Workbook 2 Hand-in

Nicolas THIERRY

April 9, 2024

Contents

1 Ex 8: Transfer Orbit

[1]:
```
import matplotlib.pyplot as plt
from math import pi

from space_base import GravBody, Probe
from numpy import linspace, sqrt, pi, cos, sin

# 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 UA_to_meters(UA):
    return UA * 1.496e11
def meters_to_UA(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:

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

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

[2]:
```
r_per = UA_to_meters(1)  # Perihelion distance = Earth-Sun distance
r_aph = UA_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
```

[2]:
```
1.262
```

[3]:
```
e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
e # Eccentricity
```

[3]:
```
0.2076069730586371
```

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

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

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

[4]:
```
v_per = sqrt(G * sun.mass * (2 / r_per - 1 / a))  # Velocity at perihelion
v_per / 1e3  # Velocity at perihelion in km/s
```

[4]:
```
32.72071038681002
```

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

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

[5]:
```
period = sqrt(4 * pi**2 * a**3 / (G * sun.mass)) / 2  # Orbital period
period / (24 * 3600)  # Half orbital period in days
```

[5]:
```
258.9982551297217
```

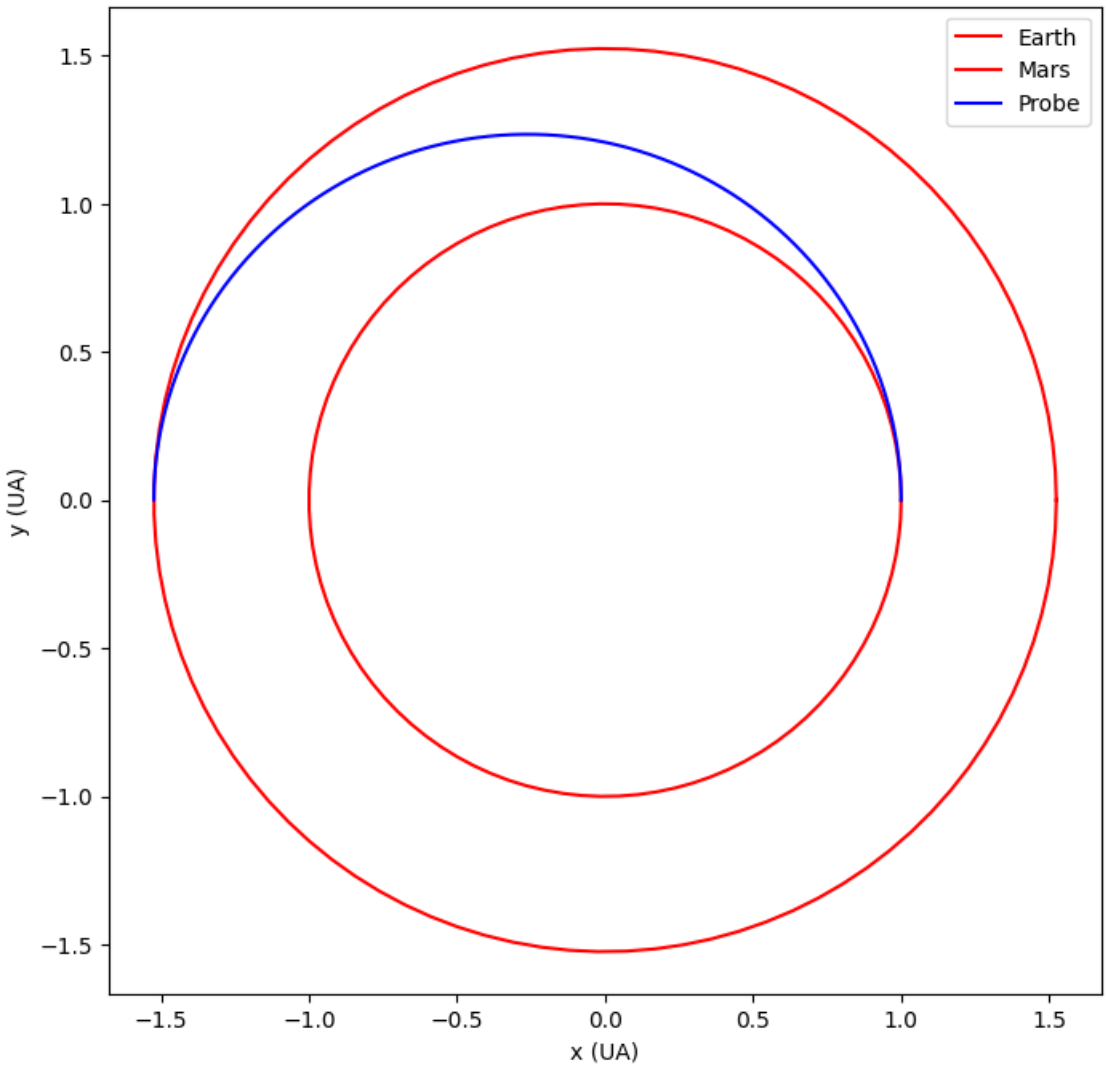
Here, we calculate the trajectory of the spacecraft with the Sun as the primary interaction body.

[6]:
```
def probeqns(_, posvel):
    r = 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.

[7]:
```
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
```

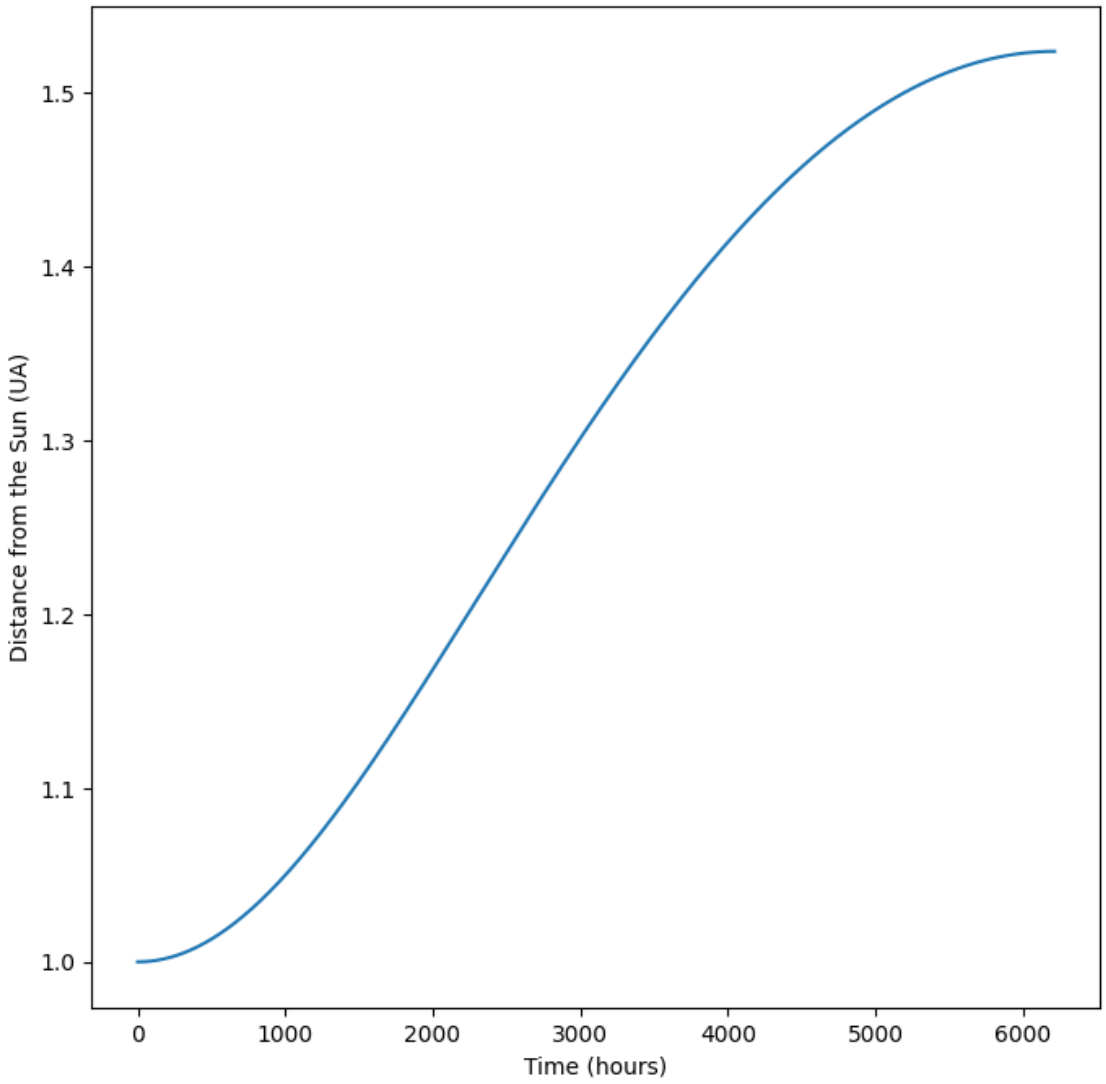
[8]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

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

plt.plot(meters_to_UA(posvel[:, 0]), meters_to_UA(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.

[9]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred
r = meters_to_UA(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
```

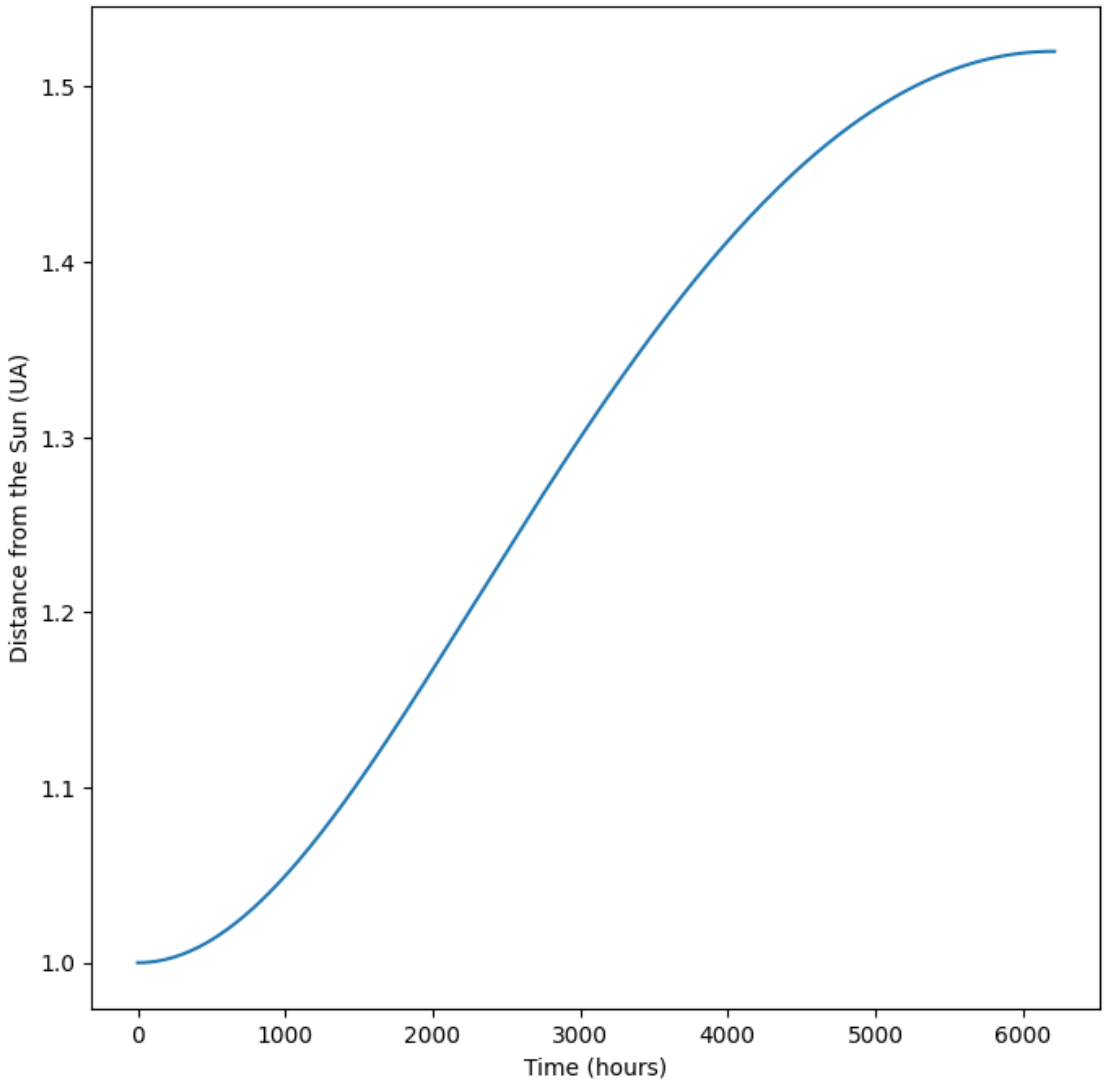


By reducing the initial velocity to 99.95% of the required velocity to reach Mars, we are quite far from the planet in absolute terms.

[10]:
```
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 = 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
```



[11]:
```
UA_to_meters(1.524 - max(r)) / 1e3 # Distance from Mars in km
```

[11]:
```
574407.1227933413
```

1.1 Asteroid $2013LA_2$ analyze

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

[12]:
```
ast_e = 0.4656 # Eccentricity of asteroid
ast_a = UA_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
```

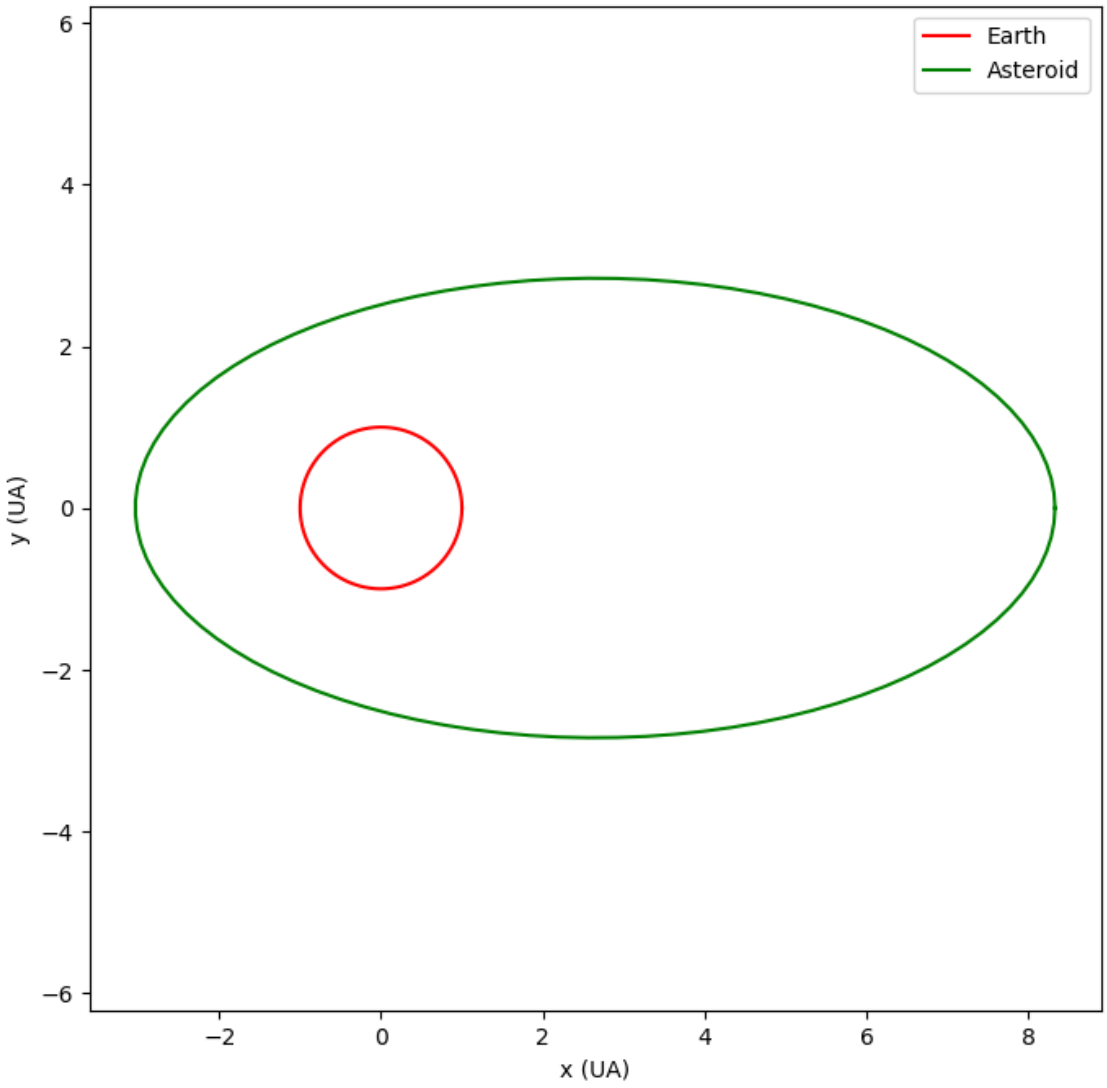
[13]:
```
ast_per / 1e3 # Perihelion distance of asteroid in km
```

[13]: 454422422.784

[14]:
```
ast_aph / 1e3 # Aphelion distance of asteroid in km
```

[14]: 1246260297.216

[15]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

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

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



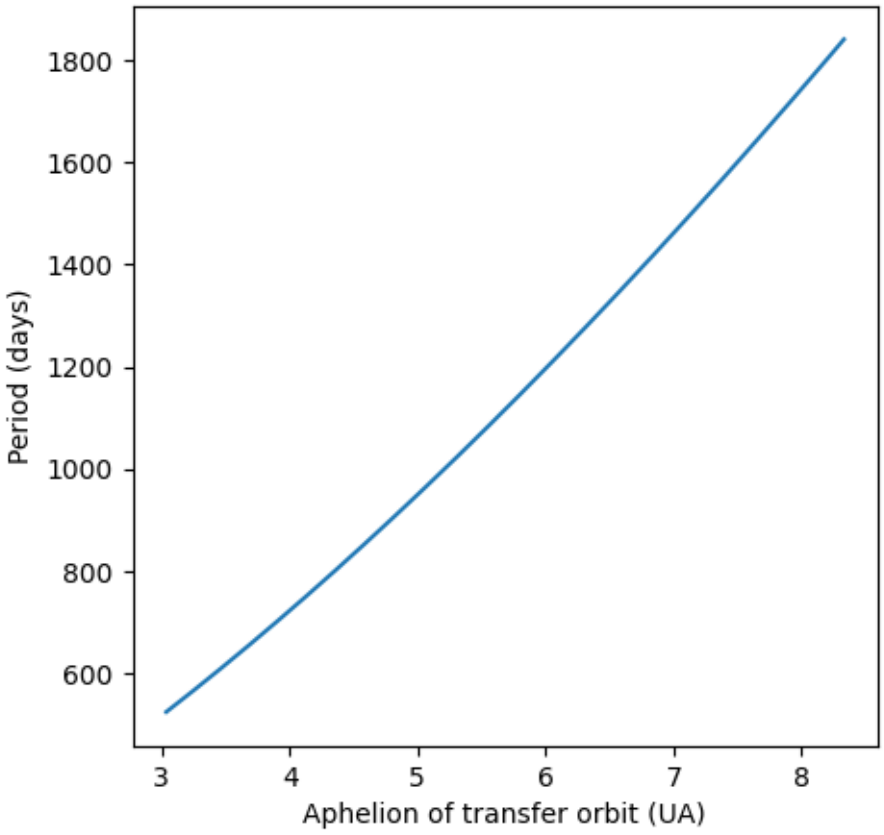
1.1.1 Least transit time

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

[16]:
```
ast_alts = linspace(ast_per, ast_aph, 10000)
a = (ast_alts + UA_to_meters(1)) / 2

plt.figure(figsize=(5, 5))  # create figure, figsize can be changed as preferred
p = 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.

[17]:
```
r_per = UA_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
```

[17]:
```
2.01879152
```

[18]:
```
e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
e # Eccentricity
```

[18]:
```
0.5046541507168606
```

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

[19]:
```
period = sqrt((4 * pi**2 * a**3)/(G * sun.mass))/2
period / (24 * 3600)  # Transit time in days
```

[19]:
```
524.0165621549162
```

[20]:
```
ast_vel = sqrt(G * sun.mass * (2 / r_aph - 1 / ast_a))  # Velocity of asteroid
vel = 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)
```

[20]:
```
32.70655681694997
```

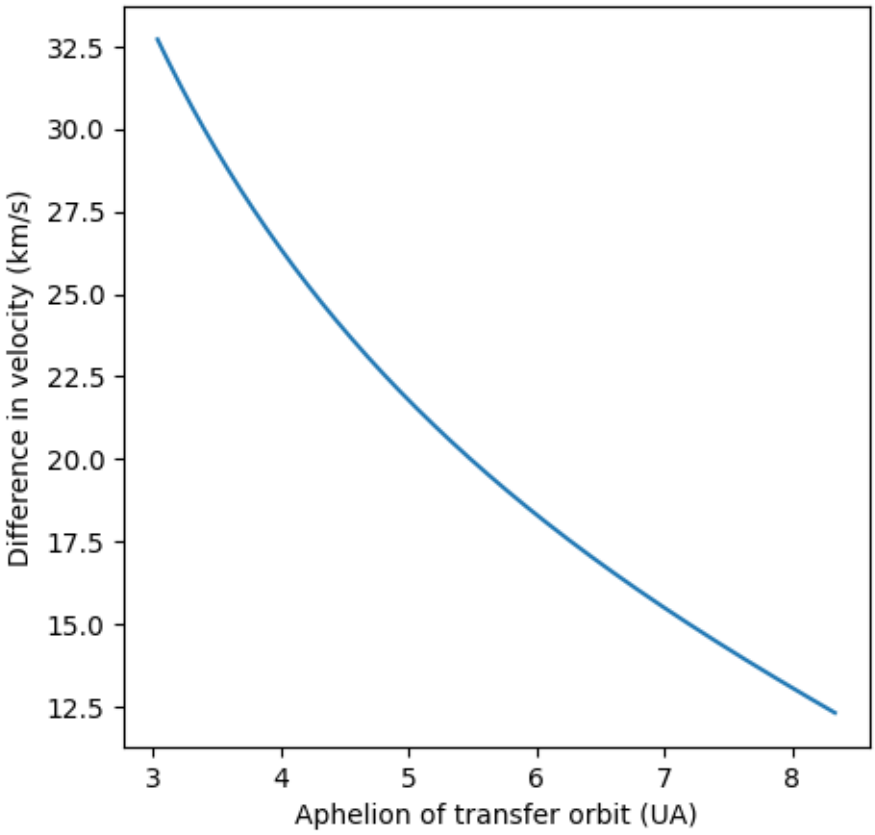
[21]:
```
xy0 = [r_per, 0]  # start position
vxy0 = [0, sqrt(G * sun.mass * (2 / r_per - 1 / a))]  # start vertical speed

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.

[22]:
```
ast_alts = linspace(ast_per, ast_aph, 10000)
ast_vel = sqrt(G * sun.mass * (2 / ast_alts - 1 / ast_a))  # Velocity of␣
 ↪asteroid
a = (ast_alts + UA_to_meters(1)) / 2
vel = 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.

[23]:
```
r_per = UA_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
```

[23]:
```
4.66530848
```

[24]:
```
e = (r_aph - r_per) / (r_aph + r_per)  # Eccentricity
e # Eccentricity
```

[24]:
```
0.7856519018437983
```

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

[25]:
```
ast_vel = sqrt(G * sun.mass * (2 / r_aph - 1 / ast_a))  # Velocity of asteroid
vel = sqrt(G * sun.mass * (2 / r_aph - 1 / a))  # Velocity of probe
```

```
abs(vel + ast_vel) / 1e3 # Difference in velocity in km/s
```

[25]: 12.31762997981529

[26]:
```
period = sqrt((4 * pi**2 * a**3)/(G * sun.mass))/2
period / (24 * 3600)  # Transit time in days
```

[26]: 1840.892185350244

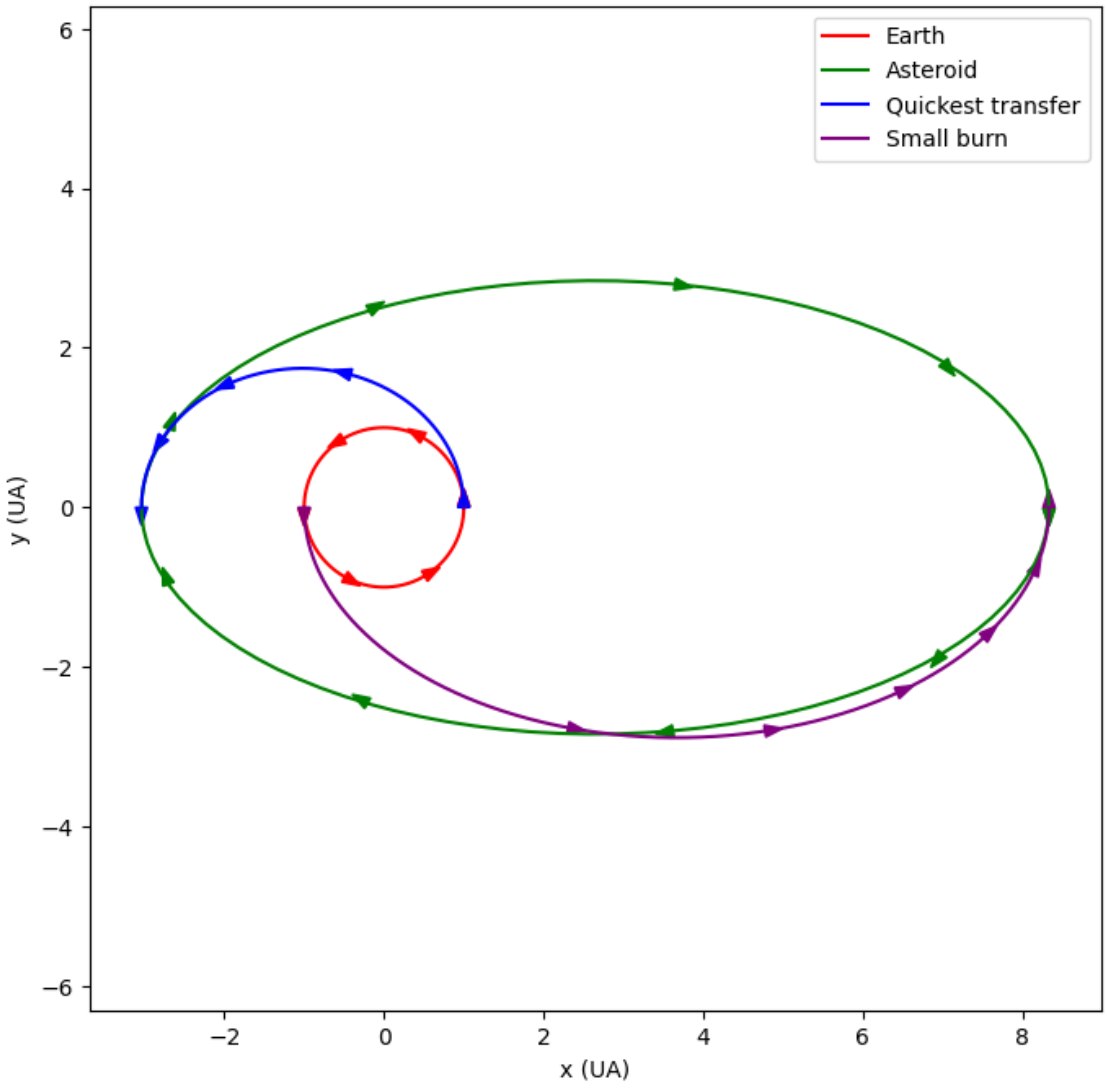
[27]:
```
xy0 = [-r_per, 0]  # start position
vxy0 = [0, -sqrt(G * sun.mass * (2 / r_per - 1 / a))]  # start vertical speed

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.

[28]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

# Plotting Earth's orbit
uang = linspace(0, 2 * pi, 100)
x = cos(uang)
y = sin(uang)
plt.plot(x, y, color='red', label='Earth')
## Add arrow to show direction of orbit
uang = linspace(0, 2 * pi, 7)
xs = cos(uang)
ys = sin(uang)
dxs = -0.01*sin(uang)
dys = 0.01*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_UA(ast_a - ast_per)
v=0
a= meters_to_UA(ast_a)
b=a / 2
t = linspace(0, 2*pi, 100)
x = u+a*cos(t)
y = v+b*sin(t)
plt.plot(x, y, color='green', label='Asteroid')
```

```
## Add arrow to show direction of orbit
uang = linspace(0, 2 * pi, 7)
t = linspace(0, 2*pi, 10)
xs = u+a*cos(t)
ys = v+b*sin(t)
dxs = 0.01*sin(t)
dys = -0.01*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_UA(quickest_posvel[:, 0]), meters_to_UA(quickest_posvel[:,␣
 ↪1]), color='blue', label='Quickest transfer')
idx = linspace(0, len(quickest_posvel)-1, 5, dtype=int)
for (x, y, dx, dy) in zip(meters_to_UA(quickest_posvel[idx, 0]),␣
 ↪meters_to_UA(quickest_posvel[idx, 1]), meters_to_UA(quickest_posvel[idx,␣
 ↪2]), meters_to_UA(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_UA(smallburn_posvel[:, 0]), meters_to_UA(smallburn_posvel[:,␣
 ↪1]), color='purple', label='Small burn')
idx = linspace(0, len(smallburn_posvel)-1, 7, dtype=int)
for (x, y, dx, dy) in zip(meters_to_UA(smallburn_posvel[idx, 0]),␣
 ↪meters_to_UA(smallburn_posvel[idx, 1]), meters_to_UA(smallburn_posvel[idx,␣
 ↪2]), meters_to_UA(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

[29]:
```
import matplotlib.pyplot as plt
from math import pi

from space_base import GravBody, Probe
from numpy import linspace, sqrt, pi, cos, sin, arccos

# 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 UA_to_meters(UA):
    return UA * 1.496e11
def meters_to_UA(meters):
    return meters / 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.

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

[30]:
```
r = 200e3 + mars.radius  # Radius distance in meters
v_init = sqrt(G * mars.mass / r)  # Velocity on initial circular orbit
v_init / 1e3  # Velocity at perihelion in km/s
```

[30]:
```
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.

[31]:
```
r_earth = UA_to_meters(1)  # Earth's radius in meters
r_mars = UA_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 = 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
```

[31]:
```
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.

[32]:
```
v_mars = 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
```

[32]: -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,

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

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

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

[33]:
```
v_per = sqrt(v_hyp**2 + 2*G*mars.mass/r)
v_per / 1e3  # Velocity at perihelion in km/s
```

[33]: 5.555702863158424

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

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

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

[34]:
```
a = G*mars.mass / (v_hyp**2)  # Semi-major axis
a / 1e3  # Semi-major axis in km
```

[34]: 6098.620611701382

[35]:
```
e = 1 + (r / a)  # Eccentricity
e # Eccentricity
```

[35]: 1.5885757171241068

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

[36]:
```
beta_angle=arccos(1/e)
beta_angle * 180 / pi  # Beta angle in degrees
```

[36]: 50.98714749340521

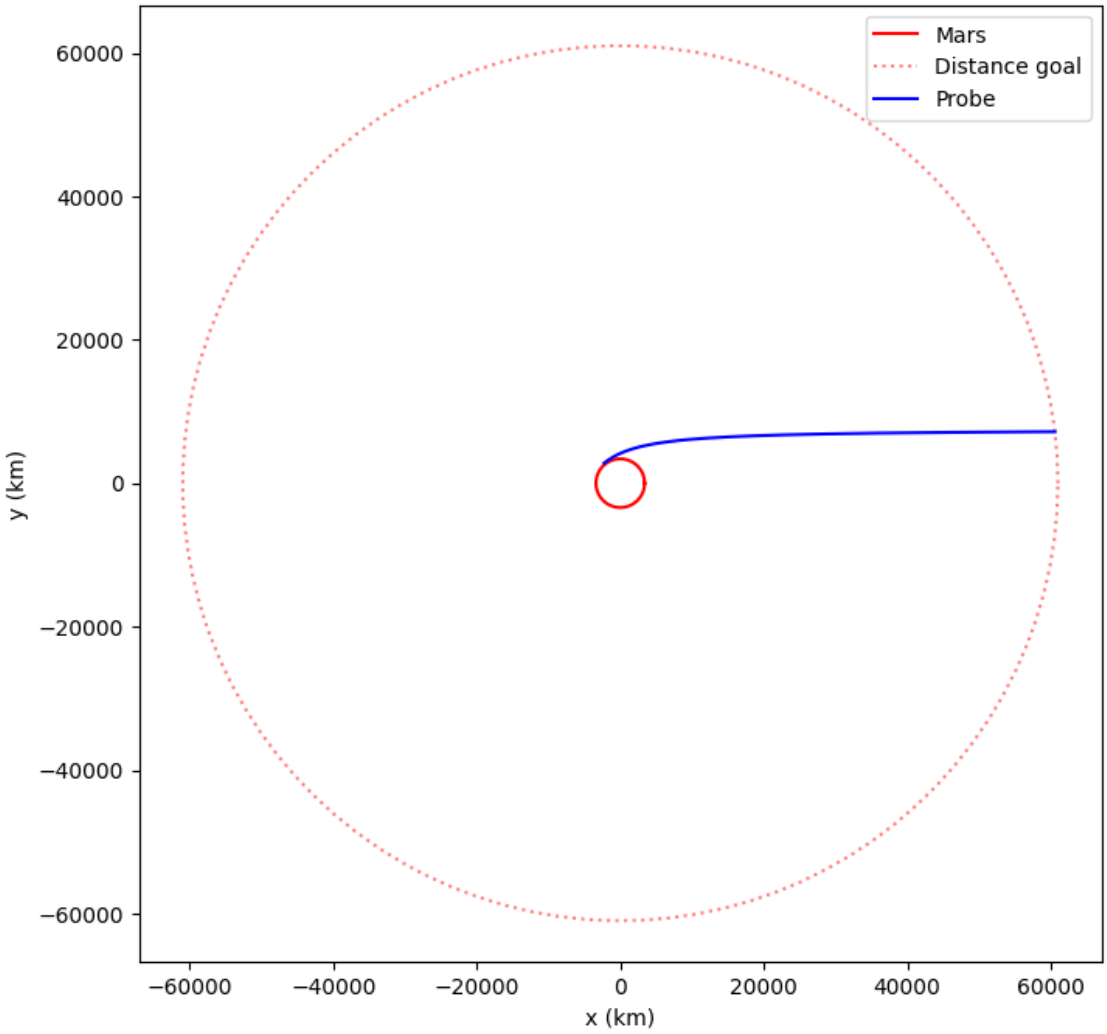
With these settings, we will fire our spacecraft in the opposite direction from Mars' orbit, as we will need to slow down to lower our perihelion to match Earth's orbit.

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.

[37]:
```
def probeqns(_, posvel):
    r = 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]
```

[38]:
```
xy0 = [-r*cos(beta_angle), r*sin(beta_angle)]  # start position
vxy0 = [sin(beta_angle)*v_per, 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
```

[39]:
```
plt.figure(figsize=(8, 8))  # create figure, figsize can be changed as preferred

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

[40]:
```
from datetime import timedelta
str(timedelta(seconds=t[len(t)-1])) # time of flight in hh:mm:ss
```

[40]:
```
'5:17:08.021686'
```

[41]:
```
from numpy.lib.stride_tricks import sliding_window_view
pos_slice = sliding_window_view(posvel[:, 0:2], window_shape = [2, 2])
total_dist = sum([ sqrt((pos[0, 0, 0] - pos[0, 1, 0])**2 +  (pos[0, 0, 1] -␣
 ↪pos[0, 1, 1])**2) for pos in pos_slice])
```

[42]:
```
sqrt(posvel[len(t)-1, 0]**2 + posvel[len(t)-1, 1]**2)/ 1e3 # Direct distance␣
 ↪from Mars center in km
```

[42]:
```
61011.50753647896
```

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

[43]:
```
total_dist / 1e3 # Total distance in km
```

[43]:
```
63492.58205912211
```

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

[44]:
```
(total_dist / t[len(t)-1]) / 1e3 # average speed in km/s
```

[44]:
```
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.

[45]:
```
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
```

[45]:
```
26
```

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

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

[46]:
```
center_of_mass = (modules[:, 1:].transpose() * modules[:, 0]).sum(axis=1) /␣
↪total_mass
center_of_mass
```

[46]:
```
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:

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

[47]:
```
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
```

[47]:
```
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.

[48]:
```
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

[49]:
```
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
```

[49]: 28

[50]:
```
center_of_mass = (modules[:, 1:].transpose() * modules[:, 0]).sum(axis=1) /␣
 ↪total_mass
center_of_mass
```

[50]:
```
array([ 0.14285714, -0.07142857,  0.42857143])
```

[51]:
```
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
```

[51]:
```
array([[366.71428571,  -2.28571429, -46.28571429],
       [ -2.28571429, 694.28571429,  -8.85714286],
       [-46.28571429,  -8.85714286, 455.28571429]])
```

[52]:
```
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]]
```

[53]:
```
w_new - w
```

[53]:
```
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.