Infinite Horizon Control Using Koopman Operator Surrogate Models

Master's Thesis in Informatics

Abdülkadir Aslan

Technical University of Munich
Department of Informatics

TECHNISCHE UNIVERSITÄT MÜNCHEN

Infinite Horizon Control Using Koopman Operator Surrogate Models

Regelsteuerung über unendliche Horizonte mit Koopman Operator Modellen

Master's Thesis in Informatics

Abdülkadir Aslan

Supervised by **Dr. Felix Dietrich**

Technical University of Munich
Department of Informatics

Submitted on:
October 15, 2022

Thesis submitted in fulfilment of the requirements for the degree of M.Sc. in Informatics.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

October 15, 2022
Munich

Abdülkadir Aslan

Acknowledgements

I would like to thank my advisor, Dr. Felix Dietrich for giving me an opportunity to work on this amazing project and driving my interest towards Control Theory. His consistent support, guidance and valuable feedback kept me excited throughout this journey. I also would like to thank my family for their consistent support and all the sacrifices they made during my years of education. I wouldn't be here without you.

Abstract

Model Predictive Control is a process control method that produces a series of control inputs at each time step by predicting the future outputs of a dynamical system among which the first one is applied to the system to move it closer to the target state. When the underlying system is linear, the optimization problem is convex and easy to solve. However, for most cases where the underlying dynamical system is non-linear or when complicated constraints must be satisfied, local optimization becomes computationally expensive.

To tackle this problem, Koopman Operator Framework can be utilized which in theory, lifts the dynamics of the system into an infinite dimensional space where the dynamics becomes linear. In practice, a finite-dimensional approximation of the Koopman Operator can be obtained by Extended Dynamic Mode Decomposition which being a purely data-driven method, is efficient to use. MPC controllers designed with the Koopman Operator can use linear optimization techniques to produce the set of control inputs at each time step for a limited horizon by estimating the future states of the system sufficiently accurate enough for the prediction horizon.

Koopman MPC still requires local optimization at every time step. In this thesis, the Infinite Horizon Control framework is investigated instead, where no local optimization is needed. An implementation of Infinite Horizon Control that uses Linear Quadratic Regulator to generate optimal control inputs is presented which aims to achieve better efficiency while still providing a reliable controller. The results of the different scenarios are evaluated by using various metrics and a specially designed one that utilizes the nature of the test cases. The IHC controller is able to accurately control the spacecraft with a fraction of the computational cost of Koopman MPC.

List of Abbreviations

API Application Programming Interface
CPU Central Processing Unit
CSV Comma Separated Values
DARE Discrete Time Algebraic Riccati Equation
DMD Dynamic Mode Decomposition
EDMD Extended Dynamic Mode Decomposition
IHC Infinite Horizon Control
JSON JavaScript Object Notation
KMPC Koopman MPC
kRPC Kerbal Remote Procedure Call
KSP Kerbal Space Program
LQR Linear Quadratic Regulator
MPC Model Predictive Control
ODE Ordinary Differential Equation
PID Proportional, Integral, Derivative
POD Proper Orthogonal Decomposition
RBF Radial Basis Function
RMSE Root Mean Squared Error
SVD Singular Value Decomposition
XLS Microsoft Excel Spreadsheet

Contents

1

Introduction

Model Predictive Control (MPC) is a process control method that produces a control input which is optimized by predicting the future outputs of the system. It has a wide range of use cases in the industry for the complex processes requiring many constraints to be satisfied that are too complex for Proportional, Integral, Derivative (PID) Controllers to deal with. As the interest in autonomous driving and space exploration grows, MPC becomes the backbone for advances in many areas of technology (Faulwasser et al., 2021).

However, MPC uses costly optimization for each input suggestion to the real system and non-linear systems make this process computationally hard to overcome since convex optimization is not possible for them. Koopman Operator Framework provides a solution to this by lifting the non-linear dynamics of the system into a higher dimension where it can be treated as a linear system. It is sufficient for the Koopman MPC (KMPC) to provide an estimator that can closely approximate the dynamics for a long enough time interval since it only needs to produce proper control inputs for a limited horizon. But what if the linearized system covers the dynamics sufficiently so that it is not needed to compute control inputs for a receding horizon on each time step? What if we can just solve the system and get optimal control inputs for an infinite horizon?

This thesis provides an implementation of Infinite Horizon Control (IHC) using Linear Quadratic Regulator (LQR) on a system linearized with a finite-dimensional approximation of the Koopman Operator obtained with Extended Dynamic Mode Decomposition (EDMD). It evaluates how accurate and efficient this approach is compared to regular MPC and KMPC. Kerbal Space Program (KSP) will be the simulation tool and existing implementations of the MPC and PID controllers will be utilized (Atukalp, 2021; Ganbarov, 2020). Collaborated data is used for training and evaluations. The next chapter will summarize the theory behind the key concepts, the third chapter will go over the environment, metrics, implementation and evaluation; and finally, the last chapter will summarize everything and discuss the future directions.

2

Theoretical Background

This chapter gives a theoretical background to the concepts of open and closed loop control systems, PID Controllers, Model Predictive Control (MPC), Infinite Horizon Control (IHC), Koopman Operator, Dynamic Mode Decomposition (DMD) and Extended Dynamic Mode Decomposition (EDMD). Finally, it explains what Kerbal Space Program (KSP), the testing environment for this thesis is and how it works.

2.1 | Proportional Integral Derivative Control

Understanding types of control systems is essential for building a solid background before moving further. The open loop control system is the first concept to be elaborated. It is the very basic control system that one can think of. Consider a car that goes from point A to point B. A trivial approach would be to calculate the distance, give the input throttle signal and expect it to arrive at point B based on the calculations. However, there may be obstacles on the road that can divert the vehicle from its goal. One of the tires may be exposed to more friction that can bend the path the car is expected to follow. There may be other factors such as wind or rain that may require the vehicle to have extra throttle to arrive at point B. The necessity of a feedback loop emerges from scenarios like this so that it is possible to alter control inputs for achieving the goal (Prasad et al., 2014).

This is where closed-loop control systems are often used. In a closed-loop control system, there is a feedback mechanism that allows the control input to be changed based on the output produced from the previous input signal. Output is provided to a controller which calculates the **error** between the output and the **reference signal**, and then produces the new input signal to get closer to the reference.

In order to generate the input signal, a control system called **PID Control** can be used that is proven to be very simple, efficient and effective. PID stands for **Proportional, Integral, Derivative** which composes three separate components for calculating new input signal. Figure 2.1 illustrates the flow of a PID Controller.

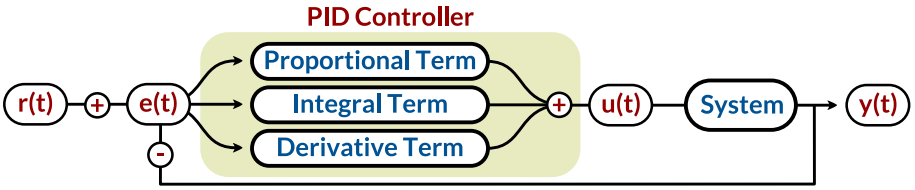

Figure 2.1: Diagram of a PID Controller.

Figure 2.1 shows that there are three paths to follow when calculating the next input signal. In the proportional path, the error term is multiplied with a constant called **Proportional Gain** (K_P). In the integral path, the error term is integrated and multiplied with another constant, **Integral Gain** (K_I) and in the derivative path, it is differentiated and multiplied with yet another constant, **Derivative Gain** (K_D). Equation 2.1 shows how these terms form the control signal:

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)d\tau + K_D \frac{de(t)}{dt}, \tag{2.1}$$

where **Proportional** path measures how much error there is in a given time and contributes to control input directly to eliminate the error. **Integral** path removes the constant errors in the system by accumulating errors over time to make them significant enough eventually. **Derivative** path takes the rate of change of the error into consideration to assure a more stable path to reference by preventing overshoots. See Figure 2.2. (Ang et al., 2005).

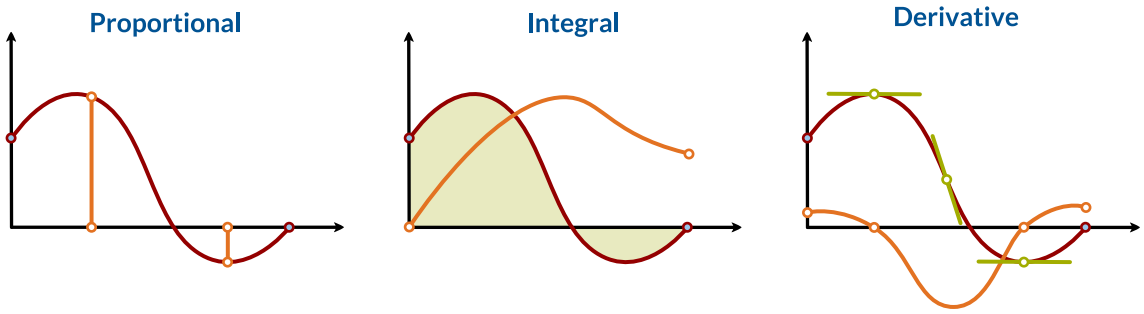

Figure 2.2: **Response** over time to **error** for each path.

2.2 | Model Predictive Control

The next concept in line to elaborate is **Model Predictive Control (MPC)** which is a process control method for closed-loop control systems. Consider the example given to explain PID Control. There is a car that needs to reach a point and there may be many factors that can divert it from the reference, as before.

MPC predicts an optimal control input sequence that takes the car closer to the reference, over a finite time horizon. After the input sequence is optimized, the first input signal is applied which brings the car to a new state that is closer to the reference and another optimization is made over the slid horizon. A set of new input signals is produced, the first signal is applied, the new state is observed, the prediction horizon is slid one time step further and this procedure is repeated over and over again until the car reaches the goal. Figure 2.3 illustrates these prediction horizons and their application to reach reference signal. Figure 2.4 shows the interaction pattern between MPC and the system it optimizes (Morari et al., 1988).

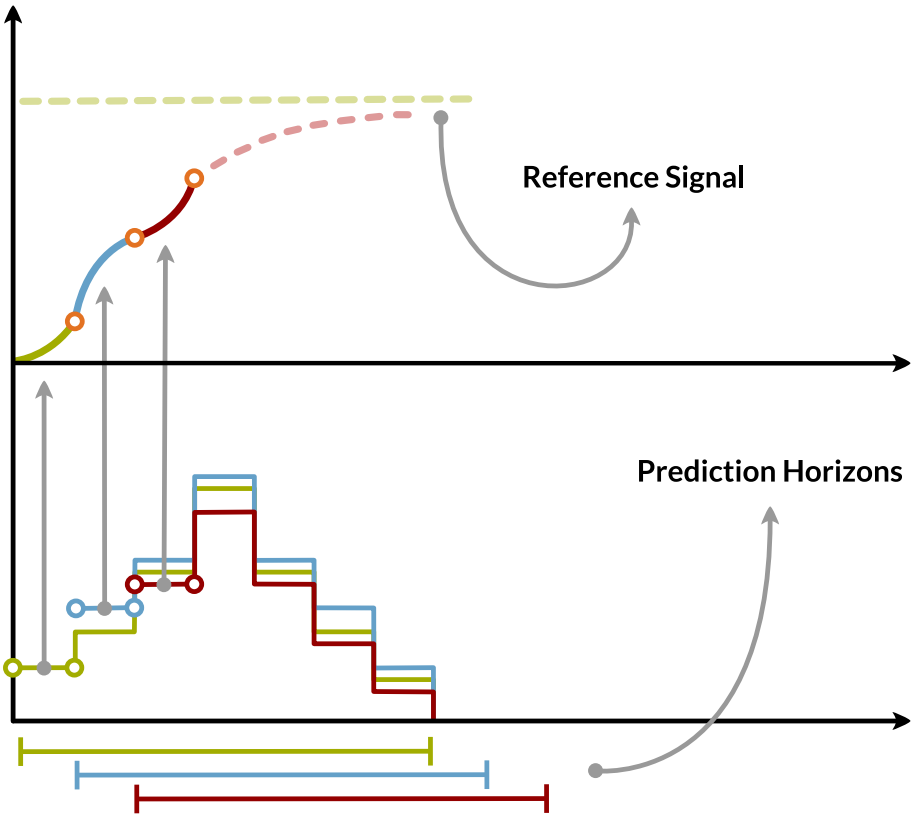

Figure 2.3: Model Predictive Control for three time steps.

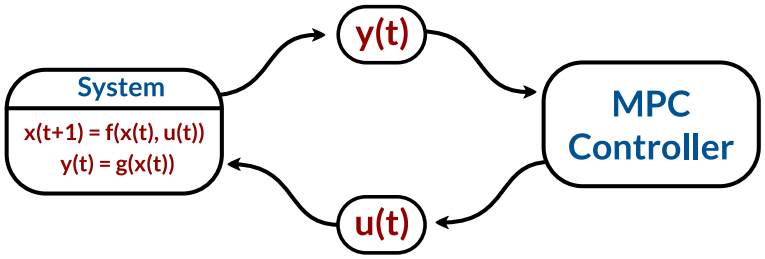

Figure 2.4: Model Predictive Control diagram.

The power of MPC comes from optimization. It optimizes the input signal while taking also the future inputs into consideration. It also enables constraints to be put on the input signal. On a regular scheme, a badly designed controller can tell the car to go a lot faster than it is able to go. With MPC, it can be imposed that the input given to the car stays within some threshold. Ability to optimize over a set of constraints and consideration of the future inputs makes MPC a very powerful strategy (Kaiser et al., 2018).

Another very important feature of MPC is that it can optimize the input signal for non-linear systems as well, though it can be costly (Findeisen and Allgöwer, 2002). Here rises an even better strategy when there is a non-linear system to optimize. The system can be linearized and optimized with MPC. It will be a lot faster and MPC can optimize the input signal very good thanks to its ability to reinitialize and optimize at each time step over and over again. This reduces the probability of making critical errors that arose from the accuracy of the approximated non-linear system (Zhakatayev et al., 2017).

Now, consider the case where there is a linear system with a lot of disturbances or a linearized system which also has its deviations since it is an approximation of a non-linear system. An optimal input signal can be calculated for a linear system using an LQR. Yet, external factors can steer the car far away from the predicted path. This is where MPC shines because it optimizes the input signal in each iteration and so, it compensates the effects of external factors over time (Mattingley et al., 2010).

When it comes to the downsides of MPC, it depends on the assumption that there is a computer with sufficient power to carry out necessary calculations fast enough since the whole optimization scheme is needed to be computed in each time step over and over again. Optimizing a linear system is the more flexible option here since it requires less resources than optimizing a non-linear system. This is why linearization is a very useful strategy when there is a non-linear system to optimize. It will produce input sequences faster, but it also needs to be considered that success can depend on how good the approximation is.

2.3 | Infinite Horizon Control

In this section, the concept of **Infinite Horizon Control (IHC)** will be discussed. First, it is important to distinguish two different approaches to control a process in terms of horizons that are **receding** and **infinite**. For the receding horizon control, optimal control inputs are recalculated in each time step for a **finite horizon**. Reaching the goal at the end is what matters here and the path can be corrected over and over again if the controller observes diversion from it (Mattingley et al., 2010).

On the contrary, infinite horizon optimal control approach does not have a chance to recalculate the dynamics of the system. The start and end points of the journey do not matter here because the whole set of equations that describe the dynamics of the system has a solution that is within reach. When an initial state and a target state are given, optimal control inputs that take the process to the target state can be calculated (Carlson and Haurie, 1987).

Before moving further, it is necessary to go over the state space representation of a linear system. There are three vectors and four matrices that define a linear control system. **State vector** x defines the state of the system, **input** or **control vector** u defines the control inputs and **output vector** y defines the output of the system which is basically what can be observed outside instead of the full internal state of the system, x. The first matrix is the **system matrix** A which defines how the state of the system evolves without any control input. The next one is the **control matrix** B which describes how the control inputs applied to the current state contribute to the next state of the system. There are two other matrices **output matrix** C and **feed-through matrix** D that define how the state and control input **directly modifies the output of the system** which is in the scope of this thesis, not the case. Equations 2.2 and 2.3 shows the state space representation of a discrete-time linear system (Brogan, 1985):

$$x_{k+1} = Ax_k + Bu_k, \tag{2.2}$$

$$y_{k+1} = Cy_k + Du_k, \tag{2.3}$$

which can be solved to obtain control inputs for the process. The linear equation $z_{k+1} = Az_k + Bu_k$ defines the linear system in this thesis and matrices C, D and output y are neglected because all the internal states of the system can be observed here through the output (Kalman, 1960). In order to find optimal control inputs for the infinite horizon, discrete-time systems, **Linear Quadratic Regulator (LQR)** can be used. But first, a performance metric needs to be defined, or in other words, a cost function.

Cost function $\mathcal{J}$ composes the weighted sums of **performance** $\mathcal{Q}$ and **actuator effort** $\mathcal{R}$ and these two matrices allow controlling the behaviour of the system. Bad performance can be penalized by adjusting $\mathcal{Q}$ and bad actuator effort can be penalized by adjusting $\mathcal{R}$. Equation 2.4 shows how the cost function is formed (Brunton and Kutz, 2019):

$$\mathcal{J} = \sum_{k=0}^{\infty} (x_k^T \mathcal{Q} x_k + u_k^T \mathcal{R} u_k). \tag{2.4}$$

Equation 2.5 shows how the optimal control sequence u_k that minimizes the cost function:

$$u_k = -\mathcal{K} x_k \tag{2.5}$$

is obtained by multiplying the **current state of the system** x_k with the **feedback matrix** $\mathcal{K}$. In order to obtain **state feedback gains** $\mathcal{K}$, solution to the **Discrete Time Algebraic Riccati Equation (DARE)** P is required which can be obtained through dynamic programming, by iterating the dynamic Riccati equation until convergence. Equation 2.6 shows DARE and Equation 2.7 shows how the feedback matrix is calculated (Kostova et al., 2013):

$$P = Q + A^T P A - A^T P B (R + B^T P B)^{-1} B^T P A, \tag{2.6}$$

$$\mathcal{K} = (R + B^T P B)^{-1} B^T P A. \tag{2.7}$$

In order to solve the discrete-time LQR problem, **Python Control Systems Library** (Fuller et al., 2021) can be used which returns the optimal gain set as shown in Equation 2.8 when system, control, performance and actuator effort matrices are provided:

$$\mathcal{K}, \mathcal{S}, \mathcal{E} = dlqr(A, B, \mathcal{Q}, \mathcal{R}), \tag{2.8}$$

where $\mathcal{K}$ is the **state feedback gains**, $\mathcal{S}$ is the **solution to Riccati equation** and $\mathcal{E}$ is the **eigenvalues of the closed loop system**. Optimal control input for the system can be obtained as in Equation 2.5. How well Infinite Horizon Control approach works can be considered as a feedback on how good the linear approximation of the underlying system is when applied on a linearized system. Moreover, a successful result on this approach can also save a lot of resources and time by getting rid of recalculating the dynamics of the system at each time step. Feedback matrix $\mathcal{K}$ can be used to obtain optimal control inputs regardless of the time step as in Equation 2.5. If the approximation is well enough, process will get closer to the target state.

2.4 | Koopman Operator

Using MPC reduces the cost and improves speed for both linear and non-linear systems. However, optimizing a linear system instead of a non-linear one takes less resources. This section will introduce the Koopman Opearator that can approximate a non-linear systems to a linear one and discuss how it can improve the speed with a compromise on accuracy.

In theory, Koopman Operator shows that it is possible for non-linear systems to be represented with linear operators in an infinite-dimensional space. Let $\mathcal{D} \subseteq \mathbb{R}^n$, let $f : \mathcal{D} \to \mathcal{D}$, and for all $k \geq 0$, consider the discrete-time system below:

$$x(k+1) = f(x(k)) \quad \text{where} \quad x(0) = x_0, \tag{2.9}$$

$$y(k) = g(x(k)), \tag{2.10}$$

where $x, y \in \mathcal{D}$ and $g : \mathcal{D} \to \mathbb{R}$. Now, let $\mathcal{G}$ be a set of real-valued functions on $\mathcal{D}$ which satisfy the following property:

$$\text{If} \quad g \in \mathcal{G} \quad \text{then} \quad g \circ f \in \mathcal{G}. \tag{2.11}$$

If 2.11 holds, then $\mathcal{G}$ becomes *compositionally complete* and since $g \in \mathcal{G}$ and $\mathcal{G}$ is compositionally complete, g is an *observable* function. Now we can define the *Koopman Operator* $\mathcal{K} : \mathcal{G} \to \mathcal{G}$ as follows:

$$\mathcal{K}(g) \triangleq g \circ f \tag{2.12}$$

and since $\mathcal{G}$ is compositionally complete, if $g \in \mathcal{G}$, then $g \circ f \in \mathcal{G}$ which makes $\mathcal{K}(g) \in \mathcal{G}$. Now, we assume that $\mathcal{G}$ is a compositionally complete vector space over $\mathbb{R}$. Then, for all $g_1, g_2 \in \mathcal{G}$ and all $a_1, a_2 \in \mathbb{R}$,

$$\begin{aligned} \mathcal{K}(a_1 g_1 + a_2 g_2) &= (a_1 g_1 + a_2 g_2) \circ f \\ &= a_1 g_1 \circ f + a_2 g_2 \circ f \\ &= a_1 \mathcal{K}(g_1) + a_2 \mathcal{K}(g_2). \end{aligned} \tag{2.13}$$

Therefore, $\mathcal{K}$ is a linear operator on $\mathcal{G}$. This provides a simple overview of the Koopman Operator. For the whole proof and numerical examples, see Bruce et al. (2019).

This thesis aims to find a finite-dimensional representation of Koopman Operator so that linear representation of the underlying system can be obtained. Non-linear state space will be lifted to a higher dimension where z_k is the *lifted state vector* at time step k, u_k is the input and the non-linear state vector x_k is considered as the output so $y_k := x_k$. The non-linear system and its linear approximation:

$$x_{k+1} = f(x_k, u_k) \quad \text{and} \quad y_k = g(x_k), \tag{2.14}$$

$$z_{k+1} = Az_k + Bu_k \quad \text{and} \quad y_k = Cz_k \quad \text{for} \quad z_0 = \phi(x_0) \tag{2.15}$$

is shown in Equations 2.14 and 2.15. Remember that in this thesis, the system is in discrete-time which will be considered for the future calculations.

Solving a linear system is a lot faster than optimizing a non-linear system and if the approximation is good enough, goal can be achieved with a lot less computational resources. Koopman Operator will serve this thesis as a linearizer for the non-linear system. Then the linear system obtained by Koopman Operator will be optimized using Infinite Horizon Optimal Control approach. Figure 2.5 illustrates how Koopman Operator works (Brunton, 2019). It is important to note that stated functions in the figure works as shown in Equation 2.16:

$$\begin{aligned} F_t &: x_k \to x_{k+1}, \\ g &: x_k \to y_k, \\ K_t &: y_k \to y_{k+1}. \end{aligned} \tag{2.16}$$

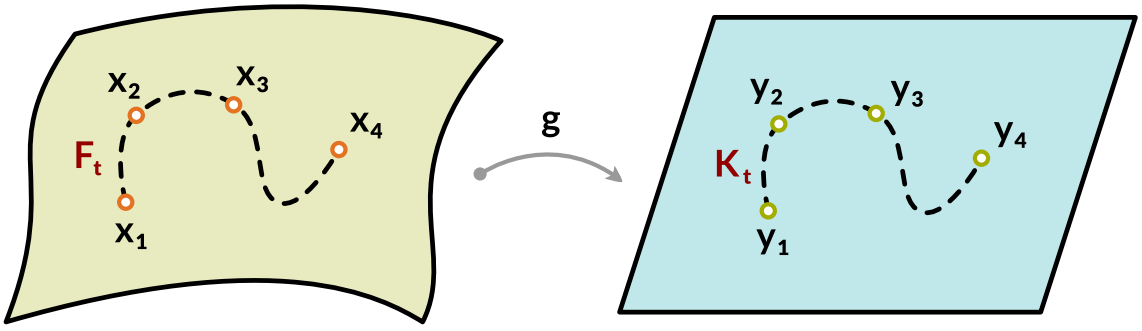

Figure 2.5: Visualization of the Koopman Operator's workflow.

2.4.1 | Dynamic Mode Decomposition

This subsection will provide the theory behind the **Dynamic Mode Decomposition (DMD)** and show how it approximates the Koopman Operator with a linear model. The reason that DMD is an algorithm that is applicable to many cases is that it is purely data driven. Without the need for any knowledge of the underlying dynamical equations of a system, data snapshots can be fed into DMD and spatial temporal modes along with a linear dynamical system defining how they evolve in time can be fetched (Schmid, 2010).

DMD takes the data of a dynamical system evolving in time to work. Snapshots taken at different time steps of the dynamical system are reshaped and stacked into matrix **X** and another matrix **X′** contains the same data, just shifted one time step into the future. Columns of these matrices are evolving in time along with the dynamics of the system. These matrices are constructed as follows:

$$X = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \dots & x_{m-1} \\ | & | & & | \end{bmatrix}, \quad X' = \begin{bmatrix} | & | & & | \\ x_2 & x_3 & \dots & x_m \\ | & | & & | \end{bmatrix}. \tag{2.17}$$

The aim of DMD is to find the best linear operator **A** that advances the dynamical system one time step further, or in other words, maps **X** into **X′** as such:

$$X' \approx AX, \tag{2.18}$$

$$A = X'X^{\dagger}, \tag{2.19}$$

where $\mathbf{X}^{\dagger}$ is the pseudo inverse of **X**. Computing the matrix **A**, let alone its eigenvalues and eigenvectors, is too hard since it will be a huge matrix for most cases. What DMD does is that it approximates the leading eigendecomposition of this matrix where leading eigenvectors corresponds to the spatial temporal coherent modes, and leading eigenvalues define how these modes evolve in time. In order to compute DMD, first step is to compute the Singular Value Decomposition (SVD) of matrix **X** as in Equation 2.20:

$$X = U\Sigma V^{*} \quad \text{and} \quad X' = AU\Sigma V^{*}, \tag{2.20}$$

where matrix **U** contains the Proper Orthogonal Decomposition (POD) modes of **X** hierarchically ordered in terms of capturing the variance of **X**. From this point on, a proper **truncation value** r is selected and the reduced matrices can be used.

Equation 2.21 describes how leading singular values are filtered by computing the reduced matrices that are used to obtain an approximation of the SVD of X:

$$X = U\Sigma V^* = \begin{bmatrix} \tilde{U} & \tilde{U}_{\text{rem}} \end{bmatrix} \begin{bmatrix} \tilde{\Sigma} & 0 \\ 0 & \tilde{\Sigma}_{rem} \end{bmatrix} \begin{bmatrix} \tilde{V}^* \\ \tilde{V}^*_{\text{rem}} \end{bmatrix} \approx \tilde{U}\tilde{\Sigma}\tilde{V}^*, \tag{2.21}$$

where $U \in \mathbb{R}^{n\times n}$, $\Sigma \in \mathbb{R}^{n\times m-1}$, $V^* \in \mathbb{R}^{m-1\times m-1}$, $\tilde{U} \in \mathbb{R}^{n\times r}$, $\tilde{\Sigma} \in \mathbb{R}^{r\times r}$, $\tilde{V}^* \in \mathbb{R}^{r\times m-1}$ and notation $_{rem}$ denotes the remaining parts of these matrices. Using the SVD of matrix X, an approximation for the operator $\mathbf{A}$ and a dynamic model of the process can be constructed as in Equations 2.22 and 2.23:

$$A \approx \bar{A} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^*, \tag{2.22}$$

$$x_{k+1} = \bar{A}x_k. \tag{2.23}$$

The next step in DMD is to project matrix $\bar{\mathbf{A}}$ onto the dominant singular vectors $\tilde{\mathbf{U}}^*$ and $\tilde{\mathbf{U}}$ to get matrix $\tilde{\mathbf{A}}$ which is a lower order linear dynamical system that describes how the POD modes evolve in time. This matrix is smaller than $\bar{\mathbf{A}}$ and has the same eigenvalues. Equation 2.24 shows how to obtain $\tilde{\mathbf{A}}$:

$$\tilde{U}^*X'\tilde{V}\tilde{\Sigma}^{-1} = \tilde{U}^*\bar{A}\tilde{U} = \tilde{A}. \tag{2.24}$$

Since the matrix $\tilde{\mathbf{A}}$ has the same eigenvalues as matrix $\bar{\mathbf{A}}$, eigendecomposition of $\tilde{\mathbf{A}}$ can be used to get eigenvalues of $\bar{\mathbf{A}}$ as in Equation 2.25:

$$\tilde{A}W = W\Lambda. \tag{2.25}$$

Now that the eigenvalues of $\tilde{\mathbf{A}}$ are calculated, dynamic modes that represent the dominant coherent structures of the system can be obtained, which is described by the eigenvectors of $\bar{\mathbf{A}}$ that is an approximation of $\mathbf{A}$. See Equation 2.26:

$$\Phi = X'\tilde{V}\tilde{\Sigma}^{-1}W. \tag{2.26}$$

What DMD achieves is that it extracts the leading dynamic modes of the system without actually calculating the matrix $\mathbf{A}$. These DMD modes (Φ in Equation 2.26) represent the spatial correlations between each snapshot of the system and the eigenvalues of matrix $\mathbf{A}$ (Λ in Equation 2.25) defines the growth, decay and oscillations of these modes over time. See Tu et al. (2014) for the detailed computations and proofs.

2.4.2 | DMD and EDMD For Control

Now that the basics of DMD is explained, it is time to add control inputs to the equation. This subsection will go over the theory behind computing system and control matrices for a dynamic process using DMD, give an introduction to EDMD and introduce an example on how these algorithms perform. Dynamic Mode Decomposition is able to approximate both system and control matrices just by processing snapshots of the control inputs and outputs of the system. The snapshots of the process, $\mathbf{X}$ and $\mathbf{X}'$ are already introduced. Now, another matrix that contains the snapshots of control inputs will be introduced as in Equation 2.27:

$$\Upsilon = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_{m-1} \\ | & | & & | \end{bmatrix}, \tag{2.27}$$

where $\mathbf{\Upsilon} \in \mathbb{R}^{l \times m-1}$ since each input snapshot has a length of l and the dynamics of the system will be as in Equation 2.28 where $\mathbf{G}$ and $\mathbf{\Omega}$ are constructed as follows:

$$X' \approx AX + BY = G\Omega \quad \text{where} \quad G = \begin{bmatrix} A & B \end{bmatrix} \quad \text{and} \quad \Omega = \begin{bmatrix} X \\ Y \end{bmatrix}. \tag{2.28}$$

In order to get the best fit solution for the operator $\mathbf{G}$ which contains process dynamics $\mathbf{A}$ and control inputs $\mathbf{B}$, SVD can be utilized as before. See Equations 2.29, 2.30 and 2.31:

$$G = X'\Omega^{\dagger} \quad \Longleftrightarrow \quad \begin{bmatrix} A & B \end{bmatrix} = X' \begin{bmatrix} X \\ Y \end{bmatrix}^{\dagger}, \tag{2.29}$$

$$\Omega = U\Sigma V^* \approx \tilde{U}\tilde{\Sigma}\tilde{V}^* = \begin{bmatrix} \tilde{U}_1 \\ \tilde{U}_2 \end{bmatrix} \tilde{\Sigma}\tilde{V}^*, \tag{2.30}$$

$$G \approx \bar{G} = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}^*. \tag{2.31}$$

Note that the **truncation value** p for $\mathbf{\Omega}$ is larger than the truncation value for $\mathbf{X}$. Now that the SVD of $\mathbf{\Omega}$ is found, linear operator $\tilde{\mathbf{U}}$ can be broken into two seperate components to approximate system and control matrices $\mathbf{A}$ and $\mathbf{B}$ as follows:

$$G = \begin{bmatrix} A & B \end{bmatrix} \approx \begin{bmatrix} \bar{A} & \bar{B} \end{bmatrix} = \bar{G} = \begin{bmatrix} X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}_1^*, & X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}_2^* \end{bmatrix}, \tag{2.32}$$

where $G \in \mathbb{R}^{n\times(n+l)}, \tilde{U}_1 \in \mathbb{R}^{n\times p}, \tilde{U}_2 \in \mathbb{R}^{l\times p}$. Unlike the DMD algorithm, $\tilde{\mathbf{U}}$ cannot be used for defining the subspace states evolve. In the context of control, they define the **input space**. In order to get a reduced order model, reduced order subspace of the **output** should also be utilized so that a **reduced order** dynamics of the system and control matrix can be obtained. To find the reduced order subspace, another SVD should be performed as in Equation 2.33:

$$X' \approx \hat{U}\hat{\Sigma}\hat{V}^*, \tag{2.33}$$

where $\hat{U} \in \mathbb{R}^{n\times r}$, $\hat{\Sigma} \in \mathbb{R}^{r\times r}$, $\hat{V}^* \in \mathbb{R}^{r\times m-1}$ and the **truncation value** r for the second SVD is smaller than the first, p. The reduced order approximations of the system and control matrices can be computed as follows:

$$\tilde{A} = \hat{U}^*\bar{A}\hat{U} = \hat{U}^*X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}_1^*\hat{U}, \tag{2.34}$$

$$\tilde{B} = \hat{U}^*\bar{B} = \hat{U}^*X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}_2^*, \tag{2.35}$$

where $\tilde{A} \in \mathbb{R}^{r\times r}$ and $\tilde{B} \in \mathbb{R}^{r\times l}$. Now, the reduced order system dynamics can be defined as follows:

$$\tilde{x}_{k+1} = \tilde{A}\tilde{x}_k + \tilde{B}u_k. \tag{2.36}$$

Next thing in line is to find the dynamic modes of $\mathbf{A}$ which can be done by solving the eigenvalue decomposition $\tilde{A}W = W\Lambda$ and the dynamic modes of $\mathbf{A}$ can be obtained with a little modification to the equation before, as follows:

$$\Phi = X'\tilde{V}\tilde{\Sigma}^{-1}\tilde{U}_1^*\hat{U}W. \tag{2.37}$$

Dynamic Mode Decomposition and how it can be used in the context of control is summarized until this point. For the detailed proof and examples, see Tu et al. (2014). From now on, an extension of DMD will be utilized as a method to approximate Koopman Operator. **Extended Dynamic Mode Decomposition (EDMD)** improves the accuracy of DMD by utilizing a dictionary of observables on a finite space in which the Koopman Operator can be approximated (Li et al., 2017). The choice of dictionary determines the accuracy of EDMD and the optimal choice depends on the underlying dynamical system. Possible choices of the elements of this dictionary can be polynomials, Fourier modes, radial basis functions or spectral elements (Williams et al., 2015). The next section will investigate DMD and EDMD with a basic implementation that relies on the software package **datafold** which contains the implementation of EDMD utilized in the rest of this thesis (Lehmberg et al., 2020).

2.4.3 | DMD and EDMD on Limit Cycle

This section will summarize the example implementation of DMD and EDMD on the Limit Cycle, from **datafold** tutorials, to provide an understanding on how EDMD improves the performance over DMD (Lehmberg et al., 2020). Data for this tutorial sampled from a Hopf ODE described as in Equation 2.38:

$$\begin{aligned} \dot{y}_0 &= -y_1 + y_0(\mu - y_0^2 - y_1^2), \\ \dot{y}_1 &= y_0 + y_1(\mu - y_0^2 - y_1^2) \end{aligned} \tag{2.38}$$

with $\mu = 1$. ODE is solved with different initial conditions to sample time series data from the Hopf system. Figure 2.6 below shows how the sampled time series data looks like, which has two components for each time step, x_1 and x_2.

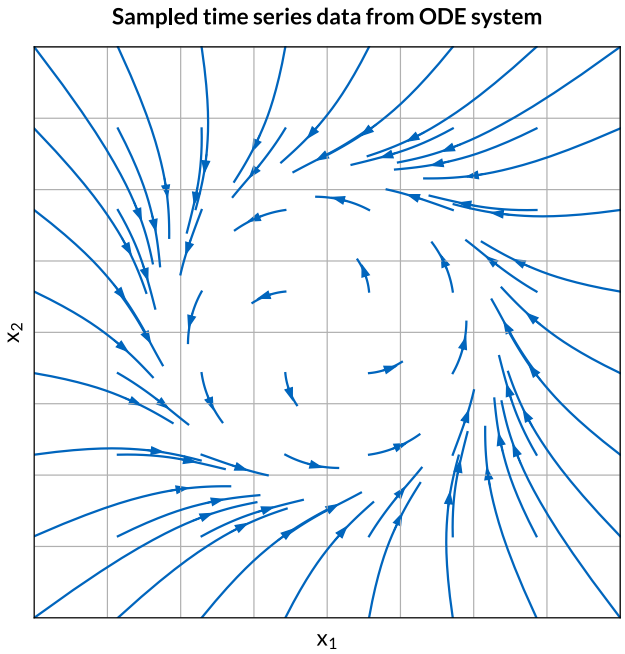

Figure 2.6: Plot of the sampled training data for DMD and EDMD.

First, a **DMD** is performed that uses identity dictionary to decompose spatio-temporal modes of the dynamical system. As can be seen in the Figure 2.7, when the observable functions only include state identities x_1 and x_2, Koopman matrix is not capable of describing the dynamics of the underlying system.

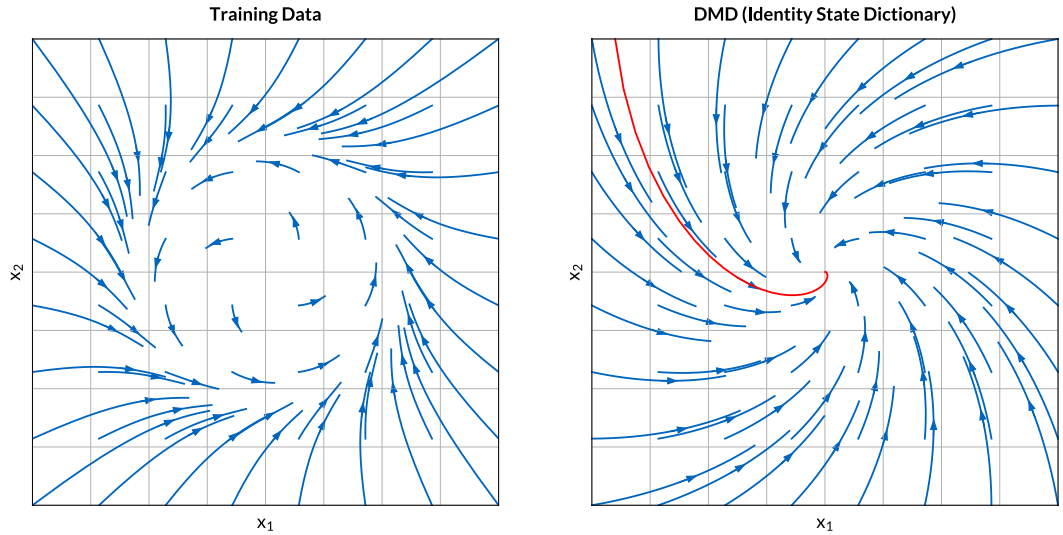

Figure 2.7: Data sampled from the DMD model with an out of sample prediction shown in red.

Now, an **EDMD** model will be trained with the same data and the extension is that first, time series data will be processed in a dictionary that is formed with polynomial features. A **degree 3** polynomial features dictionary of the states x_1 and x_2 contains the following composed spaces: x_1, x_2, x_1^2, x_1x_2, x_2^2 x_1^3, $x_1^2x_2$, $x_1x_2^2$ and x_2^3. Figure 2.8 shows the data sampled from the constructed model.

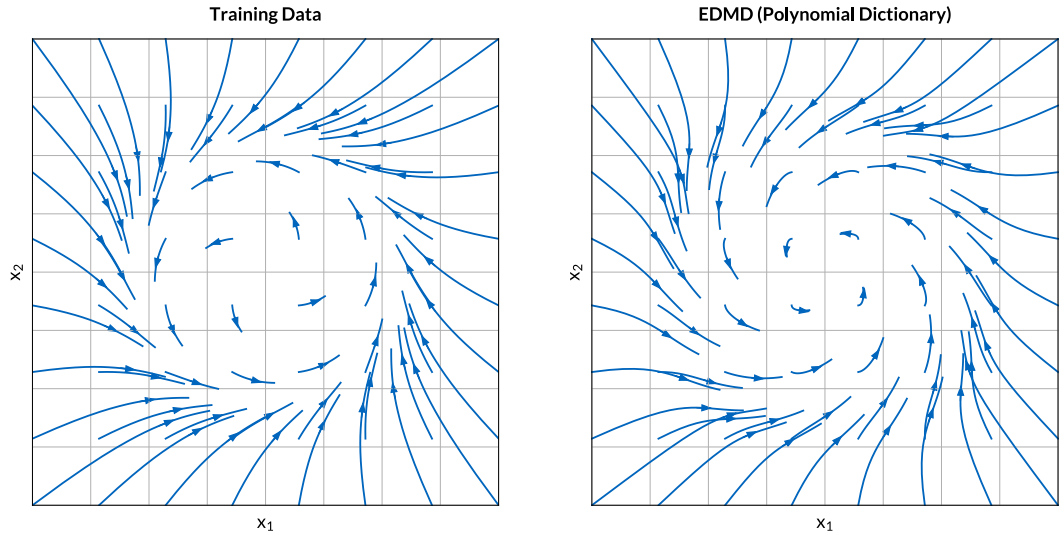

Figure 2.8: Data sampled from the EDMD model with **polynomial features dictionary**.

Reconstruction is improved, but still not ideal. The model cannot capture the limit cycle and some time series cross which is not expected. As stated previously, or in Williams et al. (2015), choice of dictionary is important to capture the true dynamics. In the last part of this section, **Radial Basis Functions (RBFs)** will be utilized to construct dictionary, see Figure 2.9.

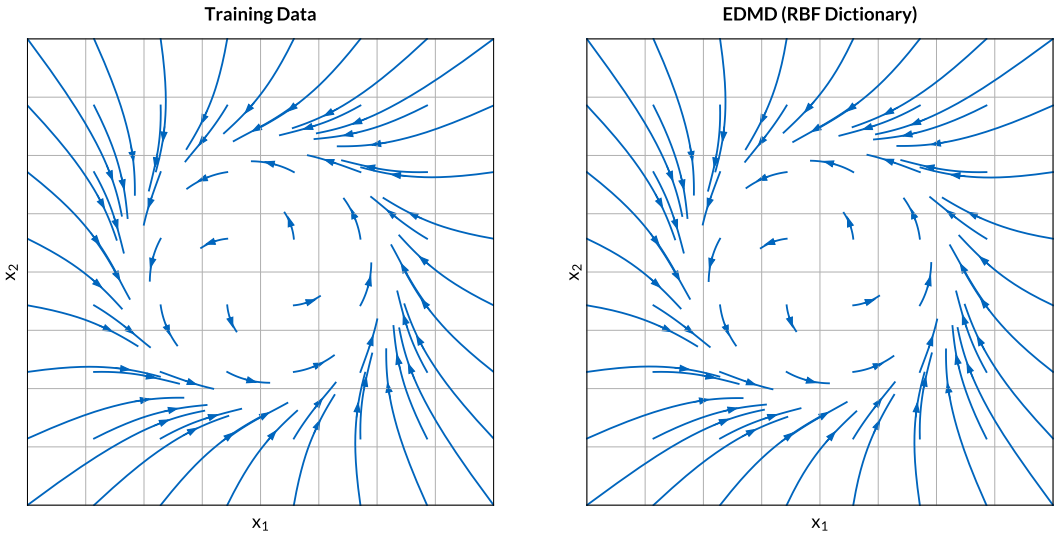

Figure 2.9: Data sampled from the EDMD model with **RBF dictionary**.

Finally, an out of sample prediction is compared with ground truth in Figure 2.10. See **EDMD on Limit Cycle** tutorial in **datafold** documentation for more (Lehmberg et al., 2020).

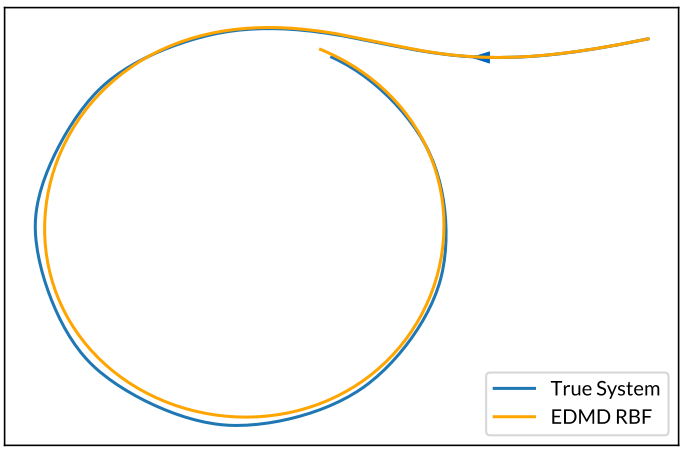

Figure 2.10: Out of sample prediction compared with the **true system**.

2.5 | Kerbal Space Program

Testing suite for this thesis is the **Kerbal Space Program (KSP)** which is a video game that let's players create their own spaceship, install modifications to it, launch or land and test the behaviour of their creation. Players need to consider many important aspects such as weight balance, maintaining a trajectory, escaping the atmosphere etc. while building their spaceship since the mechanics of the game are based on the laws of physics and it is also necessary to input proper control actions to successfully complete the intended journey. The data collected from Kerbal Space Program is collaborated between students working on consecutive projects in this area. See Figure 2.11 to see the game's relevant parts for this project.

Figure 2.11: Interface of the KSP.

1: Altitute. **2:** Vertical Speed. **3:** kRPC Interface. **4:** Speed. **5:** Throttle. **6:** Pitch, Yaw, Roll. **7:** Fuel.

Interaction with the game is performed using a mod called **Kerbal Remote Procedure Call (kRPC)**. It supports many languages, among which the **Python** libraries will be used for this research. It functions by running a server in the game to which users connect for executing remote procedures. It provides action inputs and read the state of the spaceship in this thesis.

3

Infinite Horizon Control Using Koopman Operator Surrogate Models

This thesis aims to implement Infinite Horizon Control (IHC) to reduce the computational resources required by MPC and KMPC while still providing a reliable process controller. The results of the implementation will be evaluated by performing simulations and comparing results with the MPC and KMPC implementations.

This chapter contains a description of the environment and; implementation and evaluation of Infinite Horizon Control to land a spaceship in Kerbal Space Program. In particular, Section 3.1 will introduce the development environment where the previous work and packages the project rely on will be introduced along with the simulation tools. Section 3.2 will describe the metrics that are used to evaluate performance of the controller. Section 3.3 will go over the implementation and finally, Section 3.4 will evaluate the performance of the implementation using the metrics described in Section 3.2.

3.1 | Environment

This project is a continuation of the previous work on Model Predictive Control (MPC) by **Ali Ganbarov** and **Kaan Atukalp** as well as the Koopman MPC implementation provided by **Dr. Felix Dietrich** as a code base. MPC and KMPC implementations are used for comparisons with the Infinite Horizon Control (IHC) implementation that is integrated over the existing code base. See (Ganbarov, 2020) and (Atukalp, 2021).

Kerbal Space Program (KSP) is used as the simulation tool and kRPC plugin enabled controlling the spaceships constructed in the game for testing different scenarios. The whole project is implemented in **Python**.

3.1.1 | Aircraft Dynamics

In this subsection, numerical data related to the aircraft used in the experiments throughout the project and the simulation dynamics will be discussed. Kerbal Space Program (KSP) allows users to create their own spaceships. In order to use for training and testing, a spaceship we call **'Aurora'** has been created in the game.

Aurora is a relatively simple aircraft that is composed of 9 parts. From top to bottom, it has a **Small Nose Cone**, **Mk1 Command Pod**, two **FL-T400 Fuel Tanks**, one **LV-T30 Liquid Fuel Engine** and four **LT-2 Landing Struts**. It has a height of **7.5 meters**, width and length of **4.6 meters** and it weighs **7 tons**. It is possible to give 4 different control inputs to Aurora, **thrust**, **pitch**, **yaw** and **roll**. The axes of these control inputs originate at the center of gravity of the aircraft and are perpendicular to each other except for 'thrust' which has the same axis as 'roll'. Figure 3.1 shows the directions of these control inputs.

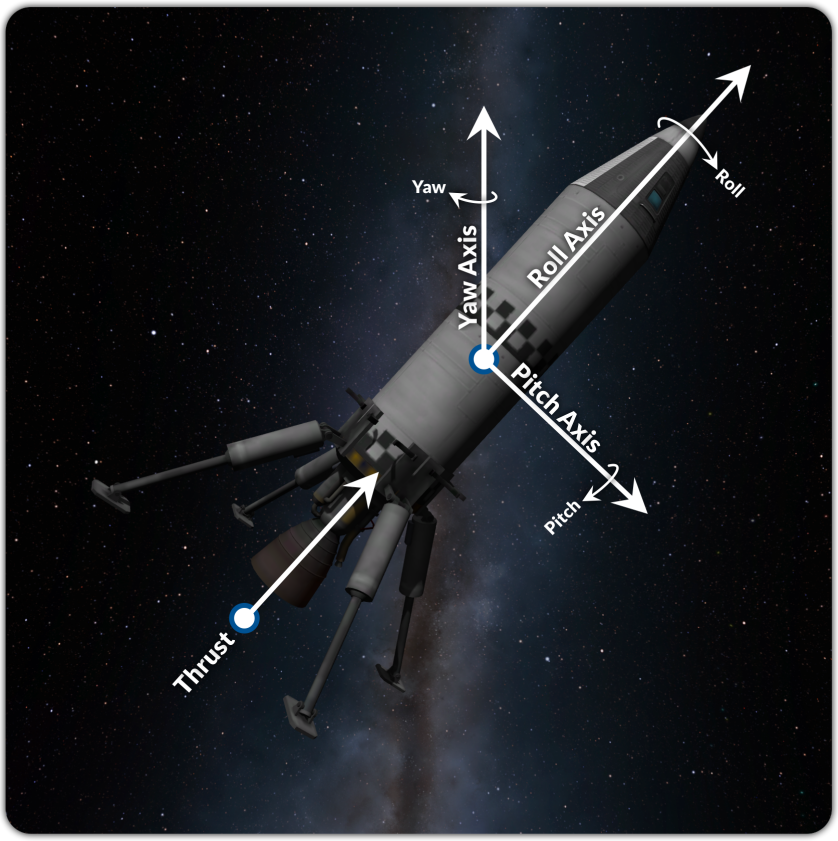

Figure 3.1: Directions of the control inputs acting on the aircraft.

Among these four control inputs, PID controllers implemented by Ali Ganbarov will be used to control pitch, yaw and roll of the aircraft. Infinite Horizon Control (IHC), MPC and Koopman MPC controllers will determine the **input throttle** to control **thrust force** acting on the aircraft to reach the target values and achieve a safe landing.

There are also four different forces acting on the aircraft; **weight**, **drag**, **lift** and **thrust**. Weight pulls the aircraft towards the center of the planet. Lift and drag are the aerodynamic forces that depend on the shape and velocity of the aircraft, and density of the atmosphere aircraft is travelling in. Lift is perpendicular to the direction of the airflow and drag acts on the opposite direction of motion. Thrust is the only force that is controllable in the aircraft and pushes the aircraft against the direction of the exhausting fuel. See Ganbarov (2020) for detailed explanations.

The API of kRPC plugin allows users to fetch a rich set of measurements from the game at any moment during the flight. Throttle, roll, pitch, yaw, altitude, direction, thrust, drag, mass, vertical velocity and mass are the measurements that can be obtained through kRPC and since the dynamics of the game follows Newtonian physics, quantities like **weight** (See Equation 3.2) and **acceleration** (See Equation 3.1) can be calculated using Newton's motion laws (Newton et al., 1848) as follows:

$$a_t = \frac{T + sign(v_t) \times D - W}{m}, \tag{3.1}$$

$$W = G\frac{m \times M}{(R + y)^2}, \tag{3.2}$$

where a_t is the acceleration, T is the thrust, v_t is the velocity, D is the drag, m is the mass, y is the altitude and W is weight of the aircraft; M is the mass and R is the radius of the planet and finally, G is the gravitational constant. The planet that the spaceship lifts of and lands is called **Kerbin** which has a mass of 5.29×10^{22} kilograms and a radius of **600** kilometers. It is important here to note that in order to obtain **vertical** component of the acceleration, projection to the desired orbit needs to be performed or the desired components of the forces need to be used in calculations. Apart from utilizing the forces acting on the aircraft, acceleration, or its vertical component through projection can also be calculated as follows:

$$a_t = \frac{v_t - v_{t-\Delta t}}{\Delta t}, \tag{3.3}$$

where Δt is the time between velocity measurements. Equation 3.3 shows another formulation used to obtain **vertical acceleration** throughout this project.

3.1.2 | Existing Setup

The existing implementation already connects the game through kRPC. It allows interfering the game at any point through running the `run_landing()` method of the `Controller` object. It also logs the measurements throughout the journey as a **CSV** file that can be processed later for further analysis.

The `main.py` script is the entrypoint for the application which takes an argument to define which mode the script should be running. It can be `MPC`, `KMPC` or `IHC` for which the last one is implemented to serve the purpose of this project. It also accepts an optional argument to specify a save game file to connect with. If not specified, it connects to the current session and takes over the control of the aircraft.

Ali Ganbarov implemented fundamental MPC controller that works with two different drag models as well as a PID controller that takes care of orienting the aircraft. **Kaan Atukalp** trained various predictors to generate control inputs for the specified horizons and created another controller called `ControllerML` that works with trained models. See Atukalp (2021) for details. **Dr. Felix Dietrich** provided pre-trained Koopman system and control matrices and implementation of the Koopman MPC.

There are four control inputs that can be given to the aircraft, as denoted before. Pitch, roll and yaw are controlled through PID controllers implemented and tuned by **Ali Ganbarov**. Check Ganbarov (2020) for the details. What this project focuses on is determining accurate values for **throttle** input to achieve convenient **thrust** values that take the aircraft to the desired targets and land it safely.

The existing code base needed some restructuring to make it more understandable and easy to work with. In order to achieve this, separate log handlers are merged into a single class. Controllers, handlers, wrappers, predictors, models, settings and utility functions are moved into different files and class structure is updated. Predictors and data gathering functions implemented by **Kaan Atukalp** merged under a class called `Estimators`. Notebooks for data analysis and various data collected by former programmers are restructured to ensure each sell produces proper outputs with a proper order without an issue. Other notebooks that contributed to Theoretical Background are merged under a single notebook. How to work with the existing code base and notes about plugins, packages and various keynotes to deal with unexpected issues during installations are documented in the `README.md` file to speed up the on-boarding process of the next person working on this project.

3.2 | Metrics

In this chapter, the statistics used to evaluate the performance of Infinite Horizon Control will be discussed. The aim of this thesis is to provide a method to control an aircraft that is both resource efficient and reliable. Firstly, the computational resources used throughout the flight scenarios will be investigated. As the measurements are prune to bias, it is important to evaluate the results relative to each other instead of looking at them individually. This approach will provide insights about the improvements of the new implementation over the older ones.

The most trivial statistics used are the mean μ and the standard deviation σ. Mean value gives insight about the overall magnitude of the measurements whereas standard deviation shows how much they deviate from the mean. The mean value can be obtained as follows:

$$\mu = \frac{\sum x_i}{N}, \tag{3.4}$$

where N is the number of measurements. Since the data collected in this thesis are discrete, we only need the formula for the discrete-time cases. Together with the standard deviation, which can be calculated as follows:

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}, \tag{3.5}$$

they provide extremely valuable insights about the investigated data sets. In the upcoming chapter, mean and standard deviation will not only give broad insights about the scenarios, they will also serve as an anchor point for other statistics computed. For instance, as mentioned before, computing relative performance increase of a method over the other is an essential statistic that depicts how successful the new method is. The relative performance increase (in percentages) of **method 1** (m_1) over the **method 2** (m_2) can be calculated as follows:

$$\text{Performance Increase } (\%) = \frac{-(p_{m_1} - p_{m_2})}{p_{m_2}} \times 100, \tag{3.6}$$

where p_{m_1} and p_{m_2} denotes the performances of **methods 1** and **2** respectively. It is worth noting that performance here can stand for any two measurements that one wants to compute how much of a performance increase one provides over the other. It is also important to note that Equation 3.6 considers **smaller values to be better**. One can say that the performance of the first method (m_1) is better as much as the percentage value calculated using the Equation 3.7, from the performance of the second method (m_2).

Another measurement used throughout this project shows how much one value is larger than the other, or in other terms, how much of an increase one is over the other. Computing this statistic is also trivial and similar to Equation 3.6 which is as follows:

$$\text{Increase } (\%) = \frac{v_2 - v_1}{v_1} \times 100, \tag{3.7}$$

where v_1 denotes the first measurement and v_2 denotes the second measurement. One can say that the measured statistics is increased as the calculated percentage using the Equation 3.7 when the new value of the measurement is v_2 and the old value is v_1.

Even though it is trivial, also worth mentioning that metrics computed will also be compared in terms of factor of each other so that another insight about the improvement can be obtained.

The statistics introduced so far will be used to infer that the new implementation is more resource efficient than the older ones. The next thing to measure is how reliable the new implementation is. In order to achieve this, trajectories followed by the aircraft on each flight scenario as well as the vertical velocities will be compared with the reference which includes a target velocity and a target altitude for each time step.

However, it is challenging to obtain an unbiased score for how well the path is followed. Since the target states are slided based on the current state of the aircraft, whenever target state changes, there emerges a huge gap between the target values and the current values. Controller needs to adjust throttle to reach the new target state first, then it needs to provide appropriate throttle values to follow this targets.

In order to acquire statistics that give more reliable scores for reference following performance, time series data will be split into two sets of fragments. For vertical velocities, **first set** will include the parts where controller is trying to get the target velocity value and as soon as it reaches a predetermined proximity of the target, fragment will be extracted; the next fragment starts at this point until the the point where the target state changes, which will construct the next fragment that will be sent to the **second set**. We will call the first set as the **Approach Set**, and the second set as the **Proximity Set**.

The **Approach Set** will be evaluated in terms of how fast the controller reaches the target value. The time it takes the controller to reach $\pm 10\%$ of the target value will be computed for each approach fragment of each method. Since the target values have the same intervals, the fragments of each method's time series data that correspond to the same target state, will be normalized as follows:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{3.8}$$

where x is an element of the set that contains fragments of data corresponding to the time intervals that has the same target state. After a score is calculated for each **approach fragment**, an average value of the approach scores will be calculated for each method, by computing the mean value of approach scores for that method; and will be stored as the **Approach Score**, one for each method.

As for the **Proximity Sets**; what matters is how good the controller can keep the aircraft stable at the target state. In order to get a well defining value for this, Root Mean Squared Error (RMSE) will be computed for each **proximity fragment** between the real trajectory and the target trajectory of the time series data. As the diversion from the reference gets larger, penalty value of the RMSE gets larger as well. It will be an appropriate statistic to get an insight on how good the controller follows reference **after** the reference value is reached. The score obtained for each fragment will be normalized among methods and will be averaged within methods to get a **Proximity Score**, again, one for each method.

Approach scores and proximity scores will be combined to get an insight on how good the reference is reached and how stable it is followed. If the controller applies too much throttle to get to the target state quickly, it may reach a velocity that is hard for it to control. The velocity overshot created in this scenario will increase the RMSE error of the upcoming proximity fragment and since it is proportional to the square of the value, larger deviations will result in larger penalties. In order for a method to get a good **overall score** which will be the **average of approach and proximity scores**, controller of that method both have to provide a good approach and a stable trail of the target.

Another metric calculated that gives an insight about how good the trajectory is compared to the reference is a simplified version of the aforementioned split approach. This time, the time series data will be split only according to the target state intervals and RMSE will be calculated for each method, for each interval. Data will be normalized again among the methods and averaged within the methods, as before.

For each score computed, lower means there is less error which in this scale, refers to a better model. After the final scores are computed, relative performance of the models will be evaluated, again, in terms of factors of improvement. Figure 3.2 illustrates the approaches to achieve both scores. After the application of this process on each segment, **approach** and **proximity** scores are obtained for each model.

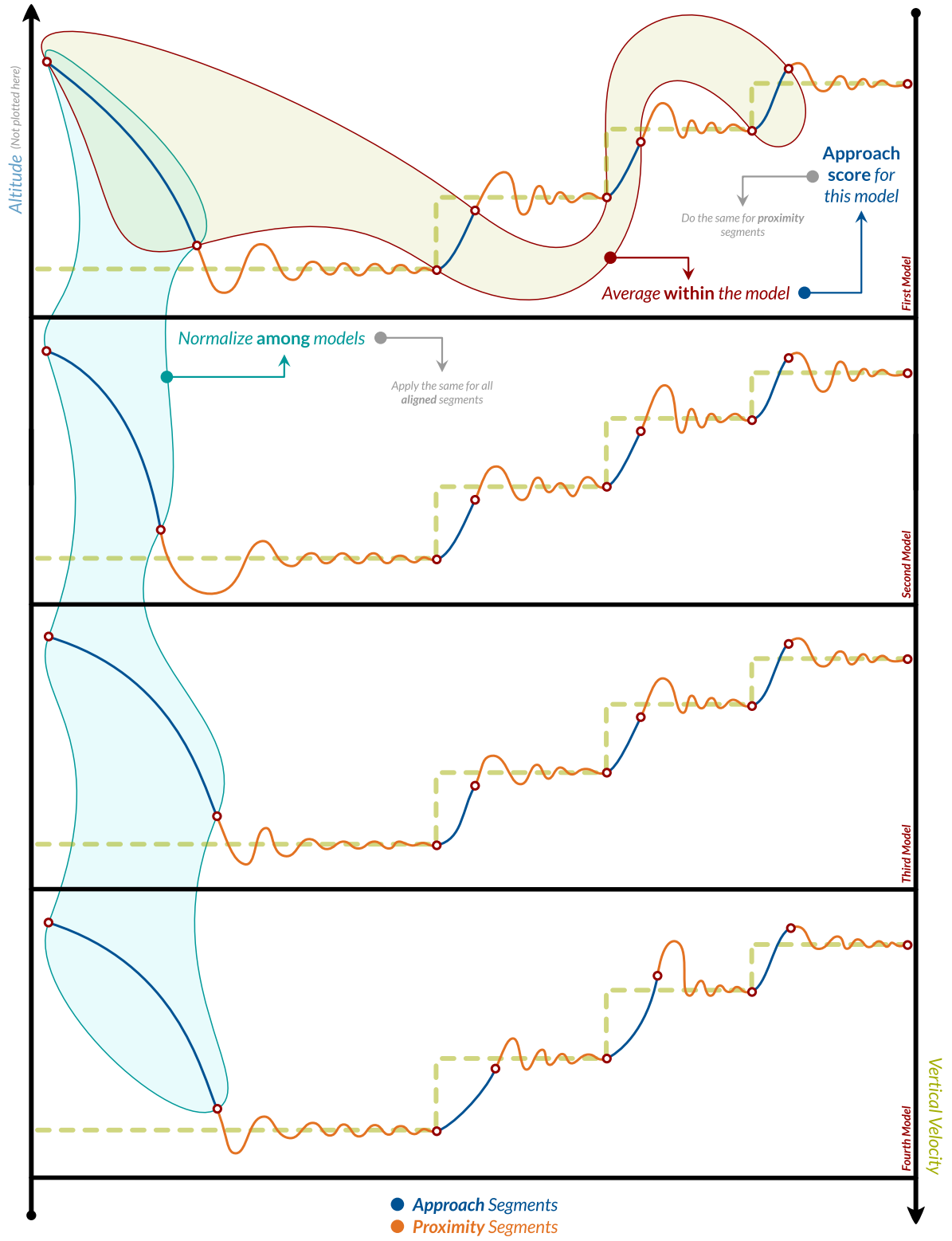

Figure 3.2: Illustration of the algorithm computing **approach** and **proximity** scores.

3.3 | Implementation

This section will explain how the system and control matrices are obtained using Extended Dynamic Mode Decomposition (EDMD) and how the linear system is solved using Linear Quadratic Regulator (LQR) to obtain control inputs at each step. It will also explain how crucial data that are going to be used to evaluate performance of the implementation using metrics explained in Metrics section, are gathered.

3.3.1 | Data Preprocessing

The first phase of achieving the intended results is to gather and process the data. Thankfully, **Ali Ganbarov** and **Kaan Atukalp** cooperated to provide a collaboration database that contains many flight data from various scenarios which can be used to obtain system and control matrices. Since EDMD is a purely data driven method, data used during training is very important to achieve proper results. It is necessary to provide a rich dataset to avoid memorizing. We have performed our tests on the system and control matrices that are provided by **Dr. Felix Dietrich**, but also we have performed EDMD over a larger dataset to get a system more robust over different scenarios. We have compared the results of both approaches in the Evaluation section.

There are two different types of files available in the collaboration database. One type contains data in the form of XLS files and the other contains data in the form of CSV files. Even though file types are different, they both contain all the essential information during the flight scenarios. Existing implementation logs flight records in the form of CSV files and for the training, XLS files collected by **Kaan Atukalp** are used.

To begin with, time stamps in the data are evenly spaced using interpolation (See Atukalp (2021)) by calling `format()` method of the `DataFormatter` class. Time delta is defined as **0.25 seconds** and then, offset is removed from the column containing time stamps to make it start from zero. Afterwards, vertical acceleration values are computed by calling `get_vertical_acceleration()` method which first, computes weight of the aircraft using Equation 3.2 and uses that value to compute vertical acceleration using Equation 3.1. Next step is to construct `TSCDataFrame` object from the processed data and use `normalize_ts()` method provided by **Dr. Felix Dietrich** to normalize it using following equation:

$$x_{normalized} = \frac{x - x_{min}}{x_{max} \times 2} - 1 \tag{3.9}$$

which is the final component of the method `construct_tsc()` that processes raw data and produces `TSCDataFrame` containing formatted training data for EDMD.

3.3.2 | Obtaining System & Control Matrices

The next phase for implementation is to obtain system and control matrices. There already exist a system and control matrix pair with its normalizer to use in computing optimal control values. Even though it works great for the scenarios where aircraft starts its motion towards the planet, when the script is run during an upward trajectory, it fails to give proper inputs. In order to deal with this issue, data from all the scenarios are used to train EDMD.

First off, a dictionary of observables need to be selected. Polynomial features with a degree of **2** is chosen with an included bias. `EDMDControl` method from **datafold** is used to get the system and control matrices. As denoted before, there will be only one control input determined by the controller implemented here which is the **thrust**. As detailed in the Theoretical Background chapter, `EDMDControl` method stacks the control and state columns to obtain a solution to the problem. It expects the user to specify control and state columns of the data given. In this setting, state columns are **altitude**, **vertical velocity**, **mass**, **vertical component of the drag** and **vertical acceleration** which will be used to create polynomial features dictionary for EDMD whereas there is only one column for control which is **thrust**. After fitting the data using `EDMDControl`, system and control matrices are saved into `sys_matrix` and `control_matrix` fields of the object returned by the `fit()` method. They are dumped into a JSON file that is later processed by the remaining components of the implementation.

3.3.3 | Infinite Horizon Control

The final section of this chapter will focus on the implementation of Infinite Horizon Control using the system and control matrices obtained in the previous section. In order to get the optimal control inputs for the aircraft, Linear Quadratic Regulator (LQR) is used which is implemented as a method under the `IHC` object that is created as a class that fits into the existing implementation structure.

The optimal control input is calculated by the `get_val()` method of the `IHC` class. The method takes the `current_state` and the `desired_state` as inputs as well as the `current_time` which is used together with the class field `last_time` to determine time delta. The velocity of the aircraft at the previous time step is used together with the calculated time delta to obtain **vertical acceleration** as described in the Equation 3.3. Lifting function of the class is used to lift the `current_state` with features **altitude**, **velocity**, **mass**, **drag** and **vertical acceleration** into a higher dimension, as done in EDMD using polynomial features with **degree 2**. The same is applied to the `target_state`, then the obtained lifted states are stored into variables `actual_state` and `desired_state` respectively.

The next step is to compute `state_error` by subtracting `desired_state` from the `actual_state`. Then, the optimal control input can be calculated by multiplying **state feedback gains** matrix $\mathcal{K}$ with the `state_error`. After the optimal control input is obtained, current time is saved into the class variable `last_time` and the current velocity is saved into the variable `last_vel` to be used for calculating vertical acceleration in the next time step.

The heart of the LQR is to obtain the feedback gains matrix $\mathcal{K}$. As described in the Section 2.3 of Chapter 2, first step of achieving this is finding a solution $\mathcal{S}$ to Discrete Time Algebraic Riccati Equation (DARE) which is a non-linear equation that can be solved using dynamic programming. DARE generally has more than one solution but the aim is to get one stabilizing solution if exists. I order to obtain a stabilizing solution to DARE, `solve_discrete_are()` method of `scipy` is used for which 4 matrices are passed: the state transition matrix **A**, control matrix **B**, state cost matrix **Q** and finally, control cost matrix **R**. Matrices **Q** and **R** can be used to set different penalty values for each of the lifted states and the control inputs. In the scope of this thesis, all the lifted states treated equally and matrix **Q** is set to be a 21×21 diagonal matrix with ones on the diagonal. The control cost matrix **R** has one element which determines how much **thruster effort** is penalized. Unless specified otherwise, it is set to be **0.01** throughout the experiments.

After method `solve_discrete_are(A, B, Q, R)` returns a solution to DARE, the stabilizing solution is used to obtain **state feedback gains** matrix $\mathcal{K}$ as follows:

$$\mathcal{K} = (B^T \mathcal{S} B + R)^{-1} B^T \mathcal{S} A \tag{3.10}$$

and the **eigenvalues of the closed loop system** $\mathcal{E}$ can be calculated with `scipy.linalg.eigvals()` function by passing the matrix $A - B\mathcal{K}$ as argument. This concludes obtaining the **optimal gain set** $\mathcal{K}$, $\mathcal{S}$, $\mathcal{E}$. The final thing left to do is to compute **optimal control input** U^* that takes the aircraft to the desired target state which can be calculated as follows:

$$\mathcal{U}^* = K \times E(s), \tag{3.11}$$

where $E(s)$ denotes the **state error** that can be obtained by subtracting `desired_state` from the `actual_state`. Solving DARE has various approaches that takes different amounts of work to obtain a stabilizing solution. The `solve_discrete_are()` function of `scipy` uses **QZ Algorithm** to obtain a solution to DARE. See Laub (1979) and Van Dooren (1981) for the details on how the algorithm is implemented and see Benner (2000) for understanding how the accuracy of QZ decomposition is improved.

At this point, with the optimal control inputs calculated, aircraft is successfully landing from reasonable initial conditions such as altitude between **1000 meters** and **20000 meters** and vertical velocity less than **100 meters per second**. It can be further discussed what the reasonable initial conditions are, but except for scenarios in which the crash is inevitable, IHC does a pretty good job landing the aircraft. Various scenarios will be investigated in Section 3.4.

At each time step, optimal control inputs can be calculated with the implementation so far to reach the desired altitude and vertical velocity. What matters next is how the reference values at each state are calculated. Theoretically, Infinite Horizon Control can obtain all control inputs that eventually achieve the target at once if the system and control matrices are perfectly describing the dynamics of the underlying system. However, approximations made to acquire these matrices makes it unrealistic to expect this solution to work since the smallest error can cause catastrophic results. Different scenarios showing that this is, indeed, the case are discussed in the next section. Instead of planning the whole flight at once, current state is updated at each time with the real values and one optimal control input set is generated at each time step using fixed **state feedback gains** matrix $\mathcal{K}$ that only needs to be calculated once for the whole flight. This approach also requires intermediate targets, as existing MPC and Koopman MPC implementations did. The target values for the altitude can be computed using the following equation:

$$y_{target} = \begin{cases} y - 500, & \text{if } y > 1000 \\ 0, & \text{else} \end{cases} \tag{3.12}$$

for which the values are in **meters** and the target values for the vertical velocity can be computed based on the current altitude, as follows:

$$v_{target} = \begin{cases} -150, & \text{if } y > 1000 \\ -50, & \text{if } 1000 \geq y > 150 \\ -20, & \text{if } 150 \geq y > 20 \\ -3, & \text{if } 20 \geq y > 5 \\ 0, & \text{else} \end{cases} \tag{3.13}$$

for which the units are **meters/seconds**. Until this point, implementation lands the aircraft successfully but since one of the primary objectives of this thesis is to reduce the resources used to obtain control input while keeping a reliable controller, it is essential to correctly measure how much resources are used to produce each control input.

For achieving this objective, some standard libraries of Python are used. First off, `process_time()` method of the `time` library gives an insight of how much CPU time elapsed to obtain a single optimal control input. This method only includes the time processor was actually busy. It shows how many clock cycles passed between two points.Then, `timeit()` computes average time it takes for a method to run by actually running it a number of times and calculating the average. This function disables the garbage collector and since the processor will be occupied mostly computing the desired function over and over again, it also minimizes the influence of other tasks. Though, it is not reasonable to run the same function over and over again at each time step, so; this function is only used for several scenarios to give a broad unbiased oversight of the resource usage. The final library is called `tracemalloc` which gives many insights on computational resources used **over time**. However, it injects itself deep into code and adds a considerable overhead to computations which effect the results gathered. For that, as in the case with `timeit()`, it will only be used on several scenarios to gain insights on memory usage over time for different control methods. Algorithm 1 shows how resource calculation is integrated to the main loop. Note that logs are accumulated in memory, but output is generated only after the flight is over.

Algorithm 1: Gathers the computational resources used by the controllers.

Data:
Target handler: *target_handler*,
Termination handler: *termination_handler*,
Log handler: *log_handler*,
Controller: *controller*,
Vessel status: *status*
Result:
Log files containing information about flight and resources used.
while *True* **do**
  target ← *target_handler(status)*
  if *termination_handler(status, target)* **then**
    break
  end
  controller ← *controller.update(status)*
  cpu_time_start ← *time.process_time()*
  tracemalloc.start()
  new_inputs ← *controller.get_new_inputs(status, target)*
  cpu_time_elapsed ← *(time.process_time() - cpu_time_start)*
  average_memory, peak_memory ← *tracemalloc.get_traced_memory()tracemalloc.stop()*
  timeit(stmt='get_new_inputs(status, target)' number=n)
  log_handler.write(status, new_inputs, cpu_time_elapsed, average_memory, peak_memory)
end
log_handler.save()

In the final subsection of Evaluation, there arises a need to fit a polynomial that takes **time delta** and **throttle** as inputs and generates a prediction for how much will the **mass** of the fuel that will be consumed during **time delta** when **throttle** value is given, will be. This model will be used to generate control inputs and simulate flights by only using the initial conditions. In particular, the aim is to find the coefficients of the following equation:

$$m = f(t,u) = a_0 + a_1 t + a_2 u + a_3 t^2 + a_4 t^2 u + a_5 t^2 u^2 + a_6 u^2 + a_7 t u^2 + a_8 t u, \tag{3.14}$$

where m stands for **mass**, t stands for **time delta** and u stands for **throttle**. In order to obtain the set of coefficients that best describes the relation, **least squares solution** to the following equation is obtained:

$$Ax = B, \tag{3.15}$$

where the vector x contains the coefficients a_0 ... a_8 that approximately solves the Equation 3.15. Algorithm 2 shows how the model is fitted to the data to get the coefficients of the polynomial. Note that the first row of matrix A contains a row vector of ones with the same dimension as the vectors X and Y.

Algorithm 2: Obtains coefficients of the polynomial model of change in mass.

Data: Time Deltas: t, Throttles: u, Masses: m
Result: Solution to least squares problem $Ax = B$: x
$T \leftarrow [\, t; \ldots ; t \,]$
$U \leftarrow [\, u^T, \ldots u^T \,]$
$T, U \leftarrow T.flatten(),\ U.flatten()$
$M \leftarrow m^T m$

$$A \leftarrow \begin{bmatrix} 1 \\ T^T \\ U^T \\ (T \odot T)^T \\ (T \odot T \odot U)^T \\ (T \odot T \cdot U \odot U)^T \\ (U \odot U)^T \\ (T \odot U \odot U)^T \\ (T \odot U)^T \end{bmatrix}^T$$

$B \leftarrow M.flatten()$
solve_least_squares(A, B)

3.4 | Evaluation

In the last section of the main chapter of this thesis, using the metrics described in Chapter 2, the performance of the Infinite Horizon Control implementation will be evaluated. In particular, first subsection will compare the computational resources used by Infinite Horizon Control (IHC), MPC with one dimensional drag model, MPC with two dimensional drag model and Koopman MPC. There will be three different scenarios: landing from **3000 meters**, landing from **5000 meters** and landing from **10000 meters**. The second subsection will compare the trajectories and vertical velocities through time for the aforementioned four different methods and give statistics about how well they performed. In the final subsection, an attempt to use approximated system and control matrices to generate a complete flight plan from the initial conditions will be evaluated and discussed.

3.4.1 | Computational Resources

In this subsection, computational resources used by different implementations will be evaluated and discussed. First thing to denote here is that measuring resources used by a piece of code is a process that is vulnerable to bias. There are certain measures taken to reduce bias as much as possible, but what matters most is that relatively, analysis performed here gives valuable insights about computational efficiency of different methods.

Firstly, **CPU Time** is measured in seconds and the method used here excludes times that the processor was idle. Statistics collected are converted into milliseconds and plotted for each method in a single chart, shown at the top for each scenario. Next thing to measure was memory usage. As explained before, `tracemalloc` module provides valuable insights but adds an overhead to the process. Average and peak memory usages for a single optimal control input to be generated are measured for each method and plotted in the middle and bottom charts, respectively. At last, CPU Time statistics were measured once again for the same scenarios but without `tracemalloc` in order to avoid overhead and see how much it affects the overall performance.

As can be clearly seen in the Figures 3.3, 3.4 and 3.5, Infinite Horizon Control approach uses a lot less resources than the other approaches to compute control inputs. In terms of **CPU Time**, **Peak** and **Average Memory** usages, Infinite Horizon Control has a clear advantage over the other methods. In the Table 3.1, resource usage by different methods is summarized. Mean and standard deviation values gives insights about the methods' performance. The ways these methods are implemented have impacts on resource usage patterns. Take a look at the relative trajectory, velocity and target plots of the same scenarios in the Section 3.4.2 simultaneously for better understanding of the behaviours.

As an elaboration of how the implementation changes resource usage patters, consider MPC implementations versus Koopman MPC and IHC. MPC methods have larger standard deviations which is because when the aircraft gets closer to target, they need to act more to stay in proximity of the target, whereas other methods have same dimensional matrix computations at each step which makes CPU Time for Koopman MPC and Infinite Horizon Control more stable over time compared to MPC.

Table 3.1: **Mean** and **Standard Deviation** values of different resources for different methods.

Method	CPU (μ)	CPU (μ)	CPU (σ)	CPU (σ)	Memory (μ)	Memory (σ)
MPC	34.17	16.26	20.52	10.97	6.38	**1.95**
	43.7	21.09	16.72	8.84	6.89	2.43
	42.61	20.36	18.95	9.61	6.76	2.23
Average	**40.16**	**19.24**	**18.73**	**9.81**	**6.68**	**2.20**
MPC *(2D Drag)*	42.79	17.6	25.62	11.42	6.69	2.33
	53.91	22.37	20.67	9.62	6.67	2.31
	52.95	22.12	24.71	10.43	6.85	2.33
Average	**49.88**	**20.70**	**23.67**	**10.49**	**6.74**	**2.32**
Koopman MPC	20.75	10.56	2.08	1.44	8.89	2.29
	20.71	10.58	1.66	1.27	8.69	2.26
	20.36	10.47	1.98	1.3	8.88	2.66
Average	**20.61**	**10.54**	**1.91**	**1.34**	**8.82**	**2.40**
IHC	**1.17**	**0.71**	**0.48**	**0.22**	**1.32**	2.00
	1.23	**0.7**	**0.53**	**0.21**	**1.56**	**2.25**
	1.21	**0.71**	**0.51**	**0.23**	**1.45**	**2.10**
Average	**1.20**	**0.71**	**0.51**	**0.22**	**1.44**	**2.12**

Unbiased (without the `tracemalloc` module) measurements are shown in **orange**, **best** measurements are in **bold**. **CPU Time** values are in **milliseconds** and **Memory Usage** values are in **kilobytes**. From top to bottom of each row: **3000**, **5000** and **10000** meters landings.

At this point, it is also important to discuss the bias introduced during measurement processes. The CPU Time values here measure the time it takes the CPU to compute a single optimal control input, but it is worth denoting that there are other processes also running in the background during measurement. The tests are repeated by running the minimum background processes during measurements, but it still cannot capture the exact resources **only** used by the intended code piece. For that, it is important to evaluate these values relative to each other instead of evaluating them individually. In the Table 3.2, it can be seen how much Infinite Horizon Control increased relative performance.

Table 3.2: **Relative** performance increase of IHC compared to the other methods.

Method	CPU (μ)		CPU (μ)		Memory (μ)	
	(%)	($*$)	(%)	($*$)	(%)	($*$)
MPC	97.00	33.37	**96.33**	**27.22**	78.38	4.63
MPC *(2D Drag)*	97.59	41.45	**96.59**	**29.29**	78.57	4.67
Koopman MPC	94.16	17.12	**93.29**	**14.91**	83.64	6.11

Unbiased measurements are shown in **bold**. First values (%) depicts performance increases as percentages, whereas the second ones ($*$) as factors.

The next important bias to elaborate occurs during **peak** and **average** memory usage measurements. As denoted before, in order to capture memory usage, `tracemalloc` module is used which injects itself into the code to get an accurate measure, however it also increases the runtime of the code. **Peak** and **average** memory values correctly capture how much the targeted code piece uses these resources but when memory measurements are active, CPU Time measurements are not reliable. Table 3.3 summarizes and Figure 3.6 illustrates the bias introduced. Also denote that as the resources used increase, the amount of bias introduces increases as well which is the expected behaviour.

Table 3.3: **Increase (%)** in CPU time due to bias.

Method	Increase in **CPU Time** due to bias (%)		
	$3K$	$5K$	$10K$
MPC	110.15	107.21	109.28
MPC *(2D Drag)*	143.13	140.99	139.38
Koopman MPC	96.50	95.75	94.46
IHC	64.79	75.71	70.42

Bias refers to the measurements taken while `tracemalloc` module is active. Values refer to **3000**, **5000** and **10000** meters landings.

As a closing remark of this subsection, the IHC implementation presented in this thesis greatly reduces resource usage and may be a viable option when the resources are scarce. The next thing that determines whether this implementation was a success or not is that whether the new resource efficient method is reliable enough or not. It will be elaborated in the next subsection, Trajectory Comparisons.

Figure 3.3: Resources used for the landing from **3000 meters** scenario.

Figure 3.4: Resources used for the landing from **5000 meters** scenario.

Figure 3.5: Resources used for the landing from **10000 meters** scenario.

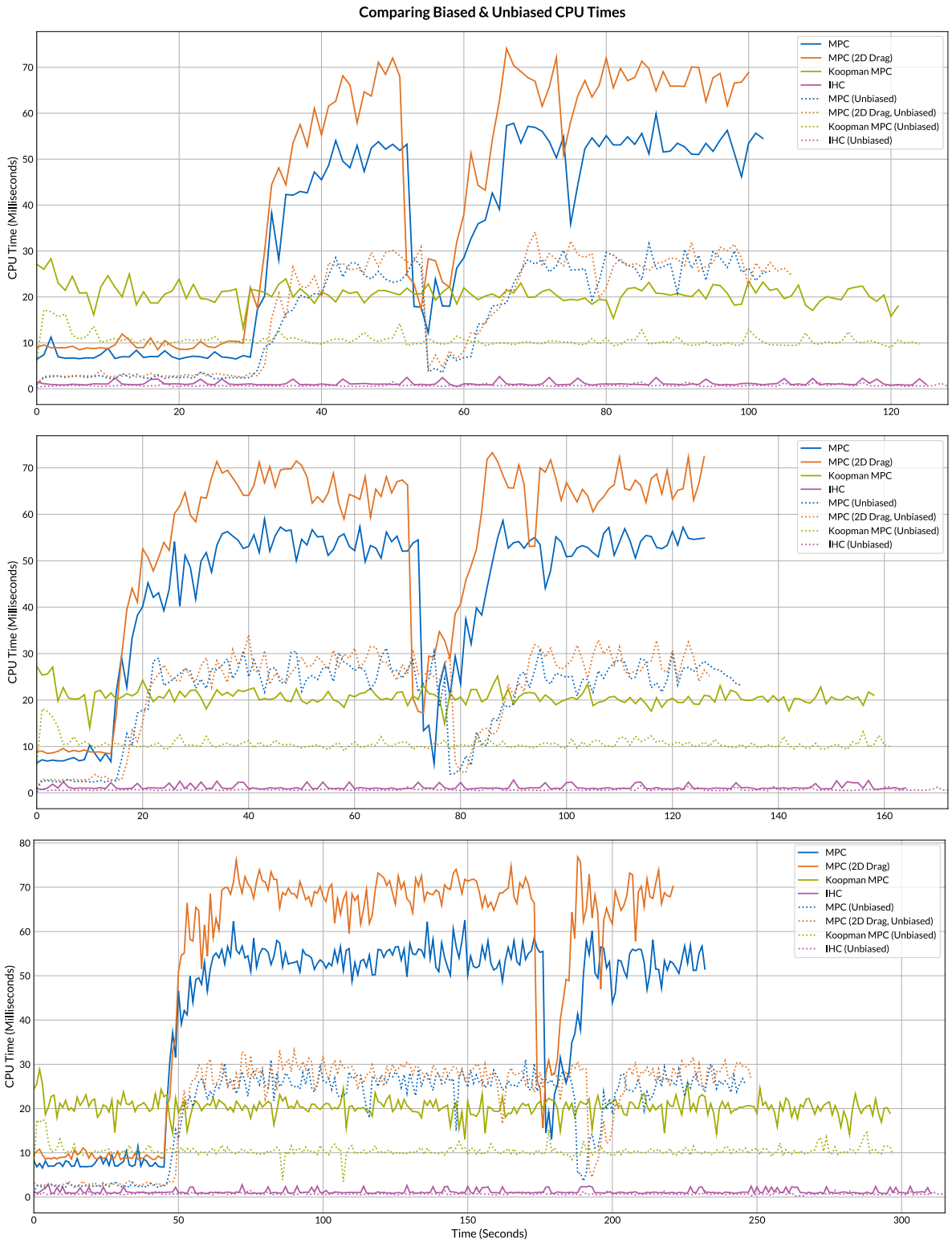

Figure 3.6: Comparison of CPU times between biased and unbiased (**dotted**) measurements. From top to bottom: **3000**, **5000** and **10000** meters landings.

3.4.2 | Trajectory Comparisons

As emphasized before, aim of this thesis is to present an implementation of Infinite Horizon Control that is both resource efficient and reliable. In the previous subsection, it was clearly shown that implementation presented here greatly reduces resource usage, both in terms of **CPU** and **Memory**. We were able to see around 95% reduction in CPU time and around 80% reduction in the average memory usage.

In this subsection, the trajectories followed by different scenarios will be investigated to determine how reliable each method is. In particular, **Altitude** and **Vertical Velocity** values will be plotted together with their target values to give an insight on how quickly each method reaches the reference and how good they can stay on it. There are two different approach used to evaluate performance of the controllers. These approaches were introduces in detail before, in the Metrics section. The first one splits each flight data into **approach** and **proximity** fragments and evaluates corresponding fragments together to get an estimation of the performance.

Unfortunately, flight data during the flights of MPC methods had hard time to reach the target altitude which made it hard to properly split the fragments. In order to deal with this problem, the first method that splits fragments into two sets is only used to get **approach** and **proximity scores** of Infinite Horizon Control and Koopman MPC.

Table 3.4: Flight Times for each model's fragments.

Method	1^{st} *Fragment* $(150m/s)$	2^{nd} *Fragment* $(50m/s)$	3^{rd} *Fragment* $(20m/s)$	4^{th} *Fragment* $(3m/s)$	**Total Flight Time** (s)
MPC	17,48	11,50	4,81	1,40	**35,19**
	27,05	12,86	5,26	1,97	**47,14**
	67,07	13,71	4,96	1,97	**87,72**
MPC *(2D Drag)*	16,29	7,02	3,79	4,30	**31,41**
	23,85	7,29	3,98	2,77	**37,88**
	58,21	6,82	4,62	5,33	**74,98**
Koopman MPC	16,98	12,71	4,84	2,13	**36,66**
	27,29	13,18	5,44	2,17	**48,08**
	68,08	14,04	5,19	1,80	**89,12**
IHC	17,48	11,50	4,81	1,40	**35,19**
	27,05	12,86	5,26	1,97	**47,14**
	67,07	13,71	4,96	1,97	**87,72**

From top to bottom of each row: **3000**, **5000** and **10000** meters landings.

For comparing all methods with each other, flight data split into fragments according to sliding targets. RMSE values are computed for each scenario, normalized among the methods and averaged within each method. There are **four** fragments for each flight scenario. Vertical speed targets for these fragments are **150m/s**, **50m/s**, **20m/s** and **3m/s** in order. Table 3.4 shows the time it takes for the aircraft to complete each fragment, for different controllers, as well as total flight times in seconds whereas Table 3.5 shows the RMSE values together with their normalized scores for each segment of the flight scenarios.

Table 3.5: **RMSE** values with **normalized** scores for each fragment of methods and scenarios.

Method	1^{st} $(150m/s)$		2^{nd} $(50m/s)$		3^{rd} $(20m/s)$		4^{th} $(3m/s)$	
	(RMSE)	(∼)	(RMSE)	(∼)	(RMSE)	(∼)	(RMSE)	(∼)
MPC	49,61	**0,21**	74,72	**0,98**	17,58	**0,91**	4,71	**0,05**
	32,81	**0,97**	73,47	**0,99**	16,16	**1,00**	3,85	**0,00**
	55,14	**0,92**	73,97	**0,98**	15,17	**0,90**	3,81	**0,00**
MPC *(2D Drag)*	49,64	**0,21**	75,40	**1,00**	18,31	**1,00**	4,48	**0,00**
	33,25	**1,00**	74,07	**1,00**	15,82	**0,95**	4,89	**0,28**
	56,13	**1,00**	74,68	**1,00**	15,86	**1,00**	3,91	**0,03**
Koopman MPC	48,87	**0,00**	33,38	**0,00**	10,15	**0,00**	7,24	**0,55**
	17,13	**0,00**	30,76	**0,00**	9,38	**0,00**	6,96	**0,84**
	44,32	**0,00**	28,13	**0,00**	9,11	**0,00**	7,43	**1,00**
IHC	52,45	**1,00**	37,73	**0,10**	10,92	**0,09**	9,52	**1,00**
	19,93	**0,17**	32,63	**0,04**	9,56	**0,03**	7,54	**1,00**
	46,09	**0,15**	30,12	**0,04**	9,20	**0,01**	7,24	**0,95**

As can be easily seen in Figures 3.7, 3.8, 3.9 and 3.10; it is obvious that the best performing controller is the Koopman MPC. It reaches the target velocities through a smooth path while reducing over shots by adjusting the speed as it gets closer to the proximity of the target. IHC method also maintains a smooth path; however, due to the approximation and the fact that it cannot re-adjust itself at each time step, it cannot follow the target values as close as Koopman MPC method. Though, as can be seen in the plots and in the various tests we have performed, it still seems to be a reliable option. Especially when the resources are scarce for it is extremely efficient in terms of CPU time and memory usage, as explained before. Moreover, as can be seen in the plots and in Table 3.6, IHC is performing far better than both implementations of MPC for which they fail to reach the target values smoothly and keep the ship steady. Even though MPC implementations are able to land the aircraft for most scenarios, the existing implementations are not very reliable since they fail to maintain close trajectory to the reference.

Table 3.6: **Average** performances for different landing scenarios.

Height	Method			
	MPC	MPC *(2D Drag)*	Koopman MPC	IHC
3000 Meters	0,5369	0,5536	**0,1369**	0,5494
5000 Meters	0,7398	0,8085	**0,2108**	0,3109
10000 Meters	0,6997	0,7569	**0,2500**	0,2884
Average	0,6588	0,7063	**0,1992**	0,3829

Lower value refers to the better controller.

Table 3.6 is showing that as expected, the best performing operator is Koopman MPC while Infinite Horizon Control controller follows it closely. It is hard to come up with a metric that says how much reliable the new method is, but statistics we gathered here together with the scenarios observed shows that IHC can be a reliable option.

This sections last analysis will focus on IHC and Koopman MPC implementations relative performance. Existing MPC implementations has shown clearly poor performance compared to these two methods. Even though the metrics calculated before supports the claim that KMPC is a better option in terms of accuracy, it is worth digging deeper into these methods' performances. The first approach described in the Metrics section that classifies fragments into two sets and performs calculations on them will be applied here to compare two methods in depth.

It is worth noting that there are only two methods to compare at this point. So, Equation 3.8 which shows how the measurements of the fragments will be normalized among methods becomes a voting mechanism since it will only output **0** or **1**. The better method in each comparison will get a 0 whereas the other one gets 1. At the end, the votes will be summed to get metrics for comparison. It is also worth emphasizing that lower value means better performance here too, as before.

The upper part of the Table 3.7 shows which controller performed better in each segment of the trajectories. It is worth noting here that there is no need to plot the same table for the Koopman MPC since where IHC gets a **0**, KMPC gets **1**. Lower part of the table shows average **proximity** and **approach** scores of both methods for different landing scenarios as well as an overall score pair for both implementations. Finally, Table 3.8 shows raw values of error statistics for each particular flight fragment as well as their averages. Again, for both, lower is better.

Table 3.7: Analysis of trajectory fragments for all scenarios of **IHC** and **KMPC**.

Height	Infinite Horizon Control *(Fragments)*							
3000 Meters	1	**1**	0	**0**	1	**1**	1	**1**
5000 Meters	1	**1**	0	**1**	1	**0**	1	**0**
10000 Meters	1	**1**	0	**0**	1	**1**	1	**1**

Height	*IHC*		*KMPC*	
3000 Meters	0,5	**1**	0,5	**0**
5000 Meters	0,75	**0,5**	0,25	**0,5**
10000 Meters	0,5	**1**	0,5	**0**
Average	**0,58**	**0,83**	**0,42**	**0,17**

Bold or **blue** values refer to the **proximity fragments** whereas the **orange** ones and the rest refer to the **approach fragments**. For the table on the top, **zero** means current controller performed better whereas **one** means the opposite.

In the lights of the analysis performed so far, even though **IHC** did not perform as well as **KMPC**, it has shown to be a reliable alternative, especially when the resources are scarce.

Table 3.8: Calculated error metrics for all fragments of **IHC** and **KMPC**.

Height	Infinite Horizon Control *(Fragments)*							
3000 Meters	10,95	**3,68**	1,12	**0,57**	6,89	**1,88**	1,86	**1,74**
5000 Meters	6,11	**3,10**	1,38	**0,56**	5,51	**1,09**	1,11	**1,25**
10000 Meters	15,52	**2,51**	0,56	**0,84**	4,52	**1,38**	1,25	**1,16**
Average	10,86	**3,10**	1,02	**0,66**	5,64	**1,45**	1,41	**1,38**
Height	Koopman MPC *(Fragments)*							
3000 Meters	9,94	**3,28**	1,51	**1,49**	4,41	**0,90**	0,35	**0,87**
5000 Meters	5,26	**2,76**	1,48	**0,32**	2,65	**1,46**	0,71	**1,36**
10000 Meters	15,09	**2,45**	0,57	**0,86**	1,65	**0,84**	0,72	**0,74**
Average	**10,10**	**2,83**	1,19	**0,89**	2,90	**1,07**	0,59	**0,99**

Bold ones are the **RMSE** values that refer to the **proximity fragments** whereas others are the **time it takes the aircraft to get to the proximity of the target** *(in seconds)*, which refer to the **approach fragments**.

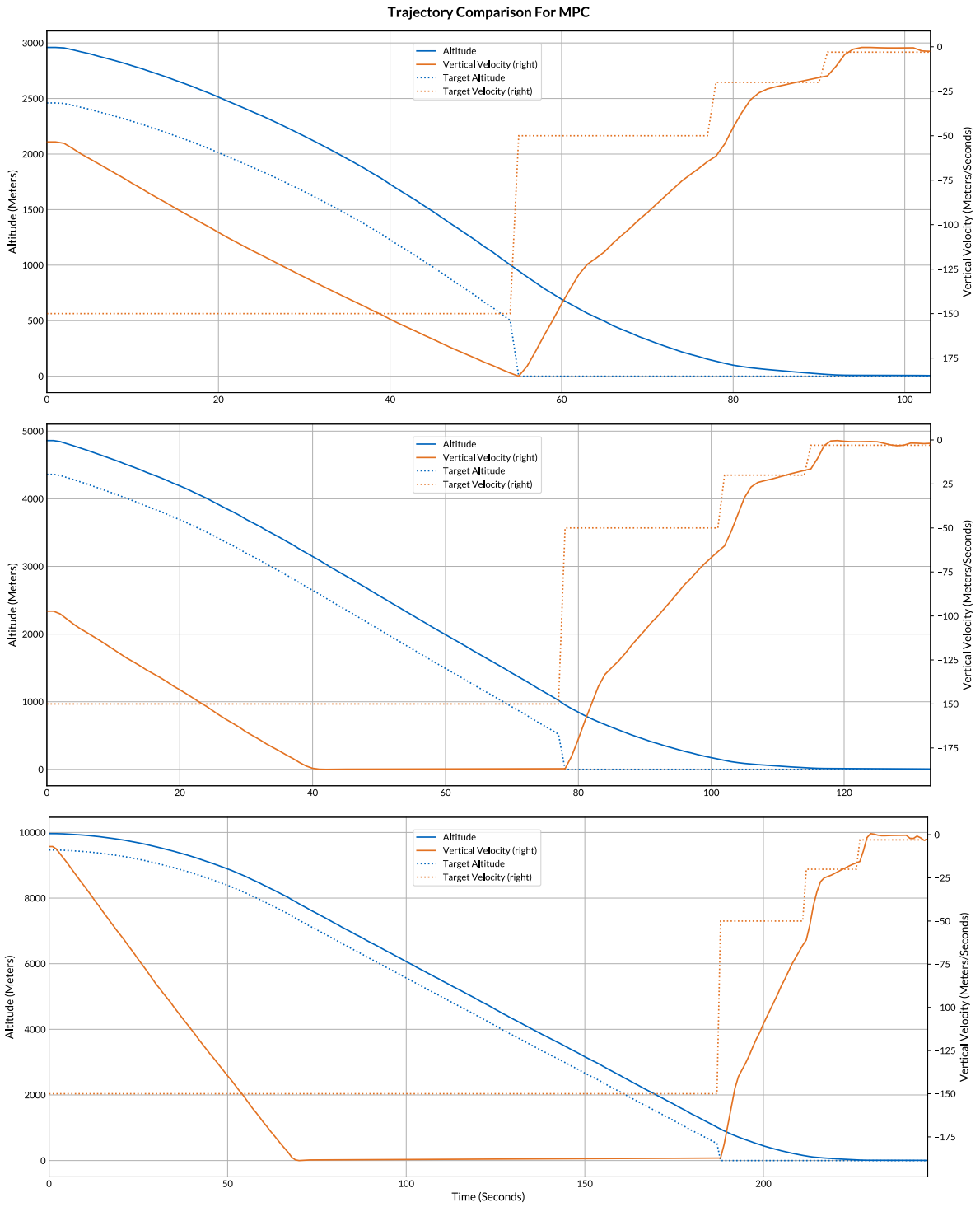

Figure 3.7: Comparison of trajectories for **MPC**.

Altitude on the left, **Vertical Velocity** on the right.
From top to bottom: **3000**, **5000** and **10000** meters landings.

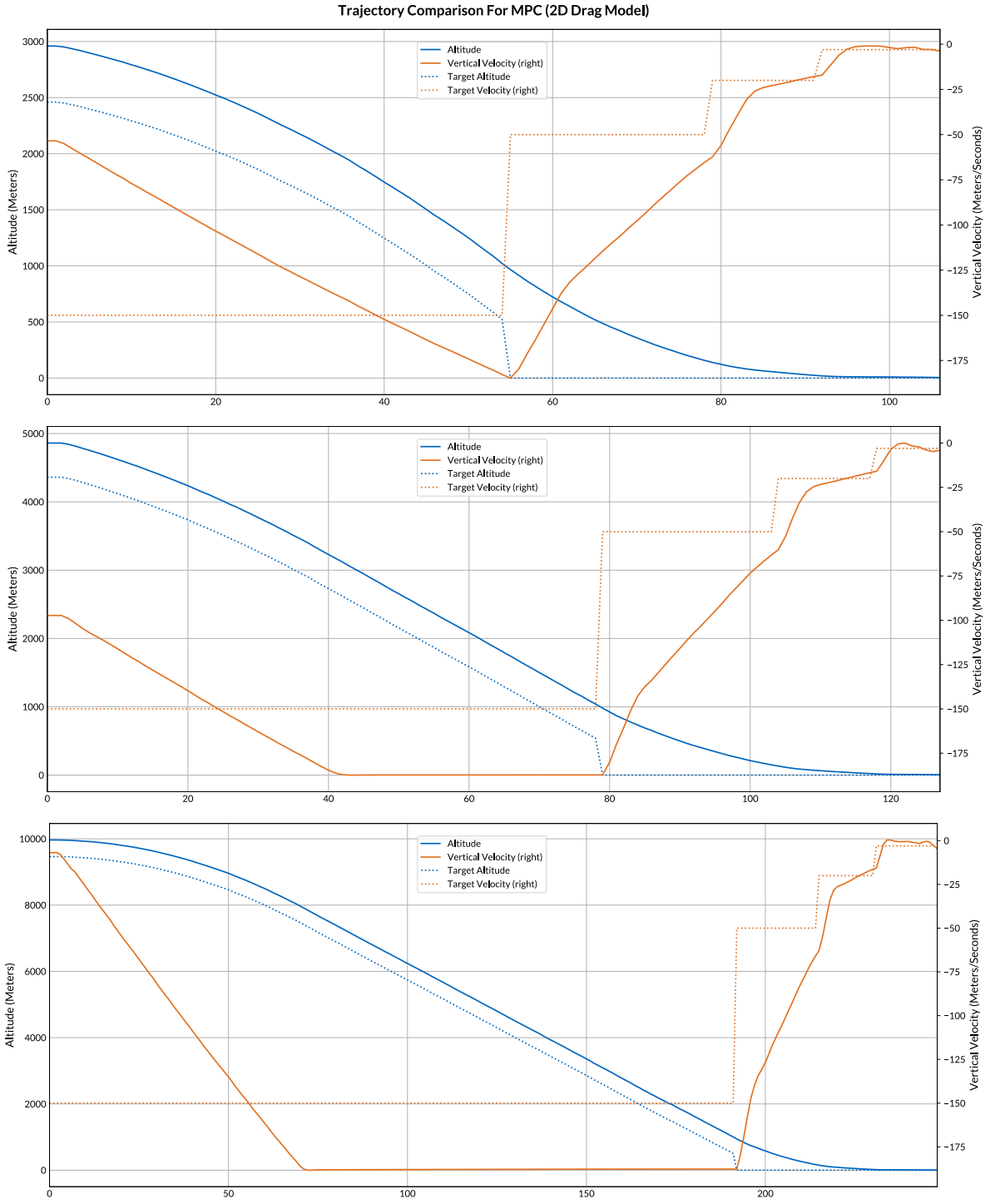

Figure 3.8: Comparison of trajectories for **MPC (2D Drag Model)**.

Altitude on the left, **Vertical Velocity** on the right.
From top to bottom: **3000**, **5000** and **10000** meters landings.

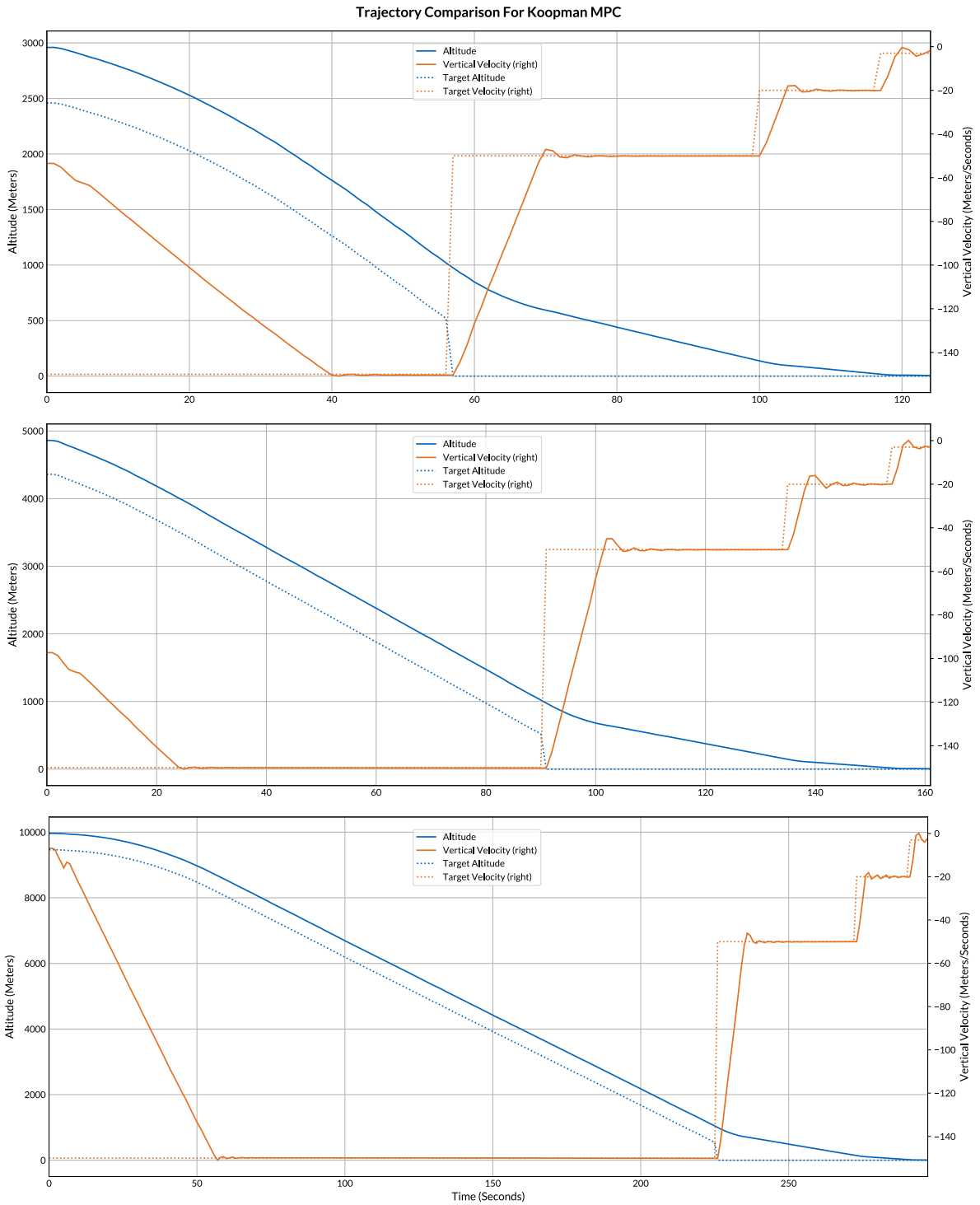

Figure 3.9: Comparison of trajectories for **Koopman MPC**.

Altitude on the left, **Vertical Velocity** on the right.
From top to bottom: **3000**, **5000** and **10000** meters landings.

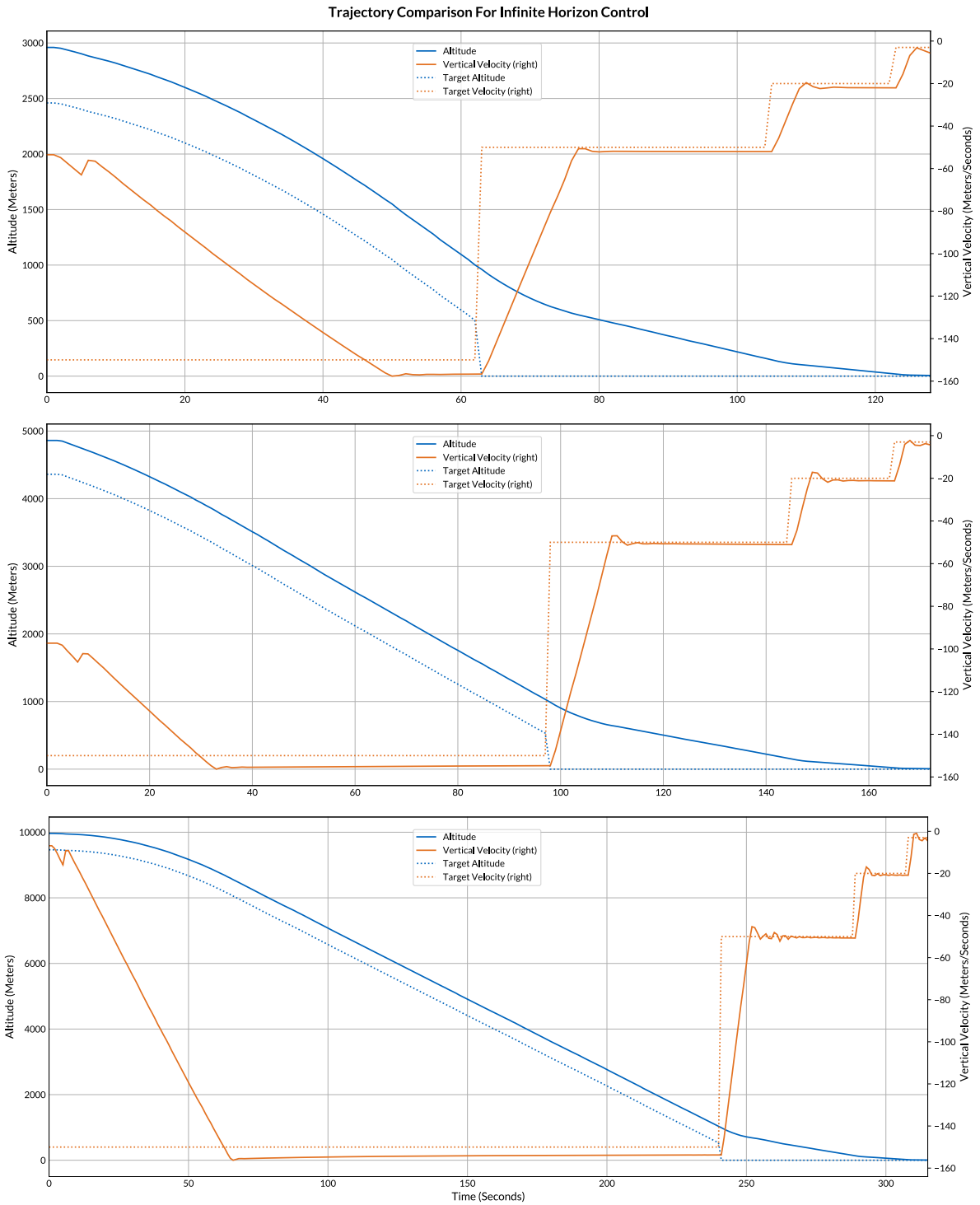

Figure 3.10: Comparison of trajectories for **Infinite Horizon Control**.

Altitude on the left, **Vertical Velocity** on the right.
From top to bottom: **3000**, **5000** and **10000** meters landings.

3.4.3 | Infinite Horizon Control

In the final subsection, Infinite Horizon Control implementation will be used to construct a complete flight plan that presumably takes the aircraft to the desired state. The only known variable will be the **initial state** and the Koopman matrix will perform state transformations and compute optimal control inputs to reach the target. There will be two different test cases. The first one is that there is only one target throughout the whole flight and it is the **zero** altitude and velocity. The second one is the same as the previous flight scenarios where there is a sliding target generator that generates a specific target based on the current state of the aircraft.

The current state of the aircraft contains four values: **Altitude**, **Vertical Velocity**, **Mass** and **Drag**. The first attempt was to compute how all these four values evolve using the **system** and **control** matrices. The results were extremely different than the expected scenarios. Mass and drag values are changing rapidly in a bizarre way and corrupts all predictions. In order to eliminate unrealistic evolution of these values, polynomial models are generated and used. There already exists a 2D polynomial drag model implemented by **Ali Ganbarov**. This model is used to obtain drag values at each time step using **vertical velocity** and **altitude** predictions. See Ganbarov (2020) for details on how this model was implemented.

Since the drag model did not prevent the distortion of the values through time, a mass model is implemented to model how the mass of the aircraft changes depending on the **time delta** and the **throttle** value. Implementation details can be found under Section 3.3.3. As the throttle value increases, the total mass reduced will increase as well due to the fuel burn. Also, since the throttle value stays the same until the next control input comes, time delta also affects how much fuel burn, which is proportional to the mass loss. The change of mass versus time delta and throttle can be seen in Figure 3.11.

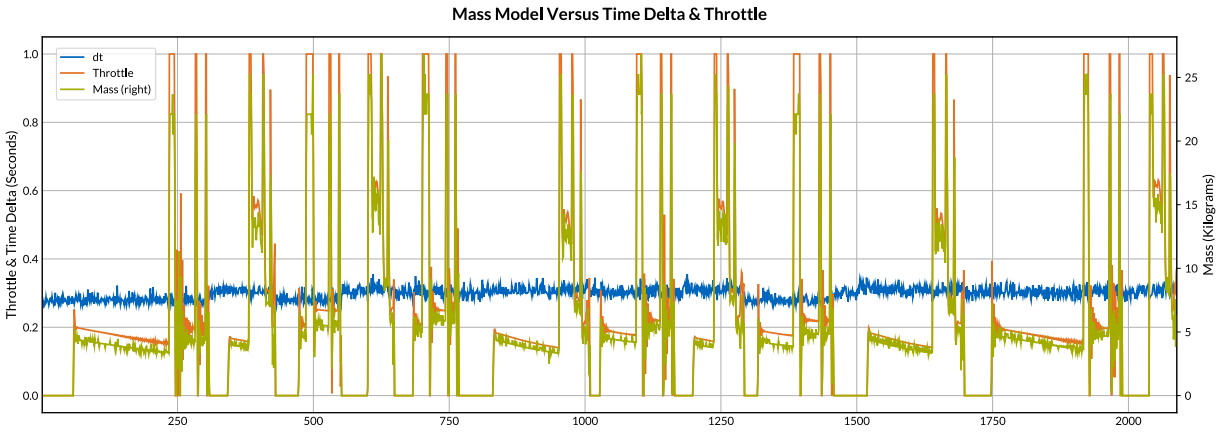

Figure 3.11: Comparison between mass delta, time delta and throttle.

The coefficients of the second degree polynomial that describes how much mass is lost based on certain throttle and time delta values is found using **The Method of Least Squares** which is explained in detail at the end of the Section 3.3.3. The resulting models predictions can be seen in Figure 3.12.

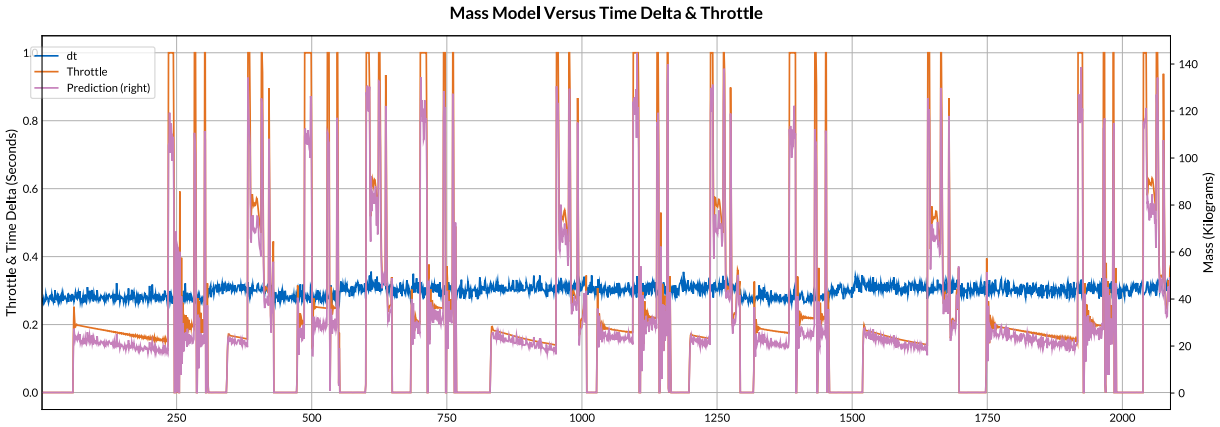

Figure 3.12: 2D Polynomial Mass Model.

Using the aforementioned mass and drag models, there leaves only two variables to estimate in order to obtain a prediction of the next state of the aircraft: **vertical velocity** and **altitude**. Recall the matrices Q and R from the Equation 2.4. The former is used to induce penalty on particular states. In our implementation, this is a matrix of dimension 21×21 with ones on the diagonal since the state space with **five** variables is lifted into a higher dimension. Each lifted state is treated equally in all the scenarios so far, and beyond. The latter on the other hand, is a 1×1 matrix that contains a value which can be used to penalize **thruster effort**. **PID Controller** is responsible for **pitch**, **yaw** and **roll** whereas **IHC** only controls one input, which is the **throttle**. **9** different penalty values with an **exponential** increase are tested for the flight scenarios in the Figures 3.13, 3.14, 3.15 and 3.16. It is worth noting that results in this section are **inconclusive** and there is a high possibility of a bug in the code we cannot spot.

The first test case has a single target state, **zero** velocity and altitude. In order to test this, simulation is run from the initial state for all values of thruster effort penalty. Figure 3.13 shows altitude, velocity, mass and throttle changes through time for all values of matrix R. As for the second case, the same sliding target handler is used to generate the same targets, as it did for all the previous tests, based on the current altitude. The initial state of the aircraft is passed to the simulation, results are fetched, then closed loop system is run to actually control the aircraft and fetch the data of the flight starting from the same initial condition. Figures 3.14, 3.15 and 3.16 shows the change in altitude, velocity and throttle through time for different altitudes.

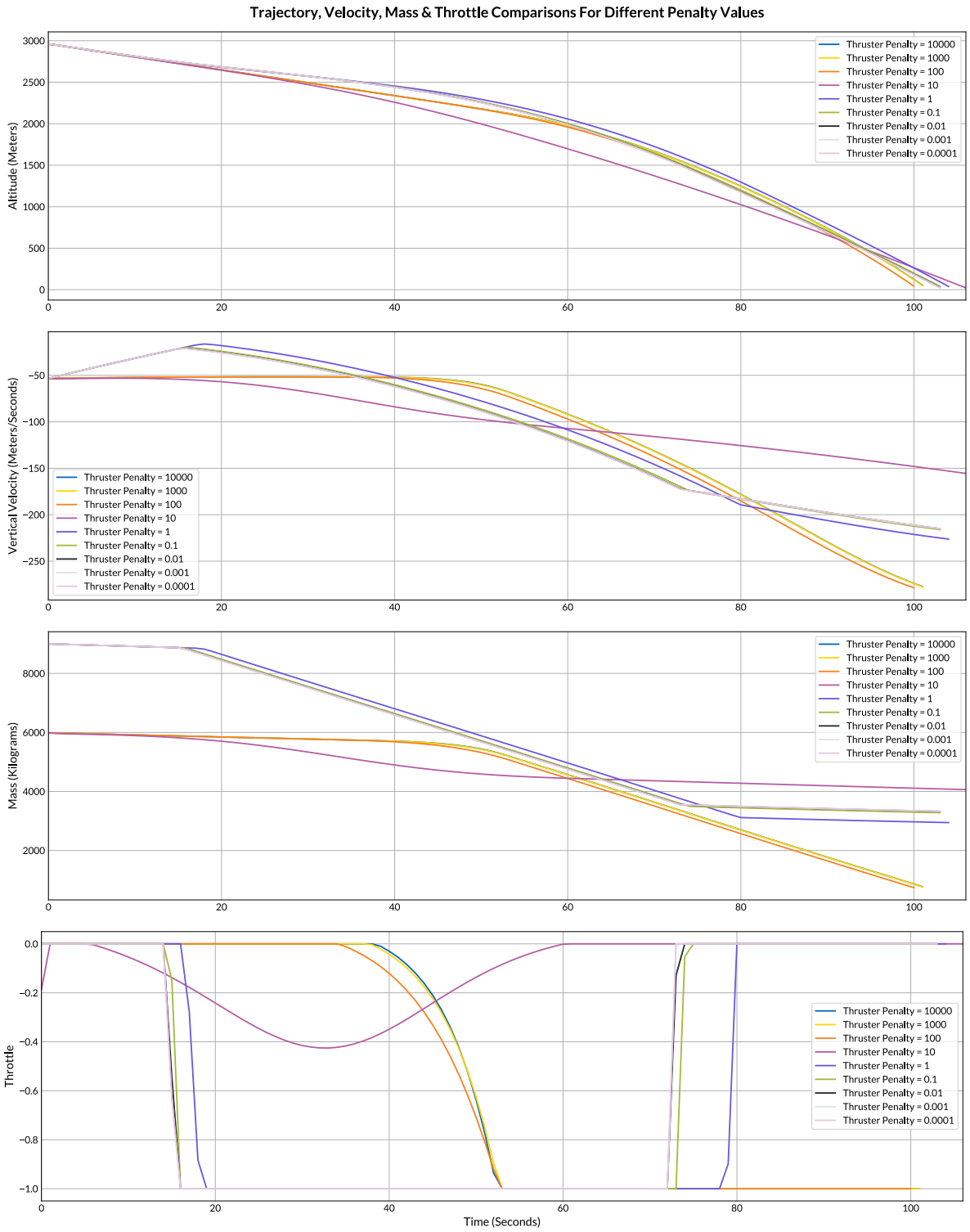

Figure 3.13: **Altitude**, **Vertical Velocity**, **Mass** and **Throttle** change during simulations.

Effects of different **thruster efforts** are illustrated. There is a single target velocity and altitude that are both **zero** all the time for all flights.

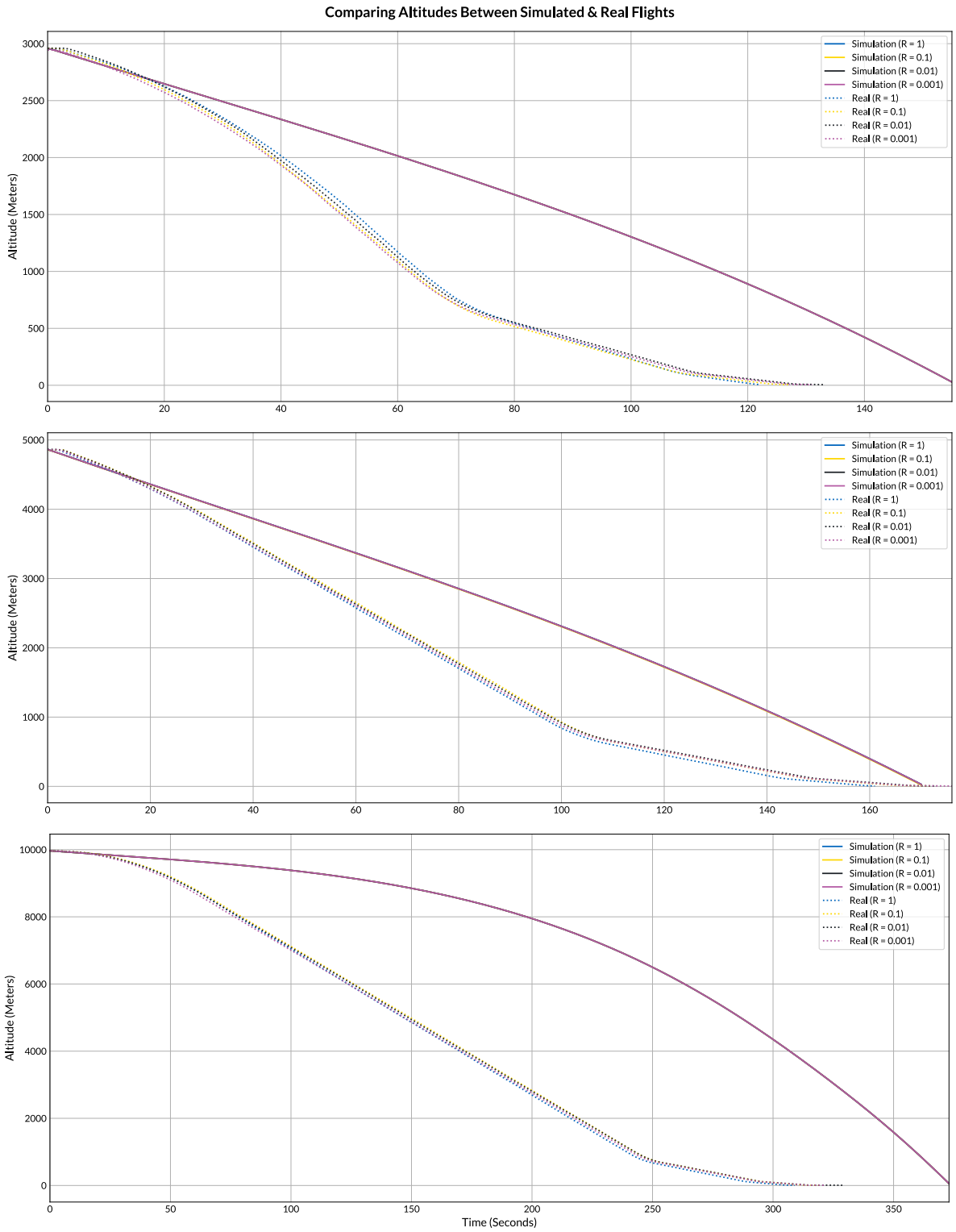

Figure 3.14: Change in **Altitudes** through time for different landing scenarios.

From top to bottom: **3000**, **5000** and **10000** meters landings.

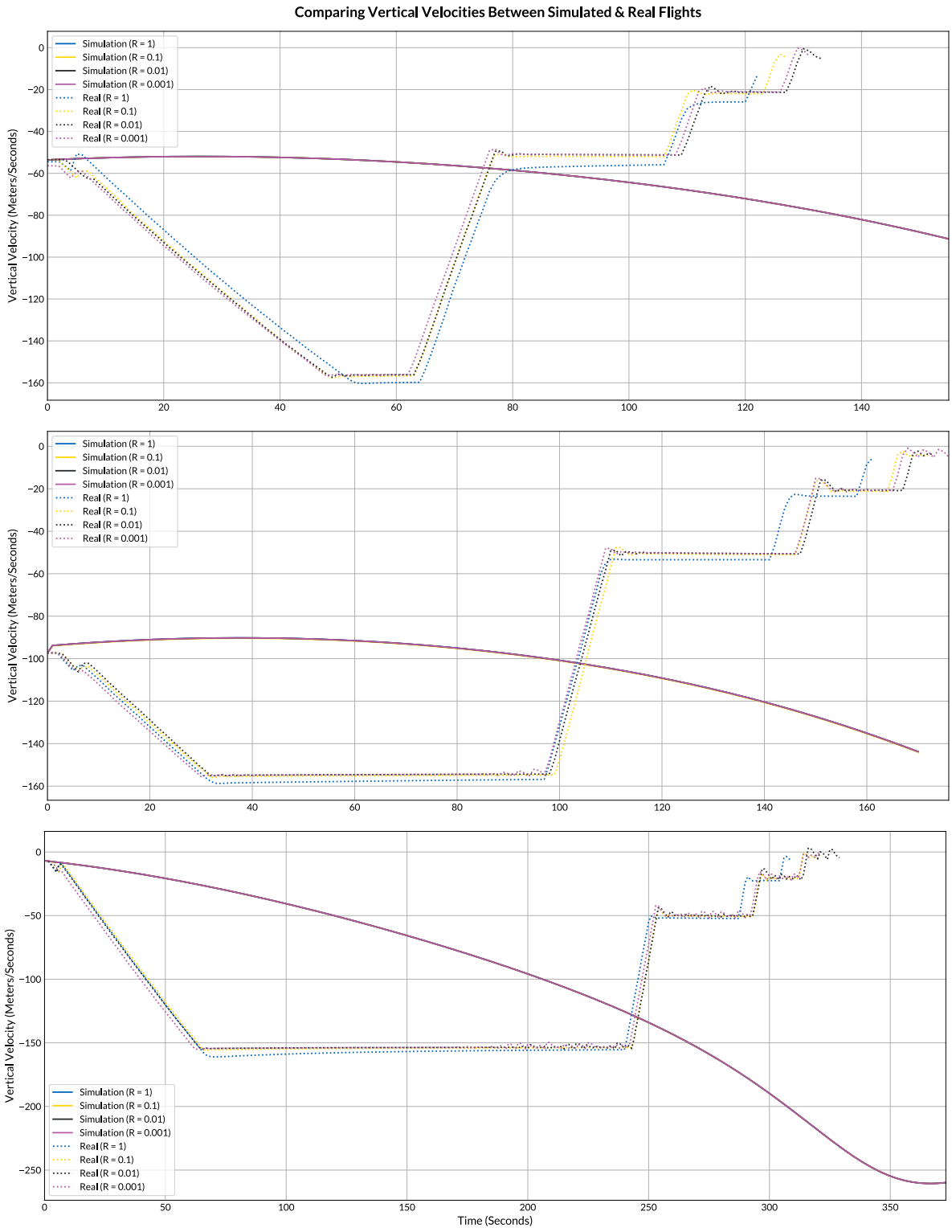

Figure 3.15: Change in **Vertical Velocities** through time for different landing scenarios. From top to bottom: **3000**, **5000** and **10000** meters landings.

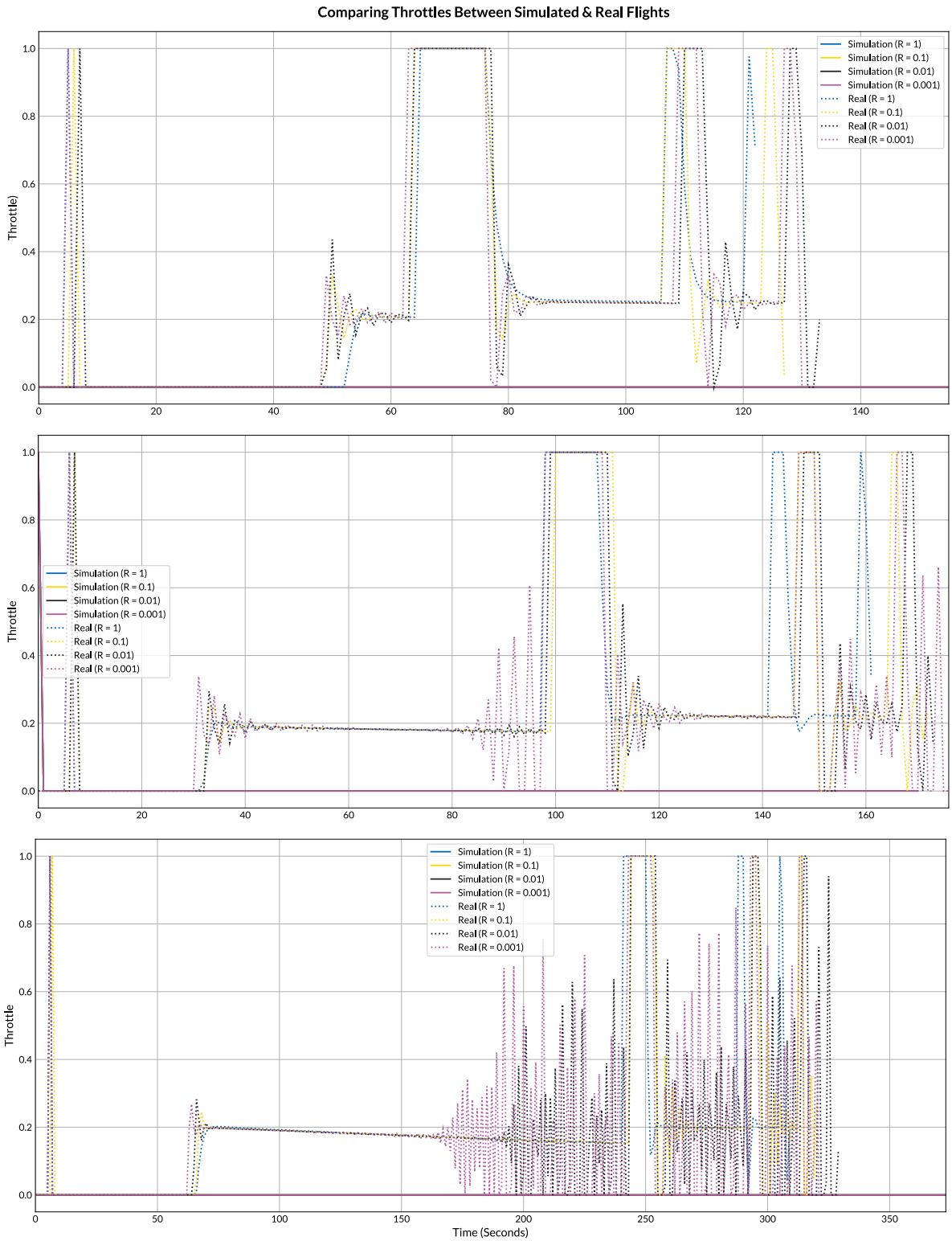

Figure 3.16: Change in **Throttle** values through time for different landing scenarios. From top to bottom: **3000**, **5000** and **10000** meters landings.

4

Conclusion

This thesis has shown that Koopman Operator approximated through the use of EDMD can capture the dynamics of the underlying system accurately so that the equations describing the system dynamics can be used to obtain control inputs directly by utilizing LQR instead of optimizing over a prediction horizon at each time step. It also proves that this approach extremely reduces computational resources used by the controller.

Infinite Horizon Control approach that utilizes LQR to obtain control inputs for the approximated linear system reduces CPU Time to generate a single control input by a factor of **14** compared to Koopman MPC, by a factor of **27** compared to MPC and a factor of **29** compared to the MPC implementation with 2D Drag model. In terms of Memory Usage, IHC uses **83%** less memory than KMPC and **78%** less than both MPC implementations. In order to determine whether it is a reliable method to use, an evaluation metric is designed to provide scores that are in line with the nature of the testing scenarios. Interpretation of the computed confidence scores and plotted time series data for different test cases shows that Infinite Horizon Control using Koopman Operator Surrogate Models is both reliable and extremely resource efficient option compared to the other models.

Even though the results are very promising in this thesis, they can be further improved. Obtaining a solution to Discrete Time Algebraic Riccati Equation is the most resource consuming part of this implementation and there are various approaches in the literature that suggest better computation times. Investigation and implementation of different methods to solve DARE can take this project one step forward. At its current state, this project provides an efficient and reliable controller for landing an aircraft. However, the biggest weakness of all the aforementioned implementations is the inability to land at a specific location which would be an important goal to achieve for the future work, but would require much more detailed flight planning procedures on top of the regular control framework.

References

Kiam Ang, G. Chong, and Yun Li. Pid control system analysis, design, and technology. *Control Systems Technology, IEEE Transactions on*, 13:559 – 576, 08 2005. doi: 10.1109/TCST.2005.847331.

Kaan Atukalp. Automated feature selection and learning of a spaceship model for model predictive control. Master's thesis, Technical University of Munich, 2021.

Peter Benner. Symplectic balancing of hamiltonian matrices. *SIAM Journal on Scientific Computing*, 22, 03 2000. doi: 10.1137/S1064827500367993.

W.L. Brogan. *Modern Control Theory*. QPI series. Prentice-Hall, 1985. ISBN 9780135903162. URL `https://books.google.de/books?id=PPBQAAAAMAAJ`.

Adam L. Bruce, Vera M. Zeidan, and Dennis S. Bernstein. What is the koopman operator? a simplified treatment for discrete-time systems. pages 1912–1917, 2019. doi: 10.23919/ACC.2019.8814306.

Steven L. Brunton. Notes on koopman operator theory. 2019.

Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019. doi: 10.1017/9781108380690.

D. A. Carlson and A. Haurie. *Dynamical Systems with Unbounded Time Interval in Engineering, Ecology and Economics*. Springer Berlin Heidelberg, 1987. doi: https://doi.org/10.1007/978-3-662-02529-1_1.

T. Faulwasser, M.A. Müller, and K. Worthmann. *Recent Advances in Model Predictive Control: Theory, Algorithms, and Applications*. Lecture Notes in Control and Information Sciences. Springer International Publishing, 2021. ISBN 9783030632809. URL `https://books.google.de/books?id=Z7P4zQEACAAJ`.

Rolf Findeisen and Frank Allgöwer. An introduction to nonlinear model predictive control. 01 2002.

Sawyer Fuller, Ben Greiner, Jason Moore, Richard Murray, René van Paassen, and Rory Yorke. The python control systems library (python-control). In *2021 60th IEEE Conference on Decision and Control (CDC)*, pages 4875–4881, 2021. doi: 10.1109/CDC45484.2021.9683368.

Ali Ganbarov. Autonomous spaceship navigation and landing using model predictive control. Master's thesis, Technical University of Munich, 2020.

E. Kaiser, J. N. Kutz, and S. L. Brunton. Sparse identification of nonlinear dynamics for model predictive control in the low-data limit. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 474(2219):20180335, nov 2018. doi: 10.1098/rspa.2018.0335.

R.E. Kalman. On the general theory of control systems. *IFAC Proceedings Volumes*, 1(1):491–502, 1960. ISSN 1474-6670. doi: https://doi.org/10.1016/S1474-6670(17)70094-8. 1st International IFAC Congress on Automatic and Remote Control, Moscow, USSR, 1960.

Snezhana Kostova, Ivan Ivanov, Lars Imsland, and Nonka Georgieva. Infinite horizon lqr problem of linear discrete time positive systems. *Proceeding of the Bulgarian Academy of Sciences*, 66, 01 2013.

A. Laub. A schur method for solving algebraic riccati equations. *IEEE Transactions on Automatic Control*, 24(6):913–921, 1979. doi: 10.1109/TAC.1979.1102178.

Daniel Lehmberg, Felix Dietrich, Gerta Köster, and Hans-Joachim Bungartz. datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283, 2020. doi: 10.21105/joss.02283.

Qianxiao Li, Felix Dietrich, Erik M. Bollt, and Ioannis G. Kevrekidis. Extended dynamic mode decomposition with dictionary learning: A data-driven adaptive spectral decomposition of the koopman operator. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(10):103111, oct 2017. doi: 10.1063/1.4993854.

Jacob Mattingley, Yang Wang, and Stephen Boyd. Code generation for receding horizon control. pages 985 – 992, 10 2010. doi: 10.1109/CACSD.2010.5612665.

Manfred Morari, Carlos E. Garcia, and David M. Prett. Model predictive control: Theory and practice. *IFAC Proceedings Volumes*, 21(4):1–12, 1988. ISSN 1474-6670. doi: https://doi.org/10.1016/B978-0-08-035735-5.50006-1. IFAC Workshop on Model Based Process Control, Atlanta, GA, USA, 13-14 June.

I. Newton, A. Motte, and N.W. Chittenden. *Newton's Principia: The Mathematical Principles of Natural Philosophy*. Newton's Principia: The Mathematical Principles of Natural Philosophy. D. Adee, 1848.

Lal Prasad, Barjeev Tyagi, and Hari Gupta. Optimal control of nonlinear inverted pendulum system using pid controller and lqr: Performance analysis without and with disturbance input. *International Journal of Automation and Computing*, 11:661–670, 12 2014. doi: 10.1007/s11633-014-0818-1.

Peter J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656: 5–28, 2010. doi: 10.1017/S0022112010001217.

Jonathan H. Tu, , Clarence W. Rowley, Dirk M. Luchtenburg, Steven L. Brunton, and J. Nathan Kutz and. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, 2014. doi: 10.3934/jcd.2014.1.391.

P. Van Dooren. A generalized eigenvalue approach for solving riccati equations. *SIAM Journal on Scientific and Statistical Computing*, 2(2):121–135, 1981. doi: 10.1137/0902010.

Matthew O. Williams, Ioannis G. Kevrekidis, and Clarence W. Rowley. A data–driven approximation of the koopman operator: Extending dynamic mode decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, jun 2015. doi: 10.1007/s00332-015-9258-5.

A. Zhakatayev, B. Rakhim, O. Adiyatov, A. Baimyshev, and H. A. Varol. Successive linearization based model predictive control of variable stiffness actuated robots. In *2017 IEEE International Conference on Advanced Intelligent Mechatronics (AIM)*, pages 1774–1779, July 2017. doi: 10.1109/AIM.2017.8014275.