Gradient-Based Orbital Transfer Optimisation with MCMC-Driven Random Walkers for Uncertainty Evaluation

Name: Tszon Tseng (21065076)

Department of Physics, Astronomy, Mathematics

University of Hertfordshire

Dated: 18 April 2025

Abstract

Orbital transfer optimisation is an important topic for future deep space missions and machine learning applications. We explore how thrust changes in different directions of a spacecraft would affect the Clohessy-Wiltshire (CW) linear model. Then, we successfully find an optimal solution: $v_{optimised} = 9376.19\,ms^{-1}$, $\theta_{optimised} = 35.51^{\circ}$ that would optimise the required thrust Δv and burn angle θ for an interceptor spacecraft in LEO (an altitude h of 2000 km) to intercept a target spacecraft in MEO (2000 km $< h <$ 35786 km) using the gradient descent algorithm, followed by the Markov Chain Monte Carlo (MCMC)-based random walkers for evaluating the final convergent values and uncertainties: $v_{optimised} = (9443.89 \pm 3.16)\ ms^{-1}, \theta_{optimised} = (34.21 \pm 0.06)^{\circ}$. The simulations are presented in insightful figures and animations.

Contents

I. INTRODUCTION

On 15 April 2024, NASA and ESA proposed a Mars Sample Return Mission to collect Martian soil, rocks and atmosphere samples to Earth in the 2030s (Madni et al. 2024). This involves an interplanetary docking between the Mars ascent vehicle and an orbiter. Simulating the relative spacecraft's motion accurately is vital for researchers to gain more insight into how the spacecraft behaves dynamically in their orbits for a more advanced docking process.

There are various models describing the relative motion problem in the research areas, the most common one is the **Clohessy-Wiltshire (CW) linear model** (Clohessy and Wiltshire 1960), a valid model for short-time periods where only the tidal, Coriolis, centrifugal and Euler forces are taken into account when deriving the equations of orbit. My project will follow the quantitative analysis performed by Carroll 2019, who derived the CW equations and explained the nature of the spacecraft's orbit. More realistic treatments, including the gravitational force due to the Earth's unevenly distributed density and the oblateness of the two poles, are ignored for simplicity.

The first focus of this project is to simulate how the spacecraft's orbits evolve as time elapses. After obtaining a set of general solutions by solving the differential equations in three spatial coordinates, we are interested in seeing how different the interceptor's orbit is compared to the target and what changes will be brought to the orbital shape if the interceptor continuously fires its thrusters in a chosen direction. We will investigate the direction in which the interceptor should fire for two chosen locations of the two spacecraft while saving as much fuel and transfer time as

possible.

Then, we implement one of the most crucial techniques used in machine learning and deep learning, which is called the **gradient descent algorithm** to optimise the transfer orbit. To quantify uncertainties and errors associated with the optimised trajectories, we delve into the **Markov Chain Monte Carlo (MCMC) method**, which uses Bayesian inference of statistics for estimating the uncertainty (Godsill 2001).

Throughout the project, I used Visual Studio Code as my coding software, which uses a user-friendly interface with various extensions, including LaTeX and Jupyter Notebook for programmers and helps keep the code and my report written in an orderly manner. After completing the project, I will upload my source code to the GitHub platform. Also, I am planning to implement version control and develop an entire package, allowing further discussion for my project with the public and any possible updates to the source code in the future if necessary.

II. PRINCIPLES OF ORBITAL MECHANICS

Imagine there is already one spacecraft in a stable orbit around Earth, which we call the *target*, our **chief** spacecraft. And now we have another spacecraft launched into space, which we call the *interceptor*, our **deputy** spacecraft. Let's first define two frames of reference: one is the inertial static Earth frame S^*, and the other is the target's non-inertia rotating frame S.

To quantitatively analyse the motion of orbiting bodies, we can derive general solutions to the linearised differential equations in a 3D Cartesian coordinate system, x, y, and z-axis that are still approximately accurate for small distances between the two spacecraft. The x-axis is parallel to the spacecraft's orbital velocity, the y-axis points radially away from the Earth's origin, and the z-axis points perpendicularly to the orbital plane, parallel to the angular velocity vector (Ω or ω) of the spacecraft (Fig. 1).

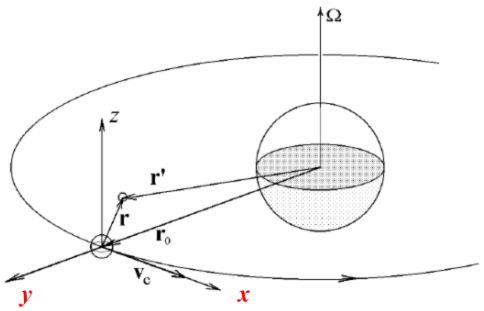

Figure 1: The orbital frames of the target and interceptor spacecraft (Butikov 2001). Note that the original x, y-axis are **swapped** to fit in our set-up. The orbital velocity v_c lies on the x-axis. r_0 denotes the distance between the target and Earth's centre, $r^{'}$ denotes the distance between the interceptor and Earth's centre, and r is the position of the interceptor relative to the target.

To determine the equations of orbits of the two spacecraft, we need to understand what forces are contributing to the total net force acting on the two objects. For simplicity, we only account for the tidal, Coriolis, centrifugal and Euler forces. Tidal force arises from the difference in gravitational pull between two locations, while the Coriolis, centrifugal, and Euler forces appear because the two spacecraft in a reference frame rotate relatively to Earth.

Following the approach from Carroll 2019, the location of the target and interceptor as seen from Earth's frame S^* are described by Eqs. (1) and (2), where R_0 is the total distance measured from the Earth's origin to the target's altitude. Eq. (3) gives the location of the interceptor as seen from frame S.

$$\vec{R} = X^*\hat{i}^* + Y^*\hat{j}^* = R_0\hat{j} \tag{1}$$

$$\vec{r}^* = x^*\hat{i}^* + y^*\hat{j}^* + z^*\hat{k}^* \tag{2}$$

$$\vec{r} = x\hat{i} + y\hat{j} + z\hat{k} \tag{3}$$

Note that the starred variables are the quantities observed from Earth's frame S^* and the unstarred variables are those seen from the target's rotating frame S.

The equations relating two different frames of reference S and S^* in terms of positions, velocities and accelerations as functions of time are defined in Eqs. (4), (5) and (6).

$$\vec{r} = \vec{r}^* - \vec{R}^* \tag{4}$$

$$\frac{d\vec{r}}{dt} = \frac{d^*\vec{r}^*}{dt} - \vec{\omega}\times\vec{r} - \frac{d^*\vec{R}^*}{dt} \tag{5}$$

$$\frac{d^2\vec{r}^*}{dt^2} = \underbrace{\frac{d^{*2}\vec{r}^*}{dt^2} - \frac{d^{*2}\vec{R}^*}{dt^2}}_{\text{Tidal}} \underbrace{- \vec{\omega}\times(\vec{\omega}\times\vec{r})}_{\text{Centrifugal}} \underbrace{- 2\vec{\omega}\times\frac{d\vec{r}}{dt}}_{\text{Coriolis}} \underbrace{- \frac{d^*\vec{\omega}}{dt}\times\vec{r}}_{Euler} \tag{6}$$

Applying Newton's 2nd law of motion to the interceptor as seen from frame S^*, we get Eq. (7) where m_i is the mass of the interceptor. Examples of the non-gravity force term include drag and thrust. Kepler's 3rd law tells us Eq. (8), where a^* is the semi-major axis seen from Earth's frame. The angular velocity vector by definition is Eq. (9), where G is the gravitational constant and M_E is the mass of Earth.

$$m_i\frac{d^{*2}\vec{r}^*}{dt^2} = -\frac{GMm_i}{r^{*3}}\vec{r^*} + \vec{F}_{non-gravity} \tag{7}$$

$$P^2 = \frac{4\pi^2 a^{*3}}{GM_E} = (\frac{2\pi}{\omega_0})^2 \tag{8}$$

$$\vec{\omega} = -\omega_0\hat{k}, where\ \omega_0 = \sqrt{\frac{GM_E}{R_0^3}} \tag{9}$$

$\vec{\omega}$ is just a constant, the last term in Eq. (6) becomes zero and makes the orbit circular. The centrifugal force per unit mass is Eq. (10). The gravitational force acting on the target provides the required centripetal acceleration as seen from the Earth's frame, which gives Eq. (11).

$$-\vec{\omega}\times(\vec{\omega}\times\vec{r}) = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ 0 & 0 & -\omega_0 \\ -\omega_0 y & \omega_0 x & 0 \end{vmatrix} = \omega_0^2\vec{r} - \omega_0^2\hat{k} \tag{10}$$

$$\frac{d^{*2}\vec{R}^*}{dt^2} = -\frac{GM_E}{R_0^3}\vec{R}^* = -\omega_0^2\vec{R}^* \tag{11}$$

Use Eqs. (4) and (7), we can rewrite Eq. (6) into Eq. (12).

$$\frac{d^2\vec{r}^*}{dt^2} = (1 - \frac{R_0^3}{r^{*3}})\omega_0^2\vec{r}^* - \omega_0^2\hat{k} - 2\vec{\omega}\times\frac{d\vec{r}}{dt} + \frac{\vec{F}_{non-gravity}}{m_i} \tag{12}$$

Assuming the two spacecraft are sufficiently close to each other, expand the first (tidal) term in Eq. (12) in binomial series and ignore any terms higher than second order to linearise the acceleration equation:

$$\begin{aligned}(1-\frac{R_0^3}{r^{*3}})\omega_0^2\vec{r}^* &\approx \{1-\frac{1}{[(\frac{x}{R_0})^2+(1+\frac{y}{R_0})^2+(\frac{z}{R_0})^2]^{\frac{3}{2}}}\}\\ &\quad\times\omega_0^2(\vec{r}+R_0\hat{j})\\ &\approx 3\omega_0^2 y\hat{j}\end{aligned} \tag{13}$$

The Coriolis force per unit mass becomes:

$$-2\vec{\omega}\times\frac{d\vec{r}}{dt}=\begin{vmatrix}\hat{i}&\hat{j}&\hat{k}\\0&0&-\omega_0\\\dot{x}&\dot{y}&\dot{z}\end{vmatrix}=-2\omega_0\dot{y}\hat{i}+2\omega_0\dot{x}\hat{j} \tag{14}$$

Eventually, Eq. (12) becomes:

$$\frac{d^2\vec{r}^*}{dt^2}=(-2\omega_0\dot{y})\hat{i}+(3\omega_0^2y+2\omega_0\dot{x})\hat{j}+(-\omega_0^2z)\hat{k}+\frac{\vec{F}_{non-gravity}}{m_i} \tag{15}$$

We now have all the forces in the unit vector form: Eq. (15). Re-express the radial acceleration in terms of x, y, and z components, and we get Eqs. (16) – (18). These are called the **Hill equations** (Hill 1878). Different components of the forces on the right-hand side of the equations will be set to zero for now, they will become useful as we discuss in Section III. where the interceptor can fire its thrusters in different directions to change the shape of its orbit for docking.

$$\ddot{x}=-2\omega_0\dot{y}+\frac{F_x}{m_i} \tag{16}$$

$$\ddot{y}=3\omega_0^2y+2\omega_0\dot{x}+\frac{F_y}{m_i} \tag{17}$$

$$\ddot{z}=-\omega_0^2z+\frac{F_z}{m_i} \tag{18}$$

We can see that the differential equation in the z-direction is independent of motions in the x and y-axis. Using the typical boundary condition (i.e. $z=z_0$ and $\dot{z}=\dot{z_0}$ at $t=0$), the solution to Eq. (18) can be obtained immediately:

$$z(t)=z_0\cos(\omega_0t)+\frac{\dot{z_0}}{\omega_0}\sin(\omega_0t) \tag{19}$$

To solve Eqs. (16) and (17), we can integrate either of the equations with respect to time first and insert it back into the other equation. Integrate Eq. (16) and substitute back into Eq. (17), and again using the boundary conditions ($x=x_0$, $\dot{x}=\dot{x_0}$, $y=y_0$ and $\dot{y}=\dot{y_0}$ at $t=0$), we get:

$$\begin{aligned}y(t)=&-(3y_0+\frac{2\dot{x_0}}{\omega_0})\cos(\omega_0t)+\frac{\dot{y_0}}{\omega_0}\sin(\omega_0t)\\&+(4y_0+\frac{2\dot{x_0}}{\omega_0})\end{aligned} \tag{20}$$

Differentiate Eq. (20) and put it back into Eq. (16), apply the boundary conditions again, it leads to:

$$\begin{aligned}x(t)=&(6y_0+4\frac{\dot{x_0}}{\omega_0})\sin(\omega_0t)+(2\frac{\dot{y_0}}{\omega_0}\cos(\omega_0t))\\&+(x_0-2\frac{\dot{y_0}}{\omega_0})-\underbrace{(3\dot{x_0}+6\omega_0y_0)}_{\text{Drift term}}t\end{aligned} \tag{21}$$

The general solutions we just derived describe the motion of the interceptor from the target's frame S. Altogether with their time derivatives, they are called the CW equations. See the full derivations in Appendix A2.

The interceptor follows an approximately circular orbit around the Earth. It will become a spiralling ellipse falling towards the Earth if the offset between the two spacecraft is large since the small distance assumption is no longer valid for the CW linear model.

From Eq. (21), there is one time-evolving term attached to it, which is the drift velocity v_{drift} that the interceptor has parallel to the x-axis as it travels along its elliptical orbit. The constant terms in Eq. (20) can be rewritten as the y-coordinate of the ellipse's centre y_c, and it leads to Eq. (22). For every orbital period, the elliptical centre of the orbit displaces by the amount shown in Eq. (23). The interceptor's orbit drifts parallel to the x-axis, it only happens when y_c doesn't equal zero, that is when its semi-major axis a^* is not the same as R_0.

$$v_{drift}=-(3\dot{x_0}+6\omega_0y_0)=-\frac{3}{2}\omega_0y_c \tag{22}$$

$$\Delta x_c=v_{drift}T=v_{drift}(\frac{2\pi}{\omega_0})=-3\pi y_c \tag{23}$$

As seen in Fig. 2, these drifting ellipses are called the **Whifferdill turns** as seen from the target's frame, which can be attributed to the consequences of Kepler's 3rd law. We know that $a^3\propto T^2$, so when $a^*>R_0$ or $a^*<R_0$, the interceptor's period will be longer or shorter than the target's. Near the apogee, the interceptor is at its farthest distance away from the ellipse's focus, so it moves momentarily slower than the target by the conservation of total mechanical energy, which explains why the interceptor appears to be lagging and moves along the negative x-axis. The converse would happen near the perigee.

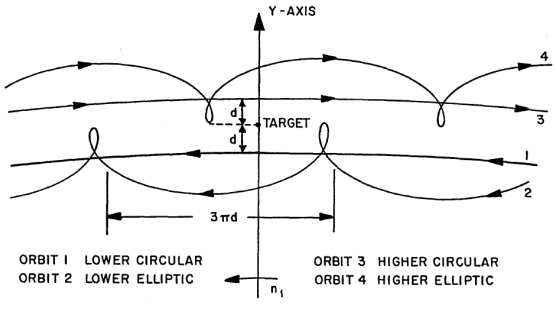

Figure 2: The Whifferdill turns as seen from the target's frame S (Aldrin 1963). d is the distance from the target to the axis of symmetry of the Whifferdill loops.

The physics behind the CW equations can be understood this way. The $-2\omega_0\dot{y}$ term in Eq. 16 is a radial Coriolis term. It is not in the same tangential direction as the acceleration term $\ddot{x}$ and therefore induces a transverse acceleration. Similarly, the $2\omega_0\dot{x}$ term in Eq. 17 is also a Coriolis term. The spring-like restoring term $3\omega_0^2y$ appears since the

gravitational and centrifugal forces no longer balance when $y > 0$. Lastly, Eq. 18 is the classic simple harmonic equation, which induces out-of-plane periodic oscillation with the orbital frequency ω_0.

III. PARAMETERISATION OF ORBITS

To visualise a realistic orbit of the interceptor, we need to carefully select the initial conditions when substituting into the general solutions since the linear CW equations only hold when the offset between the target and the interceptor is sufficiently small. As seen in Fig. 3, we can simulate the same effect of drifting ellipse as discussed before where the interceptor drifts parallel to the x-axis along the line $y = y_c = 4y_0 + 2\frac{\dot{x_0}}{\omega_0}$ from the target's perspective, which looks like an elongated spring extending itself in space in the x-y plane.

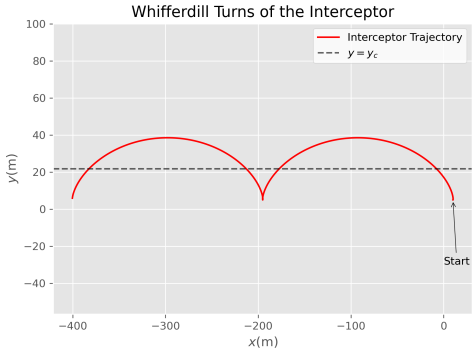

Figure 3: Simulation of the Whifferdill turns as seen from the target's rest frame S. We use the initial conditions $x_0 = 10\,m, \dot{x_0} = 0.001\,ms^{-1}, y_0 = 5\,m, z_0 = 5\,m, \dot{y_0} = \dot{z_0} = 0$.

Eq. set (24) describes the motion of the target spacecraft in the Earth's frame S^*. There is always an inclination angle ϕ of the target and interceptor with respect to the Earth's equator for a generalised case since the chief and deputy spacecraft do not necessarily stay on the equator. To visualise the orbital motion of the interceptor in the Earth's static frame, we need to compute a transformation between two Cartesian coordinate systems. The x_{rel}, y_{rel} and z_{rel} positions of the interceptor we obtained from the CW equations are the relative coordinates defined in the target's Hill frame S, we have to multiply these coordinates by some trigonometric rotation factors of ($\pm\ \omega_{target}t$) in a matrix form before adding them respectively to the target's inertial coordinates in each direction (Curtis 2019).

$$
\begin{aligned}
x_{S^*_{target}}(t) &= r_{target}\cos(\omega_{target}t) \\
y_{S^*_{target}}(t) &= r_{target}\sin(\omega_{target}t) * \cos(\phi_{target}) \\
z_{S^*_{target}}(t) &= r_{target}\sin(\omega_{target}t) * \sin(\phi_{target})
\end{aligned} \tag{24}
$$

In our setup, $\hat{i_{S,i}}, \hat{j_{S,i}}, \hat{k_{S,i}}$ in Eq. set (25) are the position vectors of the interceptor in the target's frame. $x_{rel}, y_{rel}, z_{rel}$ in Eq. (26) are coordinates of the interceptor in the target's frame, whereas $\hat{i}, \hat{j}, \hat{k}$ are the position vectors of the interceptor defined in the Earth's frame. The z-axis is commonly shared between the inertial Earth's frame and the target's rotating frame, so we can apply Eq. (25) to relate the unit vectors between the Earth's static frame and the target's rotating frame.

$$
\begin{pmatrix} \hat{i}_{S,i} \\ \hat{j}_{S,i} \\ \hat{k}_{S,i} \end{pmatrix} = \begin{bmatrix} \cos(\omega_{target}t) & \sin(\omega_{target}t) & 0 \\ -\sin(\omega_{target}t) & \cos(\omega_{target}t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{pmatrix} \tag{25}
$$

To make $\hat{i}, \hat{j}, \hat{k}$ the subject, we need to compute the inverse of the rotation matrix and multiply both sides of Eq. (25), which gives:

$$
\begin{pmatrix} \hat{i} \\ \hat{j} \\ \hat{k} \end{pmatrix} = \begin{bmatrix} \cos(\omega_{target}t) & -\sin(\omega_{target}t) & 0 \\ \sin(\omega_{target}t) & \cos(\omega_{target}t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} \hat{i}_{S,i} \\ \hat{j}_{S,i} \\ \hat{k}_{S,i} \end{pmatrix} \tag{26}
$$

Eventually, we get the position of the interceptor in the Earth's frame:

$$
\begin{aligned}
x_{S^*_{interceptor}}(t) &= x_{target}(t) \\
&\quad + [x_{rel}(t)\cos(\omega_{target}t) - y_{rel}(t)\sin(\omega_{target}t)] \\
y_{S^*_{interceptor}}(t) &= y_{target}(t) \\
&\quad + [x_{rel}(t)\cos(\omega_{target}t) + y_{rel}(t)\sin(\omega_{target}t)] \\
&\quad * \cos(\phi_{interceptor}) \\
z_{S^*_{interceptor}}(t) &= z_{target}(t) + z_{rel}(t) * \sin(\phi_{interceptor})
\end{aligned} \tag{27}
$$

Assume the target has an altitude = 400 km and the interceptor's semi-major axis is the same as the radius of the target's orbit, then $R_0 = a_0 = 6.771 \times 10^6\ m$.

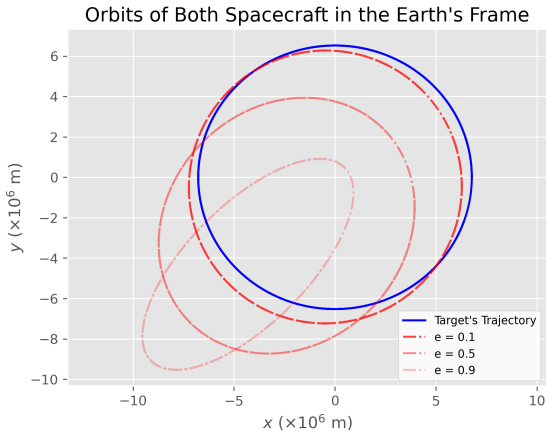

Figure 4: Orbital motion of the target and the interceptor spacecraft with different values of eccentricity e for $\phi_{interceptor} = 0$ when viewed from the Earth's frame S^*. These ellipses are plotted by solving Kepler's equation (Meeus 1991).

The spiral in Fig. (5), which grows radially outwards, describes the offset in x and y-coordinates between the two spacecraft. It heavily depends on the initial components of velocity when substituting the boundary conditions in the differential equations, since even a subtle offset in relative velocity would accumulate enormously as time progresses.

Now, we can parameterise the equations of motion by trying different initial conditions, i.e. the initial position, the inclination of the interceptor spacecraft and the direction where its thruster fires. We have three available options for the thrusting direction: either parallel to the target's orbital velocity (tangentially, along the x-axis); towards or away from the Earth (radially, along the y-axis); or perpendicular to the orbital plane of the target (out-of-plane, along the z-axis).

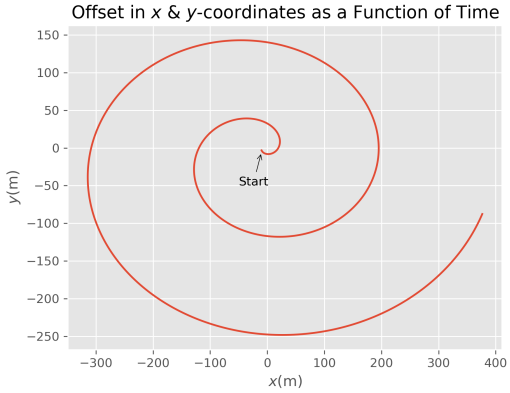

Figure 5: Offset between the target and the interceptor spacecraft. Their deviation in positions grows spirally as time evolves.

We are interested in seeing what effects these propulsions will bring to the interceptor's orbit. These will offer vital insights into optimising the docking process by firing the interceptor's engine in the most proper orientation while consuming as little fuel as possible.

To supply the interceptor with thrust, we replace the non-gravity per unit mass terms $\frac{F_x}{m_i}, \frac{F_y}{m_i}, \frac{F_z}{m_i}$ appearing on the right-hand side in Eqs. (16) – (18) with some arbitrary constants. They are chosen to be 0.05 since this magnitude is suitable enough not to cause dramatic changes to the orbits' settings for better visualisation.

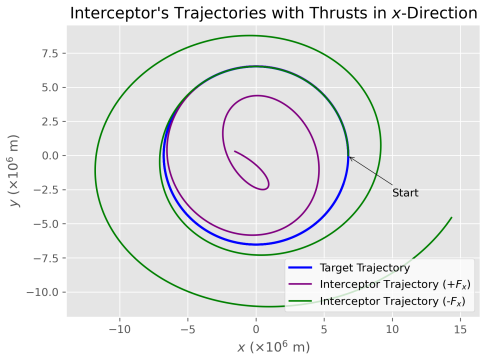

Figure 6: The interceptor fires a constant thrust in the positive or negative x direction.

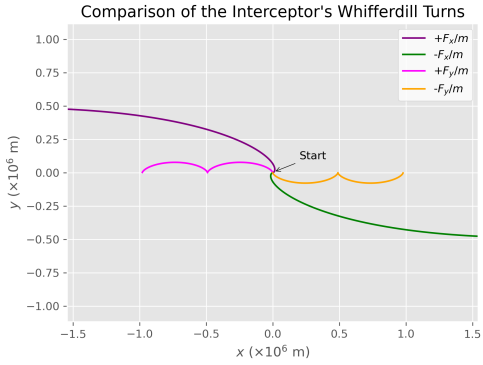

Figure 7: The comparison of different Whifferdill turn effects when the interceptor continuously fires in positive or negative, x or y-directions.

Assume the continuous thrusts provided by the spacecraft's engine are constant with time, for example, ion thrusters, which give a steady boost to the spacecraft for a long duration. If we fire the interceptor's engine in the positive x-direction, the spacecraft (the purple line) spirals inwards at a much faster rate (Fig. 6); its Whifferdill loop pattern as seen from the target's frame disappears and becomes a loose string extending away from the origin in the negative x-direction (Fig. 7).

The Whifferdill loop pattern seen when firing in the negative x-direction (i.e. the green line) becomes a loose string again but extends itself in the positive x-direction.

If we fire the interceptor's engine in either the positive or negative y-direction, the interceptor's orbit as viewed from the Earth (Fig. 8) does not look too different from our previous setup where no thrusts are applied to the interceptor; and its Whifferdill loops (Fig. 7, the magenta and orange lines) appear to be enlarged in the negative x-direction and rotated 180 degrees along the x-axis compared to the no-thrust scenario.

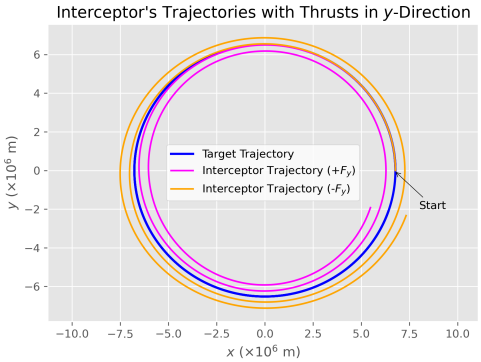

Figure 8: The interceptor fires a constant thrust in the positive or negative y direction.

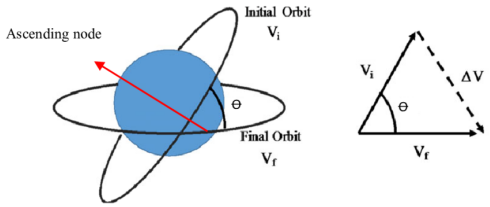

Figure 9: Schematics of changing the orbital inclination (Nongo, N. Ikpaya, and I. Ikpaya 2021). v_i and v_f are the circular speeds of the initial and final orbits, respectively. θ is the change in inclination involved.

IV. TRANSFER OPTIMISATION

As discussed in Section III., by changing the direction of firing, we realised that thrusting along the tangential direction allows for an orbit transfer on a larger scale, while propulsion made along the radial direction later helps achieve manoeuvres in the docking process of the spacecraft.

To achieve a certain change in the interceptor's inclination angle θ, firing perpendicularly to the orbital plane at apogee is the most energy-efficient approach since the extra burn required Δv would be minimal when the spacecraft is moving the slowest at the speed v_i (Fig. 9). These insights would be invaluable for optimising out-of-plane transfer in the future.

A. Hohmann & Bi-Elliptic Transfers

Over the centuries, researchers have proposed different approaches to achieve orbit transfer between two spacecraft based on their energy efficiencies and the speeds of the transfer process. When deciding the most suitable orbit transfer for docking, we need to take coplanarity, phase synchronisation and relative velocity alignment into consideration.

There are two main methods for in-plane orbit transfer, Hohmann and Bi-elliptic Hohmann. The **Hohmann transfer**, a two-impulse manoeuvre (Fig. 10) which is considered the most energy-efficient way since the Δv burns required are minimal to achieve transfer between two circular or elliptical orbits for relatively small differences in altitude in the same orbital plane (Walter 1925). The spacecraft first fires tangentially along the direction of flight at point A, then the spacecraft fires again in the forward direction at the apogee (point B) after flying half of the elliptical orbit to make its orbit circular.

However, it is time-consuming and no longer energy-efficient when used for transferring objects that have a large difference in inclination between orbital planes and a great change in altitude.

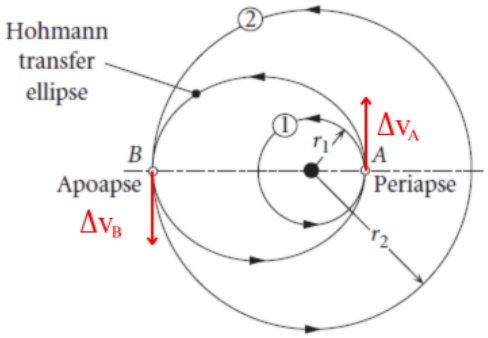

Figure 10: Schematics of Hohmann transfer. The original diagram is added with arrows to indicate the Δv burns (Curtis 2019).

Building on the Hohmann transfer, the **Bi-elliptic Hohmann transfer** (Fig. 11) is a three-impulse manoeuver with the spacecraft initially setting off from a lower orbit and then flying along two intermediate coaxial semi-ellipses to reach a much higher circular orbit. It first fires at point A as before, but with a greater magnitude of thrust to reach a very high apogee (point B) such that the Δv_B burn is minimised, Then, the spacecraft needs to fire opposite to its direction of travel at point C to circularise its orbit.

To achieve the optimal energy efficiency when choosing the best strategy between these two transfer options, we have to consider the ratio of the outer radius r_c to the inner radius r_a (Fig. 12). The Hohmann transfer is more energy efficient for a ratio less than ~ 11.94, while the bi-elliptic transfer is more energy efficient for a ratio greater than ~ 15. Within the range of these two ratios, it is energetically more favourable for a greater apoapsis r_b to adopt a bi-elliptic transfer, and for a smaller value of r_b to adopt the standard Hohmann transfer.

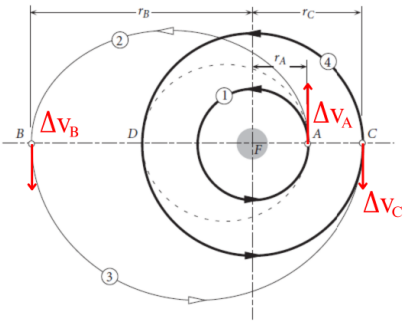

Figure 11: Schematics of Bi-elliptic Hohmann transfer. The original diagram is added with arrows to indicate the Δv burns (Curtis 2019).

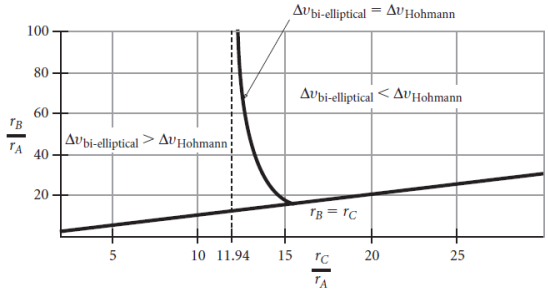

Figure 12: Comparison of the energy efficiency between the Hohmann and Bi-elliptic transfers (Curtis 2019).

B. Transfer Optimisation with Gradient Descent

To optimise the trajectories required for the docking process, we implement a machine learning algorithm called the **gradient descent** technique, an unconstrained first-order optimisation method used to locate the minimum of a loss (cost) function by choosing an appropriate step size that the machine should take, which is also called the learning rate (Deisenroth, Faisal, and Ong 2020). A suitable choice of learning rate would not be too low or too high. As seen in Fig. (13) , it takes an infinitely long time to update repeatedly for a very small learning rate. When searching with a learning rate that is too large, it skips the minimum and even leads to a divergence, ending up with an oscillation back and forth forever in the valley.

There is no actual formula for finding an optimal learning rate for any problem, we have to work that out by testing with some initial values. This process is called hyperparameter tuning, which aims to find a set of parameters for a specific learning algorithm that would optimise the model performance.

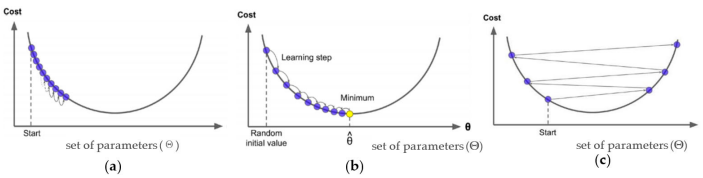

Figure 13: Diagram showing how to choose an appropriate learning rate (Cusido et al. 2022).

Back to our orbital transfer problem, Table 1 below shows the initial conditions of the interceptor and the target spacecraft in our setup. The interceptor is in a circular

orbit around the Earth, while the target spacecraft revolves around the Earth in an ellipse, and they both share a common orbital plane.

	Interceptor	Target
Distance r from Earth's centre (km)	6371 + 2000	—
Semi-major axis a (km)	—	33185.5
Eccentricity e	0	0.55
x_0 (km)	6371 + 2000	$r_{apoapsis}$ = 51437.53
v_{y0} (km s^{-1})	6.898	1.867

Table 1: Initial conditions of the interceptor and target spacecraft. The interceptor sits in the boundary (an altitude of 2000 km) of low-earth orbit (LEO), and the target sits within the mid-earth orbit (MEO) with an altitude $<$ 35780 km.

The dashed black and green lines in Fig. 14 show two possible Hohmann transfers, one firing forward to the target's perigee and the other firing backward to the target's apogee. However, Hohmann transfer optimises only the fuel consumption, not the transfer time. Therefore, we are interested in finding a pair of values for the initial velocity and burn angle that would optimise both Δv burns (the fuel) and transfer time.

To determine the optimal transfer time and the best aiming angle for which the interceptor should fire, we aim to optimise the cost function:

$$J(v, \theta) = \Delta v_1 + \Delta v_2 + \lambda t_{transfer} \tag{28}$$

λ is the penalty factor, or we call the **Lagrange multiplier** which weights how strongly we penalise longer time-of-flight $t_{transfer}$ while allowing the Δv burns and $t_{transfer}$ to change in different firing angles, where Δv_1 is the velocity change to leave the initial circular orbit, and Δv_2 is the burn to finalise the interceptor's orbit upon arrival. Or equivalently, we aim to find:

$$(v^*, \theta^*) = \arg\min_{v,\theta} J(v, \theta) \tag{29}$$

The travel time to intersect the target's orbit depends on both the Δv and the burn angle of the interceptor. We need to compute partial derivatives of the cost function J with respect to v and θ numerically to optimise the cost function. To evaluate the gradients, we use the **finite-difference method** to calculate the central difference of two points which are infinitesimally near to each other by a small step $\Delta\theta$ or Δv. For each parameter $p \in \{v, \theta\}$:

$$\frac{\partial J}{\partial v} \approx \frac{J(v + \Delta v) - J(v - \Delta v)}{2\Delta v} \tag{30}$$

$$\frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \Delta\theta) - J(\theta - \Delta\theta)}{2\Delta\theta} \tag{31}$$

Then, our gradient descent algorithm updates v and θ in the direction that minimises J. Denote η_v and η_θ as the learning rates for v and θ respectively:

$$v_{n+1} = v_n - \eta_v \frac{\partial J}{\partial v}(v_n, \theta_n) \tag{32}$$

$$\theta_{n+1} = \theta_n - \eta_\theta \frac{\partial J}{\partial \theta}(v_n, \theta_n) \tag{33}$$

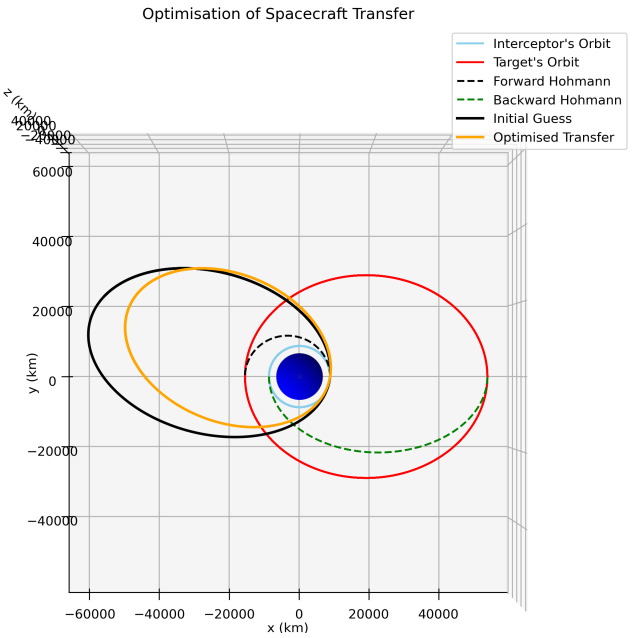

Figure 14: Comparison of possible transfers.

First, we make the initial guess of what the initial velocity and burn angle would be. The initial velocity of the interceptor should be at least $>\sim$ 7809 ms^{-1} to take off from Earth. A good guess for the burn angle would be in the range from 0 to 45 degrees. Too large a guess for the theta value would not be wise since the y-component of the initial velocity is a function of $\cos\theta$, and the interceptor might not even reach the target's orbit as a large angle would effectively decrease the take-off velocity.

Now, we have to tune the hyperparameters of the gradient descent function carefully. Depending on how far away the initial guess is from the global minimum, we need to fine-tune the learning rates η (step sizes) for v and theta accordingly and adjust the number of iterations to achieve the desired convergence. Be aware that the scaling of η_v and η_θ is different since v can occupy a much vaster parameter space, but a sensible parameter grid for θ values is much narrower as trigonometric functions are sensitive to smaller values of angles. With high learning rates, we explore the parameter space aggressively so we do not need to update the parameter values too many times to reach the global minimum, but we may risk ignoring the fine details and oscillating between the 'valleys' during the descent process; with small learning rates, we search the parameter space more thoroughly but at the expense of computation time since we need more iteration steps to reach the global minimum.

High η_v and η_θ (large steps) can cause big jumps. Therefore, we introduce a splitting of each gradient step in our program into multiple mini updates per iteration, or we call the 'sub-steps' for a more stable search of parameter values through the grid.

Using the parameter values listed in Table 2, if we plot the cost function J against the number of iterations, it shows a curve with a smoothly decreasing gradient and attains a convergence at the end (Fig. 15).

Parameters	λ	Tolerance	η_v	η_θ
Values	1×10^{-3}	1×10^{-5}	0.2	2×10^{-7}

Δv	$\Delta\theta$	No. of iterations	Sub-steps
0.5	0.01	500	20

Table 2: Parameters of the implemented gradient descent model.

	Forward Hohmann	Backward Hohmann
x_0 (km)	6371 + 2000	- (6371 + 2000)
v_{y0} ($\mathrm{m\,s^{-1}}$)	$v_i = 6898.29$	$v_i = 6898.29$
θ (°)	0	0
Δv_1 ($\mathrm{ms^{-1}}$)	911.10	2148.93
$+\ \Delta v_2$ ($\mathrm{ms^{-1}}$)	4377.58	1472.35
$= \Delta v_{total}$	5288.68	3621.28
t_{transfer} (min)	104.35	429.01

	Initial guess	**Optimised**
x_0 (km)	6371 + 2000	6371 + 2000
v_{y0} ($\mathrm{m\,s^{-1}}$)	9074.98	8915.26
v ($\mathrm{ms^{-1}}$)	9300.00	**9376.19**
θ (°)	25.00	**35.51**
Δv_1 ($\mathrm{ms^{-1}}$)	2401.71	**2477.90**
$+\ \Delta v_2$ ($\mathrm{ms^{-1}}$)	1634.66	**129.52**
$= \Delta v_{total}$	4036.37	2607.52
t_{transfer} (min)	33.99	**33.42**

Table 3: Summary of different transfer results. Notice the significant improvements in the Δv_1, Δv_2 burns and $t_{transfer}$ in the optimised transfer. Note that $v_{y0} = v_i + (v - v_i)\cos\theta$, where v_i is the circular speed of the interceptor and v is the speed on the parameter grid in Fig. 16.

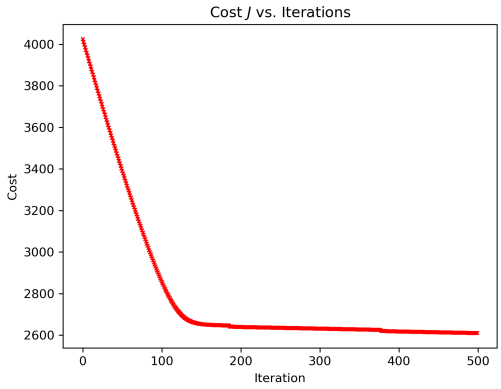

Figure 15: Changes in the cost function J over gradient descent steps. Notice that it is not a perfectly horizontal line after reaching the valley (around 130 steps) of the cost function. The descent process becomes much slower when the minimum is close.

We can inspect the local and global minimum by looking at the contour surface plot of the cost function J (Fig. 16). The white dashed line in Fig. 16 shows the ordinary Hohmann transfer, which sits inside one of the deepest basins in the contour plot. Note that any deep valley lying on the left-hand side of the white dashed line is not a feasible solution to our problem since the initial velocity of the interceptor with any values of theta would not be enough to reach the target's elliptical orbit.

From Table 3, we can see that the take-off burn for the optimised orbit is the greatest among all, but its arrival burn is the smallest compared to different transfers. Moreover, the transfer time drastically reduces from around 1 hour and 44 minutes for the Hohman transfer to just slightly over 30 minutes with the optimised orbit. Fig. 14 visualises both the guess (black line) and optimised (orange line) transfers.

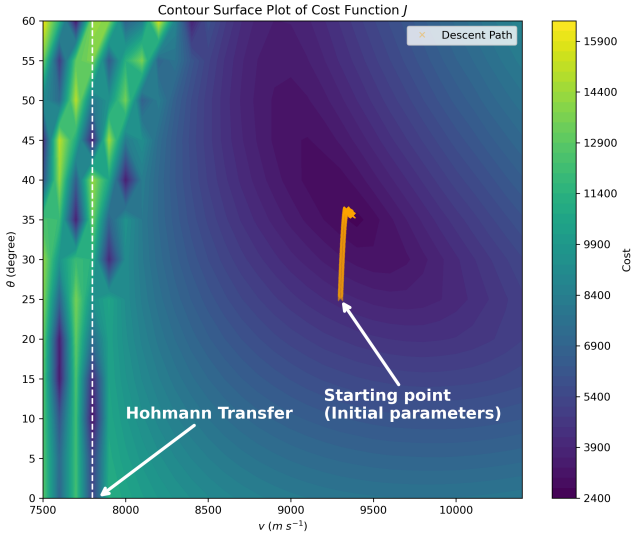

Figure 16: Gradient descent over the contour surface plot of the cost function J. The white dashed line on the left shows the cost of a classic Hohmann transfer.

C. Frequentist vs. Bayesian Approach

Classically, **hypothesis testing** in the frequentist approach depends on what we call the statistical significance α, which is a standard to determine whether or not we should reject our null hypothesis when compared to the p-value in each experiment (Pek and Van Zandt 2020). For every experimental design, two beliefs act against each other: one is called the null hypothesis H_0, an existing belief which is already proven to be true, and the other is called the alternative hypothesis H_A, a newly raised 'challenger' idea for the researchers. A p-value is the probability of obtaining a result, assuming H_0 is true. To reject H_0, we need to achieve a p-value smaller than α in a statistical test. α is usually set to be 0.05, but it can also take values of 0.005, 0.1 and 0.2 depending on how stringent the nature of the experiment is (Ioannidis 2019). One downside of the frequentist approach is that we rely greatly on fixed values, e.g. p-values, which means one can manipulate the results we want from our experiments by changing the p-value.

On the other hand, the Bayesian approach focuses on probabilities and often intertwines with inference, which is updating one's belief about something as new information becomes available (Rouder et al. 2017). We can include subject domain expertise to build our custom models. It is capable of naturally handling uncertainty since the parameter in the Bayesian approach is the distribution itself. Moreover, it remains statistically significant even with a small amount of data, which often coincides with the frequentist results.

Firstly, we need to set up a **prior** function $p(x_1, x_2, ...)$, a probability distribution (e.g. uniform, normal or beta distribution) given what we already know about some certain parameters $(x_1, x_2, ...)$ before seeing the actual data. Then, a **likelihood** function $\mathcal{L}$ describing how well the predictions made by the prior match the observed data can be

calculated. Using Bayes' theorem (Jeffreys 1998), we can compute the posterior distribution (Eq. 35) by multiplying the prior and likelihood together and dividing by the scaling factor (i.e. the normalisation constant) of the dataset.

$$P(Params|data) = \frac{P(Data|params) * P(Params)}{P(Data)} \quad (34)$$

$$Posterior = \frac{Prior * Likelihood}{Normalisation\,constant} \quad (35)$$

D. Uncertainties Evaluation with the MCMC Method

The **Monte Carlo method** uses random simulations to evaluate a quantity numerically. The more simulations, the more accurate the results will be. For example, to determine the irrational number π, we could inscribe a circle of a chosen radius with a square, and then we count how many random dots fall inside the circle. Assuming we use enough points to approximate the areas, using the area ratio between the circle and square, we can determine the value of π.

The **Markov Chain Monte Carlo (MCMC)** method uses Bayesian statistics as discussed in Sec. (C.) to sample randomly from an unknown **posterior** distribution, which is useful for estimating errors in modelling. The **Markov chain** is a set of random walks around some guesses of the parameters, which is used to model a sequence of states which transitions from one to another with a given probability. If we run the MCMC for sufficient iterations, the chains will eventually converge.

In our model, we implement an **Affine-Invariant Ensemble Sampler (AIES)** (i.e. invariant under linear transformations) using the *emcee* package in Python to sample the random walks (Huijser, Goodman, and Brewer 2015). We propose the new position for a current walker after the next walk $\vec{x_i'}$ by randomly selecting another walker $\vec{x_j}$:

$$\vec{x_i'} = \vec{x_j} + z(\vec{x_i} - \vec{x_j}) \quad (36)$$

z is a scaling factor selected randomly from the density distribution $g(z)$, where a is an adjustable constant:

$$g(z) \propto \begin{cases} \frac{1}{\sqrt{z}}, & \text{if } z \in \left[\frac{1}{a}, a\right], \\ 0, & \text{otherwise.} \end{cases} \quad (37)$$

Typically, a is chosen to be 2, and it applies well to general cases (Foreman-Mackey et al. 2013).

Then, the set of walkers will move if the acceptance probability $A > 1$, or otherwise, the walkers will stay at the same position.

$$A = \min\left[1, z^{n-1}\frac{p(\vec{x_i'})}{p(\vec{x_i})}\right] \quad (38)$$

n is the number of dimensions in the parameter space. The factor of z^{n-1} ensures the affine invariance. It makes the sampler robust to asymmetric (e.g. skewed, elongated) parameter distributions when compared to the standard **Metropolis-Hastings algorithm**, which requires extra tuning of the model.

In the context of our problem, we first define our joint prior function as $p(v, \theta)$, which takes a uniform distribution for both the burn velocity and burn angle. Assuming $p(v)$ and $p(\theta)$ are two independent uniform distributions:

$$p(v) = \begin{cases} \frac{1}{(15000 - 7809)}, & 7809 \le v \le 15000, \\ 0, & \text{otherwise.} \end{cases} \quad (39)$$

$$p(\theta) = \begin{cases} \frac{1}{(\frac{\pi}{2})}, & 0 \le \theta \le \frac{\pi}{2}, \\ 0, & \text{otherwise.} \end{cases} \quad (40)$$

$$Joint\,prior\,p(v, \theta) = p(v) \times p(\theta) \quad (41)$$

The optimised solution in the deepest valley might not be deep enough due to the number of layers we used in the contour surface plot (Fig. 16). The MCMC method helps us to precisely determine where the most optimised solution lies in the parameter grid by simulating a set of random walkers.

Therefore, we define our likelihood $\mathcal{L}$ (Eq. 42) as a function of (v, θ), which turns the cost function J into a probabilistic distribution, where σ is the scaling (Boltzmann) factor. We want to restrict random walkers to just exploring the low-cost region. The higher the cost, the less likely random walkers will find their way to high-cost regions.

$$Likelihood\,\mathcal{L}(v, \theta) \propto \exp -\frac{J(v, \theta)}{2\sigma^2} \quad (42)$$

More commonly, we transform the likelihood function into log space:

$$\log \mathcal{L}(v, \theta) = -\frac{J(v, \theta)}{2\sigma^2} + constant \quad (43)$$

Finally, the posterior function becomes:

$$Posterior = p(v, \theta \,|\, model) \propto p(v, \theta) \times \mathcal{L}(v, \theta) \quad (44)$$

Or in log-space:

$$\log p(v, \theta \,|\, model) = \log p(v, \theta) - \frac{J(v, \theta)}{2\sigma^2} + constant \quad (45)$$

This posterior is the function from which the random walkers will sample based on some initial guess of the parameters (v, θ).

From the trace plots below in Fig. 17, we can see that the walkers start from the optimised solution ($v \approx 9380$ ms^{-1}, $\theta \approx 35.5$ °), then parameters converge at around (9443 ms^{-1}, 34.2 °).

In the end, we use the mean to approximate the convergent parameters and the standard deviation to determine their uncertainty ranges. We find that the convergent values of the optimised velocity burn ($v_{optimised}$) and burn angle ($\theta_{optimised}$) are:

$$\begin{aligned} v_{optimised} &= (9443.89 \pm 3.16)\, ms^{-1}, \\ \theta_{optimised} &= (34.21 \pm 0.06)\,^\circ \end{aligned} \quad (46)$$

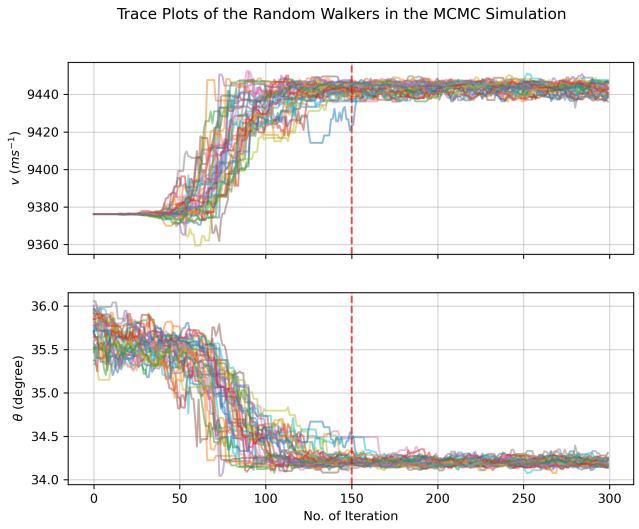

Figure 17: Simulation of random walkers to search for the final converged parameters (v, θ). We observe a sign of stable convergence at around 150 iterations (red dashed lines) for both parameters.

V. CONCLUSION

We explored the physics behind the CW model, which is a useful linear approximation for two spacecraft that are sufficiently close to each other. This can be applied to two spacecraft that require a final rendezvous to bring them together for velocity alignment and synchronisation in space missions. We have successfully optimised an in-plane orbital transfer using the gradient descent algorithm and evaluated the uncertainties for the Δv burn and the burn angle θ in the optimised solution using the MCMC-based random walkers. These promising results provide immense insights for time-critical missions that require spacecraft to arrive at the destination when the balance between fuel consumption and travel time is crucial for carrying out urgent satellite maintenance and supporting onboard crews.

Further work on this project in the future includes the phase synchronisation and manoeuvre between two spacecraft in the same orbit. We are also looking forward to finding an optimised solution to an out-of-plane transfer as a challenge.

ACKNOWLEDGEMENTS

I would express my immense gratitude to my supervisor, Dr. James Collett, for very insightful discussions and guidance throughout the project, and Professor Detlef Mueller for the valuable feedback and review to polish the entire work.

Bibliography

Aldrin, B. (1963). "Line-of-sight guidance techniques for manned orbital rendezvous". PhD thesis. Massachusetts Institute of Technology. DOI: 1721.1/12652.

Butikov, E.I. (2001). "Relative motion of orbiting bodies". In: *American Journal of Physics* 69.1, pp. 63–67. DOI: 10.1119/1.1313520.

Carroll, B.W. (2019). "The delicate dance of orbital rendezvous". In: *American Journal of Physics* 87.8, pp. 627–637. DOI: 10.1119/1.5115341.

Clohessy, W.H. and R.S. Wiltshire (1960). "Terminal guidance system for satellite rendezvous". In: *Journal of the aerospace sciences* 27.9, pp. 653–658. DOI: 10.2514/8.8704.

Curtis, H.D. (2019). *Orbital mechanics for engineering students*. Butterworth-Heinemann. DOI: 10.1016/C2011-0-69685-1.

Cusido, J. et al. (2022). "Predicting hospital admissions to reduce crowding in the emergency departments". In: *Applied Sciences* 12, p. 10764. DOI: 10.3390/app122110764.

Deisenroth, M.P., A.A. Faisal, and C.S. Ong (2020). *Mathematics for machine learning*. Cambridge University Press. DOI: 10.1017/9781108679930.

Foreman-Mackey, D. et al. (2013). "emcee: the MCMC hammer". In: *Publications of the Astronomical Society of the Pacific* 125.925, p. 308. DOI: 10.1086/670067.

Godsill, S.J. (2001). "On the relationship between Markov chain Monte Carlo methods for model uncertainty". In: *Journal of computational and graphical statistics* 10.2, pp. 230–248. DOI: 10.1198/10618600152627924.

Hill, G.W. (1878). "Researches in the lunar theory". In: *American journal of Mathematics* 1.1, pp. 5–26. DOI: 10.2307/2369430.

Huijser, D., J. Goodman, and B.J. Brewer (2015). "Properties of the affine invariant ensemble sampler in high dimensions". In: *arXiv preprint arXiv:1509.02230*, p. 2. DOI: 10.48550/arXiv.1509.02230.

Ioannidis, J. P.A. (2019). "Publishing research with P-values: Prescribe more stringent statistical significance or proscribe statistical significance?" In: *European Heart Journal* 40.31, pp. 2553–2554. DOI: 10.1093/eurheartj/ehz555.

Jeffreys, H. (1998). *The theory of probability*. OuP Oxford. DOI: 10.1038/109132a0.

Madni, A. et al. (2024). "Surface timeline management and analysis for the Mars Sample Return Mission". In: *2024 IEEE Aerospace Conference*. IEEE, pp. 1–12. DOI: 10.1109/AERO58975.2024.10521286.

Meeus, J.H. (1991). *Astronomical algorithms*. Willmann-Bell, Incorporated.

Nongo, S., N. Ikpaya, and I. Ikpaya (2021). "Prospects of siting a spaceport in Africa". In: *International Journal of Aerospace System Science and Engineering* 1, p. 1. DOI: 10.1504/IJASSE.2021.10036564.

Pek, J. and T. Van Zandt (2020). "Frequentist and Bayesian approaches to data analysis: Evaluation and estimation". In: *Psychology Learning & Teaching* 19.1, pp. 21–35. DOI: 10.1177/1475725719874542.

Rouder, J.N. et al. (2017). "Bayesian analysis of factorial designs". In: *Psychological Methods* 22.2, p. 304. DOI: 10.1037/met0000057.

Walter, H. (1925). *The Attainability of Celestial Bodies*. Original work in German. R. Oldenbourg.

APPENDIX

A1 Glossary

Acronyms

AIES

Affine-Invariant Ensemble Sampler. 9

CW

Clohessy-Wiltshire. 1, 3, 4, 10, 11

LEO

Low Earth Orbit. 1, 7

MCMC

Markov Chain Monte Carlo. 1, 2, 9, 10

MEO

Mid Earth Orbit. 1, 7

A2 Derivation

Derivation of the general solutions to the CW linear model:

$$\ddot{x} = -2\omega_0\dot{y} + \frac{F_x}{m_i} \tag{16}$$

$$\ddot{y} = 3\omega_0^2 y + 2\omega_0\dot{x} + \frac{F_y}{m_i} \tag{17}$$

Assume all the thrusts are equal to zero. Firstly, integrate Eq. (16) with respect to x:

$$\dot{x} = -2\omega_0(y + C_{y_0}) + \dot{x_0} = -2\omega_0 y + (2\omega_0 y_0 + \dot{x_0}) \tag{47}$$

Substitute this back into Eq. (17):

$$\begin{aligned}\ddot{y} &= 3\omega_0^2 y + 2\omega_0(-2\omega_0 y + 2\omega_0 y_0 + \dot{x_0}) \\ &= -\omega_0^2 y + 4\omega_0^2 y_0 + 2\omega_0\dot{x_0}\end{aligned} \tag{48}$$

The complementary function is therefore:

$$\ddot{y} + \omega_0^2 y = 0 \tag{49}$$

Obtain the homogeneous solution to this complementary function first:

$$\lambda = \frac{-0 \pm \sqrt{0^2 - 4(1)\omega_0^2}}{2(1)} = \pm i\omega_0$$

Solution to the complex conjugate roots:

$$y_c(t) = A\cos(\omega_0 t) + B\sin(\omega_0 t) \tag{50}$$

Let our particular integral y_p be a constant k, when $y = y_p$:

$$\omega_0^2 k = 2\omega_0\dot{x_0} + 4\omega_0^2 y_0$$

$$k = 2\frac{\dot{x_0}}{\omega_0} + 4y_0$$

Now, Eq. (49) becomes:

$$y(t) = A\cos(\omega_0 t) + B\sin(\omega_0 t) + (2\frac{\dot{x_0}}{\omega_0} + 4y_0) \tag{51}$$

Using the boundary conditions, $y = y_0$ and $\dot{y} = \dot{y_0}$ at $t = 0$:

$$\begin{aligned}y_0 &= A + 0 + (2\frac{\dot{x_0}}{\omega_0} + 4y_0) \\ A &= -(3y_0 + 2\frac{\dot{x_0}}{\omega_0}) \\ \dot{y_0} &= -0 + \omega_0 B \\ B &= \frac{\dot{y_0}}{\omega_0}\end{aligned}$$

Eventually for $y(t)$:

$$y(t) = -(3y_0 + 2\frac{\dot{x_0}}{\omega_0})\cos(\omega_0 t) + \frac{\dot{y_0}}{\omega_0}\sin(\omega_0 t) + (4y_0 + 2\frac{\dot{x_0}}{\omega_0}) \tag{52}$$

Substitute this back into Eq. (47):

$$\begin{aligned}\dot{x} &= -2\omega_0[-(3y_0 + 2\frac{\dot{x_0}}{\omega_0})\cos(\omega_0 t) + \frac{\dot{y_0}}{\omega_0}\sin(\omega_0 t) + (4y_0 + 2\frac{\dot{x_0}}{\omega_0})] \\ &\quad + (2\omega_0 y_0 + \dot{x_0}) \\ &= (6\omega_0 y_0 + 4\dot{x_0})\cos(\omega_0 t) - 2\dot{y_0}\sin(\omega_0 t) - (3\dot{x_0} + 6\omega_0 y_0)\end{aligned}$$

Integrate again, for $x(t)$ we finally get:

$$\begin{aligned}x(t) &= (6y_0 + 4\frac{\dot{x_0}}{\omega_0})\sin(\omega_0 t) + 2\frac{\dot{y_0}}{\omega_0}\cos(\omega_0 t) \\ &\quad + (x_0 - 2\frac{\dot{y_0}}{\omega_0}) - (3\dot{x_0} + 6\omega_0 y_0)t\end{aligned} \tag{21}$$

A3 Source Code

GitHub source code (final) & animation:
https://github.com/Tszon/BSc_Astrophysics_Projects/tree/main/GradientDescentOptimisation_MCMC

Perform baseline Hohmann transfers (Fig. 14):

```
# Import the required packages
import matplotlib.pyplot as plt
import numpy as np
from space_base import GravBody, Probe

# Define the physical constants in S.I. unit
G = 6.67e-11
earth = GravBody.earth()
mu = G * earth.mass

# Define the atmosphere model
def atmosphere(posvel):
    r = np.linalg.norm(posvel[0:3])
    h = r - earth.radius
    surface_dens = earth.surface_density
    scale_height = earth.scale_height
    density = surface_dens * np.exp(- h / scale_height)
    return density

# Define the gravitational field model
def gravplanetearth(posvel):
    r = np.linalg.norm(posvel[0:3])
    f = - mu / r ** 3
    return (f * posvel[0], f * posvel[1], f * posvel[2])

# Define the equation of motion for the spacecraft
def probeqnsearth(t, posvel):
    cd=1.
    A=0.01
    mass=1.

    density = atmosphere(posvel)
    current_gravity = gravplanetearth(posvel)
    v_norm = np.linalg.norm(np.array(posvel[3:6]))
    drag = - 0.5 * cd * A * density * v_norm * np.array(posvel[3:6]) / mass

    posveldot = posvel[3], posvel[4], posvel[5], \
        current_gravity[0] + drag[0], current_gravity[1] + drag[1], current_gravity[2] + drag[2]

    return posveldot

# Time settings
t_final = 3600 * 60
t_num = 5000

# Initial Conditions for the interceptor
r_i = earth.radius + 2000e3
v_i = np.sqrt(mu / r_i) # Initial velocity vy0
print(f"Interceptor v_init: {v_i:.3f} m/s.")

x0_i = r_i
y0_i = 0
z0_i = 0

vx0_i = 0
vy0_i = v_i
vz0_i = 0

# Plotting
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

interceptor = Probe(probeqnsearth, t_final, t_num, x0=x0_i, vx0=vx0_i,
            y0=y0_i, vy0=vy0_i, z0=z0_i, vz0=vz0_i, event=earth.radius)  # probe as an object
t, posvel_i = interceptor.odesolve()  # solve the differential equations
ax.plot(posvel_i[:, 0] / 1e3, posvel_i[:, 1] / 1e3, color='skyblue', label="Interceptor's Orbit")  #
    plot the interceptor

# Initial Conditions for the target
a_t = (earth.radius + 60000e3) / 2
e_t = 0.55
r_a_t = a_t * (1 + e_t)
r_p_t = a_t * (1 - e_t)

vy_t = np.sqrt(mu * (2 / r_a_t - 1 / a_t))
```

```
print(f"vy_target at r_a: {vy_t:.3f} m/s.\n")

x0_t = r_a_t
y0_t = 0
z0_t = 0

vx0_t = 0
vy0_t = vy_t
vz0_t = 0

target = Probe(probeqnsearth, t_final, t_num, x0=x0_t, vx0=vx0_t,
            y0=y0_t, vy0=vy0_t, z0=z0_t, vz0=vz0_t, event=earth.radius)
t_target, posvel_target = target.odesolve()
ax.plot(posvel_target[:, 0] / 1e3, posvel_target[:, 1] / 1e3, color='red', label="Target's Orbit")  # plot the target

# Parameters for the 1st Hohmann transfer
a_transfer_1 = (r_i + r_p_t) / 2
T_transfer_1 = np.sqrt(4 * (np.pi)**2  * a_transfer_1**3 / mu)

takeoff_burn_1 = np.sqrt(mu * (2 / r_i - 1 / a_transfer_1))
arrival_burn_1 = np.sqrt(mu * (2 / r_p_t - 1 / a_transfer_1))

print(f"Transfer 1: Takes {T_transfer_1 / 2 / 60:.3f} mins from the interceptor to the target.")
print(f"Take-off velocity: {takeoff_burn_1:.3f} m/s.")
print(f"Arrival burn: {arrival_burn_1:.3f} m/s.\n")

# Plot the forward Hohmann transfer
x0 = r_i
y0 = 0
z0 = 0

vx0 = 0
vy0 = takeoff_burn_1
vz0 = 0

# Plotting
transfer_1 = Probe(probeqnsearth, T_transfer_1 / 2, t_num, x0=x0, vx0=vx0,
            y0=y0, vy0=vy0, z0=z0, vz0=vz0, event=earth.radius)
t, posvel_transfer_1 = transfer_1.odesolve()
ax.plot(posvel_transfer_1[:, 0] / 1e3, posvel_transfer_1[:, 1] / 1e3, color='black', label='Forward Hohmann') # plot the transfer orbit

# Parameters for the backwards Hohmann transfer
a_transfer_2 = (r_i + r_a_t) / 2
T_transfer_2 = np.sqrt(4 * (np.pi)**2  * a_transfer_2**3 / mu)

takeoff_burn_2 = np.sqrt(mu * (2 / r_i - 1 / a_transfer_2))
arrival_burn_2 = np.sqrt(mu * (2 / r_a_t - 1 / a_transfer_2))

print(f"Transfer 2: Takes {T_transfer_2 / 2 / 60:.3f} mins from the interceptor to the target.")
print(f"Take-off velocity: {takeoff_burn_2:.3f} m/s.")
print(f"Arrival burn: {arrival_burn_2:.3f} m/s.")

# Plot the backwards Hohmann transfer
x0 = - r_i
y0 = 0
z0 = 0

vx0 = 0
vy0 = - takeoff_burn_2
vz0 = 0
transfer_2 = Probe(probeqnsearth, T_transfer_2 / 2, t_num, x0=x0, vx0=vx0,
            y0=y0, vy0=vy0, z0=z0, vz0=vz0, event=earth.radius)

# Plotting
t, posvel_transfer_2 = transfer_2.odesolve()
ax.plot(posvel_transfer_2[:, 0] / 1e3, posvel_transfer_2[:, 1] / 1e3, color='green', label='Backward Hohmann')

# Plotting Earth
uang = np.linspace(0, 2 * np.pi, 100)
vang = np.linspace(0, np.pi, 100)
x = earth.radius / 1e3 * np.outer(np.cos(uang), np.sin(vang))
y = earth.radius / 1e3 * np.outer(np.sin(uang), np.sin(vang))
z = earth.radius / 1e3 * np.outer(np.ones(np.size(uang)), np.cos(vang))

ax.plot_surface(x, y, z, color='blue')
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')
ax.set_zlabel('z (km)')
```

```
ax.azim = -90
ax.elev = 90

plt.axis('Equal')
plt.title('Hohmann Transfer of the Interceptor')
plt.legend()

plt.savefig('Fig. 4c Hohmann_transfer.png', dpi=300)

plt.show()
```

Define the cost function J and the gradient descent algorithm:

```
# Time settings
t_final = 3600 * 24

# Define the cost function J
def cost_function2D(v, theta, lambda_penalty=1e-3, v_i=v_i, r_p_t=r_p_t, r_a_t=r_a_t):
    """A 2D cost function based on the first, second burns and the transfer time.
    J = delta v1 + delta v2 + lambda * t_transfer

    Parameters
    ----------
    v: float
        v (m/s) value.
    theta: float
        Theta value.
    lambda_penalty: float, optional
        Lagrange multiplier or penalty factor. Default is 1e-3.
    v_i: float, optional
        Initial velocity of the interceptor.
    r_p_t: float, optional
        Periapsis of the target. Default is 14933.475 (km).
    r_a_t: float, optional
        Apoapsis of the target. Default is 51437.525 (km).

    Returns
    -------
    J: tuple
        Total cost.
    posvel_i: ndarray
        Position and velocity values over time in x, y & z planes.
    t_transfer: float
        Transfer time to the target's orbit.
    (first_burn, second_burn): tuple
        delta v1, delta v2 in the cost.
    """
    first_burn = abs(v - v_i)
    delta_v_val = v - v_i  # allowed to be negative (retrograde burn)

    vx0 = delta_v_val * np.sin(theta)
    vy0 = v_i + delta_v_val * np.cos(theta)
    vz0 = 0

    # Run the interceptor transfer
    transfer = Probe(probeqnsearth, t_final, t_num, x0=x0_i, vx0=vx0,
                y0=y0_i, vy0=vy0, z0=z0_i, vz0=vz0)
    t, posvel_i = transfer.odesolve()

    radii_int = np.linalg.norm(posvel_i[:, :3], axis=1)

    # Interceptor stops as soon as it hits the target's orbit
    condition = (radii_int >= r_p_t) & (radii_int <= r_a_t)
    if np.any(condition):
        idx = np.nonzero(condition)[0][0]
        t_transfer = t[idx]
    else:
        t_transfer = t[-1]

    if np.any(condition):
        v_int_vector = posvel_i[idx, 3:6]
        r_int = radii_int[idx]
    else:
        v_int_vector = posvel_i[-1, 3:6]
        r_int = radii_int[-1]

    # Find the target's velocity using energy conservation
    v_target = np.sqrt(mu * (2.0 / r_int - 1.0 / a_t))

    # Compute the difference in velocity vector
    v_target_vector = v_target * np.array([-np.sin(theta), np.cos(theta), 0])
```

```
69
70      second_burn = np.linalg.norm(v_int_vector - v_target_vector)
71
72      # Compute the cost J
73      J = first_burn + second_burn + lambda_penalty * t_transfer
74
75      return J, posvel_i, t_transfer, (first_burn, second_burn)
76
77  # Define the gradient descent model
78  def gradient_descent2D(
79      v_init, theta_init, delta_v=0.5, delta_theta=np.radians(0.5), lambda_penalty=1e-3,
80      max_iter=50, tol=1e-8, eta_v=0.5, eta_theta=1e-6, sub_steps=5
81  ):
82      """Automated gradient descent pipeline on (v, theta) based on 'cost_function2D'.
83
84      Parameters
85      ----------
86      v_init: float; theta_init: float
87            Initial guesses for v (m/s) and theta.
88      delta_v, delta_theta: float, optional
89            Finite-difference steps for v and theta. Defaults are 0.5 and np.radians(0.5).
90      lambda_penalty: float, optional
91            Lagrange multiplier or penalty factor. Default is 1e-3.
92      max_iter: int, optional
93            Max. no. of iterations. Default is 50.
94      tol: float, optional
95            Stopping threshold for parameter changes. Default is 1e-8.
96      eta_v,  eta_theta : float, optional
97            Learning rate for v and theta. Defaults are 0.5 and 1e-6.
98      sub_steps: int, optional
99            No. of sub-steps for mini updates. Default is 5.
100
101     Returns
102     -------
103     (v, theta): tuple
104                 Final optimised parameters.
105     [params_history]: ndarray
106                 (v, theta) per iteration.
107     [cost_history]: ndarray
108                 Cost values per iteration.
109     """
110     v = float(v_init)
111     theta = float(theta_init)
112     params_history = []
113     cost_history = []
114
115     def numeric_gradient(v_val, theta_val, lambda_penalty):
116         """Compute the partial derivatives of the cost function w.r.t. v and theta.
117
118         Parameters
119         ----------
120         v_val: float
121             v (m/s) value.
122         theta_val: float
123             Theta value.
124         lambda_penalty: float
125             Lagrange multiplier or penalty factor.
126
127         Returns
128         -------
129         [grad_v, grad_theta]: ndarray
130                               List of gradient values for v and theta.
131         """
132         # Partial derivative wrt v
133         J_plus_v, _, _, _  = cost_function2D(v_val + delta_v, theta_val, lambda_penalty)
134         J_minus_v, _, _, _ = cost_function2D(v_val - delta_v, theta_val, lambda_penalty)
135         grad_v = (J_plus_v - J_minus_v) / (2.0 * delta_v)
136
137         # Partial derivative wrt theta
138         J_plus_theta, _, _, _ = cost_function2D(v_val, theta_val + delta_theta, lambda_penalty)
139         J_minus_theta, _, _, _= cost_function2D(v_val, theta_val - delta_theta, lambda_penalty)
140         grad_theta = (J_plus_theta - J_minus_theta) / (2.0 * delta_theta)
141
142         return np.array([grad_v, grad_theta])
143
144     for iteration in range(1, max_iter + 1):
145         # Compute the gradient at the current (v, theta)
146         grad_v, grad_t = numeric_gradient(v, theta, lambda_penalty)
147
148         # Split the update into sub-steps
149         dv = (eta_v / sub_steps) * grad_v
```

```
        dt = (eta_theta / sub_steps) * grad_t

        for _ in range(sub_steps):
            # Do a small update
            v_new = v - dv
            theta_new = theta - dt

            # clamp v >= 7809 m/s, clamp theta in [0, 90 deg]
            v_new = max(v_new, 7809.0)
            theta_new = np.clip(theta_new, 0, np.radians(90))

            v, theta = v_new, theta_new

        # Evaluate cost
        J_current, _, _, _ = cost_function2D(v, theta, lambda_penalty)

        cost_history.append(J_current)
        params_history.append([v, theta])

        # Halt if both updates are sufficiently small
        if (abs(dv*sub_steps) < tol) and (abs(dt*sub_steps) < tol):
            print(f"Convergence at iteration {iteration}")
            break

    return (v, theta), np.array(params_history), np.array(cost_history)
```

Plot the cost J vs. no. of iterations (Fig. 15):

```
# Time settings
t_final = 3600 * 24  # simulation time

# Minimum burn needs to be > 911 m/s to escape the elliptical transfer
# So the test values for v would be > 7809 (~ 8000) m/s
v_init = 9300.0
theta_init = np.radians(25)

# Hyperparameters for tuning
common_lambda = 1e-3
tol = 1e-5 # 1e-5 ~ 1e-8 (BEST: 1e-5)
eta_v = 0.2 # 1e-2 ~ 1 (BEST: 0.2)
eta_theta = 2 * 1e-7 # 1e-6 ~ 1e-7 (BEST: 2 * 1e-7)
delta_v = 0.5
delta_theta = np.radians(0.01) # 0.1 ~ 0.01 (BEST: 0.01)
max_iter = 500
sub_steps = 20 # 5 ~ 20

# Run gradient descent from the initial guess
(v_optimal, theta_optimal), params_history, cost_history = gradient_descent2D(
    v_init, theta_init, delta_v=delta_v, delta_theta=delta_theta,
    lambda_penalty=common_lambda, max_iter=max_iter, tol=tol,
    eta_v=eta_v, eta_theta=eta_theta, sub_steps=sub_steps
)

print(f"Optimised v: {v_optimal:.3f} m/s.")
print(f"Optimised theta: {np.degrees(theta_optimal):.3f} degrees.")
print(f"Optimised cost: {cost_history[-1]:.3f}.")

# Plot the cost function J vs. iteration
iters = range(len(cost_history))

plt.figure()
plt.plot(iters, cost_history, 'x-', color='red', markersize=3)
plt.title(r"Cost $J$ vs. Iterations")
plt.xlabel("Iteration")
plt.ylabel("Cost")

plt.savefig('Fig. 4d Cost_vs._Iteration.png', dpi=300)

plt.show()
```

Plotting the contour surface of the cost function J (Fig. 16):

```
# Time settings
t_final = 3600 * 24

# Define ranges of v and theta for plotting
v_start = 7500
v_end = 10500
x_ticks = np.arange(v_start, v_end, 500)
y_ticks = np.arange(0, 65, 5)

vs = np.arange(v_start, v_end, 100)
thetas = np.arange(0, np.radians(65), np.radians(5))

# Map v and theta values into a meshgrid
V, T = np.meshgrid(vs, thetas)
cost_vals = np.zeros_like(V)

# Compute cost for each point in the grid
for i in range(V.shape[0]):
    for j in range(V.shape[1]):
        cost_vals[i, j], _, _, _ = cost_function2D(V[i, j], T[i, j])

# Plot the reference contour map
plt.figure(figsize=(10, 8))
cp = plt.contourf(V, np.degrees(T), cost_vals, levels=50, cmap='viridis')
plt.colorbar(cp, label='Cost')
plt.xlabel(r'$v$ ($m$ $s^{-1}$)')
plt.ylabel(r'$\theta$ (degree)')
plt.xticks(x_ticks)
plt.yticks(y_ticks)

plt.title(r'Contour Surface Plot of Cost Function $J$')
plt.show()

# Plot the actual contour map
plt.figure(figsize=(10, 8))

cp = plt.contourf(V, np.degrees(T), cost_vals, levels=50, cmap='viridis')
plt.colorbar(cp, label='Cost')
plt.xlabel(r'$v$ ($m$ $s^{-1}$)')
plt.ylabel(r'$\theta$ (degree)')
plt.xticks(x_ticks)
plt.yticks(y_ticks)
plt.axvline(7800, linestyle='--', c='white')

plt.title(r'Contour Surface Plot of Cost Function $J$')

params_history = np.array(params_history)

plt.plot(params_history[:, 0], np.degrees(params_history[:, 1]),
         'x', label='Descent Path', color='orange', linewidth=3, alpha=0.4, markersize=5)

# Annotate the Hohmann transfer
plt.annotate(text='Hohmann Transfer', xy=(7830, 0), xytext=(8000, 10),
             arrowprops={'arrowstyle': '->', 'color': 'white', 'linewidth': 3}, c='white', fontsize=15,
                  fontweight='bold')

# Annotate the optimised transfer
plt.annotate(text='Starting point\n(Initial parameters)', xy=(v_init, 25), xytext=(9200, 10),
             arrowprops={'arrowstyle': '->', 'color': 'white', 'linewidth': 3}, c='white', fontsize=15,
                  fontweight='bold')
plt.legend()

plt.savefig('Fig. 4e Contour.png', dpi=300)

plt.show()
```

Animate the gradient descent:

```
# Import additional packages for animation
import matplotlib.animation as animation
from matplotlib.animation import FFMpegWriter

fig, ax = plt.subplots(figsize=(10, 8))

# Plot the contour
cp = ax.contourf(V, np.degrees(T), cost_vals, levels=50, cmap='viridis')
cbar = plt.colorbar(cp, ax=ax)
cbar.set_label('Cost')
ax.set_xlabel(r'$v$ ($m \, s^{-1}$)')
ax.set_ylabel(r'$\theta$ (degree)')
ax.set_title(r'Contour Surface Plot of Cost Function $J$')
ax.axvline(7800, linestyle='--', c='white')

# Annotations
ax.annotate(text='Hohmann Transfer', xy=(7830, 0), xytext=(8000, 10),
            arrowprops={'arrowstyle': '->', 'color': 'white', 'linewidth': 3},
            c='white', fontsize=15, fontweight='bold')
ax.annotate(text='Starting point\n(Initial parameters)', xy=(v_init, 25),
            xytext=(9200, 10),
            arrowprops={'arrowstyle': '->', 'color': 'white', 'linewidth': 3},
            c='white', fontsize=15, fontweight='bold')

# Plot the descent path
params_history = np.array(params_history)

# Create the animation
line, = ax.plot([], [], 'x-', lw=3, markersize=5, alpha=0.4,
                label='Descent Path', color='orange')
ax.legend()

# Define a function to instantiate the animation data
def init():
    line.set_data([], [])
    return (line,)

# Define the animation function
def animate(i):
    data = params_history[:i+1, :]
    v_vals = data[:, 0]
    theta_vals = np.degrees(data[:, 1])
    line.set_data(v_vals, theta_vals)
    return (line,)

# Create the animation
ani = animation.FuncAnimation(fig, animate, frames=len(params_history),
                              init_func=init, interval=200, blit=True, repeat=False)

# Set the customised FFmpeg path
import matplotlib as mpl
mpl.rcParams['animation.ffmpeg_path'] = r'C:\Users\tseng\xxx\ffmpeg.exe'

# Customise the settings
writer = FFMpegWriter(
    fps=30,
    metadata={'artist': 'Tszon Tseng', 'title': 'Gradient Descent Animation'},
    bitrate=1800
)
ani.save('gradient_descent_mp4.mp4', writer=writer, dpi=300)

plt.show()
```

Plot the guess, optimised and Hohmann transfers (Fig. 14):

```
fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Plot the interceptor and the target
ax.plot(posvel_i[:, 0] / 1e3, posvel_i[:, 1] / 1e3, color='skyblue', label="Interceptor's Orbit")
ax.plot(posvel_target[:, 0] / 1e3, posvel_target[:, 1] / 1e3, color='red', label="Target's Orbit")

# Plot the forward and backwards Hohmann transfers
ax.plot(posvel_transfer_1[:, 0] / 1e3, posvel_transfer_1[:, 1] / 1e3, color='black', linestyle='--',
    label='Forward Hohmann')
ax.plot(posvel_transfer_2[:, 0] / 1e3, posvel_transfer_2[:, 1] / 1e3, color='green', linestyle='--',
    label='Backward Hohmann')

# Plot the transfer for the initial guess
_ , posvel_guess, t_guess, (firstburn_guess, arrivalburn_guess) = cost_function2D(v_init, theta_init)
ax.plot(posvel_guess[:, 0] / 1e3, posvel_guess[:, 1] / 1e3, color='black', linewidth=2, label='Initial
    Guess')
print(f"Time required with initial params:\n{v_init:.3f} m/s, {np.degrees(theta_init):.3f} degrees,\
    n1st burn: {firstburn_guess:.3f} m/s, 2nd burn: {arrivalburn_guess:.3f} m/s\nwould be {t_guess /
    60:.3f} mins.\n")

optimal_v = params_history[-1][0]
optimal_theta = params_history[-1][1]

# Plot the optimised transfer
_ , posvel_opt, t_optimal, (firstburn_optimal, arrivalburn_optimal) = cost_function2D(optimal_v,
    optimal_theta, lambda_penalty=common_lambda)
ax.plot(posvel_opt[:, 0] / 1e3, posvel_opt[:, 1] / 1e3, color='orange', linewidth=2, label='Optimised
    Transfer')
print(f"Time required for the optimised transfer:\n{optimal_v:.3f} m/s, {np.degrees(optimal_theta):.3f}
     degrees,\n1st burn: {firstburn_optimal:.3f} m/s, 2nd burn: {arrivalburn_optimal:.3f} m/s\nwould be
     {t_optimal / 60:.3f} mins.")

# Plot the Earth
ax.plot_surface(x, y, z, color='blue')
ax.set_xlabel('x (km)')
ax.set_ylabel('y (km)')
ax.set_zlabel('z (km)')
ax.azim = -90
ax.elev = 90

plt.axis('Equal')
plt.title('Optimisation of Spacecraft Transfer')
plt.legend()

plt.savefig('Fig. 4f Optimised_transfer.png', dpi=300)

plt.show()
```

Run the MCMC sampler for random walkers (Fig. 17):

```
# Import the MCMC package
import emcee, corner

# Define the Boltzmann factor
sigma = 1.0 # Set to 1 for simplicity

# Define the likelihood function
def log_likelihood(params, sigma=sigma):
    v, theta = params
    J, _, _, _ = cost_function2D(v, theta)
    # Returns -ve infinity in case of missing values
    if np.isnan(J):
        return -np.inf
    return -J / (2 * sigma**2)

# Define the prior
def log_prior(params):
    v, theta = params
    if 7809.0 < v < 15000.0 and 0.0 <= theta <= np.radians(90):
        return 0.0
    # Returns -ve infinity for very high values of v and theta
    return -np.inf

# Define the log-probability
def log_prob(params, sigma=sigma):
    lp = log_prior(params)
    # Check if the log-probability is finite
    # Again, returns -ve infinity for very high values of v and theta
    if not np.isfinite(lp):
        return -np.inf
    return lp + log_likelihood(params, sigma)

# Run the MCMC sampler
initial = np.array([optimal_v, optimal_theta])
n_dim = 2 # no. of dimensions (or parameters)
n_walkers = 35 # no. of random walkers
n_iter = 300

# Define the random walkers
p0 = initial + 1e-2 * np.random.rand(n_walkers, n_dim)

sampler = emcee.EnsembleSampler(n_walkers, n_dim, log_prob, args=(sigma,))
sampler.run_mcmc(p0, n_iter, progress=True)

samples = sampler.get_chain(discard=150, flat=True)

# Make trace plots
fig, axes = plt.subplots(2, figsize=(8, 6), sharex=True)

for i in range(n_walkers):
    axes[0].plot(sampler.chain[i, :, 0], alpha=0.5)
    axes[1].plot(np.degrees(sampler.chain[i, :, 1]), alpha=0.5)
axes[0].set_ylabel(r"$v$ $(ms^{-1})$")
axes[0].grid(visible=True, alpha=0.6)
axes[0].axvline(150, c='r', linestyle='--', alpha=0.7)

axes[1].set_ylabel(r"$\theta$ (degree)")
axes[1].set_xlabel("No. of Iteration")
axes[1].grid(visible=True, alpha=0.6)
axes[1].axvline(150, c='r', linestyle='--', alpha=0.7)

plt.suptitle("Trace Plots of the Random Walkers in the MCMC Simulation")
plt.savefig('Fig. 4g MCMC_traceplot.png', dpi=300)

plt.show()
```

Animate the random walkers.

```
samples = sampler.chain
# Unpack the dimensions of the chains
n_walkers, n_iter, n_params = samples.shape

fig, axes = plt.subplots(2, figsize=(8, 6), sharex=True)

axes[0].set_ylabel(r"$v$ (m s$^{-1}$)")
axes[0].grid(visible=True, alpha=0.6)

axes[1].set_xlabel("Iteration")
axes[1].set_ylabel(r"$\theta$ (degree)")
axes[1].grid(visible=True, alpha=0.6)

plt.suptitle("Random Walkers in the MCMC Simulation")

# Instantiate empty lists for v and theta values
lines_v = []
lines_theta = []

for _ in range(n_walkers):
    (l_v,) = axes[0].plot([], [], alpha=0.6)
    (l_theta,) = axes[1].plot([], [], alpha=0.6)
    lines_v.append(l_v)
    lines_theta.append(l_theta)

# Define a function to instantiate the animation data
def init():
    for l_v, l_theta in zip(lines_v, lines_theta):
        l_v.set_data([], [])
        l_theta.set_data([], [])
    return lines_v + lines_theta

# Define the animation function
def animate(i):
    x = np.arange(i + 1)
    for j in range(n_walkers):
        y_v = samples[j, : i + 1, 0]
        y_theta = np.degrees(samples[j, : i + 1, 1])
        lines_v[j].set_data(x, y_v)
        lines_theta[j].set_data(x, y_theta)
    return lines_v + lines_theta

# Create the animation
ani = animation.FuncAnimation(
    fig, animate, frames=n_iter,
    init_func=init, interval=100, blit=False, repeat=False
)

mpl.rcParams['animation.ffmpeg_path'] = (
    r'C:\Users\tseng\xxx\ffmpeg.exe'
)

writer = FFMpegWriter(fps=30, bitrate=1800)
ani.save('random_walkers_mp4.mp4', writer=writer, dpi=300)

plt.show()
```

Extract the final converged results at iteration 300 from the Markov chains:

```
v_vals_last = sampler.chain[:, -1, 0]
print(f'v values in the last iteration:\n{v_vals_last}\n')
print(f'Mean: {np.mean(v_vals_last):.3f} m/s\n')
print(f'Median: {np.median(v_vals_last):.3f} m/s\n')
print(f'Std: {np.std(v_vals_last):.3f} m/s\n')

print('---\n')

theta_vals_last = np.degrees(sampler.chain[:, -1, 1])
print(f'Theta values in the last iteration:\n{theta_vals_last}\n')
print(f'Mean: {np.mean(theta_vals_last):.3f} degrees\n')
print(f'Median: {np.median(theta_vals_last):.3f} degrees\n')
print(f'Std: {np.std(theta_vals_last):.3f} degrees\n')
```