Comparing the Difference Between Impulse and Extended Burns on Spacecraft Trajectories

July 25, 2024

1 Problem Statement

This paper aims to compare the effects of impulsive burns and continuous burns on spacecraft trajectories. Spacecraft in orbit frequently need to reposition, whether for rendezvous or other operational reasons. To instigate these orbital manoeuvres, thrust must be applied to the craft, typically in the form of a burn across a specified period, giving the change in velocity (Δv) required to complete the manoeuvre.

The modelling of these manoeuvres can be split into two methods:

1. **Impulsive Manoeuvre Modelling**: Impulsive manoeuvres are idealised manoeuvres in which thrust is applied instantaneously to change the magnitude and direction of the velocity vector. When thrust is applied, the position of the spacecraft is assumed to be static while the velocity changes [1]. This, of course, cannot exist in reality - requiring an infinite force to be applied across no change in time. Despite this, the impulsive model is often appropriate for describing the impact of high-thrust, low-burn times manoeuvres.
2. **Non-Impulsive Manoeuvre Modelling**: The second models the manoeuvre as a continuous application of thrust (and resulting acceleration) over a period of time, gradually changing the velocity. As a result of this, the position vector of the body is constantly moving while the force is applied. This is a more realistic representation of thrust modelling, albeit still making some approximations. It is best used to model low thrust, extended burn period manoeuvres [3].

2 Simulation Theory

The simulations in this section are split into impulse and non-impulse sections - outlining the process taken to provide the results found in section 3.

2.1 Hill-Clohessy-Wiltshire Relative Motion

Both simulations are modelled within a relative frame - projecting a chaser spacecraft in the reference frame of a target. This allows for simplified visual representation and simulation.

The Hill-Clohessy-Wiltshire (HCW) equations describe the relative motion of two satellites in close proximity to each other in a circular orbit around a central body. Assuming the reference satellite (chief) is in a circular orbit and the relative position vector of the deputy satellite with respect to the chief is (x, y, z), the HCW equations are given by [4]:

$$\ddot{x} - 3\omega^2 x - 2\omega\dot{y} = 0, \quad \ddot{y} + 2\omega\dot{x} = 0, \quad \ddot{z} + \omega^2 z = 0.$$

In these equations:

- ω is the orbital angular velocity of the chief satellite.
- x, y, and z represent the relative positions of the deputy satellite in the radial, in-track, and cross-track directions, respectively.
- $\dot{x}$ and $\dot{y}$ represent the relative velocities in the radial and in-track directions, respectively.
- $\ddot{x}$, $\ddot{y}$, and $\ddot{z}$ represent the relative accelerations.

These equations assume that the perturbations are small and that both satellites are point masses. The central body (Earth) exerts a central gravitational force, and the satellites are assumed to be in close enough proximity that the linearised approximation holds.

2.2 Impulsive Manoeuvre

The simulation of the impulsive manoeuvre took a simplified approach. In this case, the satellite was given initial parameters as follows:

1. An initial position vector (with x,y and z components),
2. An initial velocity vector (with x,y and z components),
3. A specified time-step for the burn to start,
4. A post-burn velocity vector (with x,y and z components).

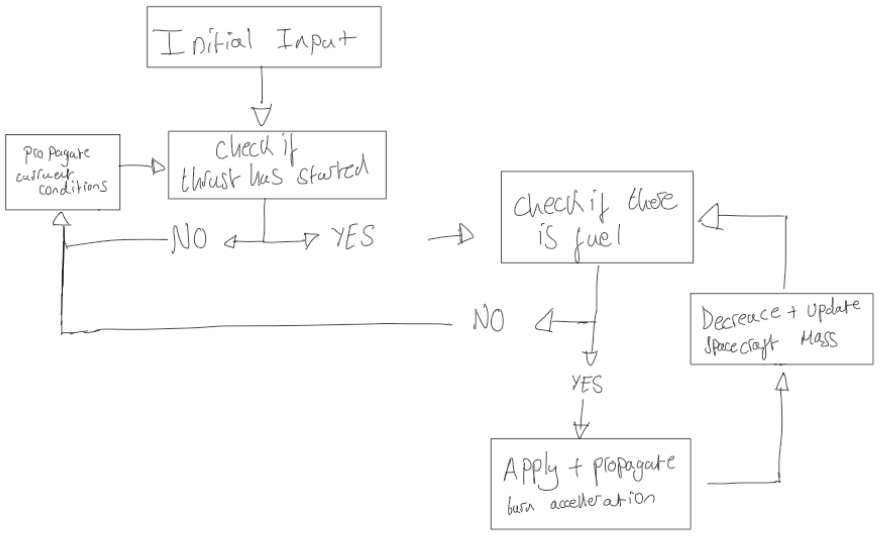
When the impulse was implemented, the velocity components were altered to those specified in the manoeuvre at a chosen time step while the position vector remained constant. The spacecraft was then propagated with its new velocity components going forward until the end of the simulation.

2.3 Non-Impulsive Manoeuvre

The simulation of the non-impulsive manoeuvre built upon the foundations of its impulsive counterpart - taking steps to apply the thrust imparted from an extended burn. As before, the body was given initial conditions, with a few changes:

1. An initial position vector (with x,y and z components),
2. An initial velocity vector (with x,y and z components),
3. A specified time-step for the burn to start,
4. A force vector for the thrust applied (with x,y and z components),
5. Wet and dry spacecraft masses,
6. A period for the burn.

At each time-step during the burn, acceleration is calculated from the spacecraft's mass and applied force, updating the velocity vector. The fuel mass used is then subtracted from the spacecraft's total mass. This process repeats until the burn ends or fuel runs out, after which the spacecraft's conditions are propagated.



Initial Input
Propagate current conditions
Check if thrust has started
NO
YES
Check if there is fuel
NO
Decreace + Update Spacecraft Mass
YES
Apply + propagate burn accelleration

Figure 1: Non-impulse code logic flowchart.

3 Results

The results section of this report displays the graphs and values output from both simulations in their respective sections. Due to the plots being three-dimensional, an effort has been made to take multiple screenshots for ease of understanding.

3.1 Simulation inputs

To facilitate a fair test, the simulation input parameters had to be as close to identical as possible between the two simulations. The input parameters used are detailed in tables 1, 2 and 3:

Table 1: Shared impulse and non-impulse standardised input parameters.

Parameter	**Value**
Satellite wet mass	5000 kg
Satellite dry mass	4000 kg
Gravitational parameter of Earth	398600.4418 km^3/s^2
Semi Major Axis	7000 km
Initial relative position vector of chaser	[10.0, 9.0, -30.0] km
Initial relative velocity vector of chaser	[3, 5, 0.1] km/s
Burn initiation time	2000 time steps
Time step	1 second
Simulation time period	10000 seconds

Table 2: Impulse input parameters.

Parameter	**Value**
Force vector of thrust	[0, 600000000, 0] N

Table 3: Non-impulse input parameters.

Parameter	**Value**
Force vector of thrust	[0, 100000, 0] N
Burn Duration	6000 seconds

3.2 Without Manoeuvre

The simulation was initially completed without either type of manoeuvre, acting as a control for later comparison.

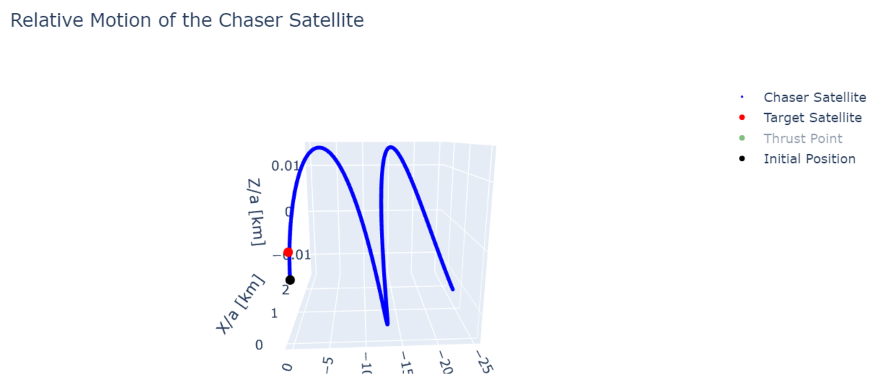


Figure 2: Simulation plot without manoeuvre (view 1).

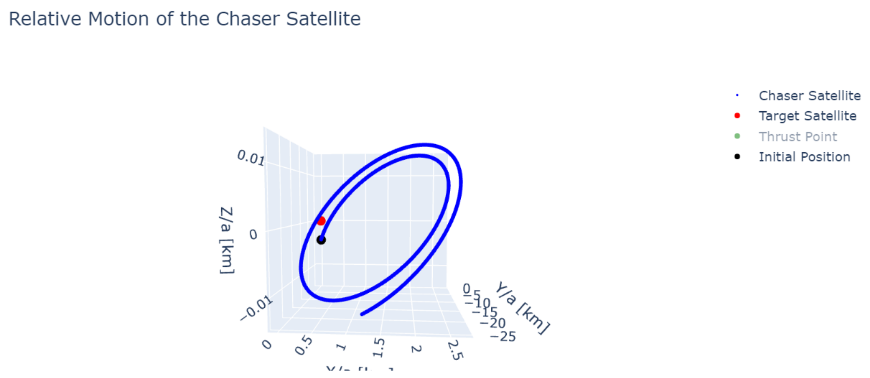


Figure 3: Simulation plot without manoeuvre (view 2).

3.3 Impulsive Manoeuvre

The manoeuvre was first modelled as an impulse, an instantaneous velocity change was imparted by a specified force.

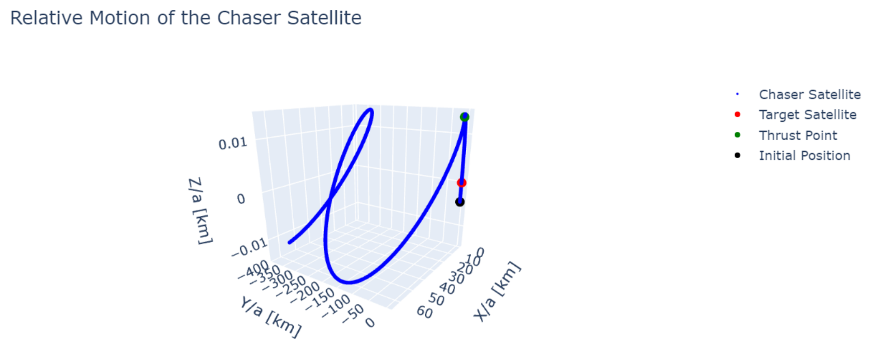


Figure 4: Simulation plot of impulsive manoeuvre (view 2).

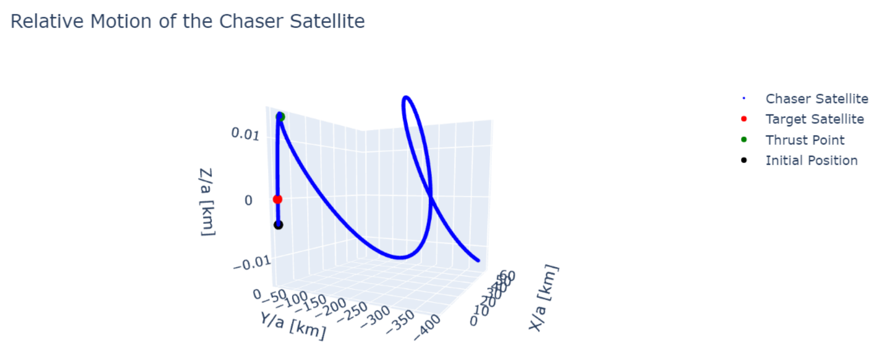


Figure 5: Simulation plot of impulsive manoeuvre (view 2).

3.4 Non-Impulsive Manoeuvre

The final simulation involved a non-impulsive manoeuvre, which applies continuous, low-thrust forces over an extended period.

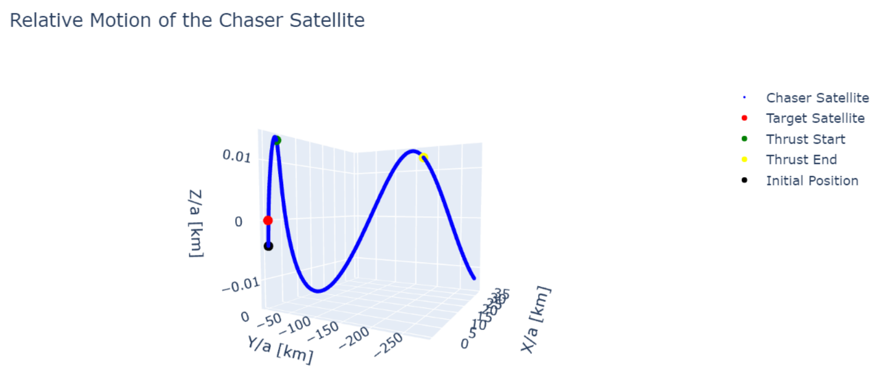


Figure 6: Simulation plot of non-impulsive manoeuvre (view 2).

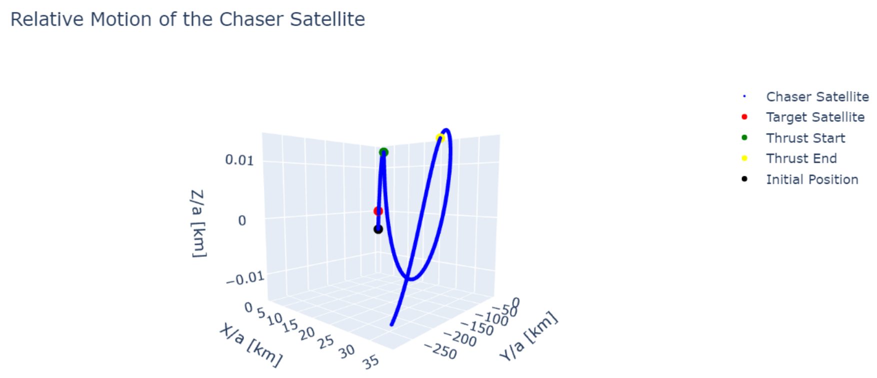


Figure 7: Simulation plot of non-impulsive manoeuvre (view 2).

4 Discussion

The comparison between impulsive and non-impulsive manoeuvres in spacecraft trajectory simulation reveals several key insights into the practical applications and limitations of each approach.

4.1 Impulsive Manoeuvres

Impulsive manoeuvres, as modelled in the simulations, provide a simplified yet effective means of understanding the immediate impact of high-thrust, short-duration burns on a spacecraft's trajectory. While not physically feasible, the instantaneous change in velocity approximates the effect of rapid burns well. This method is useful for missions requiring quick adjustments, such as collision avoidance or rapid orbital insertions.

However, the limitations of impulsive manoeuvres are also evident. The assumption of an instantaneous application of force does not account for the physical limitations of spacecraft propulsion systems. Real-world engines require finite time to ramp up and down, and during this period, the spacecraft continues to move along its trajectory. Furthermore, the simulations do not consider potential instability and misalignment issues that could arise from sudden thrust applications, which could lead to inaccuracies in the trajectory prediction if not properly accounted for.

4.2 Non-Impulsive Manoeuvres

Non-impulsive manoeuvres, on the other hand, provide a more realistic model by applying continuous, low-thrust over an extended period. This approach is particularly suitable for missions that require gradual adjustments, such as station-keeping, orbital transfers, or fine-tuning of satellite positions. The simulation results for non-impulsive manoeuvres show a smooth and gradual change in the spacecraft's velocity and position, reflecting the continuous application of thrust. This method allows for more precise control over the spacecraft's trajectory and is essential for long-duration missions where fuel efficiency and precise manoeuvring are critical.

One significant advantage of non-impulsive manoeuvres is their ability to account for the gradual depletion of fuel and the corresponding decrease in spacecraft mass, which directly affects thrust and acceleration. This dynamic aspect makes the non-impulsive model more adaptable and accurate for long-term mission planning.

4.3 Comparative Analysis

Direct comparison of the two simulation methods is possible since both simulations are set up to impart an equal force - one high thrust impulse and one across low thrust over 6000 seconds. The total displacement impacted by the impulse manoeuvre is seen to be higher than that imparted by the extended non-impulse - resulting in a different final position. This clearly shows a disparity between the two methods despite imparting the same amount of total force in the same direction.

5 Conclusion

Since this paper is intended to be an exploratory record of progress, the conclusions reached are somewhat limited. It is clear that there is a disparity between the trajectory simulation of impulse and non-impulsive manoeuvres. This is expected given the frozen nature of velocity vectors in impulse manoeuvres as opposed to their relation to time and non-impulsive manoeuvres. As discussed in the previously referenced orbital mechanics textbook [3], this lends impulsive manoeuvre simulation to high-thrust, short burns while reserving non-impulse simulation to low-thrust, extended burns.

To further develop this work, steps are being taken to implement the simulation of rotational motion in axisymmetric variable mass systems, as covered by the paper 'Geometry of motion and nutation stability of free axisymmetric variable mass systems' [2].

6 References

[1] Howard D. Curtis. "Orbital Mechanics for Engineering Students". In: Second. Butterworth-Heinemann, 2010. Chap. 6, pp. 285–348. DOI: 10.1016/B978-0-12-374778-5.00006-4. URL: https://doi.org/10.1016/B978-0-12-374778-5.00006-4.

[2] Angadh Nanjangud. “Geometry of motion and nutation stability of free axisymmetric variable mass systems”. In: *Nonlinear Dynamics* 94.3 (2018), pp. 2205–2218.

[3] Orbital Mechanics. *Nonimpulsive Orbital Maneuvers*. `https://orbital-mechanics.space/orbital-maneuvers/nonimpulsive-orbital-maneuvers.html`. Accessed: 2024-07-19.

[4] University of Texas at Austin. *Clohessy-Wiltshire Equations*. `https://www.ae.utexas.edu/courses/ase366k/cw_equations.pdf`. Accessed: 2024-07-19.

7 Appendix

7.1 Impulse Manoeuvre Python Script

```python
import numpy as np
import plotly.graph_objects as go

# Gravitational parameter (mu) and mean motion (n)
mu = 398600.4418  # km^3/s^2 (gravitational parameter of Earth)
sma = 7000  # km (semi-major axis of the target's orbit)
n = np.sqrt(mu / sma**3)  # mean motion of the target

# Initial conditions in the target's orbit reference frame
r0_chaser = np.array([10.0, 9.0, -30.0])  # Initial relative position of chaser, km
v0_chaser = np.array([3, 5, 0.1])  # Initial relative velocity of chaser, km/s
initial_mass = 5000  # Initial mass of the chaser, kg

# Thrust parameters
thrust_start_time = 2000  # Time delay before thrust starts in seconds
thrust_force = np.array([0, 600000000, 0])  # Thrust force in N (converted to
    km/s^2 later)
thrust_acceleration = thrust_force / initial_mass  # Thrust acceleration in km/s^2
thrust_acceleration = thrust_acceleration * 1e-3  # Convert to km/s^2
print("Thrust acceleration (km/s^2):", thrust_acceleration)

# Time parameters
t_final = 10000  # Total simulation time, seconds
dt = 1  # Time step, seconds

# Time array
t = np.arange(0, t_final, dt)

# Position and velocity arrays
r_chaser = np.zeros((len(t), 3))
v_chaser = np.zeros((len(t), 3))

# Set initial conditions
r_chaser[0] = r0_chaser
v_chaser[0] = v0_chaser

# Clohessy-Wiltshire equations for relative motion
def clohessy_wiltshire(r, v, t, n):
    x, y, z = r
    x_dot, y_dot, z_dot = v

    x_ddot = 3 * n**2 * x + 2 * n * y_dot
    y_ddot = -2 * n * x_dot
    z_ddot = -n**2 * z

    return np.array([x_dot, y_dot, z_dot]), np.array([x_ddot, y_ddot, z_ddot])
```

```python

# Simulation loop
for i in range(1, len(t)):
    # Get current state
    r = r_chaser[i-1]
    v = v_chaser[i-1]

    # Find accelerations using Clohessy-Wiltshire equations
    v_dot, a = clohessy_wiltshire(r, v, t[i], n)

    # Apply thrust if within thrust duration
    if thrust_start_time == t[i]:
        a += thrust_acceleration

    # Update position and velocity using Euler's method
    r_chaser[i] = r + v * dt
    v_chaser[i] = v + a * dt

# Plotting with Plotly for interactive visualization
r_chaser_scaled = np.divide(r_chaser, sma)

# Create a trace for the chaser
trace = go.Scatter3d(
    x=r_chaser_scaled[:, 0],
    y=r_chaser_scaled[:, 1],
    z=r_chaser_scaled[:, 2],
    mode='markers',
    name='Chaser Satellite',
    marker=dict(
        color='blue',
        size=2
    )
)

# Create marker for initial position
initial_trace = go.Scatter3d(
    x=[r_chaser_scaled[0, 0]],
    y=[r_chaser_scaled[0, 1]],
    z=[r_chaser_scaled[0, 2]],
    mode='markers',
    name='Initial Position',
    marker=dict(
        color='black',
        size=5
    )
)

# Create a trace for the target
target_trace = go.Scatter3d(
    x=[0],
    y=[0],
    z=[0],
    mode='markers',
    name='Target Satellite',
    marker=dict(
        color='red',
        size=5
    )
)
```

```python

# Create marker for start of thrust
thrust_start_trace = go.Scatter3d(
    x=[r_chaser_scaled[thrust_start_time, 0]],
    y=[r_chaser_scaled[thrust_start_time, 1]],
    z=[r_chaser_scaled[thrust_start_time, 2]],
    mode='markers',
    name='Thrust Point',
    marker=dict(
        color='green',
        size=5
    )
)

# Create the layout
layout = go.Layout(
    title='Relative Motion of the Chaser Satellite',
    scene=dict(
        xaxis=dict(title='X/a [km]'),
        yaxis=dict(title='Y/a [km]'),
        zaxis=dict(title='Z/a [km]')
    )
)

# Create the figure and add the traces
fig = go.Figure(data=[trace, target_trace, thrust_start_trace, initial_trace],
    layout=layout)

# Show the figure
fig.show()
```

7.2 Non-Impulse Manoeuvre Python Script

```python
import numpy as np
import plotly.graph_objects as go
import math

# Gravitational parameter (mu) and mean motion (n)
mu = 398600.4418  # km^3/s^2 (gravitational parameter of Earth)
sma = 7000  # km (semi-major axis of the target's orbit)
n = np.sqrt(mu / sma**3)  # mean motion of the target

# Initial conditions in the target's orbit reference frame
r0_chaser = np.array([10.0, 9.0, -30.0])  # Initial relative position of chaser, km
v0_chaser = np.array([3, 5, 0.1])  # Initial relative velocity of chaser, km/s

# Thrust parameters
thrust_start_time = 2000  # Time delay before thrust starts in seconds
thrust_duration = 6000    # Thrust duration in seconds
thrust_force = np.array([0, 100000, 0])  # Thrust force in N
specific_impulse = 3000  # Specific impulse of the propulsion system, seconds
fuel_mass_inital = 1000  # Initial mass of the fuel
spacecraft_mass = 4000 # Initial dry mass of the craft, kg
total_mass_initial = spacecraft_mass + fuel_mass_inital  # Initial total mass of
    the spacecraft

# Time parameters
t_final = 10000  # Total simulation time, seconds
```

```python
dt = 1  # Time step, seconds

# Time array
t = np.arange(0, t_final, dt)

# Position and velocity arrays
r_chaser = np.zeros((len(t), 3))
v_chaser = np.zeros((len(t), 3))

# Set initial conditions
r_chaser[0] = r0_chaser
v_chaser[0] = v0_chaser

# Clohessy-Wiltshire equations for relative motion
def clohessy_wiltshire(r, v, t, n):
    x, y, z = r
    x_dot, y_dot, z_dot = v

    x_ddot = 3 * n**2 * x + 2 * n * y_dot
    y_ddot = -2 * n * x_dot
    z_ddot = -n**2 * z

    return np.array([x_dot, y_dot, z_dot]), np.array([x_ddot, y_ddot, z_ddot])


# Initialize the flag variable before the loop
fuel_depleted_flag = False

#initialise current mass
current_mass = total_mass_initial

# Simulation loop
for i in range(1, len(t)):
    # Get current state
    r = r_chaser[i-1]
    v = v_chaser[i-1]

    # Find accelerations using Clohessy-Wiltshire equations
    v, a = clohessy_wiltshire(r, v, t[i], n)
    thrust_acceleration = np.array([0.0, 0.0, 0.0])
    # Calculate thrust acceleration using rocket equation
    if thrust_start_time <= t[i] <= (thrust_start_time + thrust_duration):

        if (current_mass - spacecraft_mass) > 0:
            exhaust_velocity = specific_impulse * 9.81  # Exhaust velocity in m/s
            thrust_acceleration = thrust_force / current_mass  # Thrust
                acceleration in m/s^2
            thrust_acceleration = thrust_acceleration * 1e-3  # Convert thrust
                acceleration to km/s^2
            print("Thrust acceleration (km/s^2):", thrust_acceleration)
            current_mass =
                current_mass*math.exp((-(np.linalg.norm(thrust_acceleration)*dt))/(exhaust_veloc
                 # Update mass of the spacecraft
        else:
            if not fuel_depleted_flag:
                fueldepleted = True
                print("fuel depleted at: ", t[i], "seconds")
                fuel_depleted_flag = True
            thrust_acceleration = np.array([0.0, 0.0, 0.0])  # No thrust

    # Total acceleration
    total_acceleration = a + thrust_acceleration

    # Update position and velocity
    r_chaser[i] = r + v * dt
    v_chaser[i] = v + total_acceleration * dt


print("fuel used: ", (fuel_mass_inital-(current_mass - spacecraft_mass)) , "kg,
    out of ", fuel_mass_inital, "kg")




# Plotting with Plotly for interactive visualization
r_chaser_scaled = r_chaser / sma

# Create traces for plotting
trace = go.Scatter3d(
    x=r_chaser_scaled[:, 0],
    y=r_chaser_scaled[:, 1],
    z=r_chaser_scaled[:, 2],
    mode='markers',
    name='Chaser Satellite',
    marker=dict(
        color='blue',
        size=2
    )
)

initial_trace = go.Scatter3d(
    x=[r_chaser_scaled[0, 0]],
    y=[r_chaser_scaled[0, 1]],
    z=[r_chaser_scaled[0, 2]],
    mode='markers',
    name='Initial Position',
    marker=dict(
        color='black',
        size=5
    )
)

target_trace = go.Scatter3d(
    x=[0],
    y=[0],
    z=[0],
    mode='markers',
    name='Target Satellite',
    marker=dict(
        color='red',
        size=5
    )
)

thrust_start_trace = go.Scatter3d(
    x=[r_chaser_scaled[thrust_start_time, 0]],
    y=[r_chaser_scaled[thrust_start_time, 1]],
    z=[r_chaser_scaled[thrust_start_time, 2]],
```

```python
    mode='markers',
    name='Thrust Start',
    marker=dict(
        color='green',
        size=5
    )
)

if (current_mass - spacecraft_mass) <= 0:
    thrust_end_trace = go.Scatter3d(
        x=[r_chaser_scaled[fueldepleted, 0]],
        y=[r_chaser_scaled[fueldepleted, 1]],
        z=[r_chaser_scaled[fueldepleted, 2]],
        mode='markers',
        name='Thrust End',
        marker=dict(
            color='yellow',
            size=5
        )
    )
else:
    thrust_end_trace = go.Scatter3d(
        x=[r_chaser_scaled[thrust_start_time+thrust_duration, 0]],
        y=[r_chaser_scaled[thrust_start_time+thrust_duration, 1]],
        z=[r_chaser_scaled[thrust_start_time+thrust_duration, 2]],
        mode='markers',
        name='Thrust End',
        marker=dict(
            color='yellow',
            size=5
        )
    )

# Create the layout
layout = go.Layout(
    title='Relative Motion of the Chaser Satellite',
    scene=dict(
        xaxis=dict(title='X/a [km]'),
        yaxis=dict(title='Y/a [km]'),
        zaxis=dict(title='Z/a [km]')
    )
)

# Create the figure and add the traces
fig = go.Figure(data=[trace, target_trace, thrust_start_trace, thrust_end_trace,
    initial_trace], layout=layout)

# Show the figure
fig.show()
```