retselis_set2

April 24, 2021

Problem Set #2

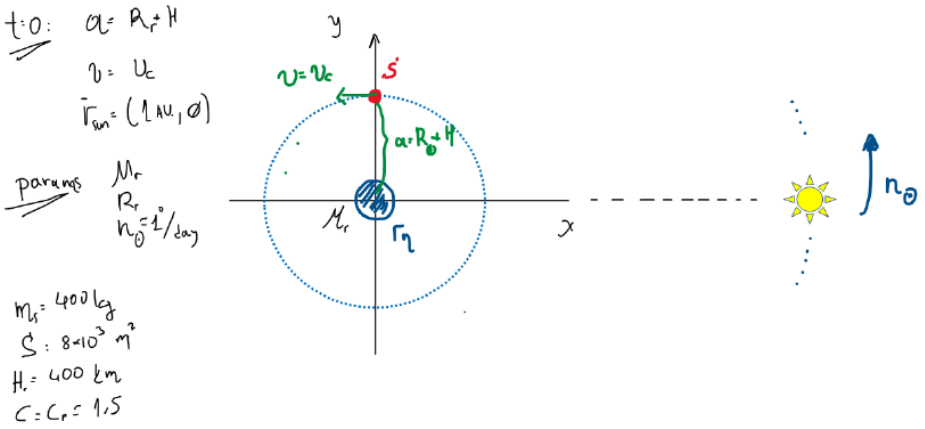
Computational Astrodynamics ($\Upsilon\Phi\Upsilon 204$)

Implemented by: **Anastasios-Faidon Retselis (AEM: 4394)**

1 Exercise 2.1

1.1 Problem statement

Assume an artificial satellite identical to the International Space Station (ISS) ($m = 400kg$, $S = 8000m^2$), in a circular low earth orbit ($h = 400km$). Assume a planar motion in Oxy, a geocentric reference frame and the initial conditions which are given in the figure below:



Write a numerical integration program for the satellite's equations of motion in cartesian coordinates, assuming only the gravitational pull of the Earth and using the initial conditions from the figure. Validate the stability of the orbit by measuring the error for the satellite's height for a propagation time equal to 6000 orbits of the satellite.

1.2 Solution

Let us first determine the initial conditions based on the figure. Assuming the earth radius to be $R_{Earth} = 6371km$:

$$a = R_{Earth} + h = 6371 + 400 = 6771km$$

Since we have a circular orbit constrained in the XY reference plane, it follows that $e = 0$, $i = 0$ and $\Omega = 0$. Finally, we can assume an initial value for the argument of pericenter of $\omega = 90°$ and an initial mean anomaly value $M = 0$, leading to the desired initial position of the spacecraft as defined by the figure above. The initial orbital elements can be found below:

Orbital Element	Value
Semi-major axis	6771000 m
Eccentricity	0
Inclination	0°
Longitude of the ascending node	0°
Argument of pericenter	90°
Mean anomaly	0°

Since no forces are acting on the spacecraft besides the gravitational force, the equation of motion can be described by the following equation:

$$\ddot{\overrightarrow{r}} + \frac{\mu}{r^2}\frac{\overrightarrow{r}}{r} = 0$$
$$\ddot{\overrightarrow{r}} = -\frac{\mu\overrightarrow{r}}{r^3}$$

This equation can be replaced by 2 first order equations for each cartesian coordinate. For clarity, we will also include the z-dimension. The equation can thus be replaced by the following equations:

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{z} = v_z$$
$$v_x = \frac{-\mu x}{\sqrt{x^2+y^2+z^2}}$$
$$v_y = \frac{-\mu y}{\sqrt{x^2+y^2+z^2}}$$
$$v_z = \frac{-\mu z}{\sqrt{x^2+y^2+z^2}}$$

The code implemented can be found below. First, we will use the previously developed transformation from orbital elements to cartesian to compute the initial values for the velocity and position vectors. We will then treat the 6 equations which we derived above as a system of differential equations and we will solve them using a fourth order Runge-Kutta integrator. We will plot a single orbit for visualization purposes and we will then validate our code, by integrating for a total of 6000 orbits for different time steps and we will log the percent error between the initial and the

final radius (equivalent to height) values. We will also log the time it took for the computer to perform each integration and discuss the results.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
# Make jupyter export images as .pdf files for higher quality
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
# Silent run for NaNs
import warnings
warnings.filterwarnings('ignore')

def orbital_elements_to_cartesian(a, e, i, Omega, omega, M, mu):
    # Computes cartesian position and velocity vector given some orbital␣
 ↪elements
    # Input:
    # a [m]
    # e []
    # i [deg]
    # Omega [deg]
    # omega [deg]
    # M [deg]
    # mu [m^3/(kg*s^2)] (GM)
    # Output:
    # r_vector, y_vector

    # Compute E using Newton-Raphson
    # Define initial value for E_0
    if M > -np.pi and M < 0:
        E_0 = M - e
    elif M > np.pi:
        E_0 = M + e
    else:
        E_0 = 0.1
    # Convert M to radians
    M = np.radians(M)
    # Define f and f dot
    f = lambda E: M - E + np.sin(E)
    fdot = lambda E: -1 + np.cos(E)
    # Stopping criteria
    N = 15  # Number of significant digits to be computed
    max_repetitions = 1000000
    es = 0.5 * pow(10, (2 - N))  # Scarborough Criterion
    ea = 100
    E_prev = E_0
    repetitions = 0
```

```
    # Main Newton-Raphson loop
    while ea > es:
        repetitions = repetitions + 1
        E_next = E_prev - (f(E_prev) / fdot(E_prev))
        ea = np.fabs((E_next - E_prev) * 100 / E_next)
        E_prev = E_next
        if repetitions > max_repetitions:
            # print('Max repetitions reached without achieving desired accuracy␣
↪for E!')
            break
    E = E_next
    # Compute x, xdot, y, ydot on the orbital plane
    x = a * (np.cos(E) - e)
    y = a * np.sqrt(1 - pow(e, 2)) * np.sin(E)
    r = np.sqrt(pow(x, 2) + pow(y, 2))
    n = np.sqrt(mu / pow(a, 3))  # Mean motion
    x_dot = -(n * pow(a, 2) / r) * np.sin(E)
    y_dot = (n * pow(a, 2) / r) * np.sqrt(1 - pow(e, 2)) * np.cos(E)
    # Rotation Matrices definition
    Omega = np.radians(Omega)
    omega = np.radians(omega)
    i = np.radians(i)
    P1 = np.array([[np.cos(omega), -np.sin(omega), 0],
                   [np.sin(omega), np.cos(omega), 0],
                   [0, 0, 1]])
    P2 = np.array([[1, 0, 0],
                   [0, np.cos(i), np.sin(i)],
                   [0, np.sin(i), np.cos(i)]])
    P3 = np.array([[np.cos(Omega), -np.sin(Omega), 0],
                   [np.sin(Omega), np.cos(Omega), 0],
                   [0, 0, 1]])
    # Compute cartesian coordinates
    x_y_vector = np.array([x, y, 0])
    x_y_dot_vector = np.array([x_dot, y_dot, 0])
    r_vector = np.matmul(np.matmul(np.matmul(P3, P2), P1), x_y_vector)
    v_vector = np.matmul(np.matmul(np.matmul(P3, P2), P1), x_y_dot_vector)
    return r_vector, v_vector


def runge_kutta_4(x_0, y_0, z_0, vx_0, vy_0, vz_0, mu, t_start, tmax, h):
    # 4th order Runge Kutta orbit propagator
    # Input:
    # x_0, y_0, z_0, namely the coordinates of the position vector [meters]
    # vx_0, vy_0, vz_0, namely the coordinates of the velocity vector [m/s]
    # mu, mu constant
    # t_start, initial time [seconds]
    # t_end, propagation end time [seconds]
```

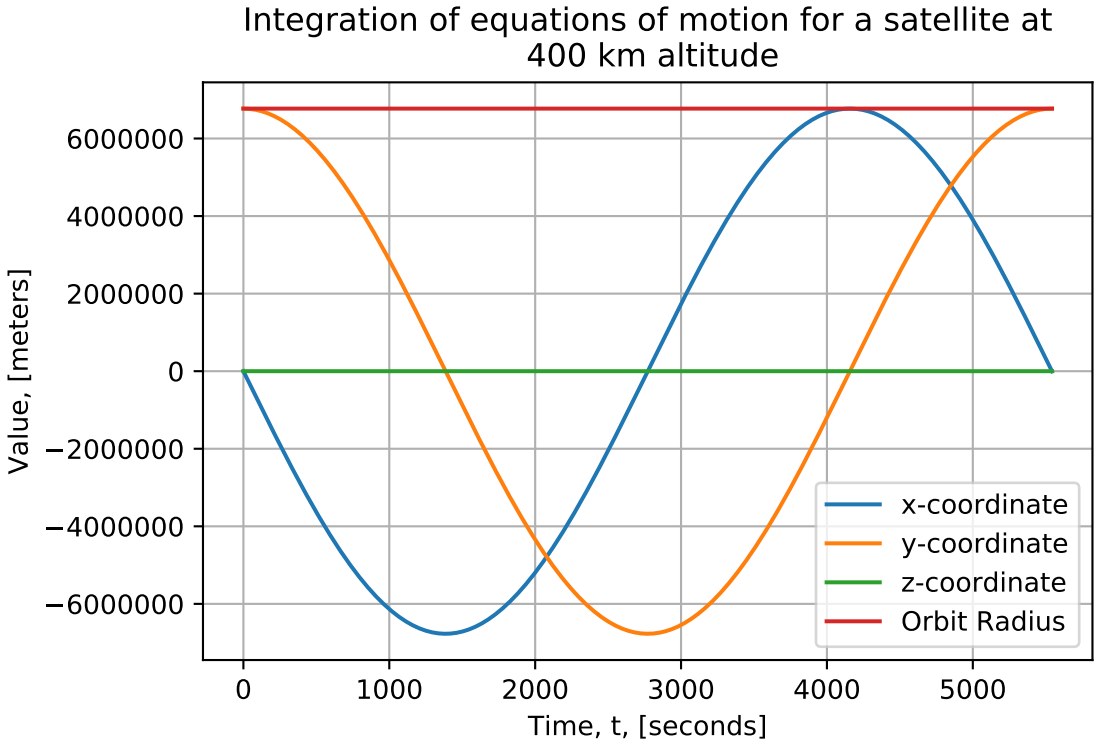
```
    # h, integration step size [seconds]
    # Output is 7 vectors containing position, velocity and time information at␣
↪each integration step

    # Log initial values
    tn = [t_start]
    xn = [x_0]
    yn = [y_0]
    zn = [z_0]
    vxn = [vx_0]
    vyn = [vy_0]
    vzn = [vz_0]
    counter = 0
    # Main RK4 loop
    while tn[counter] < tmax:
        # Calculate k1 values
        k1_x = fx(vx_0)
        k1_y = fy(vy_0)
        k1_z = fz(vz_0)
        k1_vx = fv_x(x_0, y_0, z_0, mu)
        k1_vy = fv_y(x_0, y_0, z_0, mu)
        k1_vz = fv_z(x_0, y_0, z_0, mu)
        # Calculate midpoint values
        mid_x = x_0 + k1_x * h/2
        mid_y = y_0 + k1_y * h/2
        mid_z = z_0 + k1_z * h/2
        mid_vx = vx_0 + k1_vx * h/2
        mid_vy = vy_0 + k1_vy * h/2
        mid_vz = vz_0 + k1_vz * h/2
        # Calculate k2 values
        k2_x = fx(mid_vx)
        k2_y = fy(mid_vy)
        k2_z = fz(mid_vz)
        k2_vx = fv_x(mid_x, mid_y, mid_z, mu)
        k2_vy = fv_y(mid_x, mid_y, mid_z, mu)
        k2_vz = fv_z(mid_x, mid_y, mid_z, mu)
        # Calculate next midpoint values
        mid_x = x_0 + k2_x * h / 2
        mid_y = y_0 + k2_y * h / 2
        mid_z = z_0 + k2_z * h / 2
        mid_vx = vx_0 + k2_vx * h / 2
        mid_vy = vy_0 + k2_vy * h / 2
        mid_vz = vz_0 + k2_vz * h / 2
        # Calculate k3 values
        k3_x = fx(mid_vx)
        k3_y = fy(mid_vy)
        k3_z = fz(mid_vz)
```

```
        k3_vx = fv_x(mid_x, mid_y, mid_z, mu)
        k3_vy = fv_y(mid_x, mid_y, mid_z, mu)
        k3_vz = fv_z(mid_x, mid_y, mid_z, mu)
        # Calculate next midpoint values
        mid_x = x_0 + k3_x * h
        mid_y = y_0 + k3_y * h
        mid_z = z_0 + k3_z * h
        mid_vx = vx_0 + k3_vx * h
        mid_vy = vy_0 + k3_vy * h
        mid_vz = vz_0 + k3_vz * h
        # Calculate k4 values
        k4_x = fx(mid_vx)
        k4_y = fy(mid_vy)
        k4_z = fz(mid_vz)
        k4_vx = fv_x(mid_x, mid_y, mid_z, mu)
        k4_vy = fv_y(mid_x, mid_y, mid_z, mu)
        k4_vz = fv_z(mid_x, mid_y, mid_z, mu)
        # Compute r, v values and append to list
        xn.append(xn[counter] + (h / 6) * (k1_x + 2 * k2_x + 2 * k3_x + k4_x))
        yn.append(yn[counter] + (h / 6) * (k1_y + 2 * k2_y + 2 * k3_y + k4_y))
        zn.append(zn[counter] + (h / 6) * (k1_z + 2 * k2_z + 2 * k3_z + k4_z))
        vxn.append(vxn[counter] + (h / 6) * (k1_vx + 2 * k2_vx + 2 * k3_vx +␣
↪k4_vx))
        vyn.append(vyn[counter] + (h / 6) * (k1_vy + 2 * k2_vy + 2 * k3_vy +␣
↪k4_vy))
        vzn.append(vzn[counter] + (h / 6) * (k1_vz + 2 * k2_vz + 2 * k3_vz +␣
↪k4_vz))
        # Prepare for the next iteration and reset values
        counter += 1
        tn.append(tn[counter - 1] + h)
        x_0 = xn[counter]
        y_0 = yn[counter]
        z_0 = zn[counter]
        vx_0 = vxn[counter]
        vy_0 = vyn[counter]
        vz_0 = vzn[counter]
    return xn, yn, zn, vxn, vyn, vzn, tn


def fx(v_x):
    # Assuming x'(t)=v_x
    return v_x


def fy(v_y):
    # Assuming y'(t)=v_y
    return v_y
```

```
def fz(v_z):
    # Assuming z'(t)=v_z
    return v_z


def fv_x(x, y, z, mu):
    # Assuming v_x'(t)=-mu*x/(sqrt(x^2+y^2+z^2))^3
    return -mu*x/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3)


def fv_y(x, y, z, mu):
    # Assuming v_y'(t)=-mu*y/(sqrt(x^2+y^2+z^2))^3
    return -mu*y/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3)


def fv_z(x, y, z, mu):
    # Assuming v_z'(t)=-mu*z/(sqrt(x^2+y^2+z^2))^3
    return -mu*z/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3)


# Constants definition
G = 6.6743*pow(10, -11)
M_earth = 5.977*pow(10, 24)
mu_earth = G*M_earth
R_earth = 6371000  # [meters]

# Orbit definition
H = 400000  # Altitude, [meters]
a = R_earth + H  # Semi-major axis, [meters]
i = 0  # Inclination, [deg]
e = 0  # Eccentricity, []
Omega = 0  # Longitude of the Ascending Note [deg]
omega = 90  # Argument of pericenter [deg]
M = 0  # Mean anomaly, [deg]
T = 2 * np.pi * np.sqrt(pow(a, 3)/mu_earth)  # Orbital period, [seconds]

# Compute initial position and velocity vector from orbital elements
r_vector, v_vector = orbital_elements_to_cartesian(a, e, i, Omega, omega, M,␣
 ↪mu_earth)
x = r_vector[0]
y = r_vector[1]
z = r_vector[2]
vx = v_vector[0]
vy = v_vector[1]
vz = v_vector[2]
```

```
# Propagate orbit (all times in seconds)
start_time = 0
end_time = T
time_step = 1
xn, yn, zn, vxn, vyn, vzn, tn = runge_kutta_4(x, y, z, vx, vy, vz, mu_earth,␣
 ↪start_time, end_time, time_step)

# Compute r magnitude
r = np.zeros(np.size(xn))
for i in range(0, np.size(xn)):
    r[i] = np.sqrt(pow(xn[i], 2) + pow(yn[i], 2) + pow(zn[i], 2))

# Plot results for one orbit
plt.plot(tn, xn, label='x-coordinate')
plt.plot(tn, yn, label='y-coordinate')
plt.plot(tn, zn, label='z-coordinate')
plt.plot(tn, r,  label='Orbit Radius')
plt.xlabel('Time, t, [seconds]')
plt.ylabel('Value, [meters]')
plt.title('Integration of equations of motion for a satellite at\n 400 km␣
 ↪altitude')
plt.grid()
plt.legend()
plt.show()

# Validate propagator by computing error for different step sizes for 6000␣
 ↪orbits
start_time = 0
end_time = 6000*T
time_steps = np.array([60, 50, 40, 30, 20])
print('=========================================================')
print('Propagator Validation for ' + str(end_time) + ' seconds')
print('=========================================================')
print('Time step (sec)\t\t% Error\t\tRun time (sec)')
for counter in range(0, np.size(time_steps)):
    timer_start = time.time()
    xn, yn, zn, vxn, vyn, vzn, tn = \
        runge_kutta_4(x, y, z, vx, vy, vz, mu_earth, start_time, end_time,␣
 ↪time_steps[counter])
    # Compute r magnitude
    r = np.zeros(np.size(xn))
    for i in range(0, np.size(xn)):
        r[i] = np.sqrt(pow(xn[i], 2) + pow(yn[i], 2) + pow(zn[i], 2))
    # Compute error
    r_initial = r[0]
    r_final = r[np.size(xn)-1]
```

```
    error = np.fabs(r_final-r_initial)/r_initial*100
    time_taken = time.time() - timer_start
    print('%d\t\t\t%.5f\t\t%.2f' % (time_steps[counter], error, time_taken))
```




```
=========================================================
Propagator Validation for 33255681.31219161 seconds
=========================================================
Time step (sec)         % Error         Run time (sec)
60                      0.15354         36.82
50                      0.06147         42.55
40                      0.02010         52.86
30                      0.00477         70.44
20                      0.00063         106.11
```

1.3 Discussion

From the generated figure we can verify that we used the correct initial conditions (max y-value since the spacecraft for $t = 0$ is on the positive side of the Oy axis. We also note that for one orbit the radius of the orbit remains constant, leading us to the conclusion that the propagator works as intended. To validate the propagator, we propagated the orbit for a total of 6000 orbits using different time steps, which can also be seen in the table below:

Time step [sec]	Error [%]	Run time [sec]
60	0.15354	36.44
50	0.06147	44.31
40	0.02010	55.11
30	0.00477	73.26
20	0.00063	110.85

We can observe that even if we use a time step of 60 seconds, the percentage error remains under 1% and the run time remains sufficiently small. Decreasing the time step further improves the error on the height/radius, but the run time starts to rise to unacceptably long times. Nevertheless, we have demonstrated that the integrator implemented using the fourth order Runge-Kutta method works as intended.

2 Exercise 2.2

2.1 Problem statement

For the problem of exercise 2.1, include aerodynamic drag due to the atmosphere of the Earth. For simplicity, assume that the density is given by the function $\rho = (0.1H)^{-7.5}\frac{kg}{m^3}$ for $H > 15km$ and $\rho = 0.1\frac{kg}{m^3}$ for $H < 15km$. The spacecraft properties can be seen in the figure above. Find the time taken until the spacecraft reaches the surface of the Earth. Compare this result with a prediction made using Gauss' equation (for $e = 0$, assuming the same density function $\rho(H)$. Plot the orbit on the Oxy plane.

2.2 Solution

The initial orbital elements which we will use are identical to the previous exercise, namely:

Orbital Element	Value
Semi-major axis	6771000 m
Eccentricity	0
Inclination	0°
Longitude of the ascending node	0°
Argument of pericenter	90°
Mean anomaly	0°

Since we have now included the atmospheric drag, the equation of motion can be described by the following equation:

$$\ddot{\overrightarrow{r}} + \frac{\mu}{r^2}\frac{\overrightarrow{r}}{r} = -\frac{1}{2}\rho v\left(\frac{C_D A}{m}\right)\overrightarrow{v}$$
$$\ddot{\overrightarrow{r}} = -\frac{\mu\overrightarrow{r}}{r^3} - \frac{1}{2}\rho v\left(\frac{C_D A}{m}\right)\overrightarrow{v}$$

This equation can be replaced by 2 first order equations for each cartesian coordinate. For clarity, we will also include the z-dimension. The equation can thus be replaced by the following equations:

$$\begin{aligned}
\dot{x} &= v_x \\
\dot{y} &= v_y \\
\dot{z} &= v_z \\
v_x &= \frac{-\mu x}{\sqrt{x^2+y^2+z^2}} - \frac{1}{2}\rho v\left(\frac{C_D A}{m}\right)v_x \\
v_y &= \frac{-\mu y}{\sqrt{x^2+y^2+z^2}} - \frac{1}{2}\rho v\left(\frac{C_D A}{m}\right)v_y \\
v_z &= \frac{-\mu z}{\sqrt{x^2+y^2+z^2}} - \frac{1}{2}\rho v\left(\frac{C_D A}{m}\right)v_z
\end{aligned}$$

Let us also note some important points before proceeding. We can distinguish between two cases for the atmosphere:

1. The atmosphere is stationary. In this case, we can use directly the satellite's velocity components for each coordinate in our integrator.
2. The atmosphere rotates with the earth. We will now have to compute the relative velocity to use in our integrator. The relative velocity is given by the following formula:

$$\overrightarrow{v_{rel}} = \overrightarrow{v} - \overrightarrow{v_{atm}}$$

If the atmosphere rotates with the earth, whose angular velocity is ω_E, then relative to the origin O of the geocentric equatorial reference frame, the velocity of the atmosphere will be given by the formula $\overrightarrow{v_{atm}} = \overrightarrow{\omega_E} \times \overrightarrow{r}$, resulting in:

$$\overrightarrow{v_{rel}} = \overrightarrow{v} - \overrightarrow{\omega_E} \times \overrightarrow{r}$$

We can then break apart this vector to x, y and z components and use them in the equations above. We will make a comparison between these two cases to see how different they are. We will also predict the time it takes for the spacecraft to reach the surface of the earth using Gauss' equation, namely:

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -\rho\sqrt{\mu a}\left(\frac{C_T S}{m_s}\right)$$

The integrator which we will use is similar to the one of the previous exercise (fourth order Runge-Kutta). However, to avoid any overflow issues, we will add an extra stopping condition at an altitude $h = 100km$. This stopping condition is accurate, since orbits decay rapidly at this altitude (within minutes) and furthermore any artificial satellite will start burning up due to the enormous magnitude of aerodynamic heating (thus it will not be operational). We will then plot altitude vs time diagrams for each case until the satellite decays and we will also plot a 2D view of the orbit on the XY plane.

Finally, for the coefficient of drag, a value of $C_T = 2.2$ is considered.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
# Make jupyter export images as .pdf files for higher quality
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
# Silent run for NaNs
import warnings
warnings.filterwarnings('ignore')

def runge_kutta_4(x_0, y_0, z_0, vx_0, vy_0, vz_0, mu, C_t, S, M_sc, t_start,␣
↪t_end, h, relative):
    # 4th order Runge Kutta orbit propagator
    # Input:
    # x_0, y_0, z_0, namely the coordinates of the position vector [meters]
    # vx_0, vy_0, vz_0, namely the coordinates of the velocity vector [m/s]
    # mu, mu constant
    # t_start, initial time [seconds]
    # t_end, propagation end time [seconds]
    # h, integration step size [seconds]
    # relative, 0 or 1, if 0 the integrator will not compute relative velocity␣
↪to atmosphere, if 1 it will
    # Output is 7 vectors containing position, velocity and time information at␣
↪each integration step

    min_propagation_altitude = 100000
    R_earth = 6371000
    # Log initial values
    tn = [t_start]
    xn = [x_0]
    yn = [y_0]
    zn = [z_0]
    vxn = [vx_0]
    vyn = [vy_0]
    vzn = [vz_0]
    counter = 0
    # Main RK4 loop
```

```
    while tn[counter] < t_end:
        # Calculate k1 values
        k1_x = fx(vx_0)
        k1_y = fy(vy_0)
        k1_z = fz(vz_0)
        if relative == 0:
            vx_rel, vy_rel, vz_rel = vx_0, vy_0, vz_0
        else:
            vx_rel, vy_rel, vz_rel = relative_velocity(x_0, y_0, z_0, vx_0,␣
↪vy_0, vz_0)
        k1_vx = fv_x(x_0, y_0, z_0, mu, vx_rel, vy_rel, vz_rel, C_t, S, M_sc)
        k1_vy = fv_y(x_0, y_0, z_0, mu, vx_rel, vy_rel, vz_rel, C_t, S, M_sc)
        k1_vz = fv_z(x_0, y_0, z_0, mu, vx_rel, vy_rel, vz_rel, C_t, S, M_sc)
        # Calculate midpoint values
        mid_x = x_0 + k1_x * h / 2
        mid_y = y_0 + k1_y * h / 2
        mid_z = z_0 + k1_z * h / 2
        mid_vx = vx_0 + k1_vx * h / 2
        mid_vy = vy_0 + k1_vy * h / 2
        mid_vz = vz_0 + k1_vz * h / 2
        if relative == 0:
            vx_rel, vy_rel, vz_rel = mid_vx, mid_vy, mid_vz
        else:
            vx_rel, vy_rel, vz_rel = relative_velocity(mid_x, mid_y, mid_z,␣
↪mid_vx, mid_vy, mid_vz)
        # Calculate k2 values
        k2_x = fx(mid_vx)
        k2_y = fy(mid_vy)
        k2_z = fz(mid_vz)
        k2_vx = fv_x(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k2_vy = fv_y(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k2_vz = fv_z(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        # Calculate next midpoint values
        mid_x = x_0 + k2_x * h / 2
        mid_y = y_0 + k2_y * h / 2
        mid_z = z_0 + k2_z * h / 2
        mid_vx = vx_0 + k2_vx * h / 2
        mid_vy = vy_0 + k2_vy * h / 2
        mid_vz = vz_0 + k2_vz * h / 2
        if relative == 0:
            vx_rel, vy_rel, vz_rel = mid_vx, mid_vy, mid_vz
        else:
            vx_rel, vy_rel, vz_rel = relative_velocity(mid_x, mid_y, mid_z,␣
↪mid_vx, mid_vy, mid_vz)
```

```
        # Calculate k3 values
        k3_x = fx(mid_vx)
        k3_y = fy(mid_vy)
        k3_z = fz(mid_vz)
        k3_vx = fv_x(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k3_vy = fv_y(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k3_vz = fv_z(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        # Calculate next midpoint values
        mid_x = x_0 + k3_x * h
        mid_y = y_0 + k3_y * h
        mid_z = z_0 + k3_z * h
        mid_vx = vx_0 + k3_vx * h
        mid_vy = vy_0 + k3_vy * h
        mid_vz = vz_0 + k3_vz * h
        if relative == 0:
            vx_rel, vy_rel, vz_rel = mid_vx, mid_vy, mid_vz
        else:
            vx_rel, vy_rel, vz_rel = relative_velocity(mid_x, mid_y, mid_z,␣
↪mid_vx, mid_vy, mid_vz)
        # Calculate k4 values
        k4_x = fx(mid_vx)
        k4_y = fy(mid_vy)
        k4_z = fz(mid_vz)
        k4_vx = fv_x(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k4_vy = fv_y(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        k4_vz = fv_z(mid_x, mid_y, mid_z, mu, vx_rel, vy_rel, vz_rel, C_t, S,␣
↪M_sc)
        # Compute r, v values and append to list
        xn.append(xn[counter] + (h / 6) * (k1_x + 2 * k2_x + 2 * k3_x + k4_x))
        yn.append(yn[counter] + (h / 6) * (k1_y + 2 * k2_y + 2 * k3_y + k4_y))
        zn.append(zn[counter] + (h / 6) * (k1_z + 2 * k2_z + 2 * k3_z + k4_z))
        vxn.append(vxn[counter] + (h / 6) * (k1_vx + 2 * k2_vx + 2 * k3_vx +␣
↪k4_vx))
        vyn.append(vyn[counter] + (h / 6) * (k1_vy + 2 * k2_vy + 2 * k3_vy +␣
↪k4_vy))
        vzn.append(vzn[counter] + (h / 6) * (k1_vz + 2 * k2_vz + 2 * k3_vz +␣
↪k4_vz))
        # Prepare for the next iteration and reset values
        counter += 1
        tn.append(tn[counter - 1] + h)
        x_0 = xn[counter]
```

```
        y_0 = yn[counter]
        z_0 = zn[counter]
        vx_0 = vxn[counter]
        vy_0 = vyn[counter]
        vz_0 = vzn[counter]
        r_check = np.sqrt(pow(x_0, 2) + pow(y_0, 2) + pow(z_0, 2))
        if r_check < R_earth + min_propagation_altitude:
            print('Minimum altitude reached, leading to rapid decay of the␣
↪orbit. Exiting propagation...')
            break
    return xn, yn, zn, vxn, vyn, vzn, tn


def gauss_integrator(a_0, mu, c_t, S, m_s, t_start, t_end, h):
    # Propagates the orbit's semimajor axis using Gauss' equation
    # Input:
    min_propagation_altitude = 120000
    R_earth = 6371000
    tn = [t_start]
    an = [a_0]
    counter = 0
    while tn[counter] < t_end:
        k1_a = fa(a_0, mu, c_t, S, m_s)
        mid_a = a_0 + k1_a * h / 2
        k2_a = fa(mid_a, mu, c_t, S, m_s)
        mid_a = a_0 + k2_a * h / 2
        k3_a = fa(mid_a, mu, c_t, S, m_s)
        mid_a = a_0 + k3_a * h
        k4_a = fa(mid_a, mu, c_t, S, m_s)
        an.append(an[counter] + (h / 6) * (k1_a + 2 * k2_a + 2 * k3_a + k4_a))
        counter += 1
        tn.append(tn[counter - 1] + h)
        a_0 = an[counter]
        if a_0 < R_earth + min_propagation_altitude:
            print('Minimum altitude reached, leading to rapid decay of the␣
↪orbit. Exiting propagation...')
            break
    return an, tn


def fa(a, mu, c_t, S, m_s):
    # Assuming a'(t)=-rho*sqrt(mu*a)*Ct*S/m_s
    rho = atmospheric_density(a)
    return -rho*np.sqrt(mu*a)*c_t*S/m_s


def fx(v_x):
```

```
    # Assuming x'(t)=v_x
    return v_x


def fy(v_y):
    # Assuming y'(t)=v_y
    return v_y


def fz(v_z):
    # Assuming z'(t)=v_z
    return v_z


def fv_x(x, y, z, mu, v_x, v_y, v_z, Ct, S, M_sc):
    # Assuming v_x'(t)=-mu*x/(sqrt(x^2+y^2+z^2))^3
    r_vector_magnitude = np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2))
    v_vector_magnitude = np.sqrt(pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2))
    rho = atmospheric_density(r_vector_magnitude)
    return -mu * x / pow(r_vector_magnitude, 3) -␣
↪(rho*v_vector_magnitude*Ct*S*v_x/(2*M_sc))


def fv_y(x, y, z, mu, v_x, v_y, v_z, Ct, S, M_sc):
    # Assuming v_y'(t)=-mu*y/(sqrt(x^2+y^2+z^2))^3
    r_vector_magnitude = np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2))
    v_vector_magnitude = np.sqrt(pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2))
    rho = atmospheric_density(r_vector_magnitude)
    return -mu * y / pow(r_vector_magnitude, 3) -␣
↪(rho*v_vector_magnitude*Ct*S*v_y/(2*M_sc))


def fv_z(x, y, z, mu, v_x, v_y, v_z, Ct, S, M_sc):
    # Assuming v_z'(t)=-mu*z/(sqrt(x^2+y^2+z^2))^3
    r_vector_magnitude = np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2))
    v_vector_magnitude = np.sqrt(pow(v_x, 2) + pow(v_y, 2) + pow(v_z, 2))
    rho = atmospheric_density(r_vector_magnitude)
    return - mu * z / pow(r_vector_magnitude, 3) -␣
↪(rho*v_vector_magnitude*Ct*S*v_z/(2*M_sc))


def atmospheric_density(r_magnitude):
    # Computes atmospheric density at a specific height above the Earth
    # Input is radius from the center of the Earth [m]
    # Output is rho [kg/m^3]
    R_earth = 6371
    height = r_magnitude*pow(10, -3) - R_earth
```

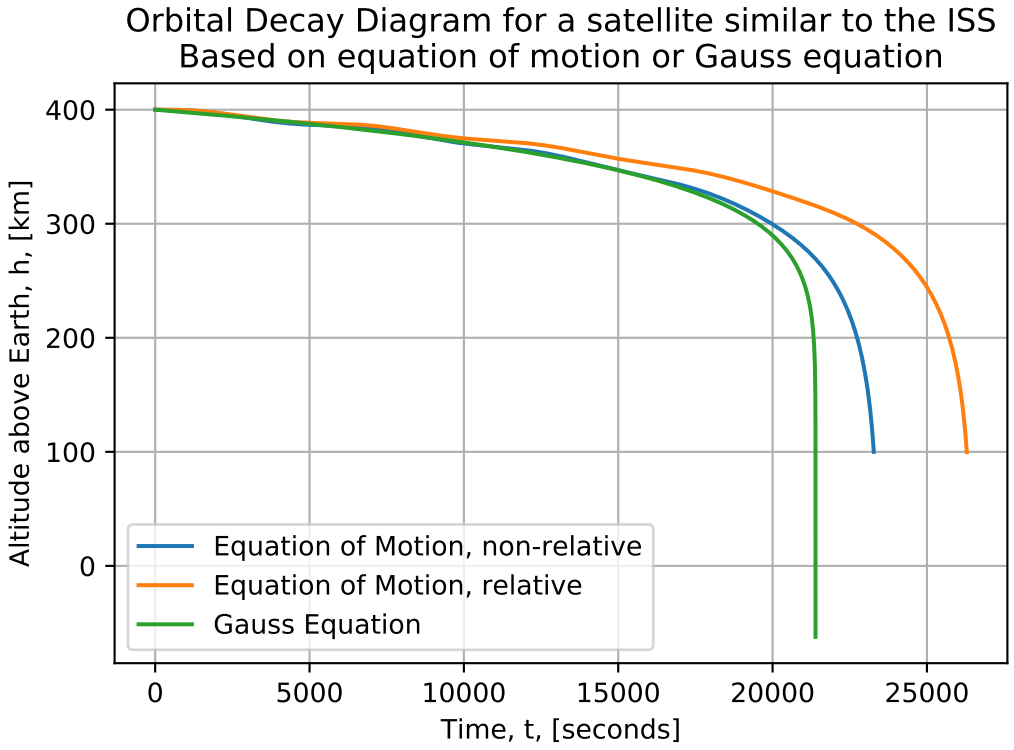
```
    if height >= 15:
        rho = pow(0.1 * height, -7.5)
    else:
        rho = 0.1
    return rho


def relative_velocity(x, y, z, vx, vy, vz):
    # Computes spacecraft's velocity relative to the atmosphere
    r = np.array([x, y, z])
    v = np.array([vx, vy, vz])
    w_z = np.array([0, 0, 7.2921150*pow(10, -5)])
    v_rel = v - np.cross(w_z, r)
    return v_rel[0], v_rel[1], v_rel[2]


# Constants definition
G = 6.6743 * pow(10, -11)
M_earth = 5.977 * pow(10, 24)
mu_earth = G * M_earth
R_earth = 6371000  # [meters]

# Orbit definition
H = 400000  # Altitude, [meters]
a = R_earth + H  # Semi-major axis, [meters]
i = 0  # Inclination, [deg]
e = 0  # Eccentricity, []
Omega = 0  # Longitude of the Ascending Note [deg]
omega = 90  # Argument of pericenter [deg]
M = 0  # Mean anomaly, [deg]
T = 2 * np.pi * np.sqrt(pow(a, 3) / mu_earth)  # Orbital period, [seconds]

# Atmospheric drag properties
C_t = 2.2
S = 8000
M_spacecraft = 400

# Compute initial position and velocity vector from orbital elements
r_vector, v_vector = orbital_elements_to_cartesian(a, e, i, Omega, omega, M,␣
 ↪mu_earth)
x = r_vector[0]
y = r_vector[1]
z = r_vector[2]
vx = v_vector[0]
vy = v_vector[1]
vz = v_vector[2]
```

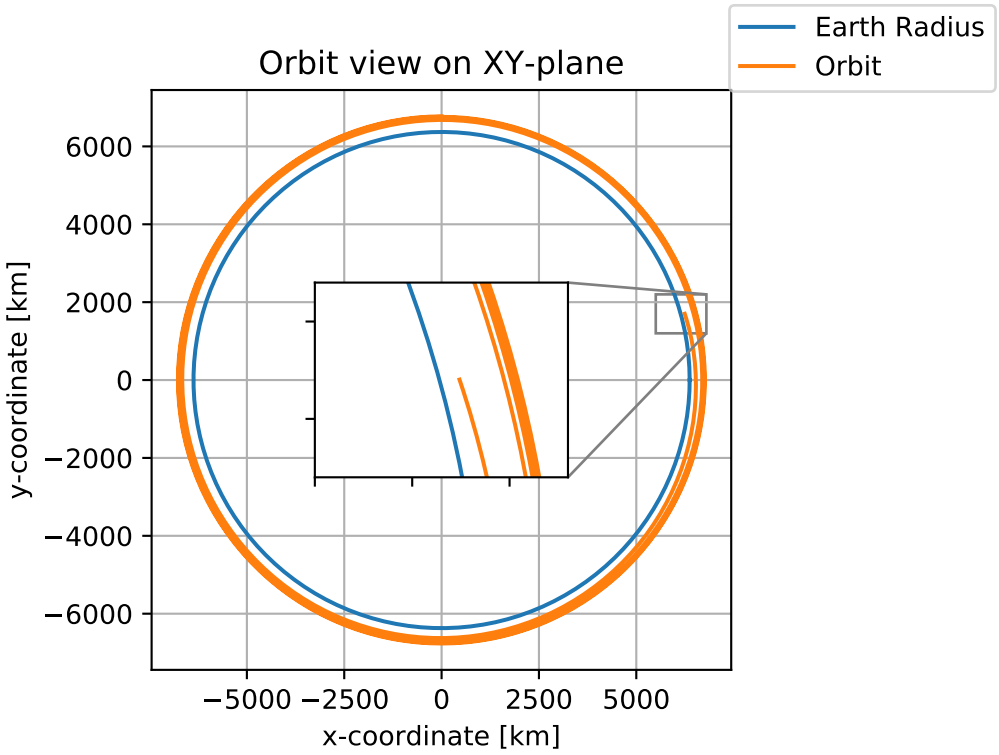
```
# Propagate orbit (all times in seconds)
start_time = 0
end_time = 10*T
time_step = 1
# Relative velocity case
xn_rel, yn_rel, zn_rel, vxn_rel, vyn_rel, vzn_rel, tn_rel = \
    runge_kutta_4(x, y, z, vx, vy, vz, mu_earth, C_t, S, M_spacecraft,␣
 ↪start_time, end_time, time_step, 1)
# Non-relative velocity case
xn, yn, zn, vxn, vyn, vzn, tn = \
    runge_kutta_4(x, y, z, vx, vy, vz, mu_earth, C_t, S, M_spacecraft,␣
 ↪start_time, end_time, time_step, 0)


# Compute r magnitude
r_rel = np.zeros(np.size(xn_rel))
h_rel = np.zeros(np.size(xn_rel))
r = np.zeros(np.size(xn))
h = np.zeros(np.size(xn))
for i in range(0, np.size(xn)):
    r[i] = np.sqrt(pow(xn[i], 2) + pow(yn[i], 2) + pow(zn[i], 2))
    h[i] = r[i] - R_earth
for i in range(0, np.size(xn_rel)):
    r_rel[i] = np.sqrt(pow(xn_rel[i], 2) + pow(yn_rel[i], 2) + pow(zn_rel[i],␣
 ↪2))
    h_rel[i] = r_rel[i] - R_earth
# Convert to xn, yn to km an
r = np.divide(r, 1000)
h = np.divide(h, 1000)
r_rel = np.divide(r_rel, 1000)
h_rel = np.divide(h_rel, 1000)
xn_rel = np.divide(xn_rel, 1000)
yn_rel = np.divide(yn_rel, 1000)

# Test with Gauss equation
start_time = 0
end_time = 10*T
time_step = 1
a_vector, t_vector = gauss_integrator(a, mu_earth, C_t, S, M_spacecraft,␣
 ↪start_time, end_time, time_step)
a_vector = np.subtract(a_vector, R_earth)
a_vector = np.divide(a_vector, 1000)

# Plot radius vs time
plt.figure()
plt.plot(tn, h,  label='Equation of Motion, non-relative')
plt.plot(tn_rel, h_rel,  label='Equation of Motion, relative')
```

```
plt.plot(t_vector, a_vector, label='Gauss Equation')
plt.xlabel('Time, t, [seconds]')
plt.ylabel('Altitude above Earth, h, [km]')
plt.title('Orbital Decay Diagram for a satellite similar to the ISS\nBased on␣
 ↪equation of motion or Gauss equation')
plt.grid()
plt.legend()

# Plot orbit on XY-plane
# Plot the earth
radius = R_earth/1000
theta = np.linspace(0, 2*np.pi, 1000)
x_earth = radius*np.cos(theta)
y_earth = radius*np.sin(theta)
# Plot the orbit
fig, ax = plt.subplots()
ax.plot(x_earth, y_earth, label='Earth Radius')
ax.plot(xn_rel, yn_rel, label='Orbit')
ax.grid()
ax.set_xlabel('x-coordinate [km]')
ax.set_ylabel('y-coordinate [km]')
ax.set_title('Orbit view on XY-plane')
ax.axis('scaled')
# Create zoom-in window
zoom1 = zoomed_inset_axes(ax, zoom = 5, loc=10)
zoom1.plot(x_earth, y_earth)
zoom1.plot(xn_rel, yn_rel)
x1, x2, y1, y2 = 5500, 6800, 1200, 2200
zoom1.set_xlim(x1, x2)
zoom1.set_ylim(y1, y2)
plt.xticks(visible=False)
plt.yticks(visible=False)
mark_inset(ax, zoom1, loc1=1, loc2=4, fc="none", ec="0.5")
fig.legend()
plt.show()

time_taken_non_relative = tn[np.size(tn)-1]
time_taken_relative = tn_rel[np.size(tn_rel)-1]
time_taken_gauss = t_vector[np.size(t_vector)-1]
orbits_taken_non_relative = tn[np.size(tn)-1]/T
orbits_taken_relative = tn_rel[np.size(tn_rel)-1]/T
orbits_taken_gauss = t_vector[np.size(t_vector)-1]/T
print("For non-relative case, using equation of motion, satellite decays after␣
 ↪%.2f seconds (or after %.2f orbits)"
      % (time_taken_non_relative, orbits_taken_non_relative))
print("For relative case, using equation of motion, satellite decays after %.2f␣
 ↪seconds (or after %.2f orbits)"
```

```
      % (time_taken_relative, orbits_taken_relative))
print("Using Gauss equation, satellite decays after %.2f seconds (or after %.2f␣
 ↪orbits)"
      % (time_taken_gauss, orbits_taken_gauss))
```

```
Minimum altitude reached, leading to rapid decay of the orbit. Exiting
propagation…
Minimum altitude reached, leading to rapid decay of the orbit. Exiting
propagation…
Minimum altitude reached, leading to rapid decay of the orbit. Exiting
propagation…
```






```
For non-relative case, using equation of motion, satellite decays after 23279.00
seconds (or after 4.20 orbits)
For relative case, using equation of motion, satellite decays after 26292.00
seconds (or after 4.74 orbits)
Using Gauss equation, satellite decays after 21391.00 seconds (or after 3.86
orbits)
```

2.3 Discussion

We note that in every case the integrator stops using the altitude stopping condition. From the first diagram, we can deduce that the relative and non-relative case have an almost identical shape, with the relative case decaying a little bit later since the atmosphere moves in the same direction as the spacecraft, resulting in a smaller relative velocity and thus the spacecraft decays at a later time in this case. We also note that the error for the time to re-entry between the non-relative case and the Gauss equation case is around 8.11%.

3 Exercise 2.3

3.1 Problem statement

Include the solar radiation pressure (remove atmospheric drag). The integration shall take place for time equal to 6000 orbits of the satellite. Assume that the sun is performing a circular motion with frequency $n_0 = 1deg/day$. Repeat the integration two times (a) without and (b) with shadowing

of the sun from the earth. Compare the diagrams $a(t)$ and $e(t)$ of the two orbits.

3.2 Solution

The initial orbital elements which we will use are identical to the previous exercise, namely:

Orbital Element	Value
Semi-major axis	6771000 m
Eccentricity	0
Inclination	0°
Longitude of the ascending node	0°
Argument of pericenter	90°
Mean anomaly	0°

We now no longer consider the atmospheric drag and we only consider the effect of the solar radiation pressure acting on our spacecraft. The equations of motions can thus be written as:

$$\ddot{\overrightarrow{r}} + \frac{\mu}{r^2}\frac{\overrightarrow{r}}{r} = \frac{\Phi A C_p}{cm}\frac{\vec{r} - \overrightarrow{r_S}}{|\vec{r} - \overrightarrow{r_S}|}$$
$$\ddot{\overrightarrow{r}} = -\frac{\mu \overrightarrow{r}}{r^3} + \frac{\Phi A C_p}{cm}\frac{\vec{r} - \overrightarrow{r_S}}{|\vec{r} - \overrightarrow{r_S}|}$$

Where $\overrightarrow{r_S}$ refers to the sun position vector and $\vec{r}$ refers to the spacecraft position vector. This equation can again be replaced by 2 first order equations for each cartesian coordinate. For clarity, we will also include the z-dimension. The equation can thus be replaced by the following equations:

$$\dot{x} = v_x$$
$$\dot{y} = v_y$$
$$\dot{z} = v_z$$
$$v_x = \frac{-\mu x}{\sqrt{x^2+y^2+z^2}} + \frac{\Phi A C_p}{cm}\frac{x - x_S}{|\vec{r} - \overrightarrow{r_S}|}$$
$$v_y = \frac{-\mu y}{\sqrt{x^2+y^2+z^2}} + \frac{\Phi A C_p}{cm}\frac{y - y_S}{|\vec{r} - \overrightarrow{r_S}|}$$
$$v_z = \frac{-\mu z}{\sqrt{x^2+y^2+z^2}} + \frac{\Phi A C_p}{cm}\frac{z - z_S}{|\vec{r} - \overrightarrow{r_S}|}$$

To compute the solar flux, we will use the following formula *Vallado, 2013*:

$$\Phi = \frac{1358}{1.004 + 0.0334\cos\left(D_{\text{ang}}\right)}\frac{\text{W}}{\text{m}^2}$$

where D_{ang} refers to the phase angle of the earth and has to be computed at each time step, basically giving the variation of the solar flux throughout a year (since Earth's orbit around the sun is not circular). We are also given the properties of the spacecraft $M_{spacecraft} = 400kg$, $A = 8000m^2$ and $C_p = 1.5$. We will again use a fourth order Runge-Kutta integrator to solve our 6 differential equations, and we will propagate the spacecraft for 6000 orbits. We will also include a notifier in case the spacecraft touches the surface of the Earth. Finally, we will use a transformation from cartesian coordinates to orbital elements in order to plot the semimajor axis $a(t)$ and the eccentricity $e(t)$.

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
import time
# Make jupyter export images as .pdf files for higher quality
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
# Silent run for NaNs
import warnings
warnings.filterwarnings('ignore')



def cartesian_to_orbital_elements(r_vector, v_vector, mu):
    # Computes cartesian position and velocity vector given some orbital␣
 ↪elements
    # Input:
    # r_vector, y_vector
    # mu [m^3/(kg*s^2)] (GM)
    # Output:
    # a [m] (Semi-major axis)
    # e [] (Eccentricity)

    # Compute magnitude and angular momentum
    r = np.sqrt(pow(r_vector[0], 2) + pow(r_vector[1], 2) + pow(r_vector[2], 2))
    v = np.sqrt(pow(v_vector[0], 2) + pow(v_vector[1], 2) + pow(v_vector[2], 2))
    h_vector = np.cross(r_vector, v_vector)
    h = np.sqrt(pow(h_vector[0], 2) + pow(h_vector[1], 2) + pow(h_vector[2], 2))
    # Compute a, e and i
    a = mu / ((2 * mu / r) - pow(v, 2))
    e_vector = (np.cross(v_vector, h_vector)/mu) - (r_vector/r)
    e = np.sqrt(pow(e_vector[0], 2) + pow(e_vector[1], 2) + pow(e_vector[2], 2))
    return a, e


def runge_kutta_4(x_0, y_0, z_0, vx_0, vy_0, vz_0, mu, t_start, tmax, h, x_sun,␣
 ↪y_sun, z_sun, A, c_p, m_sc, case):
    # 4th order Runge Kutta orbit propagator
    # Input:
```

```
    # x_0, y_0, z_0, namely the coordinates of the position vector [meters]
    # vx_0, vy_0, vz_0, namely the coordinates of the velocity vector [m/s]
    # mu, mu constant
    # t_start, initial time [seconds]
    # t_end, propagation end time [seconds]
    # h, integration step size [seconds]
    # x_sun, y_sum, z_sun, namely the coordinates of the sun position vector␣
↪[meters]
    # A, area exposed to SRP [meters^2]
    # C_p, coefficient of reflectivity []
    # m_sc, spacecraft mass [kg]
    # Case, use 0 to disable shadow from earth or 1 to enable it

    # Output is 7 vectors containing position, velocity and time information at␣
↪each integration step

    # Log initial values
    tn = [t_start]
    xn = [x_0]
    yn = [y_0]
    zn = [z_0]
    vxn = [vx_0]
    vyn = [vy_0]
    vzn = [vz_0]
    counter = 0
    ground_check = 1
    R_earth = 6371000
    # Main RK4 loop
    while tn[counter] < tmax:
        # Calculate sun position
        time = tn[counter]
        x_sun, y_sun = sun_position_calculator(x_sun, y_sun, time)
        # Calculate k1 values
        k1_x = fx(vx_0)
        k1_y = fy(vy_0)
        k1_z = fz(vz_0)
        k1_vx = fv_x(x_0, y_0, z_0, mu, x_sun, y_sun, z_sun, A, c_p, m_sc,␣
↪time, case)
        k1_vy = fv_y(x_0, y_0, z_0, mu, x_sun, y_sun, z_sun, A, c_p, m_sc,␣
↪time, case)
        k1_vz = fv_z(x_0, y_0, z_0, mu, x_sun, y_sun, z_sun, A, c_p, m_sc,␣
↪time, case)
        # Calculate midpoint values
        mid_x = x_0 + k1_x * h/2
        mid_y = y_0 + k1_y * h/2
        mid_z = z_0 + k1_z * h/2
        mid_vx = vx_0 + k1_vx * h/2
```

```
        mid_vy = vy_0 + k1_vy * h/2
        mid_vz = vz_0 + k1_vz * h/2
        # Calculate k2 values
        k2_x = fx(mid_vx)
        k2_y = fy(mid_vy)
        k2_z = fz(mid_vz)
        k2_vx = fv_x(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k2_vy = fv_y(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k2_vz = fv_z(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        # Calculate next midpoint values
        mid_x = x_0 + k2_x * h / 2
        mid_y = y_0 + k2_y * h / 2
        mid_z = z_0 + k2_z * h / 2
        mid_vx = vx_0 + k2_vx * h / 2
        mid_vy = vy_0 + k2_vy * h / 2
        mid_vz = vz_0 + k2_vz * h / 2
        # Calculate k3 values
        k3_x = fx(mid_vx)
        k3_y = fy(mid_vy)
        k3_z = fz(mid_vz)
        k3_vx = fv_x(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k3_vy = fv_y(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k3_vz = fv_z(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        # Calculate next midpoint values
        mid_x = x_0 + k3_x * h
        mid_y = y_0 + k3_y * h
        mid_z = z_0 + k3_z * h
        mid_vx = vx_0 + k3_vx * h
        mid_vy = vy_0 + k3_vy * h
        mid_vz = vz_0 + k3_vz * h
        # Calculate k4 values
        k4_x = fx(mid_vx)
        k4_y = fy(mid_vy)
        k4_z = fz(mid_vz)
        k4_vx = fv_x(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k4_vy = fv_y(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
        k4_vz = fv_z(mid_x, mid_y, mid_z, mu, x_sun, y_sun, z_sun, A, c_p,␣
↪m_sc, time, case)
```

```
        # Compute r, v values and append to list
        xn.append(xn[counter] + (h / 6) * (k1_x + 2 * k2_x + 2 * k3_x + k4_x))
        yn.append(yn[counter] + (h / 6) * (k1_y + 2 * k2_y + 2 * k3_y + k4_y))
        zn.append(zn[counter] + (h / 6) * (k1_z + 2 * k2_z + 2 * k3_z + k4_z))
        vxn.append(vxn[counter] + (h / 6) * (k1_vx + 2 * k2_vx + 2 * k3_vx +␣
↪k4_vx))
        vyn.append(vyn[counter] + (h / 6) * (k1_vy + 2 * k2_vy + 2 * k3_vy +␣
↪k4_vy))
        vzn.append(vzn[counter] + (h / 6) * (k1_vz + 2 * k2_vz + 2 * k3_vz +␣
↪k4_vz))
        r_check = np.sqrt(pow(x_0, 2) + pow(y_0, 2) + pow(z_0, 2))
        if ground_check == 1 and r_check < R_earth:
            print('Warning: Spacecraft altitude touched the surface of the␣
↪earth for time %d seconds.' % tn[counter])
            if case == 0:
                print('(No shadow case)')
            else:
                print('(Shadow case)')
            print('Continuing propagation...\n')
            ground_check = 0
        # Prepare for the next iteration and reset values
        counter += 1
        tn.append(tn[counter - 1] + h)
        x_0 = xn[counter]
        y_0 = yn[counter]
        z_0 = zn[counter]
        vx_0 = vxn[counter]
        vy_0 = vyn[counter]
        vz_0 = vzn[counter]
    return xn, yn, zn, vxn, vyn, vzn, tn


def fx(v_x):
    # Assuming x'(t)=v_x
    return v_x


def fy(v_y):
    # Assuming y'(t)=v_y
    return v_y


def fz(v_z):
    # Assuming z'(t)=v_z
    return v_z
```

```
def fv_x(x, y, z, mu, x_sun, y_sun, z_sun, A, c_p, m_sc, current_time,␣
↪shadow_enabled):
    # Assuming v_x'(t)=-mu*x/(sqrt(x^2+y^2+z^2))^3 + flux term
    c = 3*pow(10, 8)
    flux = flux_calculator(current_time)
    if shadow_enabled == 1:
        shadow = shadow_checker(x, y, z, x_sun, y_sun, z_sun)
    else:
        shadow = 1
    return -mu*x/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3) \
           + shadow*flux*A*c_p*(x-x_sun)/(c*m_sc*np.sqrt(pow(x-x_sun, 2) +␣
↪pow(y-y_sun,2) + pow(z-z_sun, 2)))


def fv_y(x, y, z, mu, x_sun, y_sun, z_sun, A, c_p, m_sc, current_time,␣
↪shadow_enabled):
    # Assuming v_y'(t)=-mu*y/(sqrt(x^2+y^2+z^2))^3 + flux term
    c = 3*pow(10, 8)
    flux = flux_calculator(current_time)
    if shadow_enabled == 1:
        shadow = shadow_checker(x, y, z, x_sun, y_sun, z_sun)
    else:
        shadow = 1
    return -mu*y/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3) \
           + shadow*flux*A*c_p*(y-y_sun)/(c*m_sc*np.sqrt(pow(x-x_sun, 2) +␣
↪pow(y-y_sun,2) + pow(z-z_sun, 2)))


def fv_z(x, y, z, mu, x_sun, y_sun, z_sun, A, c_p, m_sc, current_time,␣
↪shadow_enabled):
    # Assuming v_z'(t)=-mu*z/(sqrt(x^2+y^2+z^2))^3 + flux term
    c = 3*pow(10, 8)
    flux = flux_calculator(current_time)
    if shadow_enabled == 1:
        shadow = shadow_checker(x, y, z, x_sun, y_sun, z_sun)
    else:
        shadow = 1
    return -mu*z/pow(np.sqrt(pow(x, 2) + pow(y, 2) + pow(z, 2)), 3)


def flux_calculator(time):
    # Calculates the amount of flux received by the satellite, assuming it is␣
↪relatively close to the earth
    # The calculator assumes that the sun is initially (t=0) alligned with the␣
↪positive OX axis of the ICRF frame
    # Input is time since t=0 [seconds]
```

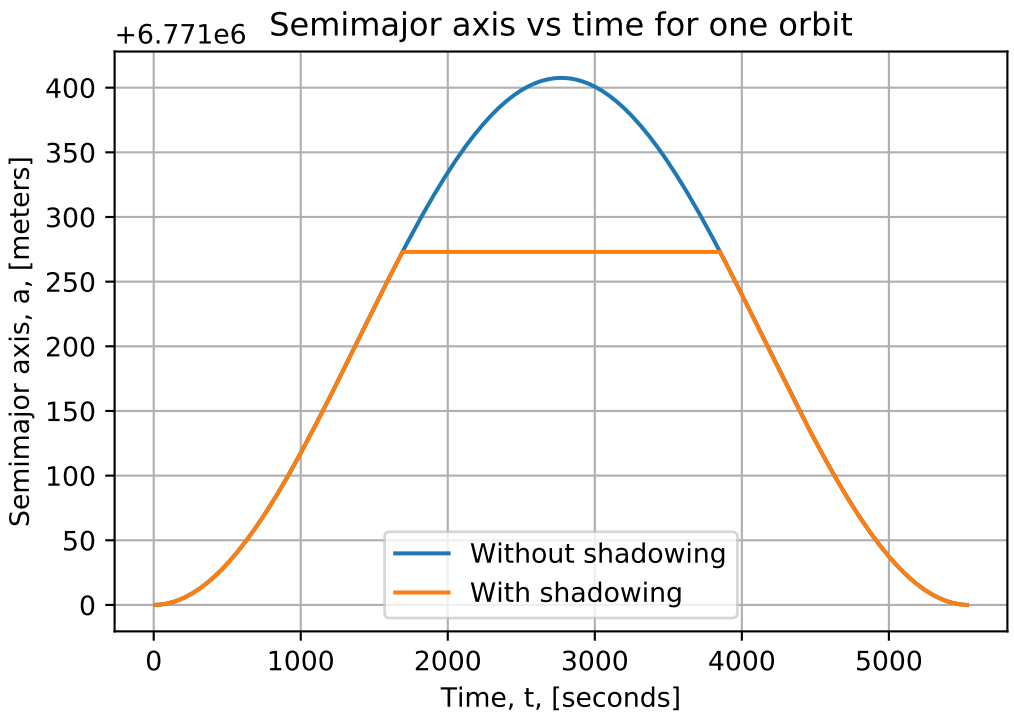
```
    # Output is flux [W/m^2
    # Convert to the desired units
    n = 1
    time = time/(60*60*24)
    phase_angle = time*n
    phase_angle = np.radians(phase_angle)
    # Make phase_angle in [0, 2*pi]
    while phase_angle >= 2*np.pi:
        phase_angle -= 2*np.pi
    flux = 1358/(1.004+0.0334*np.cos(phase_angle))
    return flux


def sun_position_calculator(x_initial, y_initial, propagation_time):
    # Rotates the sun around the earth in the ICRF frame
    # Input is the initial position vector of the sun [meters] and propagation␣
 ↪time [seconds]
    # Output is the position vector of the sun [meters] after propagation
    n_dot = 1  # [deg/day]
    propagation_time = propagation_time / (60 * 60 * 24)
    theta = n_dot*propagation_time
    theta = np.radians(theta)
    r = np.sqrt(pow(x_initial, 2) + pow(y_initial, 2))
    x_final = r*np.cos(theta)
    y_final = r*np.sin(theta)
    return x_final, y_final


def shadow_checker(x_sc, y_sc, z_sc, x_sun, y_sun, z_sun):
    # Computes if a spacecraft is inside Earth's shadow
    # Input:
    # Spacecraft position vector (x_sc, y_sc, z_sc) [meters]
    # Sun position vector (x_sun, y_sun, z_sun) [meters]
    # Output:
    # 0, if the spacecraft is in Earth's shadow
    # 1, if the spacecraft is in sunlight
    r_earth = 6371000
    # Compute vector magnitudes
    r_spacecraft = np.sqrt(pow(x_sc, 2) + pow(y_sc, 2) + pow(z_sc, 2))
    r_sun = np.sqrt(pow(x_sun, 2) + pow(y_sun, 2) + pow(z_sun, 2))
    # Compute theta values
    theta = np.arccos((x_sc*x_sun+y_sc*y_sun+z_sc*z_sun)/(r_spacecraft*r_sun))
    theta_1 = np.arccos(r_earth/r_spacecraft)
    theta_2 = np.arccos(r_earth/r_sun)
    # Decide if we are inside or outside Earth's shadow
    if theta_1+theta_2 <= theta:
        result = 0
```

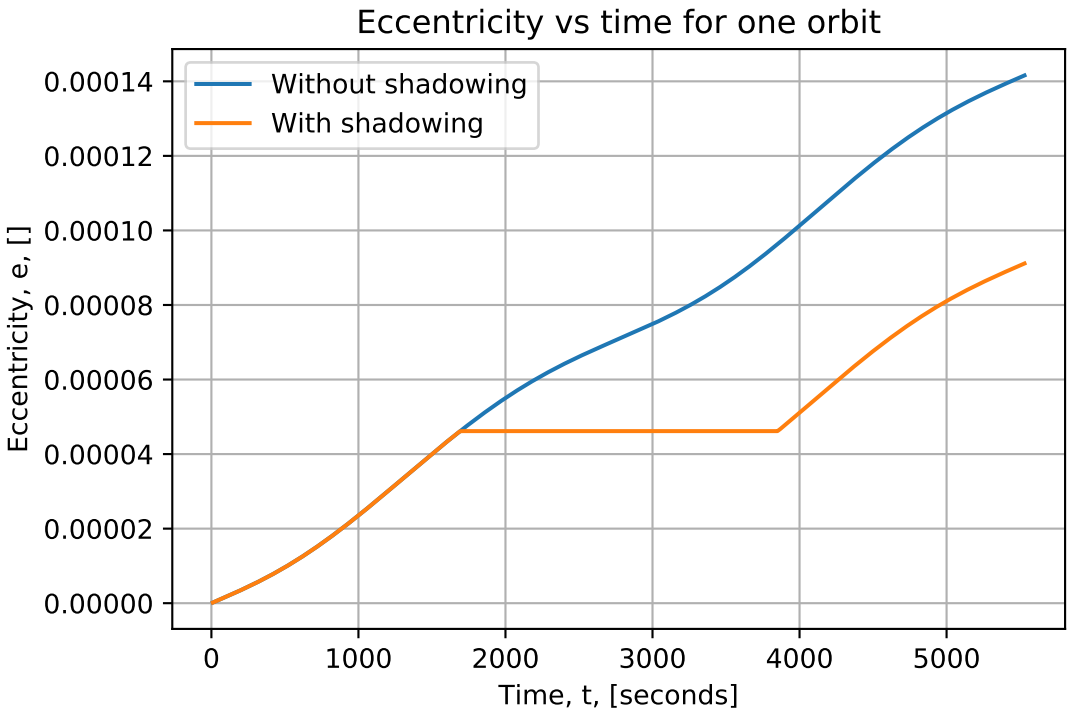
```
    else:
        result = 1
    return result


# Constants definition
G = 6.6743*pow(10, -11)
M_earth = 5.977*pow(10, 24)
mu_earth = G*M_earth
R_earth = 6371000  # [meters]

# Orbit definition
H = 400000  # Altitude, [meters]
a = R_earth + H  # Semi-major axis, [meters]
i = 0  # Inclination, [deg]
e = 0  # Eccentricity, []
Omega = 0  # Longitude of the Ascending Note [deg]
omega = 0  # Argument of pericenter [deg]
M = 0  # Mean anomaly, [deg]
T = 2 * np.pi * np.sqrt(pow(a, 3)/mu_earth)  # Orbital period, [seconds]

# Compute initial position and velocity vector from orbital elements
r_vector, v_vector = orbital_elements_to_cartesian(a, e, i, Omega, omega, M,␣
 ↪mu_earth)
x = r_vector[0]
y = r_vector[1]
z = r_vector[2]
vx = v_vector[0]
vy = v_vector[1]
vz = v_vector[2]

# Compute sun initial position

x_sun_initial = 149597870700.0
y_sun_initial = 0.0
z_sun_initial = 0.0

# Spacecraft properties

S = 8000
C_p = 1.5
M_spacecraft = 400

# Shadow? (0 is OFF, 1 is ON)
shadow_case = 0

# Propagate orbit (all times in seconds)
```

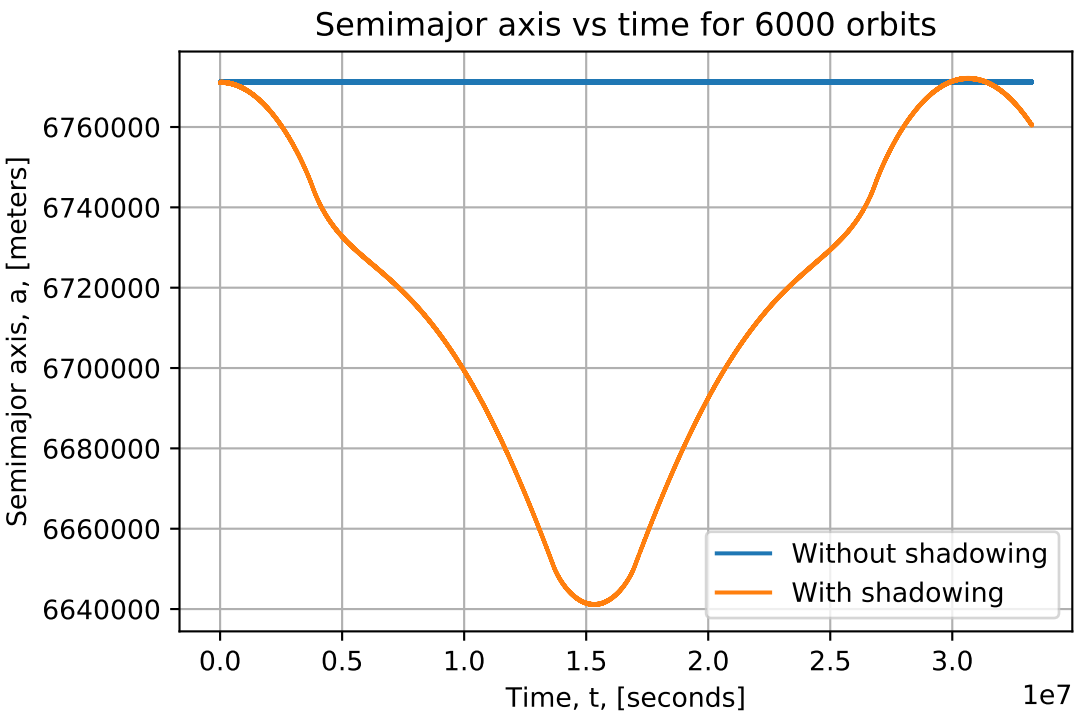
```
start_time = 0
end_time = 6000*T
time_step = 10
xn, yn, zn, vxn, vyn, vzn, tn = \
    runge_kutta_4(x, y, z, vx, vy, vz, mu_earth, start_time, end_time,␣
 ↪time_step,
                  x_sun_initial, y_sun_initial, z_sun_initial, S, C_p,␣
 ↪M_spacecraft, shadow_case)

# Propagate case and take shadow into account
shadow_case = 1
xn_s, yn_s, zn_s, vxn_s, vyn_s, vzn_s, tn_s = \
    runge_kutta_4(x, y, z, vx, vy, vz, mu_earth, start_time, end_time,␣
 ↪time_step,
                  x_sun_initial, y_sun_initial, z_sun_initial, S, C_p,␣
 ↪M_spacecraft, shadow_case)

# Log desired values
a_values = np.zeros(np.size(xn))
e_values = np.zeros(np.size(xn))
a_values_s = np.zeros(np.size(xn_s))
e_values_s = np.zeros(np.size(xn_s))
r_vector = np.column_stack((xn, yn, zn))
v_vector = np.column_stack((vxn, vyn, vzn))
r_vector_s = np.column_stack((xn_s, yn_s, zn_s))
v_vector_s = np.column_stack((vxn_s, vyn_s, vzn_s))
for counter in range(0, np.size(xn)):
    a_values[counter], e_values[counter] = \
        cartesian_to_orbital_elements(r_vector[counter][0:3],␣
 ↪v_vector[counter][0:3], mu_earth)
    a_values_s[counter], e_values_s[counter] = \
        cartesian_to_orbital_elements(r_vector_s[counter][0:3],␣
 ↪v_vector_s[counter][0:3], mu_earth)

# Plot for one period
start_index = 1
end_index = int(T/time_step)
plt.figure()
plt.plot(tn[start_index:end_index], a_values[start_index:end_index])
plt.plot(tn_s[start_index:end_index], a_values_s[start_index:end_index])
plt.grid()
plt.ylabel('Semimajor axis, a, [meters]')
plt.xlabel('Time, t, [seconds]')
plt.title('Semimajor axis vs time for one orbit')
plt.legend(['Without shadowing', 'With shadowing'])
plt.figure()
```

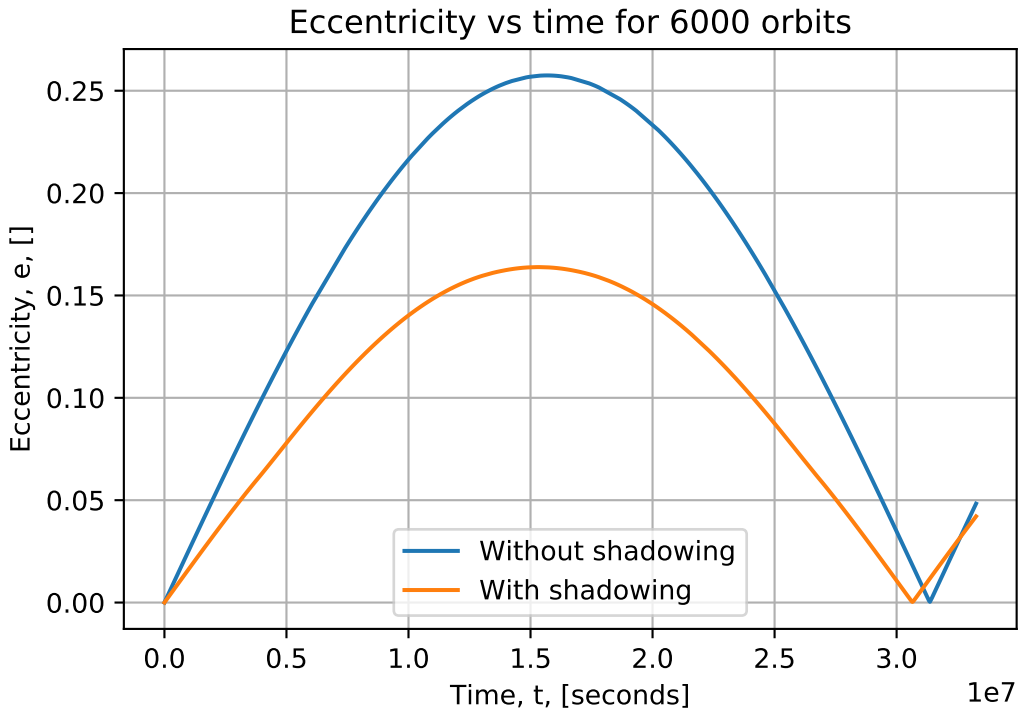
```
plt.plot(tn[start_index:end_index], e_values[start_index:end_index])
plt.plot(tn_s[start_index:end_index], e_values_s[start_index:end_index])
plt.grid()
plt.ylabel('Eccentricity, e, []')
plt.xlabel('Time, t, [seconds]')
plt.title('Eccentricity vs time for one orbit')
plt.legend(['Without shadowing', 'With shadowing'])

# Plot the entire orbit
plt.figure()
plt.plot(tn[1:np.size(xn)], a_values[1:np.size(xn)])
plt.plot(tn_s[1:np.size(xn_s)], a_values_s[1:np.size(xn_s)])
plt.grid()
plt.ylabel('Semimajor axis, a, [meters]')
plt.xlabel('Time, t, [seconds]')
plt.title('Semimajor axis vs time for 6000 orbits')
plt.legend(['Without shadowing', 'With shadowing'])
plt.figure()
plt.plot(tn[1:np.size(xn)], e_values[1:np.size(xn)])
plt.plot(tn_s[1:np.size(xn_s)], e_values_s[1:np.size(xn_s)])
plt.grid()
plt.ylabel('Eccentricity, e, []')
plt.xlabel('Time, t, [seconds]')
plt.title('Eccentricity vs time for 6000 orbits')
plt.legend(['Without shadowing', 'With shadowing'])
plt.legend(['Without shadowing', 'With shadowing'])
plt.show()
```

```
Warning: Spacecraft altitude touched the surface of the earth for time 2335090
seconds.
(No shadow case)
Continuing propagation…

Warning: Spacecraft altitude touched the surface of the earth for time 3515780
seconds.
(Shadow case)
Continuing propagation…
```


Semimajor axis vs time for one orbit
+6.771e6
Semimajor axis, a, [meters]
400
350
300
250
200
150
100
50
0
0
1000
2000
3000
4000
5000
Time, t, [seconds]
Without shadowing
With shadowing


Eccentricity vs time for one orbit
Eccentricity, e, []
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
0
1000
2000
3000
4000
5000
Time, t, [seconds]
Without shadowing
With shadowing


Semimajor axis vs time for 6000 orbits
Semimajor axis, a, [meters]
6760000
6740000
6720000
6700000
6680000
6660000
6640000
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1e7
Time, t, [seconds]
Without shadowing
With shadowing


Eccentricity vs time for 6000 orbits
Eccentricity, e, []
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1e7
Time, t, [seconds]
Without shadowing
With shadowing

3.3 Discussion

The first two plots show us the evolution of the semi-major axis $a(t)$ and the eccentricity $e(t)$ for one orbit. In both figures we can clearly see the interval for which the satellite enters the Earth's shadow, leading to no change for the value of the semi-major axis or the eccentricity throughout this interval, indicating that our code performs as expected by canceling the perturbation due to the solar radiation pressure when the spacecraft is inside the Earth's shadow. For the desired 6000 orbits, the image is different. First, we have to point out that in both cases the spacecraft collided with the surface of the planet after 2335090 seconds for the case without shadowing and after 3515780 seconds for the case with shadowing. Nevertheless, the propagation was not stopped in order to investigate what exactly happens. From the eccentricity figure, we note that the eccentricity rises quicker in the case without shadowing, due to the fact that the force is constantly being applied to the spacecraft. For the case where the shadow is taken into account, no force is applied for about 1/3 of each orbit (see the figure of $e(t)$ for one orbit), leading to slower rise of the eccentricity for this case. From the semi-major axis figure, considering the case without shadowing, we note that the oscillation we observed in the one orbit figure is maintained for the entire 6000 orbits with the semi-major axis remaining at an almost constant value. We do note that the spacecraft crashes into the Earth's surface due to the rise of the eccentricity, eventually leading to the perigee of the orbit becoming equal to the Earth's radius. For the shadowing case, we notice a drop in altitude in an oscillatory manner, leading to a variation of almost 100 km for the value of the semi-major axis. Yet again the semi-major axis value does not drop significantly, but the rise of the eccentricity again leads the spacecraft to the Earth's surface at the perigee.