Workbook 1 Hand-in

Nicolas THIERRY

April 23, 2024

Contents

0.1 Ex 1: Uniform Gravity 1
0.2 Ex 2: Realistic Gravity 3
0.3 Ex 3: Drag in a uniform atmosphere 6
0.4 Ex 4: An isothermal atmosphere model 9
0.5 Ex 5: Probe goes haywire 11
0.6 Ex 6: Drag for 3D motion 13
0.7 Ex 7: Aerobreaking 14

0.1 Ex 1: Uniform Gravity

[1]:
```python
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 $t = \frac{2*u}{g}$

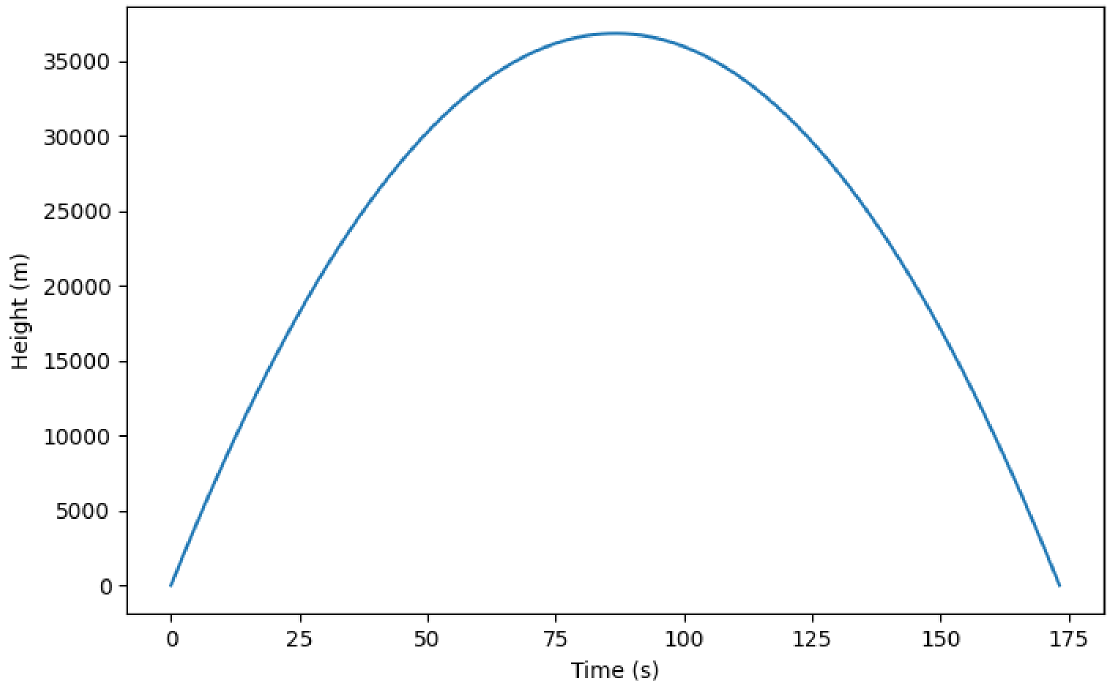
[2]:
```
t_final = (2*vx0) / gravity  # time of trajectory given
t_final # Time of flight in seconds
```

[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

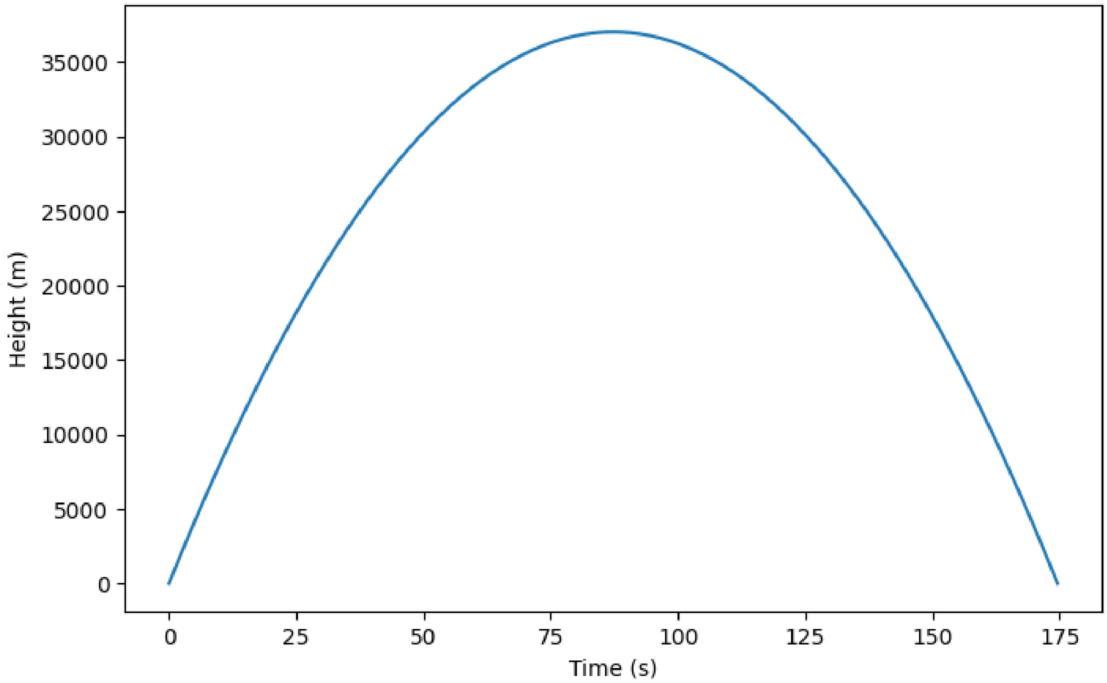
As the trajectory is modified, we can assume that time of flight will also be longer.

[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)')
```

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



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

[11]:
```python
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]:
```python
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]:
```python
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)
```

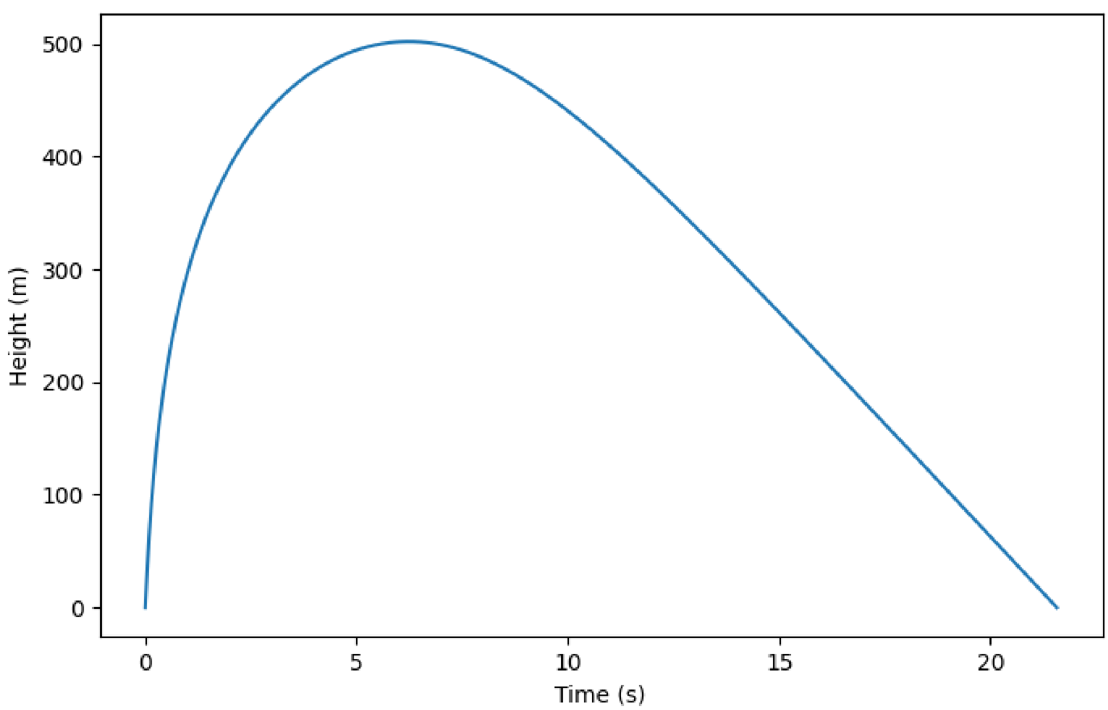
[17]: 21.56821568215682

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

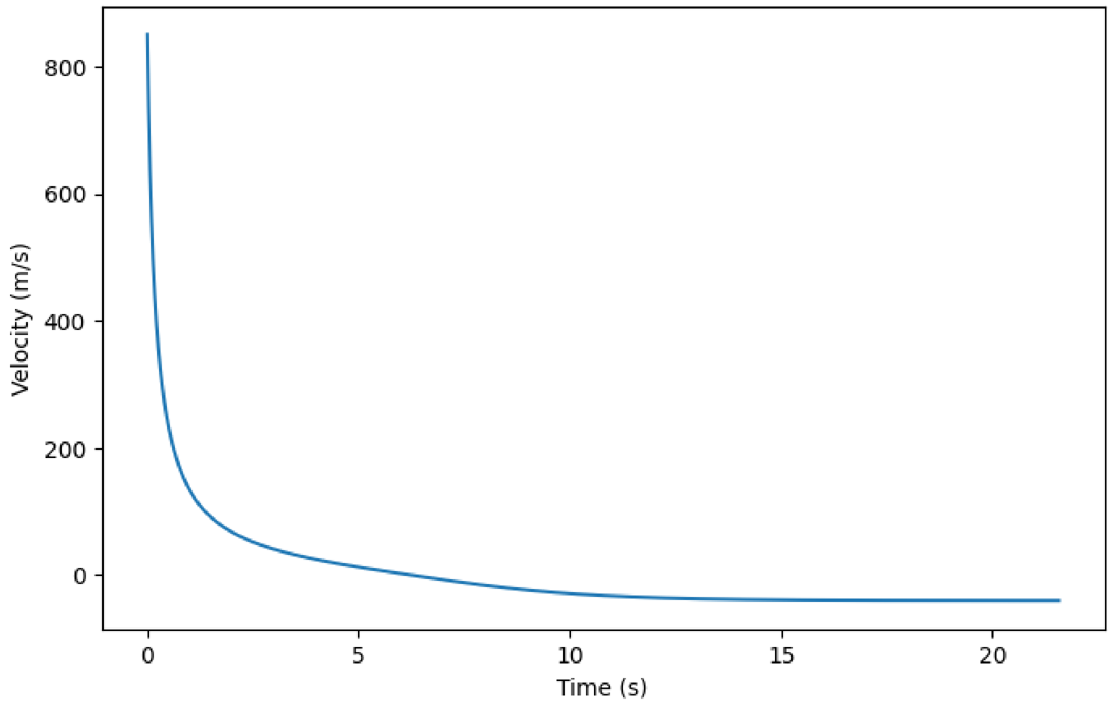
[18]:
```
posvel[t_end][0] # Altitude at the end (in m)
```

[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
```

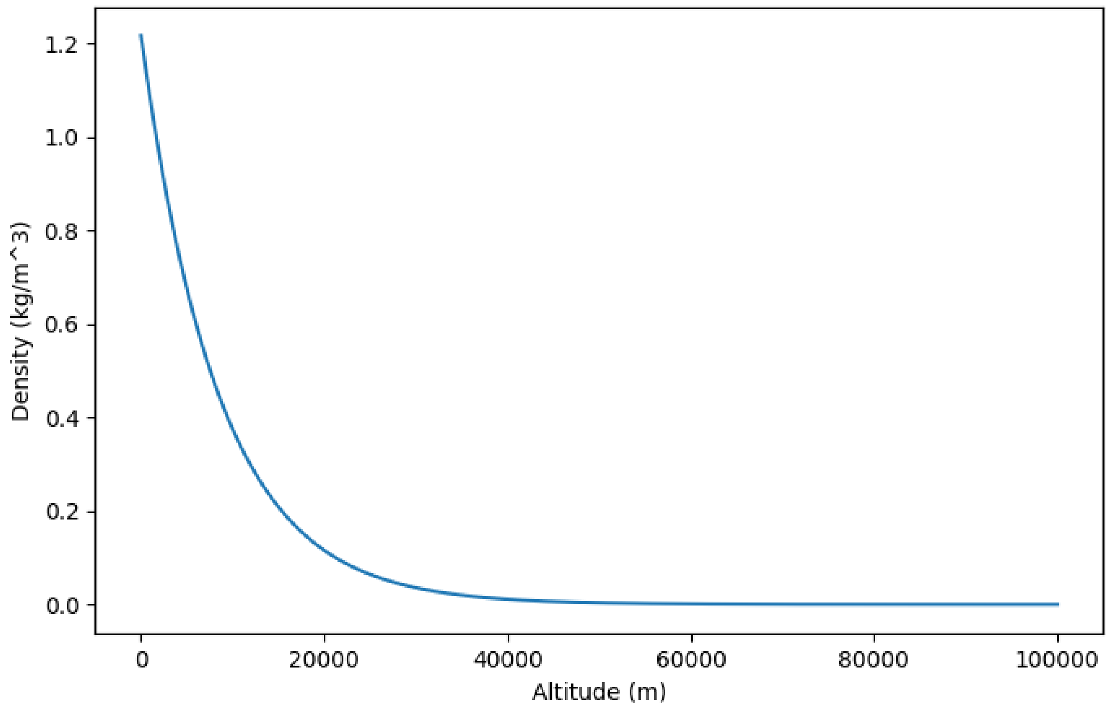
We are going to compute density as function of altitude as follows:

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

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]
```

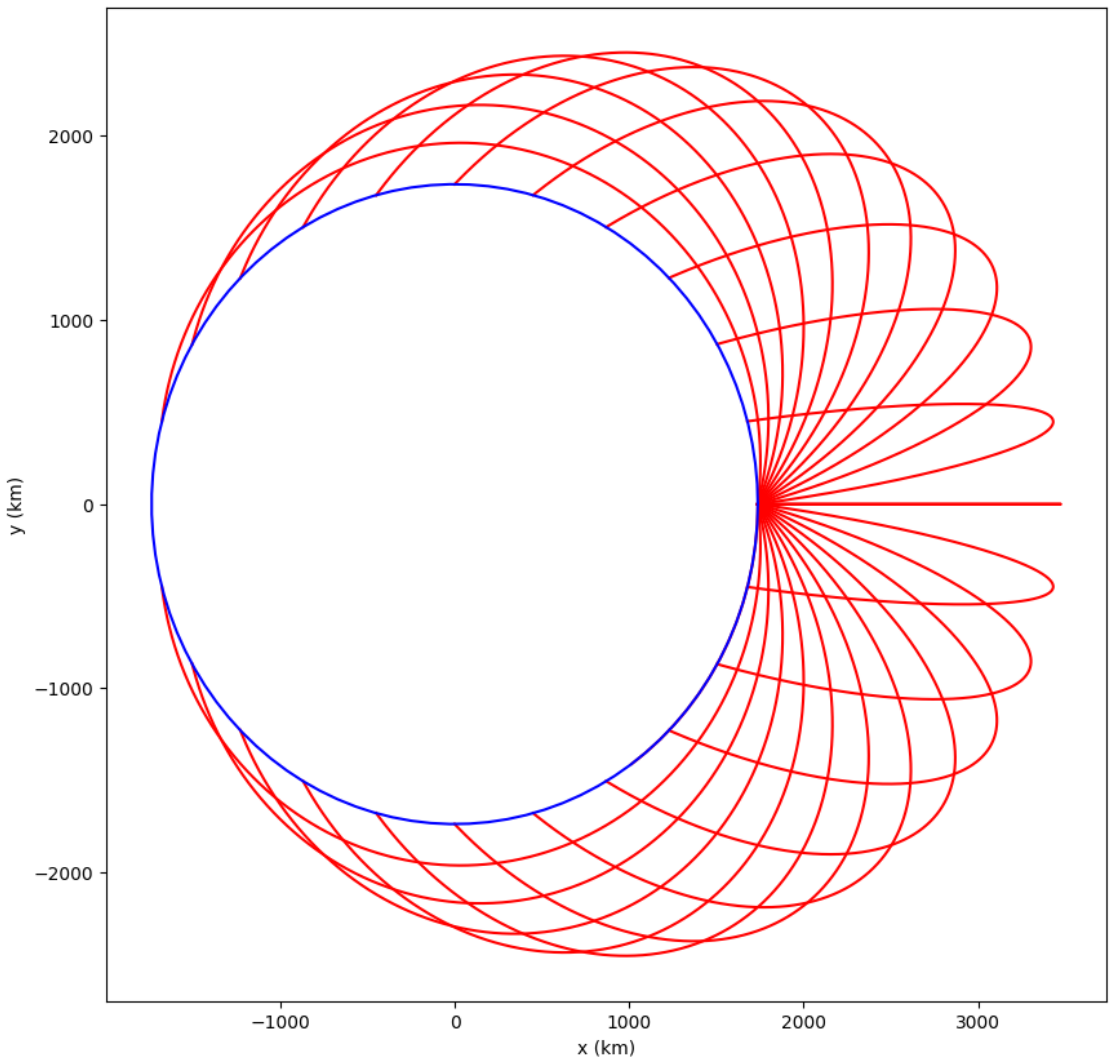
To solve this problem, we need to compute the initial velocity so that $Q = \frac{V^2 R_{moon}}{GM_{moon}}$ where $Q = 1$. Knowing that we have $\|\vec{V}\| = \sqrt{V^2}$, we can express the velocity with:

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

[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.

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

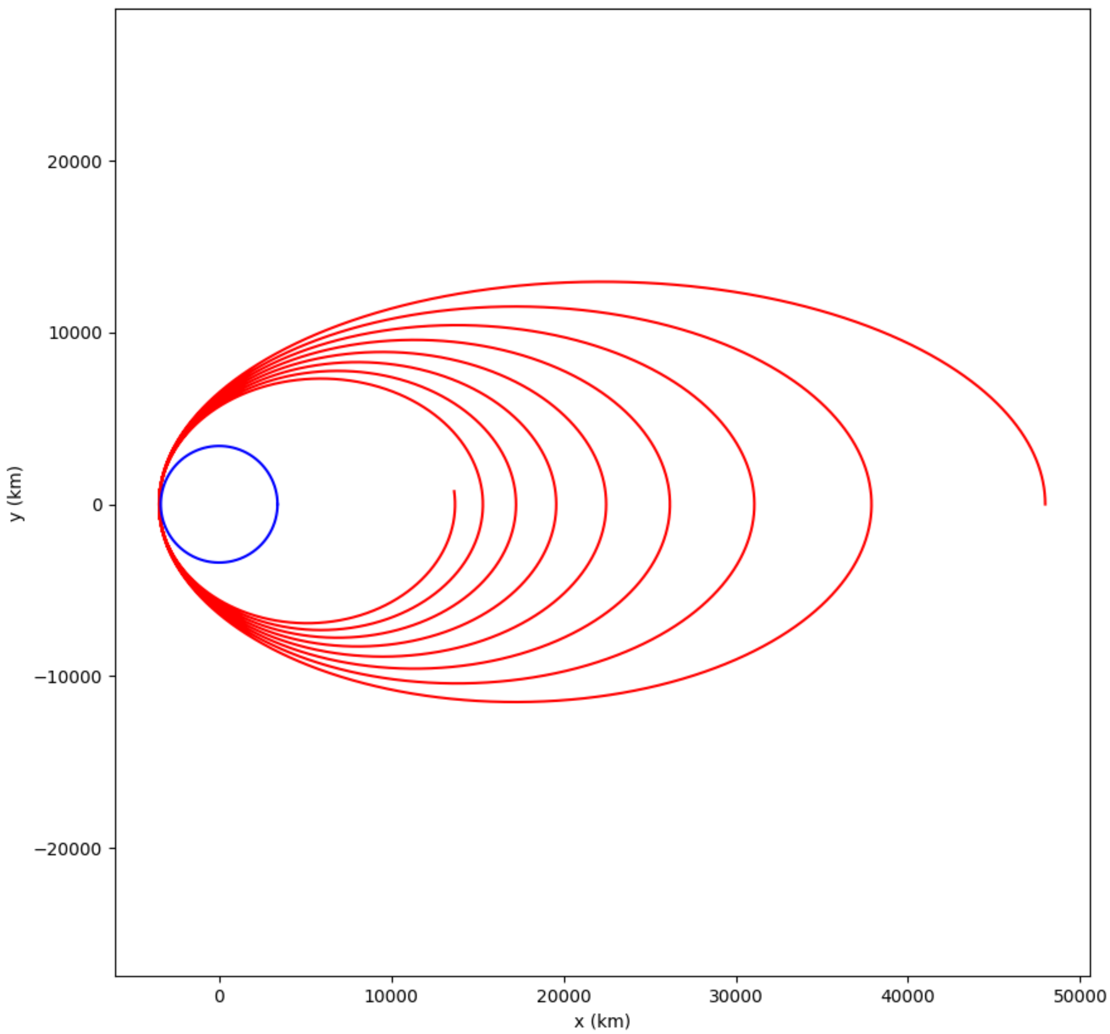
$$V = \sqrt{2GM_{planet}(\frac{1}{r} - \frac{1}{2a})}$$

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

[39]: 347.84600419637707

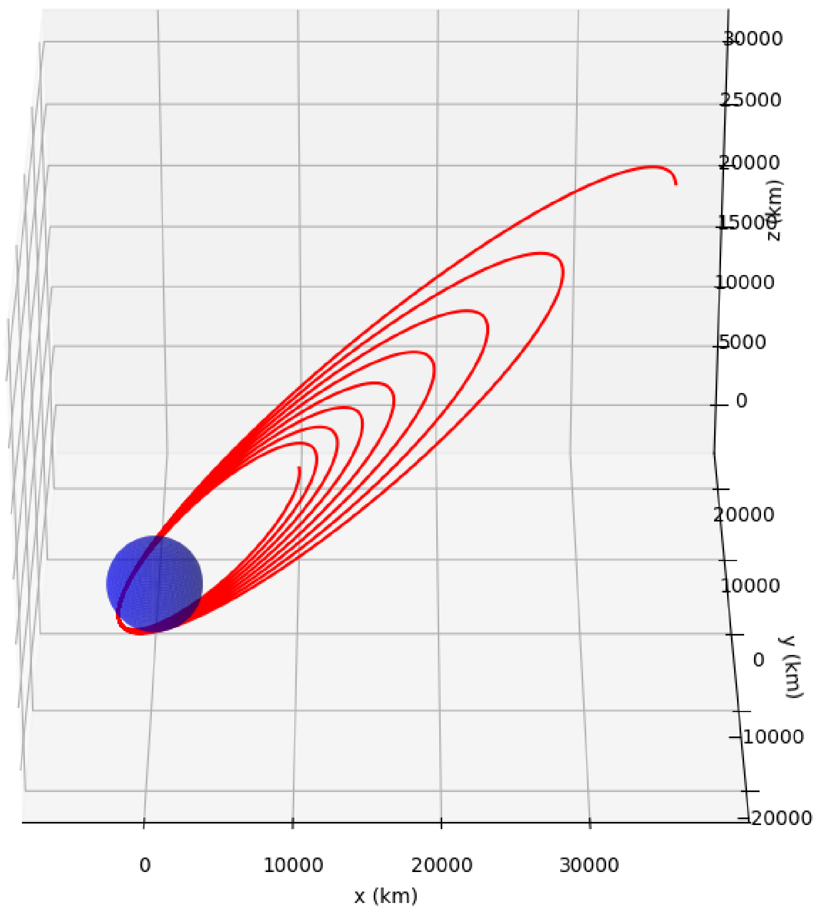
[40]:
```
# Initial Conditions
t_final = 3600 * 24 * 5 # determined trajectory time
t_num = t_final # number of steps in trajectory
xyz0 = [r_a*np.cos(30 * np.pi / 130), 0, r_a*np.sin(30 * np.pi / 130)]  # start␣
 ↪position
vxyz0 = [0, v, 0]  # start vertical speed

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.

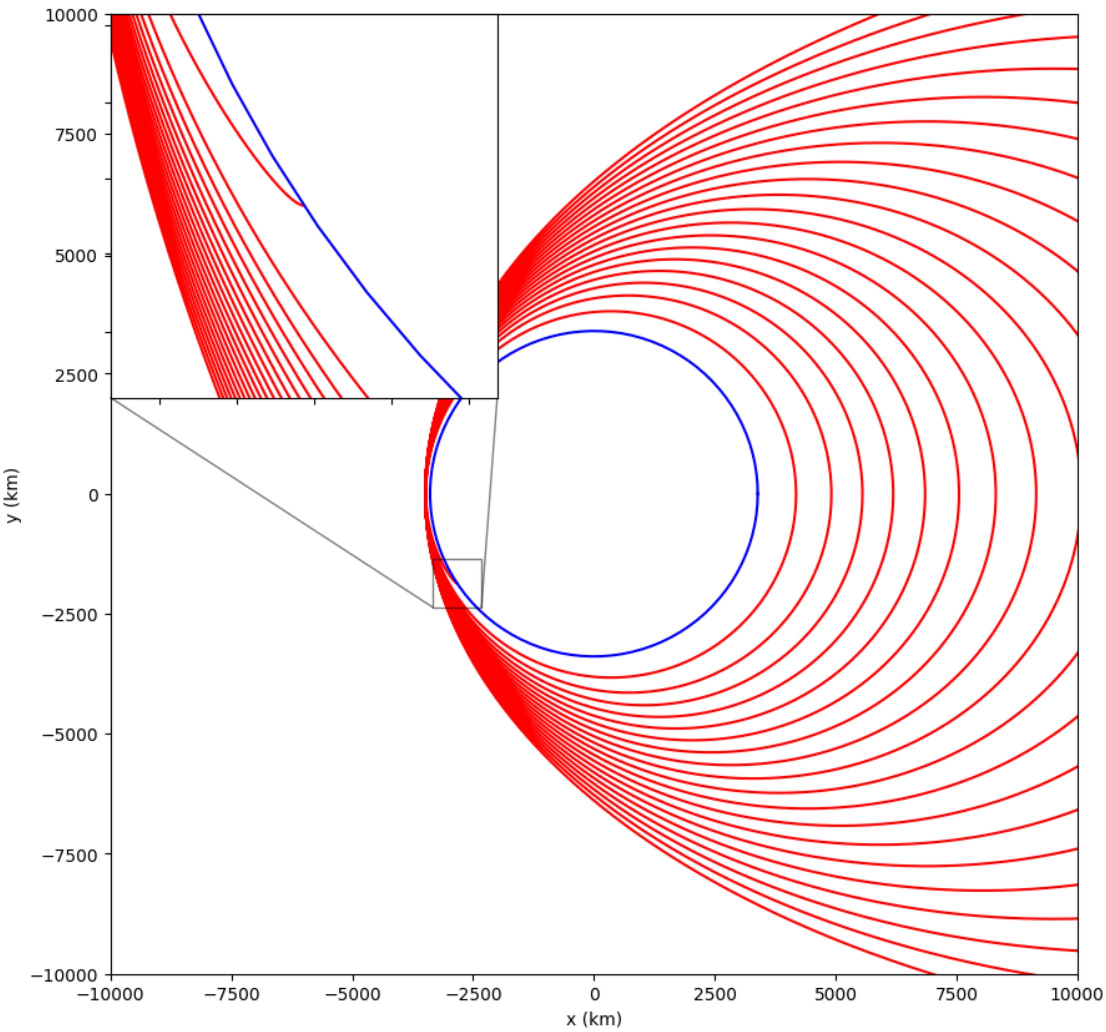
[41]:
```
fig = plt.figure(figsize=(15, 10)) # create figure, figsize can be changed as␣
↪preferred
ax = fig.add_subplot(111, projection='3d')
ax.plot(posvel[:, 0] / 1e3, posvel[:, 1] / 1e3, posvel[:, 2] / 1e3, color='red')

# 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
```

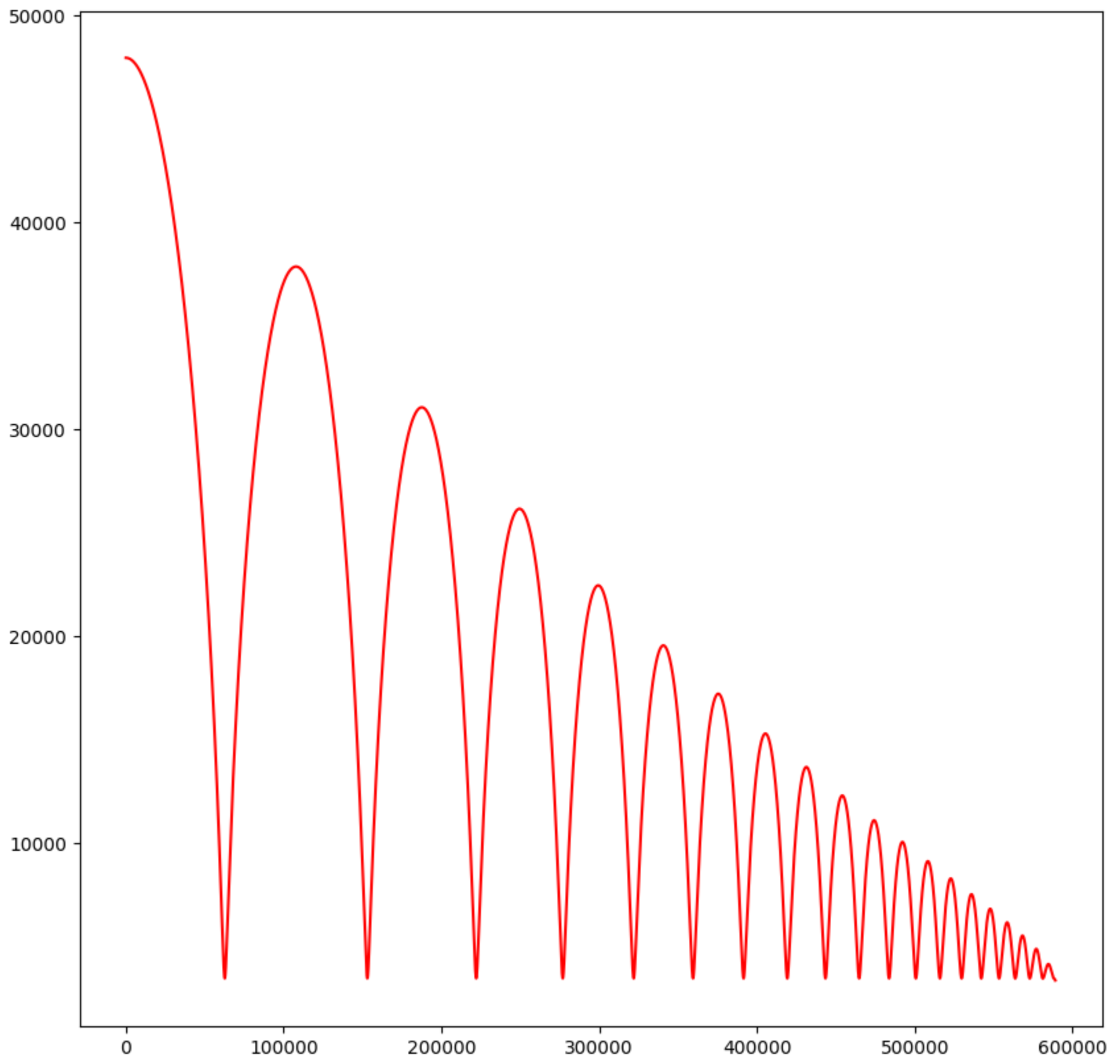
```python
t_end = len(t) - 2
altitude = np.sqrt(posvel[t_end, 0] ** 2 + posvel[t_end, 1] ** 2 + posvel[t_end, 2] ** 2) - mars.radius

# 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).