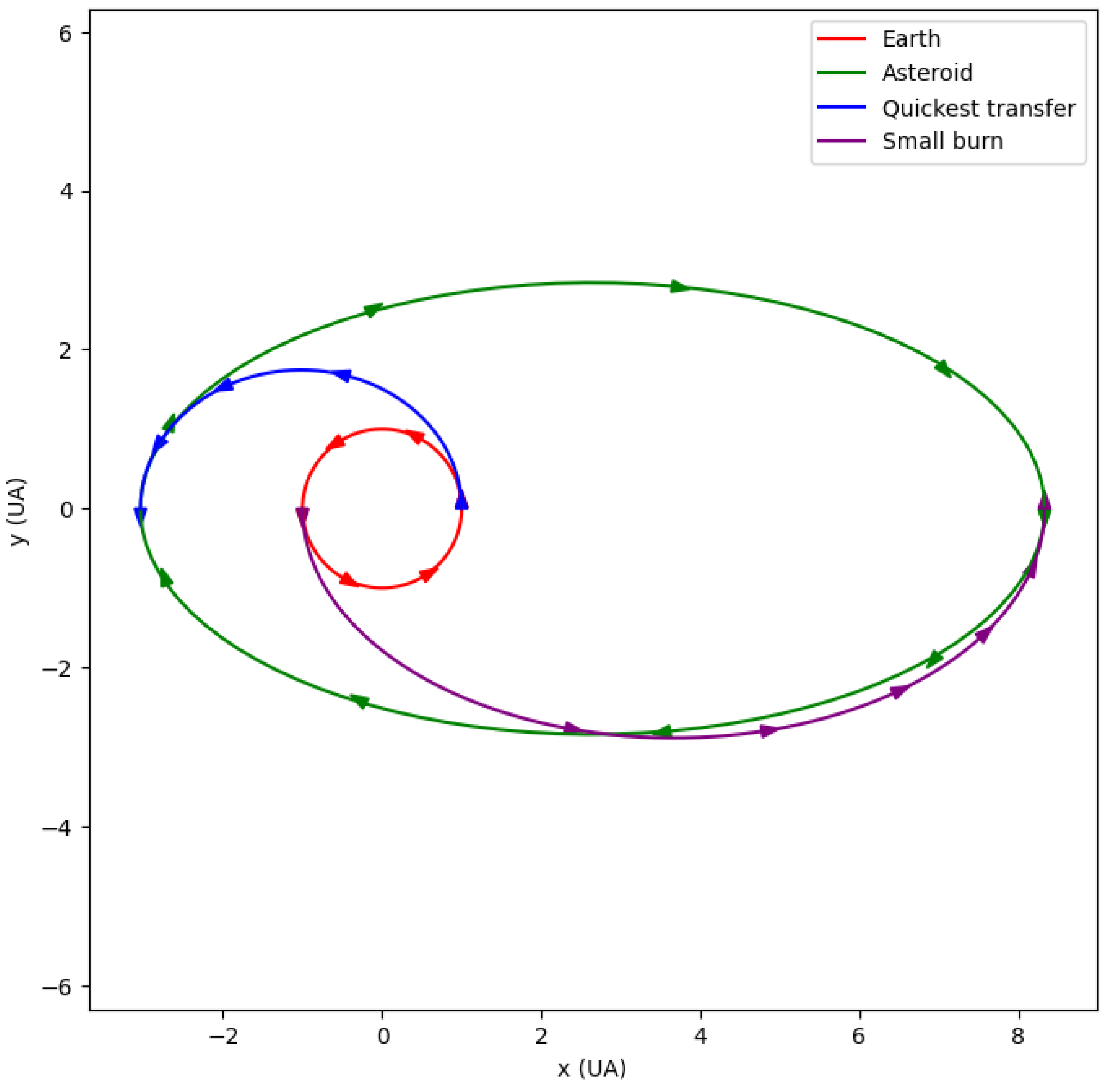
```
[74]: xy0 = [-r_per, 0]  # start position
      vxy0 = [0, -np.sqrt(G * sun.mass * (2 / r_per - 1 / a))]  # start vertical speed

      probe = Probe(probeqns, period, period / 3600, x0=xy0[0], vx0=vxy0[0],
                    y0=xy0[1], vy0=vxy0[1], event=r_aph, eventflip=True) # probe as an␣
       ↪object
      t, smallburn_posvel = probe.odesolve() # solve the differential equations
```

### 1.1.3 Final plot

Finally, we can see the two trajectories to visually confirm our simulation. It is important to understand that burn on arrival and orbit period are inversely related and we have to make a trade-off, as we cannot lower one without raising the other and vice versa.

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

      # Plotting Earth's orbit
      uang = np.linspace(0, 2 * pi, 100)
      x = np.cos(uang)
      y = np.sin(uang)
      plt.plot(x, y, color='red', label='Earth')
      ## Add arrow to show direction of orbit
      uang = np.linspace(0, 2 * pi, 7)
      xs = np.cos(uang)
      ys = np.sin(uang)
      dxs = -0.01*np.sin(uang)
      dys = 0.01*np.cos(uang)
      for (x, y, dx, dy) in zip(xs, ys, dxs, dys):
          plt.arrow(x, y, dx, dy, head_width=0.15, color='red')

      # Plotting Asteroid's orbit
      u=meters_to_AU(ast_a - ast_per)
      v=0
      a= meters_to_AU(ast_a)
      b=a / 2
      t = np.linspace(0, 2*pi, 100)
      x = u+a*np.cos(t)
      y = v+b*np.sin(t)
```

```
plt.plot(x, y, color='green', label='Asteroid')
## Add arrow to show direction of orbit
uang = np.linspace(0, 2 * pi, 7)
t = np.linspace(0, 2*pi, 10)
xs = u+a*np.cos(t)
ys = v+b*np.sin(t)
dxs = 0.01*np.sin(t)
dys = -0.01*np.cos(t)
for (x, y, dx, dy) in zip(xs, ys, dxs, dys):
    plt.arrow(x, y, dx, dy, head_width=0.15, color='green')

# Plotting probe's orbit with quickest transfer
plt.plot(meters_to_AU(quickest_posvel[:, 0]), meters_to_AU(quickest_posvel[:,␣
 ↪1]), color='blue', label='Quickest transfer')
idx = np.linspace(0, len(quickest_posvel)-1, 5, dtype=int)
for (x, y, dx, dy) in zip(meters_to_AU(quickest_posvel[idx, 0]),␣
 ↪meters_to_AU(quickest_posvel[idx, 1]), meters_to_AU(quickest_posvel[idx,␣
 ↪2]), meters_to_AU(quickest_posvel[idx, 3])):
    plt.arrow(x, y, dx, dy, head_width=0.15, color='blue')

# Plotting probe's orbit with small burn
plt.plot(meters_to_AU(smallburn_posvel[:, 0]), meters_to_AU(smallburn_posvel[:,␣
 ↪1]), color='purple', label='Small burn')
idx = np.linspace(0, len(smallburn_posvel)-1, 7, dtype=int)
for (x, y, dx, dy) in zip(meters_to_AU(smallburn_posvel[idx, 0]),␣
 ↪meters_to_AU(smallburn_posvel[idx, 1]), meters_to_AU(smallburn_posvel[idx,␣
 ↪2]), meters_to_AU(smallburn_posvel[idx, 3])):
    plt.arrow(x, y, dx, dy, head_width=0.15, color='purple')

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



## 2 Ex 9: The Martian

[76]:
```
import matplotlib.pyplot as plt
from math import pi

from space_base import GravBody, Probe
import numpy as np

# Define constants
G = 6.67e-11  # Gravitational constant
mars = GravBody(name="Mars", mass=0.64169e24, radius=3389.5e3)  # Mars as an␣
 ↪object with mass and radius
```

```
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
```

The first step is to design a circular orbit at an altitude of 200km. To do this, we can use the previous energy formula and simplify it for a perfectly circular orbit.

$$a = \frac{r_{perihelion} + r_{aphelion}}{2}$$
$$r_{perihelion} = r_{aphelion} = r = a$$
$$V = \sqrt{GM_{mars}(\frac{2}{r} - \frac{1}{a})}$$
$$V = \sqrt{\frac{GM_{mars}}{r}}$$

[77]:
```
r = 200e3 + mars.radius  # Radius distance in meters
v_init = np.sqrt(G * mars.mass / r)  # Velocity on initial circular orbit
v_init / 1e3  # Velocity at perihelion in km/s
```

[77]:
```
3.4530953751079427
```

We also need to calculate the required velocity the spacecraft must have at its aphelion to return to Earth using a Hohmann transfer. This will be our required residual velocity.

[78]:
```
r_earth = AU_to_meters(1)  # Earth's radius in meters
r_mars = AU_to_meters(1.524)  # Mars' radius in meters

# Define the probe final orbit
r_per = r_earth  # Perihelion distance
r_aph = r_mars  # Aphelion distance

a_tfr = (r_per + r_aph) / 2  # Semi-major axis
v_hyp_abs = np.sqrt(G * sun.mass * (2 / r_aph - 1 / a_tfr))  # Absolute
  ↪velocity at aphelion (relative to the Sun)
v_hyp_abs / 1e3  # Residual velocity in km/s
```

[78]:
```
21.47028240604332
```

This velocity is relative to the Sun. As we are going to simulate the departure from Mars, we need to have the velocity relative to the planet.

[79]:
```
v_mars = np.sqrt(G * sun.mass / r_mars)  # Mars' velocity
v_hyp = v_hyp_abs - v_mars  # Hypothetical velocity at aphelion (relative to
  ↪Mars)
v_hyp / 1e3  # Velocity in km/s
```

```
[79]: -2.6491694858055017
```

To find the periapsis velocity of our spacecraft around Mars so that it can escape, we can use the energy formula. We already know that $E = \frac{1}{2}V^2 - \frac{GM_{mars}}{r}$, so if $r \to \infty; E = \frac{1}{2}V_{hyp}^2$.

So,

$$\frac{1}{2}V_{hyp}^2 = \frac{1}{2}V_{per}^2 - \frac{GM_{mars}}{r_{per}}$$

$$2 * (\frac{1}{2}V_{hyp}^2 + \frac{GM_{mars}}{r_{per}}) = V_{per}^2$$

$$V_{per} = \sqrt{V_{hyp}^2 + \frac{2GM_{mars}}{r_{per}}}$$

```
[80]: v_per = np.sqrt(v_hyp**2 + 2*G*mars.mass/r)
      v_per / 1e3   # Velocity at perihelion in km/s
```

```
[80]: 5.555702863158424
```

We can also calculate $a$ and $e$ for this escape orbit using:

$$a = \frac{GM_{mars}}{V_{hyp}^2}$$

$$e = \frac{r_p}{a} + 1$$

```
[81]: a = G*mars.mass / (v_hyp**2)   # Semi-major axis
      a / 1e3   # Semi-major axis in km
```

```
[81]: 6098.620611701382
```

```
[82]: e = 1 + (r / a)   # Eccentricity
      e # Eccentricity
```

```
[82]: 1.5885757171241068
```

We can also derive the $\beta$ angle with $\cos\beta = \frac{1}{e}$.

```
[83]: beta_angle=np.arccos(1/e)
      beta_angle * 180 / pi   # Beta angle in degrees
```

```
[83]: 50.98714749340521
```

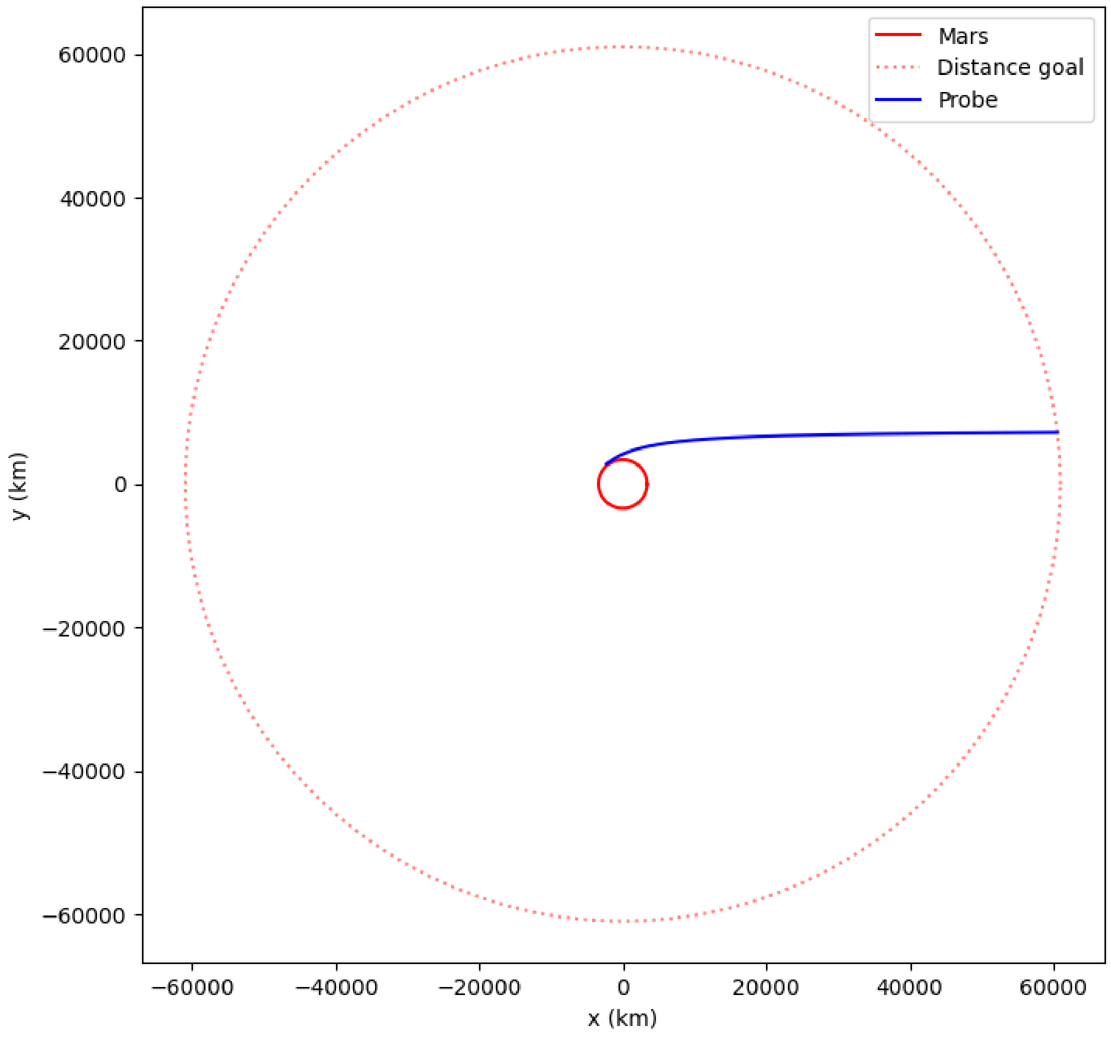
With these settings, we will fire our spacecraft in the opposite direction from Mars' orbit, as we will need to slow down to lower our perihelion to match Earth's orbit.

Finally, we can plot the trajectory until the probe reaches a point PM, which is up to 18 times the Martian radius from the centre of the planet. We can overestimate the time it will take the probe to reach this point by taking the lowest average speed (v_hyp) and using an event condition to trigger the end of the simulation at the right time.

[84]:
```python
def probeqns(_, posvel):
    r = np.sqrt(posvel[0] ** 2 + posvel[1] ** 2)
    f = -G * mars.mass / r ** 3
    gravity_force = f * posvel[0:2]
    axy = gravity_force

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

[85]:
```python
xy0 = [-r*np.cos(beta_angle), r*np.sin(beta_angle)]  # start position
vxy0 = [np.sin(beta_angle)*v_per, np.cos(beta_angle)*v_per]  # start vertical␣
 ↪speed
period = 18*mars.radius / abs(v_hyp)

probe = Probe(probeqns, period, period, x0=xy0[0], vx0=vxy0[0],
             y0=xy0[1], vy0=vxy0[1], event=mars.radius*18, eventflip=True) #␣
 ↪probe as an object
t, posvel = probe.odesolve() # solve the differential equations
```

[86]:
```python
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

# Plotting Mars
uang = np.linspace(0, 2 * pi, 100)
x = (mars.radius / 1e3) * np.cos(uang)
y = (mars.radius / 1e3) * np.sin(uang)
plt.plot(x, y, color='red', label='Mars')
# Distance from Mars wanted
x = (mars.radius*18 / 1e3) * np.cos(uang)
y = (mars.radius*18 / 1e3) * np.sin(uang)
plt.plot(x, y, color="red", linestyle=":", label='Distance goal', alpha=0.5)

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

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



As we can imagine, the real time of flight to the point is smaller.

```
[87]: from datetime import timedelta
      str(timedelta(seconds=t[len(t)-1])) # time of flight in hh:mm:ss
```

```
[87]: '5:17:08.021686'
```

```
[88]: from numpy.lib.stride_tricks import sliding_window_view
      pos_slice = sliding_window_view(posvel[:, 0:2], window_shape = [2, 2])
      total_dist = sum([ np.sqrt((pos[0, 0, 0] - pos[0, 1, 0])**2 +  (pos[0, 0, 1] -␣
       ↪pos[0, 1, 1])**2) for pos in pos_slice])
```

```
[89]: np.sqrt(posvel[len(t)-1, 0]**2 + posvel[len(t)-1, 1]**2)/ 1e3 # Direct distance␣
       ↪from Mars center in km
```

```
[89]: 61011.50753647896
```

As expected, the actual distance travelled is greater than the direct distance between the last point and the centre of Mars.

```
[90]: total_dist / 1e3 # Total distance in km
```

```
[90]: 63492.58205912211
```

By calculating the actual distance travelled by the spacecraft, we were able to determine its average speed during this journey.

```
[91]: (total_dist / t[len(t)-1]) / 1e3 # average speed in km/s
```

```
[91]: 3.3367936564991236
```

## 3 Ex 10: The moment of inertia tensor of a Space Station

Let's assume a space station define by the given modules:

| Module | Mass | x | y | z |
|---|---|---|---|---|
| Astrophysics module | 3 | 6 | -2 | 0 |
| Robotics repair plant | 4 | 3 | 1 | -2 |
| Power plant | 7 | -4 | 1 | 2 |
| Docking module | 5 | -4 | -2 | -4 |
| Communications module | 4 | 3 | 2 | -1 |
| Solar arrays | 3 | 2 | -1 | 6 |

We can then calculate the total mass, center of mass and inertia matrix of the station.

```
[92]: import numpy as np
from numpy.linalg import eig

modules = np.array([[3,  6, -2,  0],
                    [4,  3,  1, -2],
                    [7, -4,  1,  2],
                    [5, -4, -2, -4],
                    [4,  3,  2, -1],
                    [3,  2, -1,  6]])

total_mass = np.sum(modules, axis=0)[0]
total_mass
```

```
[92]: 26
```

The center of mass of the space station can be find using the mass and position off all the submodules:

$$x_{com} = \frac{\sum m_i * x_i}{m_{total}}$$
$$y_{com} = \frac{\sum m_i * y_i}{m_{total}}$$
$$z_{com} = \frac{\sum m_i * z_i}{m_{total}}$$

[93]:
```
center_of_mass = (modules[:, 1:].transpose() * modules[:, 0]).sum(axis=1) /␣
 ↪total_mass
center_of_mass
```

[93]:
```
array([0., 0., 0.])
```

Center of mass is at the origin of the coordinate system.

We can find the inertia matrix of the system with:

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

[94]:
```
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]])

for m, x, y, z in modules:
    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
```

[94]:
```
array([[298,  -6, -24],
       [ -6, 620, -20],
       [-24, -20, 446]])
```

All symmetrical inertia matrices can be diagonalised using eig vectors as the new base and the moments of inertia for these primary directions will be the corresponding eig values.

```
[95]: w,v=eig(inertia)
      print('E-value:', w)
      print('E-vector', v)
```

```
E-value: [293.95245163 447.74525937 622.302289  ]
E-vector [[-0.98681235 -0.16155017  0.01014485]
 [-0.02793964  0.10826326 -0.99372956]
 [-0.15943887  0.98090805  0.11134918]]
```

The eigenvalues and eigenvectors allow us to create a new set of axes, forming a basis, from which we can express the inertia matrix in diagonal form. Fortunately, since the inertia matrix was nearly diagonal to begin with, the eigenvectors are similar to the initial basis, making interpretation of the results easier.

Therefore, the space station exhibits the greatest inertia around the last eigenvector (primarily the y-axis) and the least inertia around the first eigenvector (primarily the x-axis). The station's stability is directly proportional to its inertia. Therefore, the greater the inertia, the more stable the station is when spinning around its axis

## 3.1 New Solar arrays mass

Now, let's update the modules tables with the below values and find the new center of mass and inertia matrix.

| Module | Mass | x | y | z |
|---|---|---|---|---|
| Astrophysics module | 3 | 6 | -2 | 0 |
| Robotics repair plant | 4 | 3 | 1 | -2 |
| Power plant | 7 | -4 | 1 | 2 |
| Docking module | 5 | -4 | -2 | -4 |
| Communications module | 4 | 3 | 2 | -1 |
| Solar arrays | 5 | 2 | -1 | 6 |

```
[96]: modules = np.array([[3,  6, -2,  0],
                          [4,  3,  1, -2],
                          [7, -4,  1,  2],
                          [5, -4, -2, -4],
                          [4,  3,  2, -1],
                          [5,  2, -1,  6]])

      total_mass = np.sum(modules, axis=0)[0]
      total_mass
```

```
[96]: 28
```

```
[97]: center_of_mass = (modules[:, 1:].transpose() * modules[:, 0]).sum(axis=1) /␣
       ↪total_mass
      center_of_mass
```

[97]: 
```
array([ 0.14285714, -0.07142857,  0.42857143])
```

[98]:
```
inertia = np.array([[0, 0, 0],
                    [0, 0, 0],
                    [0, 0, 0]], dtype=float)

for m, x, y, z in modules:
    inertia += np.array([[inertia_moment(m, y-center_of_mass[1],␣
 ↪z-center_of_mass[2]), -inertia_product(m, x-center_of_mass[0],␣
 ↪y-center_of_mass[1]), -inertia_product(m, x-center_of_mass[0],␣
 ↪z-center_of_mass[2])],
                        [-inertia_product(m, x-center_of_mass[0],␣
 ↪y-center_of_mass[1]), inertia_moment(m, x-center_of_mass[0],␣
 ↪z-center_of_mass[2]), -inertia_product(m, y-center_of_mass[1],␣
 ↪z-center_of_mass[2])],
                        [-inertia_product(m, x-center_of_mass[0],␣
 ↪z-center_of_mass[2]), -inertia_product(m, y-center_of_mass[1],␣
 ↪z-center_of_mass[2]), inertia_moment(m, x-center_of_mass[0],␣
 ↪y-center_of_mass[1])]])

inertia
```

[98]:
```
array([[366.71428571,  -2.28571429, -46.28571429],
       [ -2.28571429, 694.28571429,  -8.85714286],
       [-46.28571429,  -8.85714286, 455.28571429]])
```

[99]:
```
w_new,v_new=eig(inertia)
print('E-value:', w_new)
print('E-vector', v_new)
```

```
E-value: [346.85094272 474.82024129 694.61453027]
E-vector [[-0.91912237 -0.39396808  0.00179457]
 [-0.0160819   0.03296681 -0.99932705]
 [-0.3936438   0.91853271  0.03663629]]
```

[100]:
```
w_new - w
```

[100]:
```
array([52.89849109, 27.07498192, 72.31224128])
```

As anticipated, the moment of inertia increases with the mass of the solar arrays, making the station more resistant to external perturbations. However, even a small change in mass can have a significant impact on the inertia of the entire system.

Additionally, since the solar panels are located close to the center on the xy plane, changes in mass will have a lesser effect on the z-axis compared to the other axes. These findings are supported by the new eigenvalues and eigenvectors.

## 4 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 |
| | | -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 | | |

[101]:
```python
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
```

[101]:
```
[-0.037037037037037035, 0.7962962962962963, -0.24074074074074073]
```

[102]:
```
list(center_of_mass) # Center of mass coordinates in the form [x, y, z] in␣
↪physical space
```

[102]:
```
[-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.

[103]:
```
list(np.mean(points_remap, axis=0)) # Should be [0, 0, 0]
```

[103]:
```
[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.

[104]:
```
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)
```

[104]:
```
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.

[105]:
```
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
```

[105]:
```
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]])
```

[106]:
```
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]
```

## 4.1 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:

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

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

## 5 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 $2\frac{E}{I_x} = 2\frac{E}{I_y} = 2\frac{E}{I_z} = 1$

[107]:
```
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],
[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
```

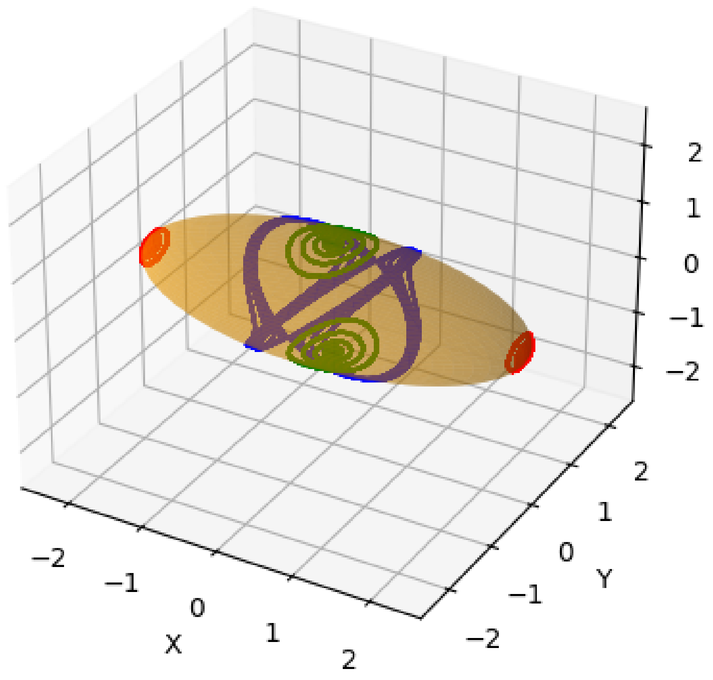
```
[107]: array([9.26067960e+18, 5.73866005e+19, 6.34453125e+19])
```

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.

```
[108]: 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)
        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%
```



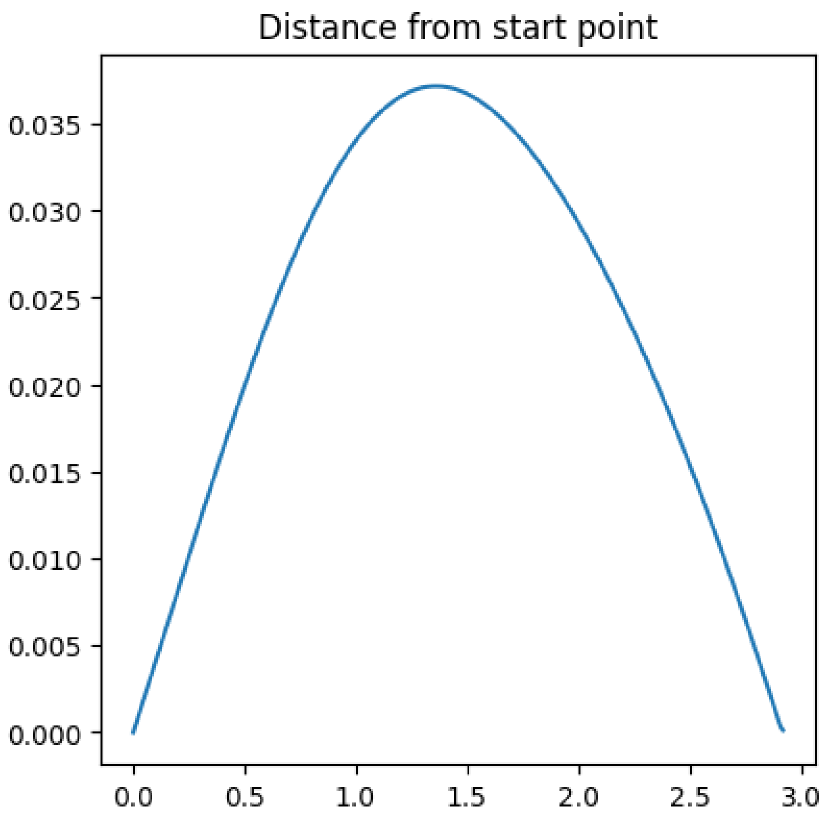
From the previous digram, it is clear that the most stable axis is the X axis and the less stable axis is the Y axis. Even really small rotation velocity around this axis will end up on high amplitude motion.

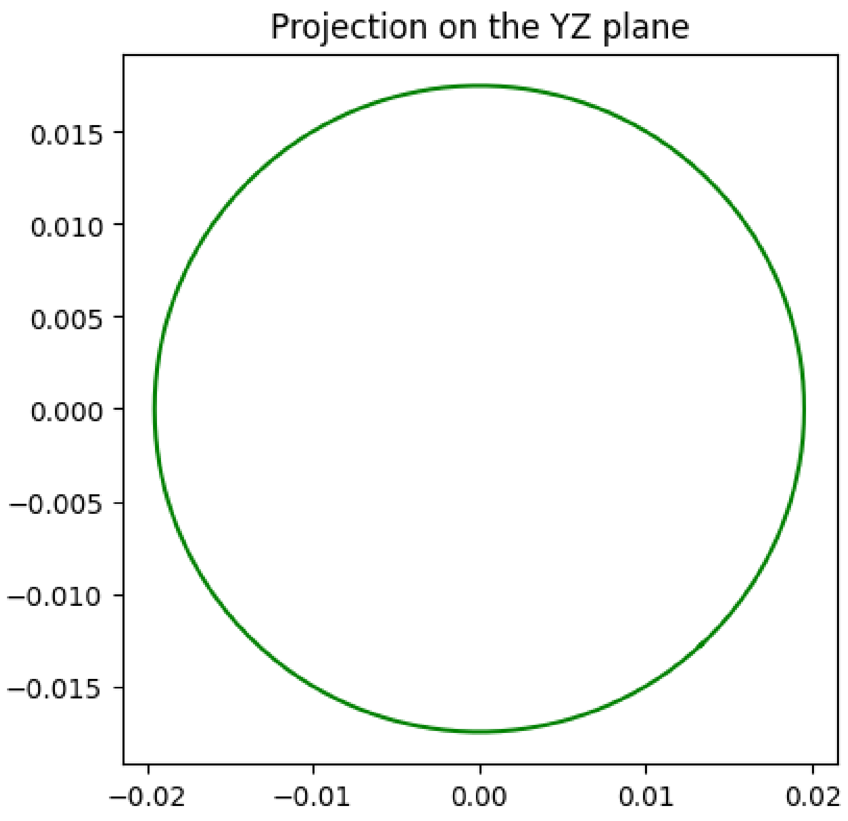
## 5.1 Period of motion

[109]:
```
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
```





[110]:
```
# 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
```

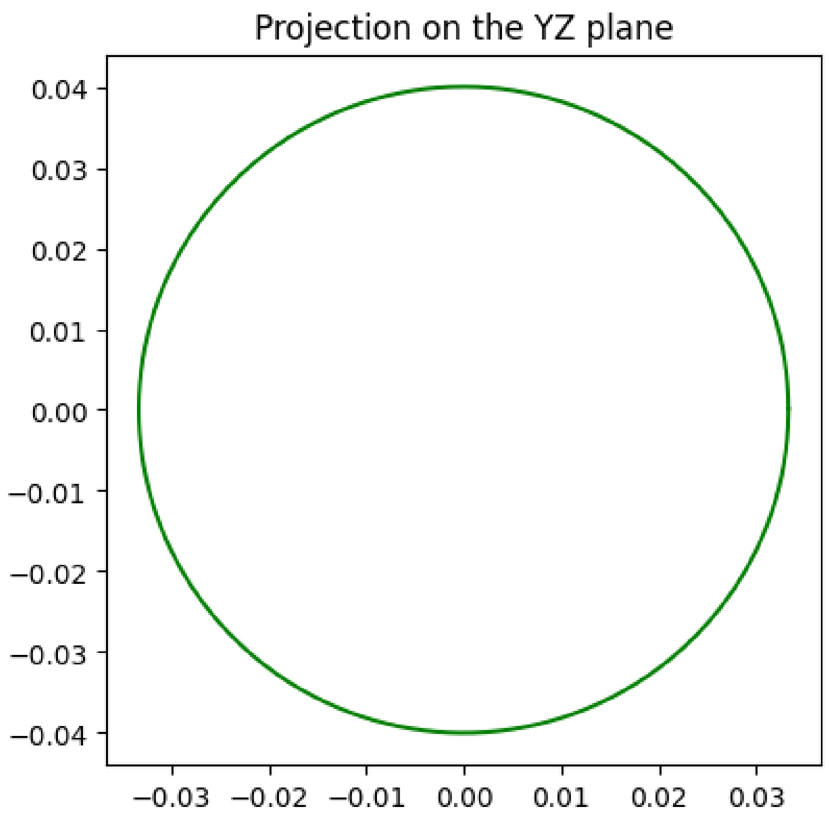
```
Period of motion: 3.0303030303030303
```

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.

[111]:
```
_, 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.

## 5.2 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.

### 5.2.1 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 $\Omega_y$ and $\Omega_z$ 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,

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

[112]:
```
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.

### 5.2.2 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 $\Omega_y$ and $\Omega_z$ 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,

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

[113]:
```
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.

## 6 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.

[114]:
```
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
```

### 6.1 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.

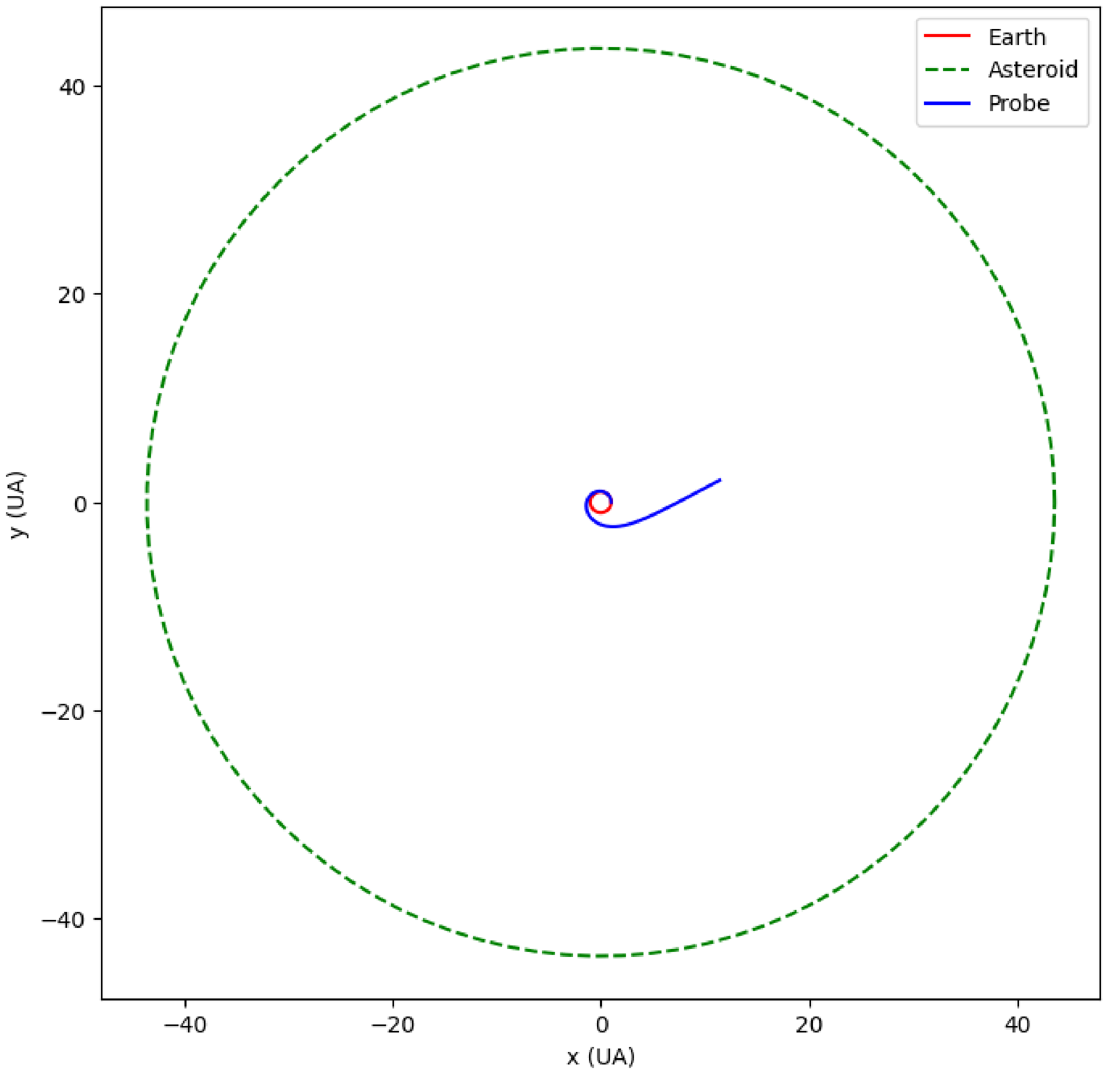
[115]:
```
mass_lost_rate = 10e-6 # kg/s
dry_mass = 300 # dry mass of the probe
def probeqns_rocket(_, posvelmass):
    Isp = 3400 # in seconds
    if posvelmass[2] <= dry_mass:
        posvelmass[5] = 0.0
    else:
        posvelmass[5] = -mass_lost_rate

    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.

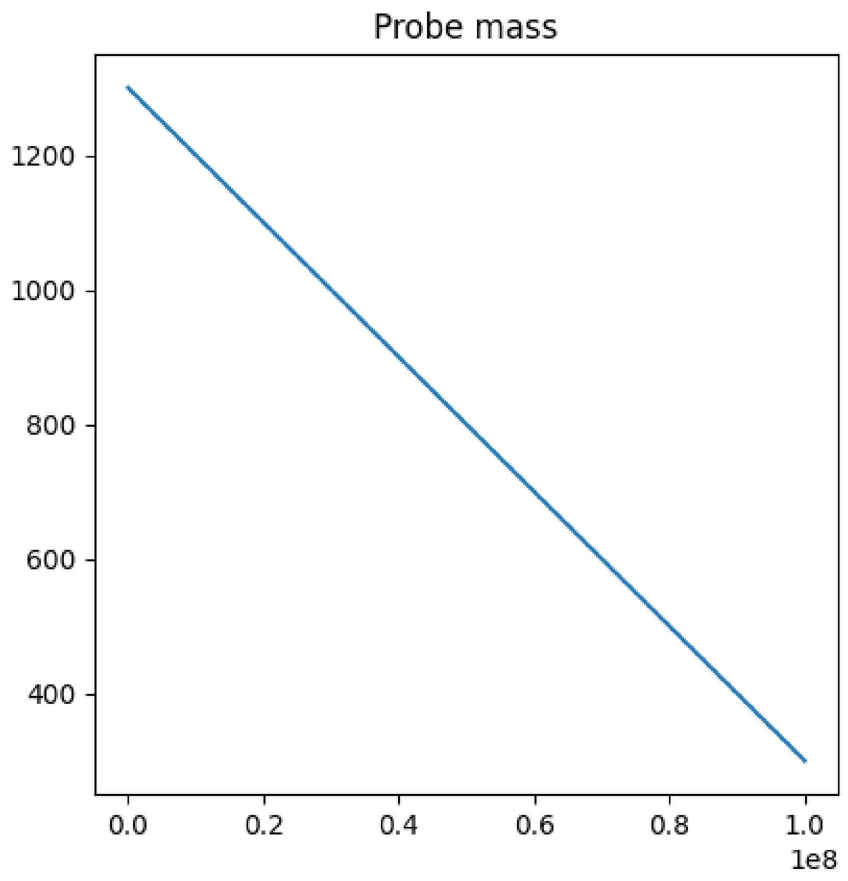
[116]:
```
fuel_mass = 1000 # kg
v0 = np.sqrt(G * sun.mass / AU_to_meters(1))  # initial speed
xymass0 = [AU_to_meters(1), 0, dry_mass + fuel_mass]  # start position
vxy0 = [0, v0]  # start vertical speed
tf = fuel_mass/mass_lost_rate  # Max burn time

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:

[117]:
```
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
```



[118]:
```
plt.figure(figsize=(5, 5))  # create figure, figsize can be changed as preferred
plt.title("Probe mass")
plt.plot(t, posvel[:, 2])
plt.show()
```



### 6.2 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.

#### 6.2.1 Minimum fuel mass

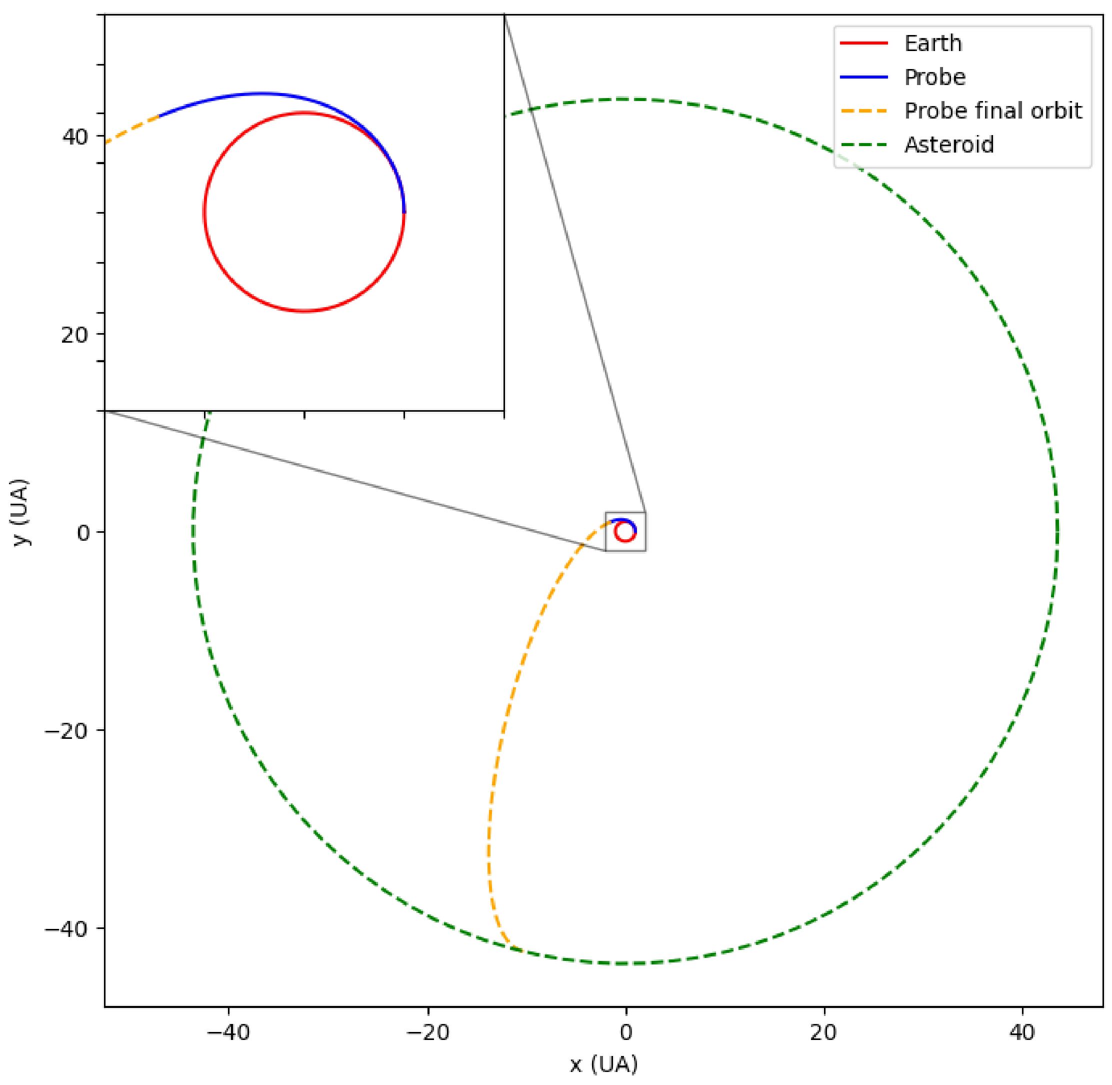
The first solution could be to try making the smallest burn possible so that after it the probe will settle in a ecliptical orbit with an apoapsis matching the asteroid's orbit. It is the burn that will use the less fuel because we only burn to raise our orbit high enough to reach the asteroid but not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

[119]:
```
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
```

[119]:
```
147.06252817521252
```

[120]:
```
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

# 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
```



```
[121]: 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
```

```
[121]: 43.59192394818552
```

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

```
[122]: 21.7959619740961
```

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

[123]: 0.9999999999996934

[124]:
```
(t_after[-1] + t[-1]) / (365*24*3600) # Total time to reach the asteroid in␣
 ↪years
```

[124]: 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.

### 6.2.2 Fastest journey

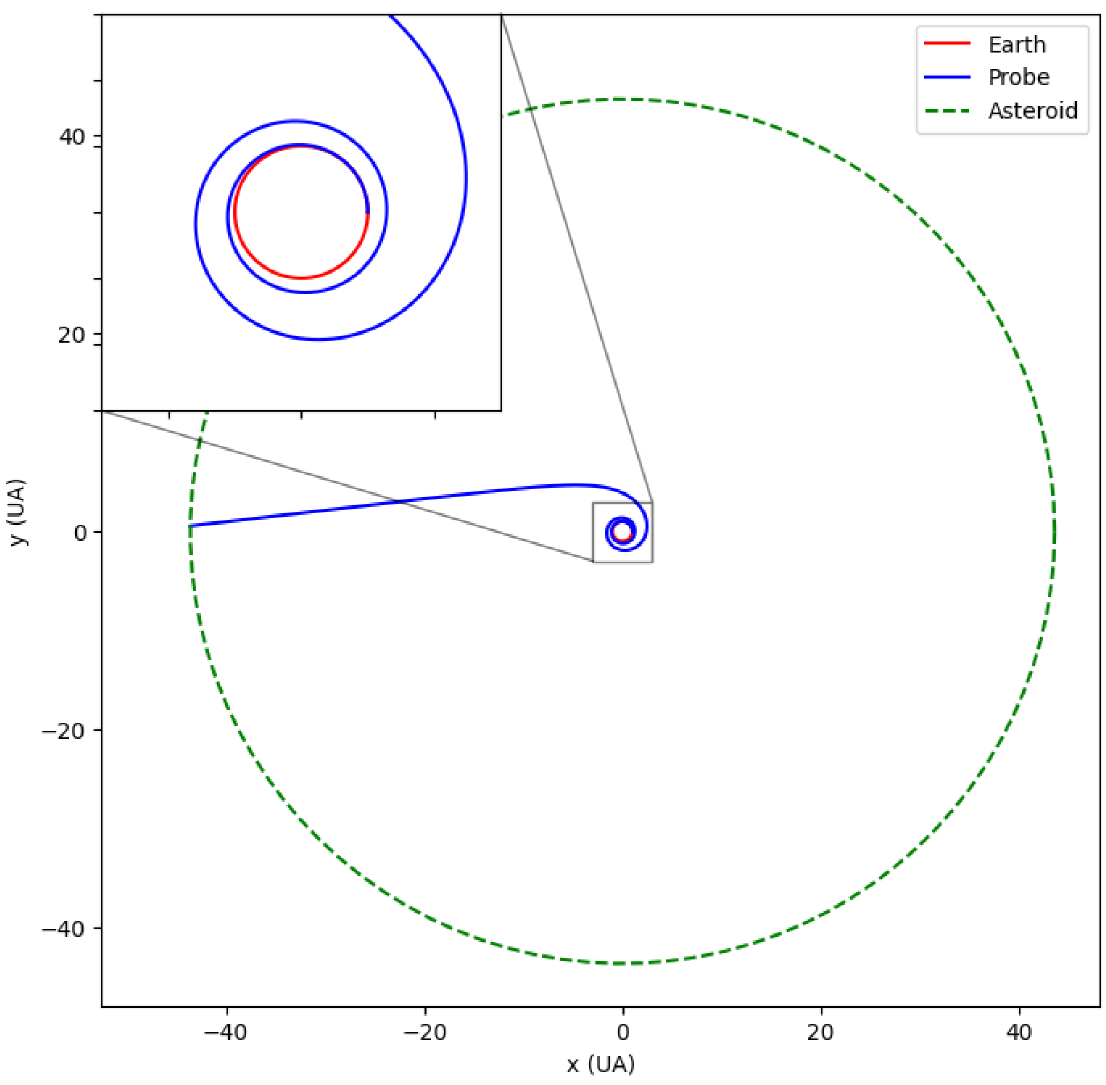
To address the travel time problem with the previous burn, it is possible to try another approach and end the burn only when the probe reaches the asteroid's orbit.

[125]:
```
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
```

[125]: 3399.801033121058

[126]:
```
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:

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

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

[127]:
```
v_final = np.linalg.norm(posvel[-1, 2:4])
v_final / 1e3 # Final speed of the probe in km/s (after burning all fuel)
```

```
[127]: 75.14672173003704
```

```
[128]: 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
```

```
[128]: -0.1581389242707825
```

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

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

```
[129]: 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
```

```
[129]: 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.

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

```
[130]: 6.377211973108335
```

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

```
[131]: 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.

## 7 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.

```
[132]: 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 $V_0 = 10.85 kms^{-1}$
- Altitude $z_0 = 300km$
- Angle $\psi_0 = 6°$ (angle between $\vec{V}$ and $\hat{\theta}$ ($\hat{\theta} \perp \vec{r}$))

```
[133]: 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 $\epsilon$ using:

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

And the specific angular momentum $h$,

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

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

[134]: -851797.5191125795
```

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

[135]: 71983.84286941901
```

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

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

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

```
[136]: 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,

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

```
[137]: 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
```

```
[137]: 0.971470304916528
```

Finally, the time of traveling from a point $(r_0, \theta_0)$ to another point $(r_0, \theta_1)$ 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.

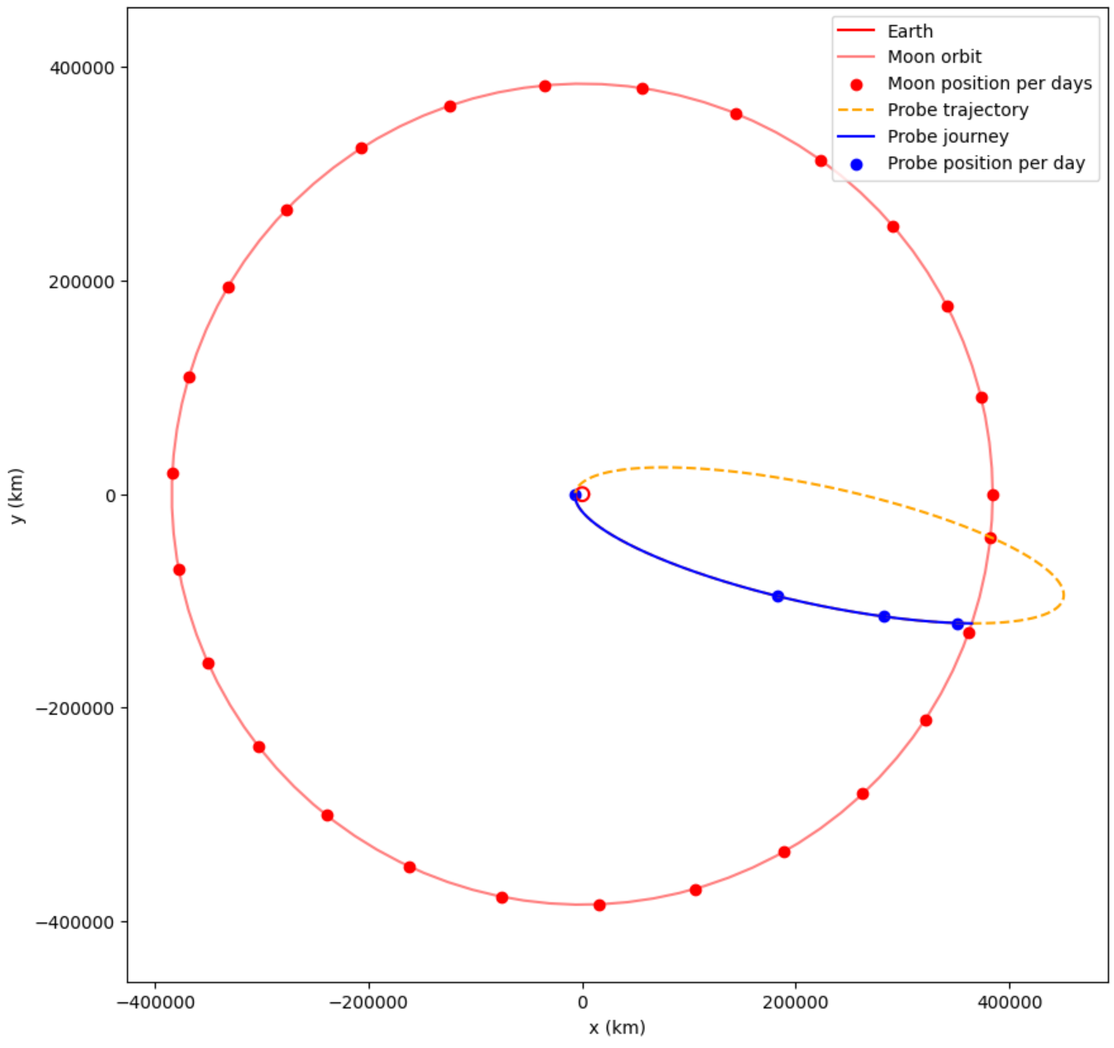
```
[138]: def flight_time(r):
           A = np.arcsin(-(G*earth.mass+2*energy0*r)/np.sqrt((G*earth.
        ↪mass)**2+2*energy0*h0**2)) + np.pi/2
           B = np.sqrt(2*energy0*(h0**2-2*G*earth.mass*r - 2*energy0*r**2))/(G*earth.
        ↪mass)
           return (G*earth.mass/(-2*energy0)**1.5) *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

       # Time of flight
       travel_time = abs(flight_time(r_moon) - flight_time(r0))
travel_time / (24*3600) # Time of flight in days
```

```
[138]: 3.2483424316374734
```

```
[139]: 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
```

```
[140]: # 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
```



[141]:
```
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
```

```
[141]: (6599.827350109479, 461056.0972096128)
```

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

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

```
[142]: (233827.96227986112, -1.4183759736418724)
```

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

```
[143]: (0.9717748583798103, -0.00030455346328228866)
```

## 7.1 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.

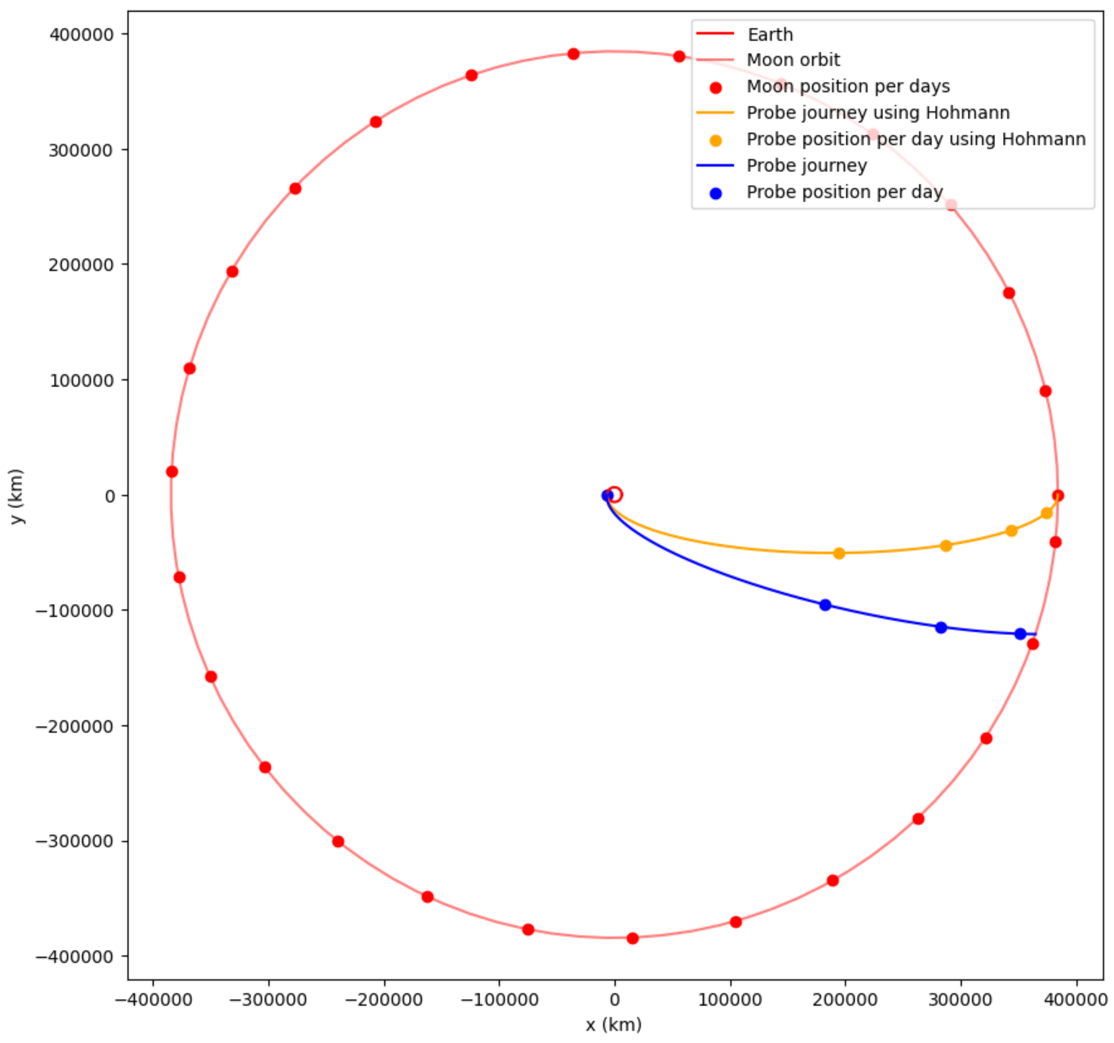
```
[144]: r_per = r0
       r_ap = r_moon
       a = (r_per + r_ap) / 2
       e = (r_ap - r_per) / (r_ap + r_per)
       journey_time = np.sqrt(4 * np.pi**2 * a**3 / (G * earth.mass)) / 2
       journey_time / (24*3600) # Time of flight in days
```

```
[144]: 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.

```
[145]: 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
y (km)
x (km)
400000
300000
200000
100000
0
−100000
−200000
−300000
−400000
−400000
−300000
−200000
−100000
0
100000
200000
300000
400000