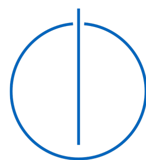
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Robust Model Predictive Control for Autonomous Spaceships with Failing Sensors

Sinan Harputluoglu



DEPARTMENT OF INFORMATICS

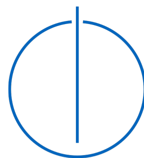
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Robust Model Predictive Control for Autonomous Spaceships with Failing Sensors

Robuste modellprädiktive Regelung für autonome Raumschiffe mit fehlerhaften Sensoren

Author: Sinan Harputluoglu
Supervisor: Prof. Dr. Hans-Joachim Bungartz
Advisor: Dr. Felix Dietrich
Submission Date: 15.11.2021



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

Munich, 15.11.2021 Sinan Harputluoglu

Acknowledgments

I would like to thank my advisor, Dr. Felix Dietrich, for his helpful support, guidance, valuable feedback, and for giving me the opportunity to work on this interesting project. Also, I would like to thank Kaan Atukalp and Ali Ganbarov for their support and enormous help when continuing this project. Finally, I would like to express my gratitude to my girlfriend and family for their continuous support.

Abstract

Controlling a system with constraints is a challenging problem. When robustness against uncertainties in the system is taken into consideration, it becomes even more challenging. To overcome this challenge, Model Predictive Control is one of the tools in control theory that can be used to tackle such challenges. Model Predictive Control (MPC) is widely used in many application areas such as plants as process control or autonomous vehicle control. The main idea behind MPC is building a mathematical model of a dynamic system and solving optimization problems online to find an optimum input to a dynamic system. MPC often takes account of finite time-horizon, thus, can estimate the future state of the system. In theory and practice, MPC is one of the successful approaches to control a constrained system. In this thesis, robustness is incorporated into MPC and a spaceship is chosen to study as the dynamic system that is controlled.

This thesis is a continuation of the previous work done in controlling a spaceship with a Model Predictive Control. This thesis describes existing models and studies how they fail when sensors fail in different ways and compares the models. Existing controllers are re-implemented to introduce disruption in the system and different landing scenarios are created. The robustness comparison procedure is implemented via simulating different failure levels of sensors which are plausible in real-world applications. Finally, the spaceship system is simulated in a space flight simulation game called Kerbal Space Program that provides realistic scenarios for space flight travel.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Theoretical Background **3**
2.1 Proportional-Integral-Derivative (PID) Controller 3
2.2 Model Predictive Control (MPC) 4
2.3 Robust MPC 7
2.3.1 Feedback MPC 8
2.3.2 Min-max robust MPC 9
2.3.3 Tube-based robust MPC 9
2.3.4 Multi-stage MPC 10
2.4 Fault Detection and Isolation 11
2.5 Simulation Environment 13
2.5.1 Kerbal Space Program 13
2.5.2 Kerbal Remote Procedure Call (kRPC) Server 14

3 Control of a Spaceship with Faulty Sensors **16**
3.1 Spaceship Dynamics 16
3.2 Simulation Environment Setup 17
3.3 Landing with Faulty Sensors 17
3.3.1 Landing Simulation Results with Noisy Sensors 18
3.3.1.1 3000 m Landing Simulation Results 19
3.3.1.2 8000 m Landing Simulation Results 26
3.3.2 Landing Simulation Results with Failing Sensors 32
3.3.2.1 3000 m Landing Simulation Results 33
3.3.2.2 8000 m Landing Simulation Results 39

4 Discussion **45**

5 Conclusion **47**

Bibliography **48**

List of Figures **51**

List of Tables **53**

1 Introduction

Autonomous control of vehicles is becoming more and more relevant. This fashion also applies to the aviation and space flight industries. Numerous solutions are implemented for autonomous control of a space flight. However, space travel and production of a spaceship have a high cost that motivates a requirement to reuse the system components. This motivation and ability to do more frequent test flights drive people to find solutions to this problem. Thus, it became a possibility in space flight history with the efforts of organizations such as Space Exploration Technologies Corporation (SpaceX). SpaceX managed to reuse some spaceship components successfully from their previous flight in 2017 [1].

Model Predictive Controller (MPC) is a widely used method for process control while satisfying a set of constraints and has numerous applications in the modern age. The term MPC refers to the idea of using an explicit model of the plant to be controlled to estimate future output behavior. This prediction capability enables online optimization of control in the difference between the predicted output and the target reference, is minimized over a future horizon, potentially subject to constraints on inputs and outputs. Today, MPC has many applications in areas such as autonomous vehicles and stability control [2], aircraft control [3], or portfolio optimization in finance [4].

The robustness of MPC to model or system uncertainty and noise is a principal concern. When a control system is stated as robust, we imply that it maintains stability and meets performance standards for a given range of model variations and signal noise. The robustness of a given control algorithm refers to an uncertainty range as well as specific stability and performance requirements.

This thesis is a continuation of Ganbaraov's [5] and Atukalp's [6] work. They managed to vertically land a spaceship using MPC, using different models from each other. However, both of these models rely on the perfect availability of state information. We have taken over their work and introduced noisy and failing sensors to the system to test and compare their models under uncertain system conditions. For this purpose, real-life plausible Gaussian noise is introduced to the sensor data and different landing scenarios are created. For the failing sensors part of the simulations, we have arranged sensors to give constant output in the middle of the landing and observed the result with different scenarios. Finally, all these simulation data are gathered and visualized.

This thesis is divided into five chapters. The first chapter introduces the topic of this thesis. The second chapter describes the theoretical background related to this thesis' subject, proposed solutions in the past, and the simulation environment. The third chapter explains the work about the comparison of different model predictive controllers under uncertain system conditions. The fourth chapters discusses results and possible solutions. Finally, the last chapter concludes this thesis and discusses future work.

2 Theoretical Background

This section describes the technical background information and is divided into five main parts. Section 2.1 describes Proportional-Integral-Derivative (PID) controller, section 2.2 describes the Model Predictive Controller (MPC), section 2.3 describes the Robust MPC and gives an overview of its different methods, section 2.4 describes the fault detection and isolation. Finally, section 2.5 describes the simulation environment.

2.1 Proportional-Integral-Derivative (PID) Controller

A proportional-integral-derivative (PID) controller is a widely used control loop mechanism that uses feedback coming from sensors or observations. The basic idea behind this controller is reading data from sensors and computing the desired actuator action by calculating proportional, integral, and derivative responses and finally, summing these three outputs to compute the final output. The control function of the PID controller can be defined as

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \tag{2.1}$$

where $u(t)$ is the control value for the next step of the system, $e(t)$ is the error value which is the difference between the desired setpoint $r(t)$ and measured process value $y(t)$, K_p is the coefficient for the proportional term (P), K_i is the coefficient for the integral term (I) and K_d is the coefficient for the derivative (D) term.

As shown in formula 2.1, the proportional term (P) is proportional to the value of $e(t)$, which means if the error is large, the output $u(t)$ will be proportionally large. The Integral term (I) takes into account past values of $e(t)$ and integrates over time and attempts to reduce residual error by introducing a control effect based on the error's past cumulative magnitude. Finally, the derivative term (D) is the estimation of the future error $e(t)$. It is proportional to the rate of change of the error $e(t)$, if there is a drastic change of the error rate, control output $u(t)$ will be large. These computations are constantly repeated in a loop until termination occurs. Overall loop structure and components of PID can be seen in figure 2.1.

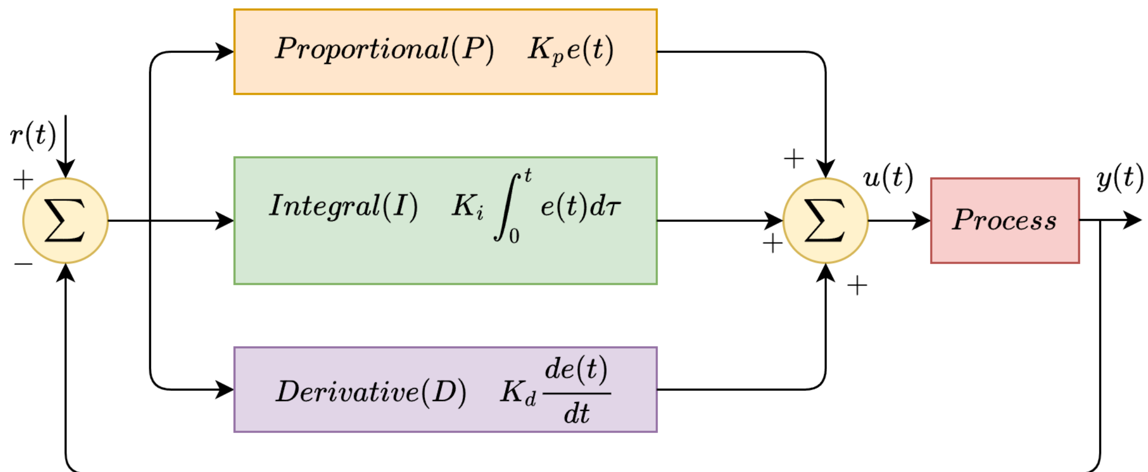


Figure 2.1: Block diagram of PID controller in a feedback loop [7]

2.2 Model Predictive Control (MPC)

Model Predictive Control (MPC), also known as receding horizon control [8] is one of the widely used and successful advanced process control mechanisms. It has been used in process industries such as chemical plants and power system balancing models [9]. The main idea behind MPC is to forecast the future behavior of the managed system over a fixed time horizon (prediction horizon) and calculate an optimal control input for the current time step that minimizes an a priori determined cost function while satisfying a set of given system constraints. This forecasting method (receding horizon estimation) is illustrated in figure 2.2.

MPC is preferred over other control methods and used in practice for many various reasons. In contrast to PID controller, MPC can deal with large time delays, MPC has a flexible and intuitive formulation, MPC can handle issues involving linear and nonlinear systems, as well as variable and multivariable systems, without changing the controller formulation, MPC uses an optimal control law that easy to implement and MPC is capable of controlling variety of simple and complex systems.

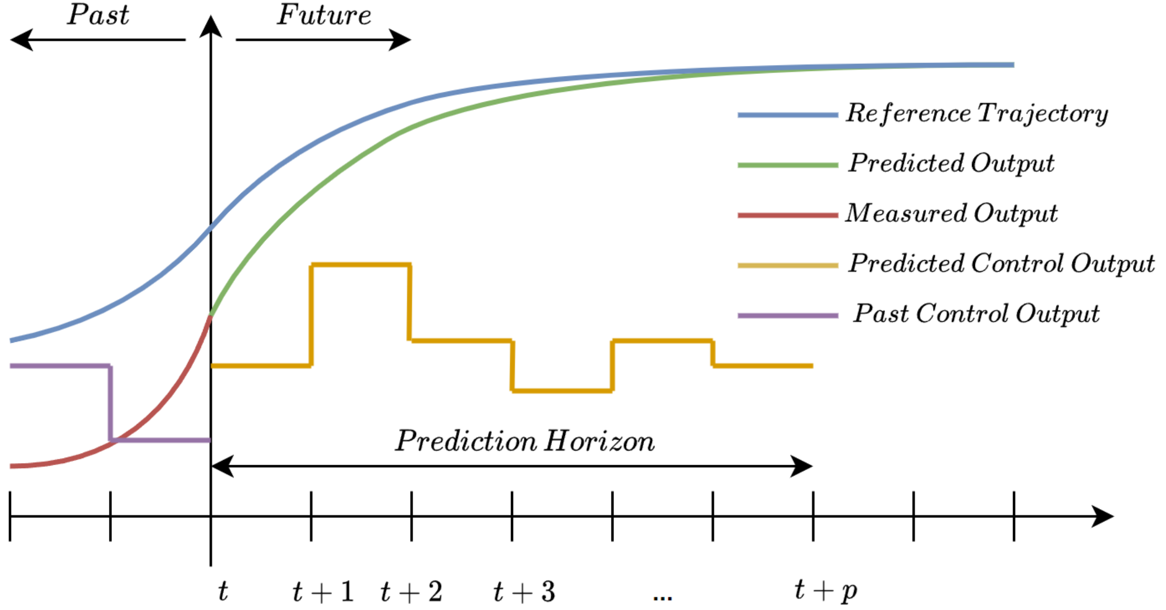


Figure 2.2: Receding horizon strategy of MPC [10]

MPC formulation is based on three components; predictive model, objective cost function, and constraints. The predictive model is the mathematic model that represents the underlying behavior of the system. A typical discrete linear time-invariant system can be modeled as

$$x_{t+1} = Ax_t + Bu_t, \tag{2.2}$$

where $x_t \in \mathbb{R}^n$ denotes the state and $u_t \in \mathbb{R}^n$ denotes the input at time instant t.

Cost function is the criteria to be optimized by the problem defined by MPC. It can have the general form

$$J_N(x_t, \overline{u}(t)) = \sum_{i=0}^{N-1} L^{stage}(x(t+i|t), u(t+1|t)) + V^{Term}(x(t+N|t)), \tag{2.3}$$

where $x(t+N|t)$ is the predicted state at $t+N$ where N is the prediction horizon, $L^{stage}(.)$ represents the stage cost, $V^{Term}(.)$ represents terminal cost and $\overline{u}(t)$ is the future control action of the predictive control horizon calculated at time t.

Constraints are usually defined on state and input as bounded sets. Constraint has to be enforced because of operational of physical limits in practice. Finally, with the addition of

constraints, MPC can be formulated as

$$\begin{aligned} &\underset{u}{\text{minimize}} \quad J_N(x_t, \overline{u}(k)) \\ &\text{subject to} \\ &x(t+i|t) \in \mathbb{U} \qquad i = 1, ..., N-1, \\ &u(t+i|t) \in \mathbb{X} \qquad i = 1, ..., N-1, \\ &x(t+N|t) \in \mathbb{X}_f \end{aligned} \tag{2.4}$$

where sets $\mathbb{X} \in \mathbb{R}^n$ and $\mathbb{U} \in \mathbb{R}^n$ denotes the constraints on the state and the input respectively. The set $\mathbb{X}_f \subseteq \mathbb{X}$ denotes the terminal constraint on the state.

Basic MPC algorithm can be defined by the following steps:

1. Model observes the current system (process).
2. Optimizer solves the problem 2.4 and finds optimal set of controls.
3. Predictive model applies the first optimal control and predicts next horizon.
4. In next time step, go to step 1.

The algorithm runs until a termination condition occurs (e.g. desired condition is met or a failure happens). There are numerous efficient solvers for the optimization problem that can be used as optimizers. A predictive model also can be any linear or non-linear (non-linear MPC) model that represents the system. This MPC algorithm is visualized in figure 2.3.

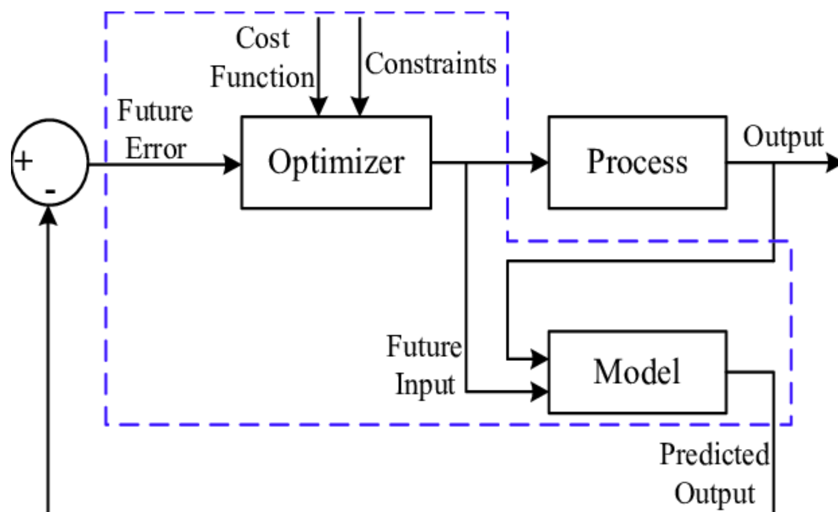


Figure 2.3: MPC control loop [11]

2.3 Robust MPC

A system is considered to be robust if stability can be assured while fulfilling the performance specifications in the face of a set of uncertainties [12]. Thus, a robust controller has to ensure constraints are not violated when the system is open to some disturbances.

In a system, uncertainty may arise in many different ways. The system may have an additive disturbance that is unknown, the state of the system may not be perfectly known, or the model of the system that is used to determine control may be inaccurate [13]. The most common uncertainties are external disturbances, noise in measurement, unmodeled non-linearities, and this thesis's main focus: failing sensors.

Conventional MPC has inherent robustness under certain conditions and it is a used approach to ignore any disturbances in the system and rely on its own robustness. This method is basically designing the controller as usual and hoping that a new control horizon will be computed while diminishing any disturbances. But, it is not guaranteed that under the existing disturbances, the system is stable or performs as well as expected [14]. Since, to solve this problem of MPC, there are works in literature that propose new MPC methods that can be stable under uncertainties in the system.

Formulation of a simple uncertain system with additive disturbance and measurement noise can be modeled by extending the linear model in equation 2.2 as

$$\begin{aligned} x_{t+1} &= Ax_t + Bu_t + Ev_t \\ y_t &= Cx_t + Dw_t, \end{aligned} \tag{2.5}$$

where w is the measurement noise and v is the unknown state disturbance at time step t. A system modeled as 2.5 can be controlled by a robust MPC and its optimization problem can be described as

$$\begin{aligned} &\min_{u} \quad J_N(x_t, \overline{u}(k)) \\ &\text{s.t.} \\ &x(t+i|t) \in \mathbb{U} \qquad i = 1, ..., N-1, \\ &u(t+i|t) \in \mathbb{X} \qquad i = 1, ..., N-1, \\ &w(t+i|t) \in \mathbb{W} \qquad i = 1, ..., N-1, \\ &v(t+i|t) \in \mathbb{V} \qquad i = 1, ..., N-1, \\ &x(t+N|t) \in \mathbb{X}_f \end{aligned} \tag{2.6}$$

where $J_N(x_t, \overline{u}(k))$ is the cost function, $w(t+i|t)$ and $v(t+i|t)$ denotes disturbance constraints on the system respectively. Since robustness is not guaranteed when the disturbances are too big in a system, sets $\mathbb{W}$ and $\mathbb{V}$ are usually assumed to be bounded.

2.3.1 Feedback MPC

The conventional MPC that is mentioned in the chapter 2.2 is an open-loop controller by design. Open-loop systems respond purely based on the input and do not use feedback from the result to self-correct. But, in a system with unknown disturbance or faults, this is a very restrictive solution.

To solve these problems, such as disturbance or faults, output-feedback MPC can be used. MPC is often designed assuming whole state information is present in the system, however, in practice, the full state cannot be measured and is not available to the controller. In these circumstances, independent algorithms are used for the state estimation such as observers, filters, and moving horizon estimation [15]. For an example of the closed-loop structure of a feedback MPC, see figure 2.4.

In feedback MPC, instead of a control sequence, a control policy is calculated. In practice, it is seen that feedback MPC performs better than conventional MPC when a system has uncertainty. But, a consequence of feedback MPC is that complexity of the optimization problem increases when the decision variable is a control policy [13].

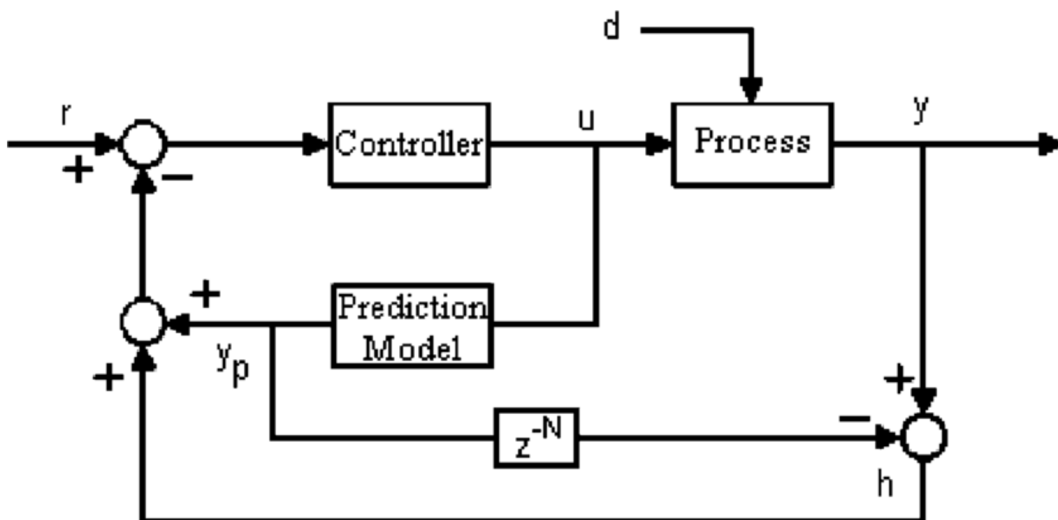


Figure 2.4: Feedback MPC control loop [16]

2.3.2 Min-max robust MPC

The min-max robust MPC is first proposed in 1987 by Campo and Morari [14]. The idea of min-max MPC is optimizing the worst case deviation in the system. Optimization problem of min-max MPC can be formulated as

$$\begin{aligned}
&\min_{u} \quad \max_{w} \quad J_N(x_t, \overline{u}(k)) \\
&\text{s.t.} \\
&x(t+i|t) \in \mathbb{X}, \qquad \forall w \in \mathbb{W} \qquad i = 1, ..., N-1, \\
&u(t+i|t) \in \mathbb{U}, \qquad \forall w \in \mathbb{W} \qquad i = 1, ..., N-1, \\
&w(t+i|t) \in \mathbb{W} \qquad i = 1, ..., N-1
\end{aligned} \tag{2.7}$$

Thus, the optimization problem is minimizing the maximum w which means worst-case performance cost. A drawback of this method is that the computational complexity which increases exponentially.

In addition, closed-loop min-max formulations (feedback min-max MPC) are also proposed [17] that improve feasibility problems, but they are complex to implement since optimization is based on searching over control policies – an infinite-dimensional problem instead of a control sequence [18].

2.3.3 Tube-based robust MPC

In an uncertain system, there are numerous possible future trajectories at every time step and each future trajectory corresponds to a specific realization of a disturbance in the system. Tube-based MPC is employed to construct a "tube" that constrains these state disturbance realizations.

The main idea of the tube-based MPC is generating a tube using a secondary feedback controller. First, conventional MPC is employed for the problem and used to determine the nominal trajectory (center of the tube). In the second step, secondary feedback controller that acts on the deviation between states is used to construct the tube around the found nominal trajectory [19][20].

There are two main advantages of tube-based MPC: first, the optimization problem is changed to finding control sequences rather than control policies, thus it reduces the computational complexity, second, the secondary feedback controller ensures that in presence of the uncertainties in the system the deviation is bounded by the tube.

2.3.4 Multi-stage MPC

Multi-stage MPC models the realizations of uncertainties in a system as scenarios. The main idea of the multi-stage MPC is considering numerous scenarios that every one of is one possible realization of all uncertain parameters at every time step within the horizon. These scenarios that are considered in MPC can be represented as a scenario tree [21] (see figure 2.5)

The basic working principle of Multi-stage MPC is that it solves the problem at every time step at the root z_0 while considering the uncertain future evolution and future decisions that might utilize the information gathered along the branches. This approach leads to an open-loop formulation of the optimal closed-loop problem for the uncertainties in the system which reduces the conservativeness of this approach [22].

Naturally, considering every possible value of uncertain parameters is infeasible, so there are some implementations of generating scenario trees for all combinations of minimum, nominal and maximum values of uncertain parameters to reduce the computational cost [23].

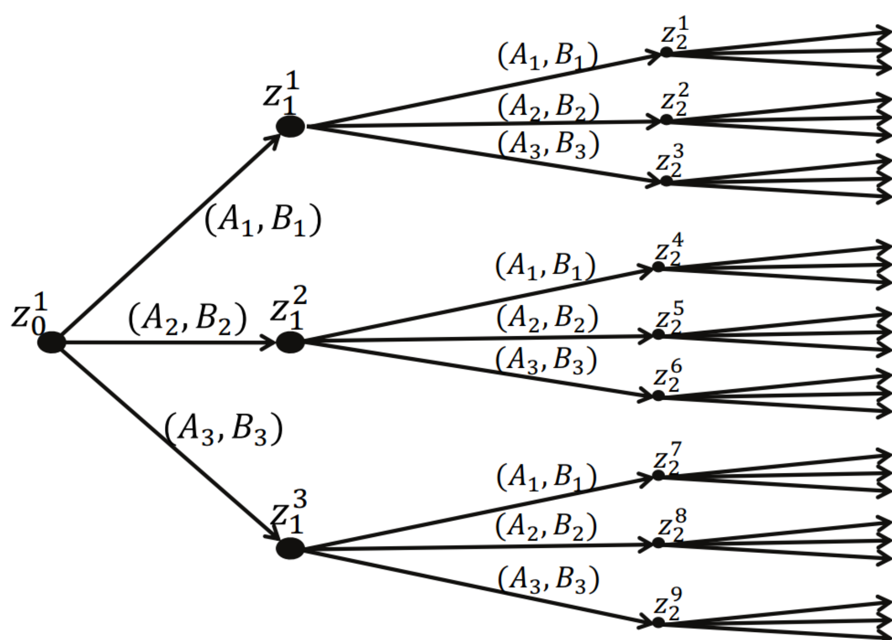


Figure 2.5: Scenario tree representation of the uncertainty evolution of multi-stage MPC [24]

2.4 Fault Detection and Isolation

A fault is defined as an unpermitted deviation of at least one characteristic property or parameter of the system from the acceptable/usual/standard condition [25]. The faults in the system may cause a failure; permanent disruption of ability to perform under its operating conditions. Many fault types may occur in a system, but the most common are actuator fault, sensor fault, and process fault itself.

A sensor fault is an irregular fluctuation in readings, such as a systematic error that affects the value given by an altimeter. An actuator fault is a malfunction of a device that affects system dynamics, such as problems in a thruster of a spaceship. Process faults are changes in the system's parameters that alter its dynamics, such as an unmodeled change in the aerodynamic coefficients of a spaceship. There are three categories of temporal aspects of faults; abrupt faults such as offset, incipients fault such as drifting parameters and intermittent fault. These faults may arise from various sources such as design error, implementation errors, wear, aging, and environmental aggressions.

Fault detection is finding out the presence of faults in a system when that fault occurs. Fault isolation is determining the type and location of these faults. This process is called fault detection and isolation (FDI). FDI employs the concept of redundancy in two ways: hardware redundancy and analytical redundancy (see figure 2.6). Hardware redundancy is comparing signals from sensor sets that have the same function to detect faults. The analytical redundancy approach uses a mathematical model of the system together with some estimation techniques for FDI [26].

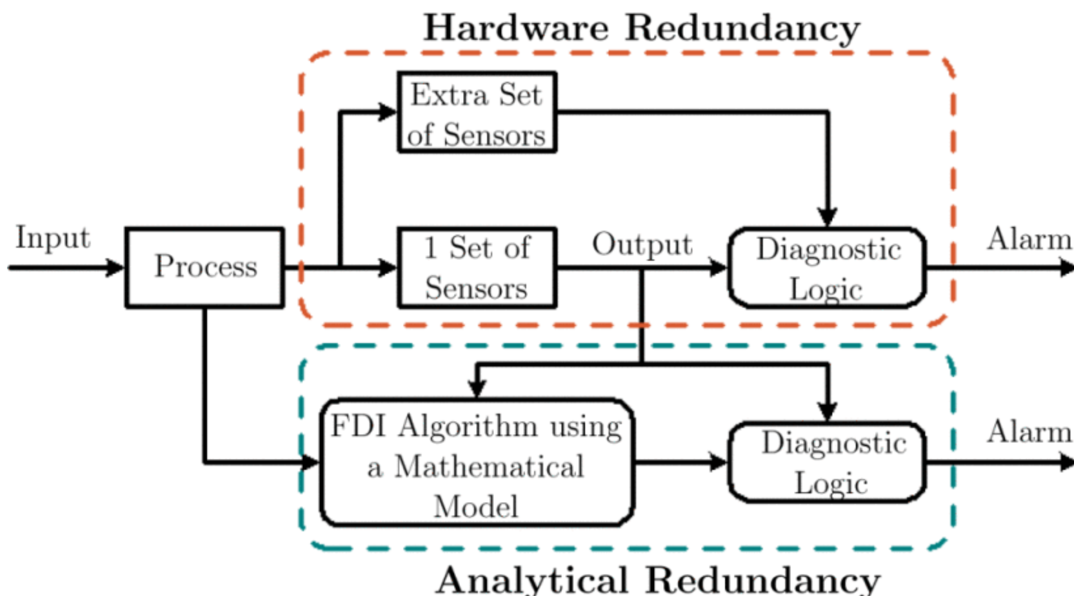


Figure 2.6: Hardware redundancy and analytical redundancy for FDI [26]

FDI approaches are generally divided into model-based and data-driven (process history) methods. Further, each of these methods can be divided into quantitative and qualitative methods (for a list of methods see figure 2.7). The quantitative methods use mathematical models to generate residuals for FDI, such as observers and Kalman filters. The qualitative methods use artificial intelligence techniques to find the difference between observed behavior and predicted behavior, such as neural networks.

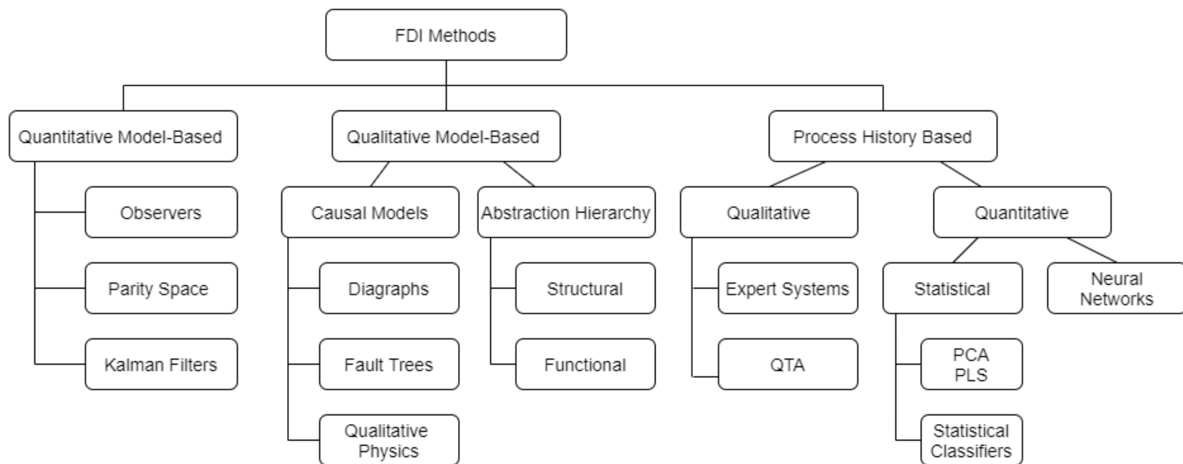


Figure 2.7: Fault detection and isolation methods [27]

After fault detection and isolation, the control law of the system needs to change for the fault-tolerant control (FTC). FTC or named as reconfiguration is changing the controller in response to the specific fault. The basic idea of fault detection, isolation, and reconfiguration (FDIR) can be explained in three steps: the generation of residuals which are the difference between measurements and estimated process output, deciding if a fault occurred in the system, and reconfiguring the system in response to the fault. These three steps illustrated in figure 2.8.

In model-based FDIR approaches, the residual generation is based on the mathematical model of the system. When a fault does not exist in a system, residuals should be small (ideally zero) and when a fault occurs, residuals should be sufficiently large. However, in practice, the model can not describe the system exactly because of disturbance and noise in the environment. Thus, residuals are never zero in a system without a fault. To overcome this problem there are two approaches: robust residual generation and robust residual evaluation.

Robust residual generation is designing a robust filter or estimator to generate residuals that are unaffected by noise and uncertainties in the system, but sensitive to faults in the system. Observer-based methods, parity relation methods are some examples of this approach. The robust residual evaluation method is designing robust hypothesis testing algorithms to evaluate the residuals that are assumed as random variables. The most basic method is deciding a fault occurs when a value of residual exceeds a threshold [26].

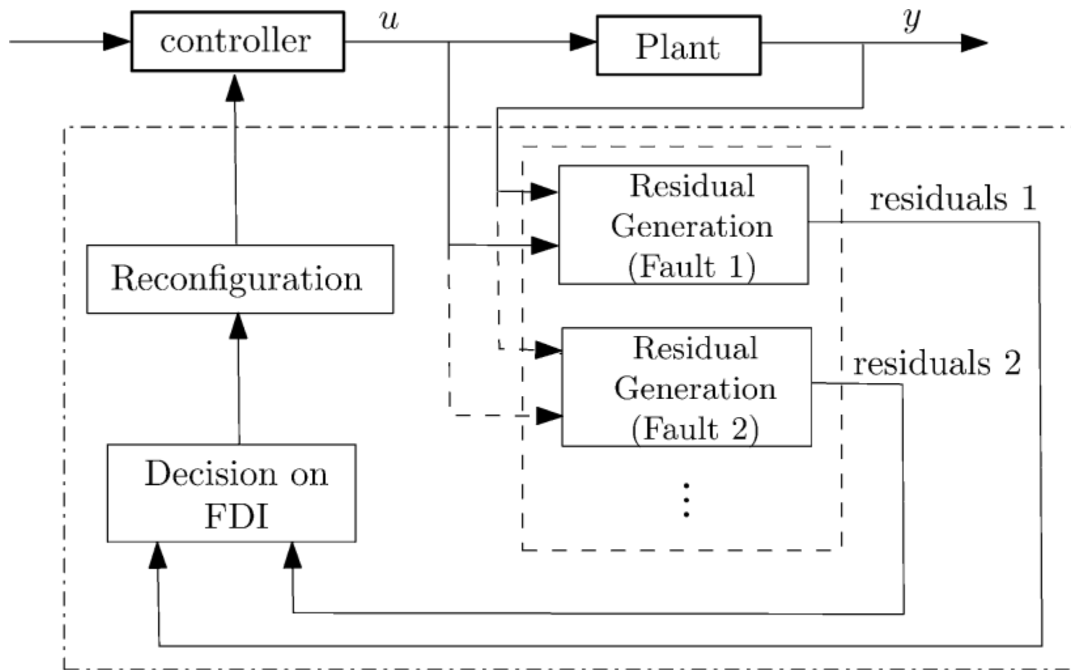


Figure 2.8: Fault detection, isolation and reconfiguration scheme [26]

2.5 Simulation Environment

2.5.1 Kerbal Space Program

Kerbal space program is a simulation video game for space flight. The game provides a realistic orbital and Newtonian physics engine in a 3-D environment [28]. Users can simulate various real-life orbital maneuvers such as orbital rendezvous. In the game, all objects are simulated based on Newtonian dynamics - except celestial bodies and in the "time warp mode", where trajectories are precomputed.

Kerbal space program provides many flight scenarios to users. Users can create planes, spaceships, and aircraft using various components provided by the game. The game has a large variety of flight components such as guidance systems, different kinds of rocket engines and fuel tanks, staging separators, etc. In addition to the possibilities of flight vehicles, the kerbal space program simulates the solar system. Users can visit and land on other planets using their space flight vehicles.

Kerbal space program provides a detailed interface to the user for the flight simulation. General interface and game scene shown in figure 2.9.

Figure 2.9: A screenshot of Kerbal Space Program game interface

The game interface has an altimeter at the top, a throttle indicator on the bottom left, and a navigation ball with a speedometer at the bottom center. In addition to the throttle indicator at the bottom left, the vehicle's principal axes -roll, yaw, and pitch- are shown at the bottom left. Users can use this information and navigate the aircraft.

Kerbal space program received a lot of attention and users created many mods for this game. These mods provide additional content such as different rocket parts, new destinations. One of the mods called kRPC is used in this thesis which is explained in section 2.5.2.

2.5.2 Kerbal Remote Procedure Call (kRPC) Server

Kerbal remote procedure call (kRPC) is a mod or plugin for Kerbal Space Program that allows running scripts to control the game. The mod library has support for many popular programming languages such as C++, Java, and Python [29].

kRPC runs a server in the game and client scripts connect to this server and use it. Clients' scripts can execute many procedures to control and interact with aircraft over a local network. kRPC server has a separate interface that shows server status in the game scene which is shown in figure 2.10.

Figure 2.10: Screenshot of kRPC mod interface

3 Control of a Spaceship with Faulty Sensors

This section presents the landings and navigation of the spaceship with faulty sensors and is divided mainly into four parts. Section 3.1 describes spaceship dynamics, section 3.2 describes the simulation environment setup for the landing of the spaceship.Finally, section 3.3 explains the landing of proposed MPCs before [6][5] with noisy and failing sensors and compares the results.

3.1 Spaceship Dynamics

In both Ganbarov's [5] and Atukalp's [6] thesis, a single model of the spaceship is used for landing, and for this thesis, the same model will be used for the robust landing. This spaceship model has a simple design that can be modeled easily with Newtonian dynamics.

Thrust, lift, and drag is the three main forces acting on the spaceship. Lift and drag forces are aerodynamic forces that are dependent on the size, shape, and velocity of the spaceship. The drag of the spaceship is modeled as a polynomial model based on altitude and vertical velocity in Ganbarov's implementation. In Atukalp's implementation, the drag model is replaced with a neural network. The spaceship and acting forces on it can be seen in figure 3.1.

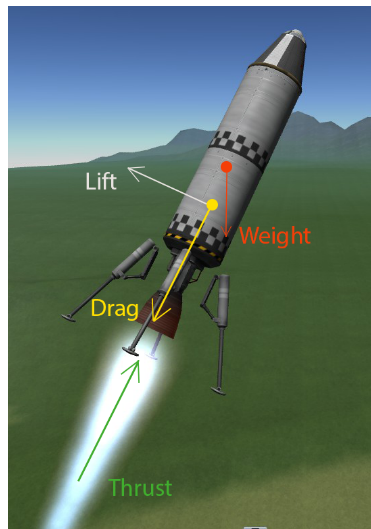


Figure 3.1: Spaceship and acting forces on it

3.2 Simulation Environment Setup

The whole simulation is done in the Kerbal Space Program environment using the kRPC module. Since the purpose of this thesis is to compare two different implementations under uncertain system conditions and propose a robust implementation, both Ganbarov's and Atukalp's controllers are set up for the simulation. The spaceship is directly imported from previous setups. The spaceship is constructed with a commanding pod, two fuel tanks, a liquid engine that has a thrust capacity of 240 kN and the spaceship has a total mass of 7.2t [5].

In Atukalp's thesis, there are numerous proposed models for the spaceship. According to Atukalp's implementation, a neural network for the altitude prediction and a linear regressor for the vertical velocity prediction gave the best results in practice [6]. Thus, in this thesis, these predictors are set up for faulty landings. In Ganbarov's thesis, a Newtonian model is proposed for altitude and vertical velocity prediction and used directly in this chapter for faulty landings.

Both of these implementations assume the perfect availability of state information. Thus, for this thesis, these controllers were implemented again to introduce noisy or faulty sensors. These controllers were implemented again in such a way that we can define random Gaussian noise with a given standard deviation to the system or we can "disrupt" a sensor in a given time.

3.3 Landing with Faulty Sensors

In this section both MPC implementations are compared in simulation environment under uncertain system conditions. These controllers' target values for the system are same and altitude target can be formulated as

$$y_{target} = \begin{cases} y - 500 & y > 5000 \\ y - 800 & 1000 < y \leqslant 5000 \\ -2 & y \leqslant 1000 \end{cases} \tag{3.1}$$

where y is altitude and y_{target} target altitude in meters. In addition, vertical velocity targets can be formulated as

$$v_{target} = \begin{cases} -200 & y > 1000 \\ 0 & y \leqslant 1000 \end{cases} \tag{3.2}$$

where y is altitude in meters and v_{target} is target vertical velocity in meters per second.

Both of these controllers control the throttle of the spaceship based on vertical velocity and altitude. Both MPCs prediction horizons are ten and for Ganbarov's implementation, the

control horizon is also ten, but for Atukalp's implementation, the control horizon is one. Spaceship's principal axes such as yaw, pitch, and roll are controlled by PID controllers which are implemented by Ganbarov [5]. PID controllers are optimized to keep the spaceship in the upright position during the landing. In these landing simulations, PID controllers are directly used from Ganbarov's implementation.

We have simulated landings starting from two different altitudes: 3000 and 8000 meters. The main reason for choosing these altitudes is that the KSP environment simulates the atmosphere in different layers to make it a more realistic simulation. These layers have a different atmospheric density which changes the drag force acting on the spaceship.

3.3.1 Landing Simulation Results with Noisy Sensors

For this part of the simulation, we have constructed Gaussian random noises with different standard deviations and added noise to the altimeter and speedometer. Since model predictions for the altitude and velocity depend on each other, firstly, we added noise to each one sensor separately and after, both sensors to observe the differences. In general, noisy sensors did not cause failure during most of the flights. Both models handled the noise and approached the ground the same as without the noise case. But, near the end of the landing, some noise spikes caused a failed landing. In this section, we have visualized the data from the end of the landing and also the whole landing data for each scenario.

Noise deviation is chosen according to plausible real-life scenarios. We have visualized model predictions, real values and constructed noisy values for each model, and compared the results. The summary of landing results can be seen in table 3.1 and table 3.2.

Landing From	Altimeter Noise	Speedometer Noise	Result
3000 m	50 m	-	Crash
3000 m	-	10 m/s	Crash
3000 m	50 m	10 m/s	Crash
8000 m	50 m	-	Crash
8000 m	-	10 m/s	Crash
8000 m	50 m	10 m/s	Crash

Table 3.1: Naive MPC simulation results with noisy sensors.

Landing From	Altimeter Noise	Speedometer Noise	Result
3000 m	50 m	-	Crash
3000 m	-	10 m/s	Crash
3000 m	50 m	10 m/s	Crash
8000 m	50 m	-	Crash
8000 m	-	10 m/s	Crash
8000 m	50 m	10 m/s	Crash

Table 3.2: ML models simulation results with noisy sensors.

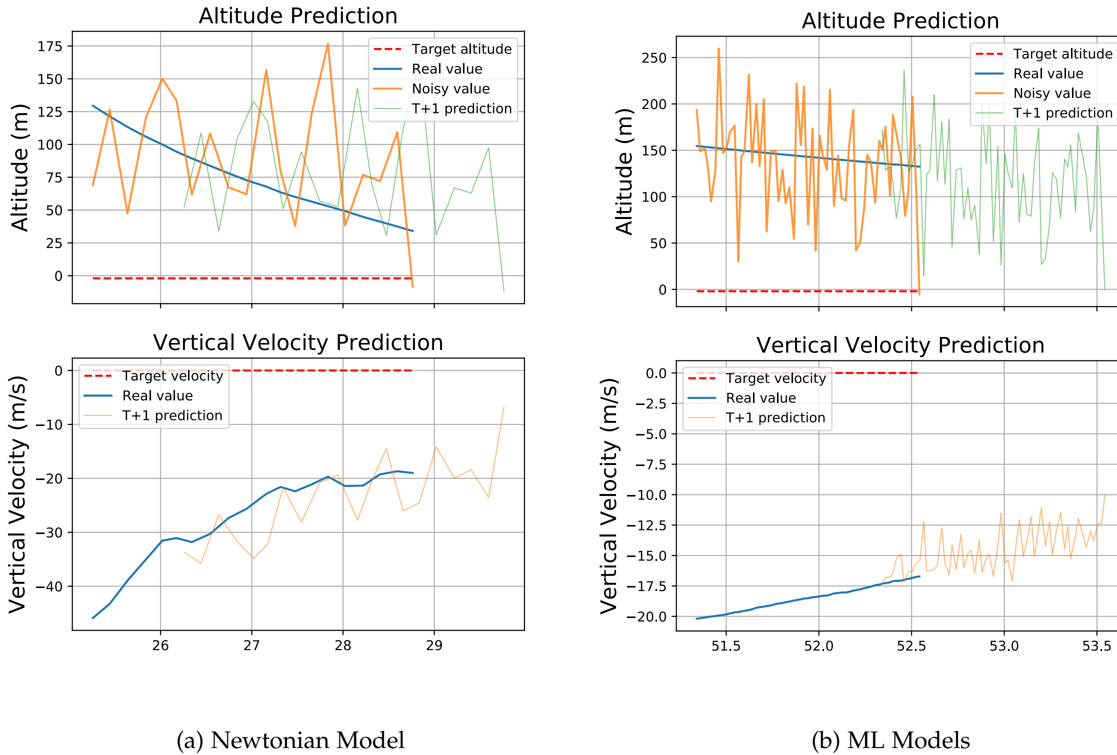
3.3.1.1 3000 m Landing Simulation Results

3K Landing Results with Altimeter Noise

We did not observe failure at the beginning of the landing simulation. Because of noise in the altimeter, target altitudes were noisy for both models. However, both models handled the noise to some degree and were able to continue the landing process. After passing the 1000 m, both models' target altitudes were zero (0), thus, the noise did not affect the targets. But, the last few hundred meters of the landings were affected by the noise and this caused a crash. Since the last few hundred meters were disturbed, we visualized that part of the landing in figure 3.2.

In the Newtonian model, controller failure happened at the 29th second of the flight. Simulated noise caused a spike in the data and, the altimeter sent the value of -4 m to the MPC. According to the controller, the target was reached and this caused the controller to disengage. But, the real value of altitude was 35 m at this time. Disengagement of the controller caused the spaceship to free-fall from that altitude without any controls. Finally, the spaceship accelerated towards to ground, causing a crash.

In the ML models, controller failure happened at the 52nd second of the flight. Similar to the Newtonian model simulation, simulated noise caused a spike in the data. In the ML model simulation, the altimeter sent the value of -2 m to the MPC. Again, according to the controller, the target was reached and this caused the controller to disengage. But, the real value of altitude was 130 m at this time. Disengagement of the controller caused the spaceship to free-fall from that altitude without any controls. Finally, the free-fall ended with a crash in this simulation.



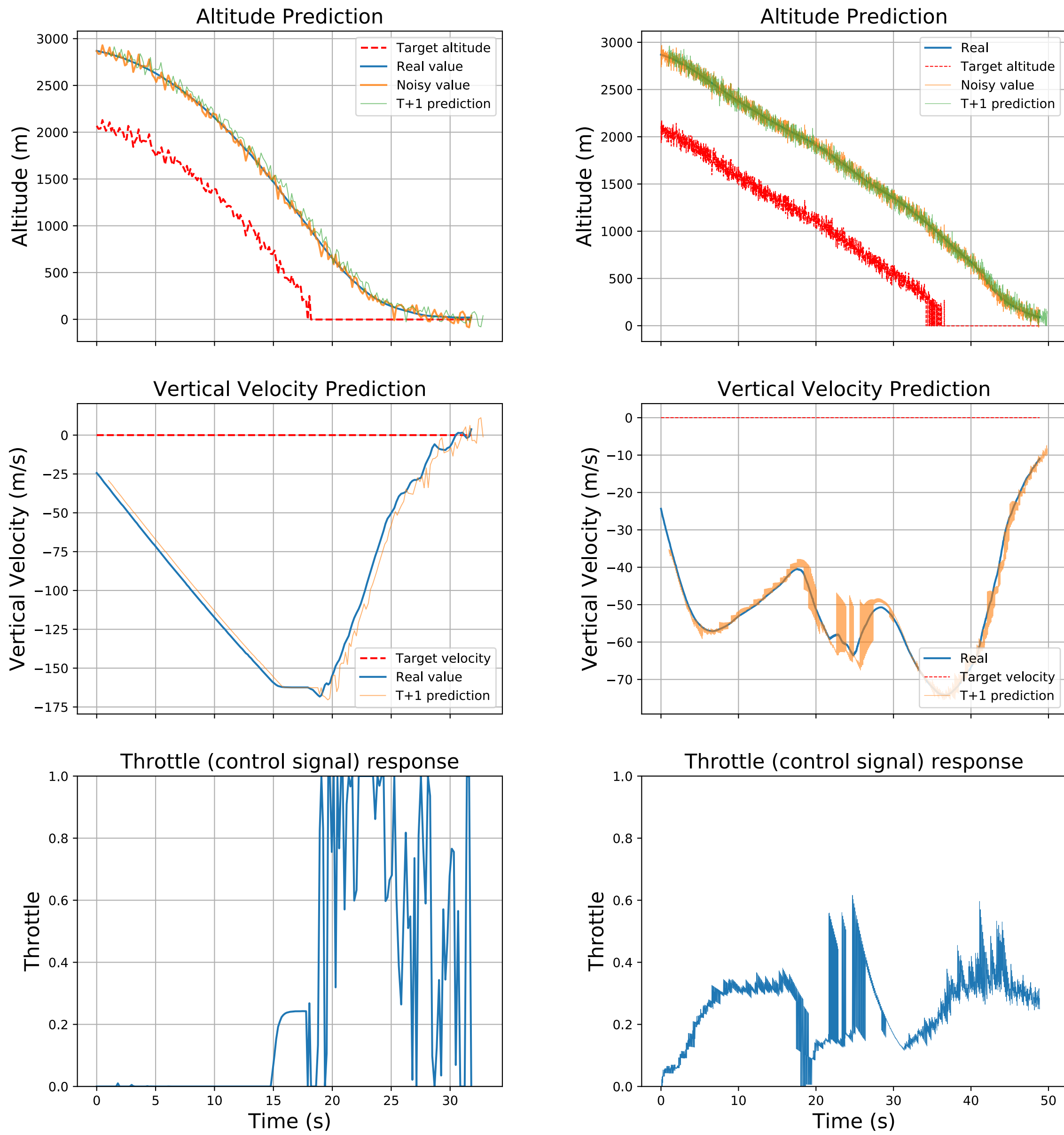
(a) Newtonian Model

(b) ML Models

Figure 3.2: Last moments of spaceship landing results from 3000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise.

Finally, we visualized the whole landing process in figure 3.3. Noise in the sensor caused noise in altitude and vertical velocity prediction for both models during the flight. We observed that the nature of ML model predictions was noisier compared to the Newtonian model. The cause of vertical velocity prediction noise is the noise in the altitude sensor. The reason is, vertical velocity predictions are dependent on altitude in both models.

One of the main differences between these models is the cost function in the controller. The Newtonian model punishes the throttle and ML models do not. In figure 3.3, we observed that there was not any noise in the vertical prediction until the 15th second in the Newtonian model. The reason is the spaceship free-falls until that point. After that, the controller tries to optimize and makes noisy predictions which also causes noisy control values. For the ML model case, vertical velocity prediction and control values were always noisy.

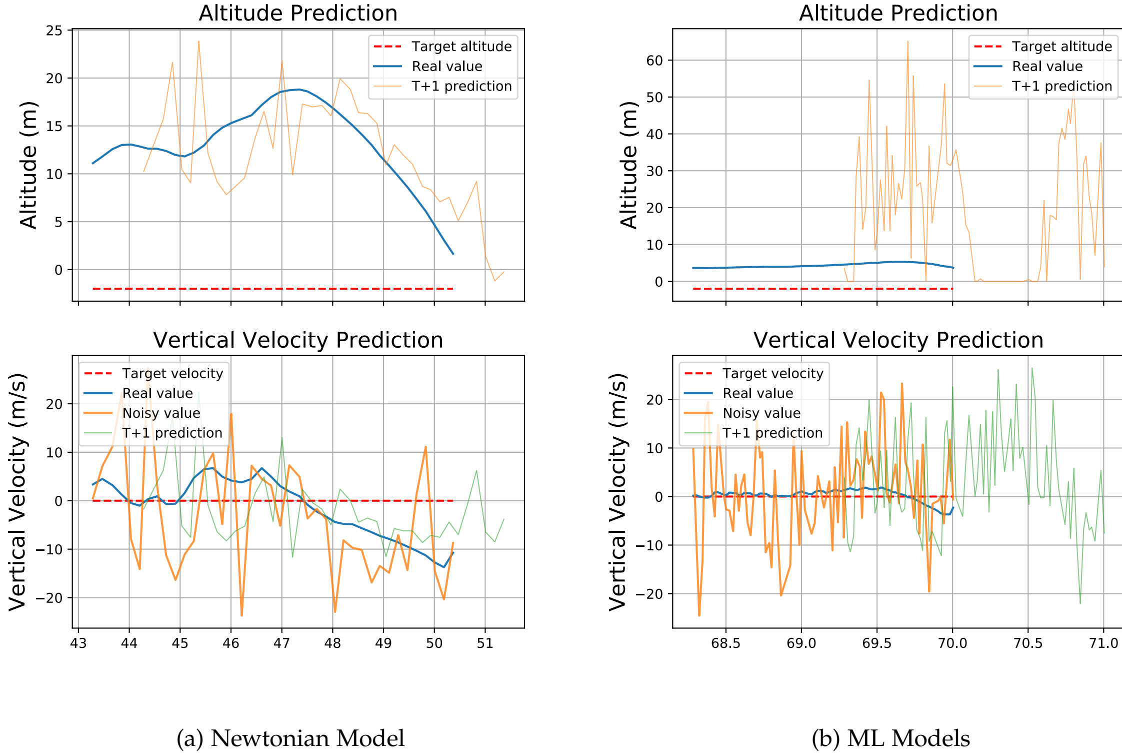


(a) Newtonian Model

(b) ML Models

Figure 3.3: Spaceship landing results from 3000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3K Landing Results with Speedometer Noise



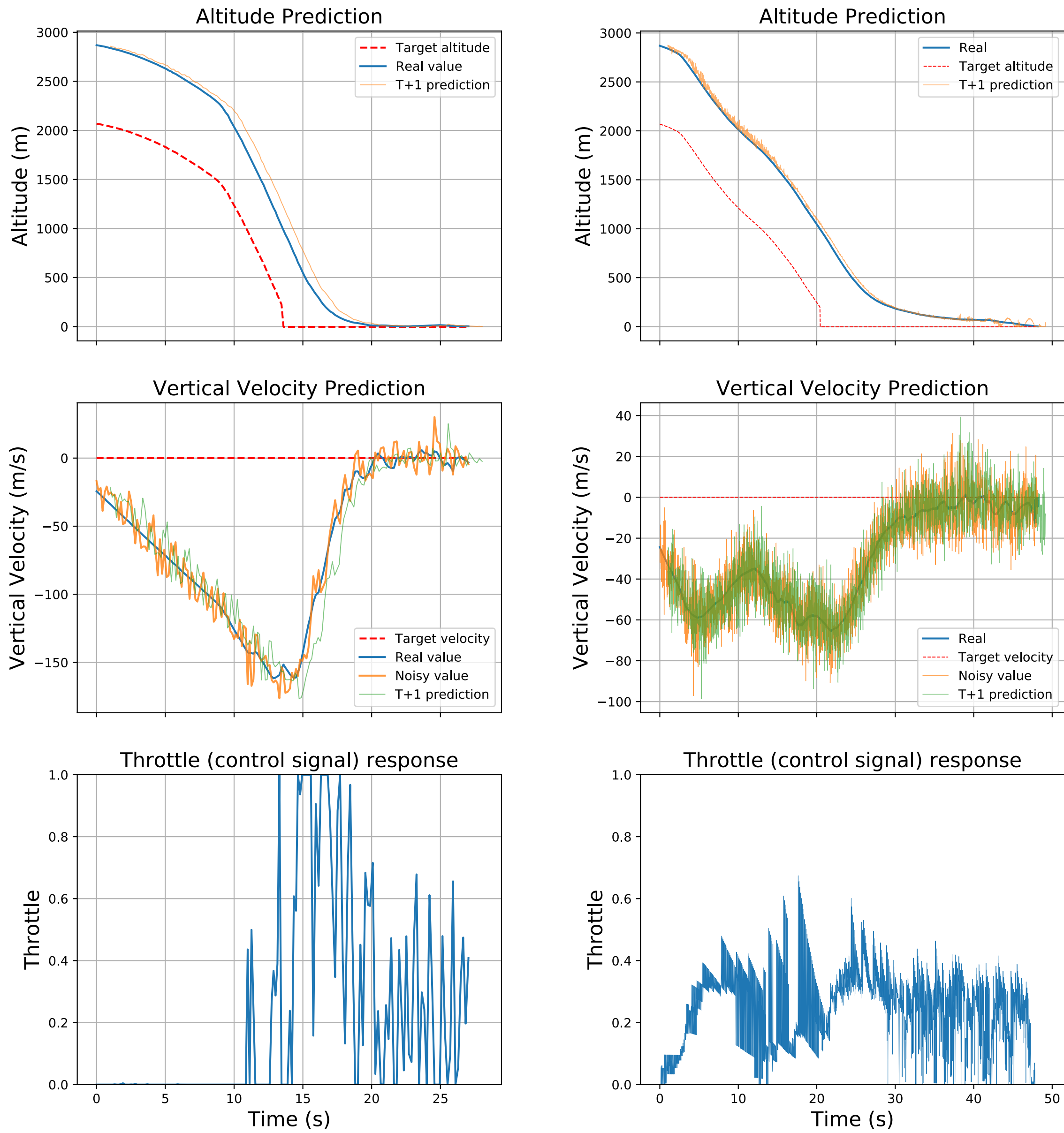
(a) Newtonian Model

(b) ML Models

Figure 3.4: Last moments of spaceship landing results from 3000 meters with two models. The speedometer is simulated with a standard deviation of 10 m/s Gaussian noise.

In our 3K speedometer noise simulation, we did not observe failure until the last few seconds of the landing. We visualized the last moments of the simulation that the crash happened in figure 3.4. In the Newtonian model, the spaceship hangs at 10 m altitude for a few seconds. The controller reached the velocity target at 44th second because of spikes in the noise, but it did not reach the altitude target. After that, a spike in the velocity causes the controller to input throttle, and the altitude rises. This unstabilized the system, including the yaw and pitch controller. Finally, the angle between the ground and the spaceship changed and caused the spaceship to crash. In ML models, we observed a similar result with the Newtonian model. The spaceship started to float around 5 m of altitude while reaching its velocity target. But, the spaceship destabilized while floating and crashed to the ground.

We visualized the whole landing process and included the control signal in figure 3.5. Noise in the sensor caused noise in altitude and vertical velocity prediction for both models. Compared to altitude noise simulation, altitude prediction was affected less for both models. The reason is velocity has little impact on altitude prediction in both models.

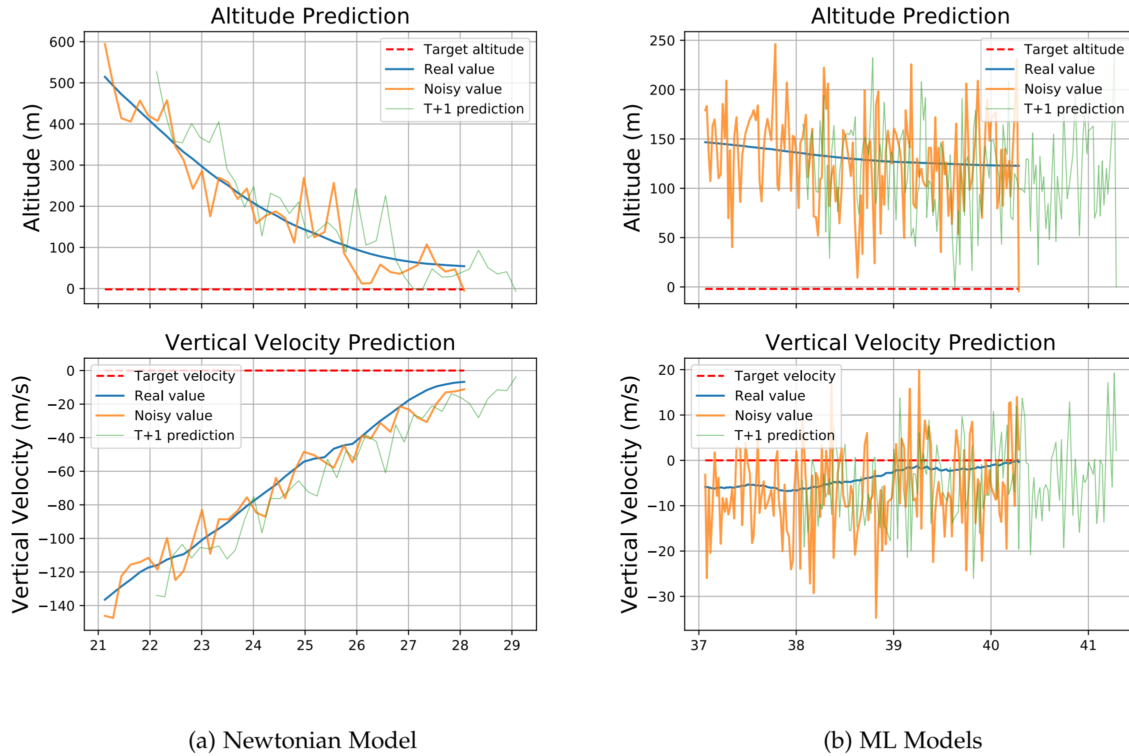


(a) Newtonian Model

(b) ML Models

Figure 3.5: Spaceship landing results from 3000 meters with two models. In both landings, speedometer (vertical velocity) is simulated with a standard deviation of 10 m/s Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3K Landing Results with Altimeter and Speedometer Noise



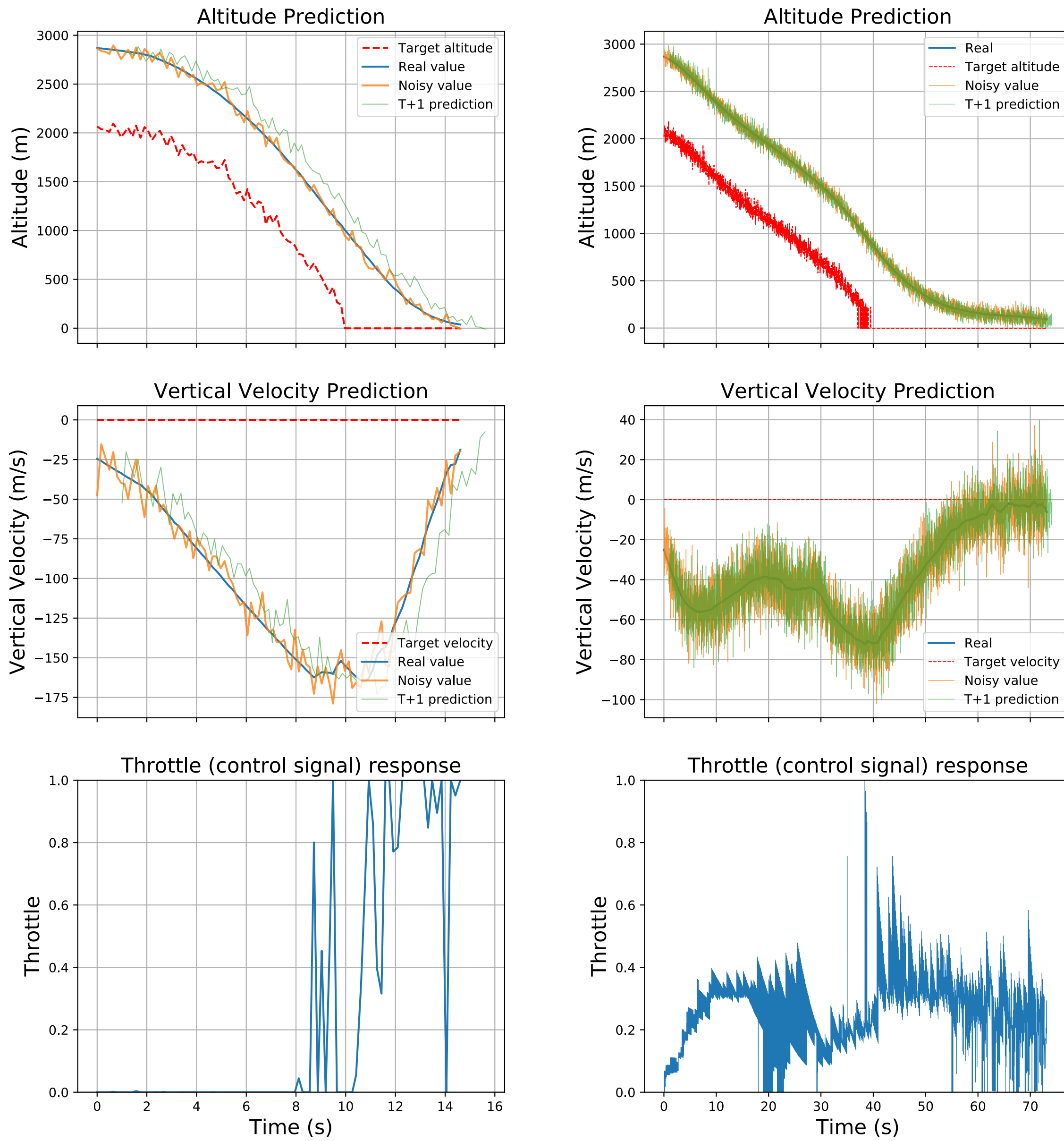
(a) Newtonian Model

(b) ML Models

Figure 3.6: Last moments of spaceship landing results from 3000 meters with two models. The speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise

We did not observe failure at the beginning of the landing simulation. We visualized the last moments of the simulation that the crash happened in figure 3.6. These landing results were similar to landing with only altimeter noise. In the Newtonian model, simulated altimeter noise caused a spike in the data, the altimeter sent the value of -6 m to the MPC and this caused the controller to disengage. Disengagement caused the spaceship to free-fall from that altitude without any controls and causing a crash. In the ML models, similar to the Newtonian model simulation, simulated altitude noise caused a spike in the data. Again, according to the controller, the target was reached and this caused the controller to disengage. This caused the spaceship to free-fall from that altitude without any controls.

The main difference between this simulation and only the altimeter noise simulation is predictions. In both models, velocity and altitude predictions were noisier. The whole simulation process with including the control signal visualized in figure 3.7.



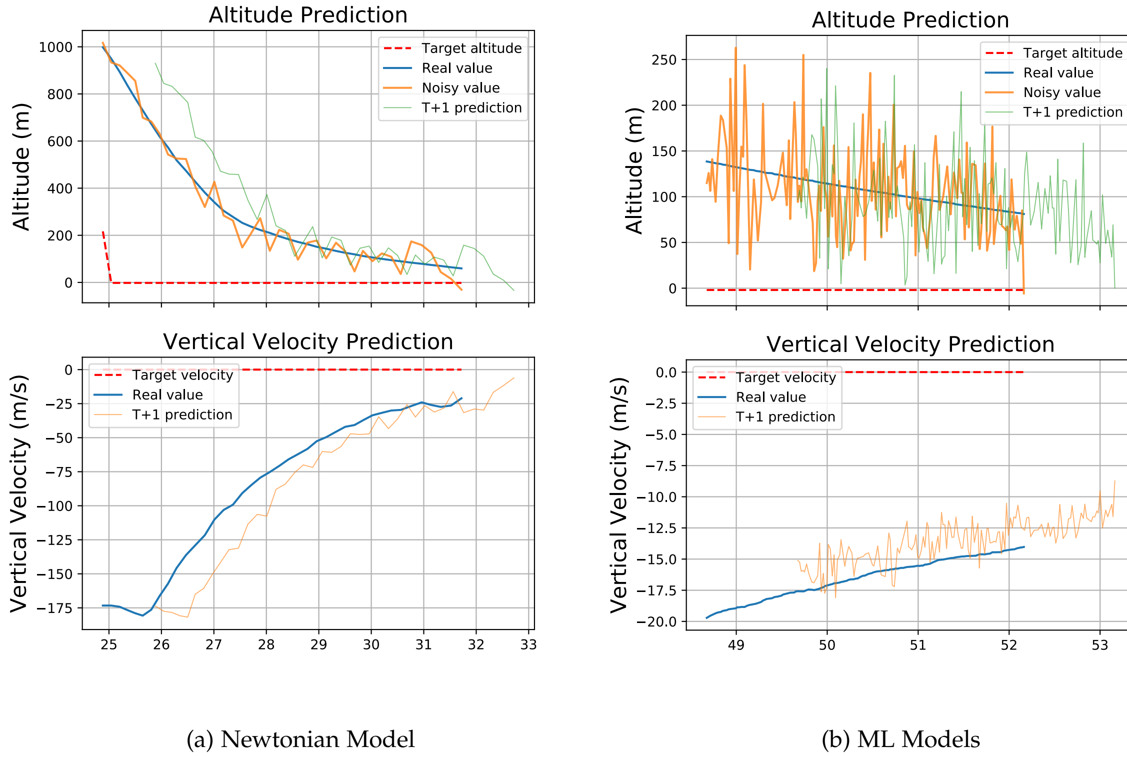
(a) Newtonian Model

(b) ML Models

Figure 3.7: Spaceship landing results from 3000 meters with two models. In both landings, speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3.3.1.2 8000 m Landing Simulation Results

8K Landing Results with Altimeter Noise



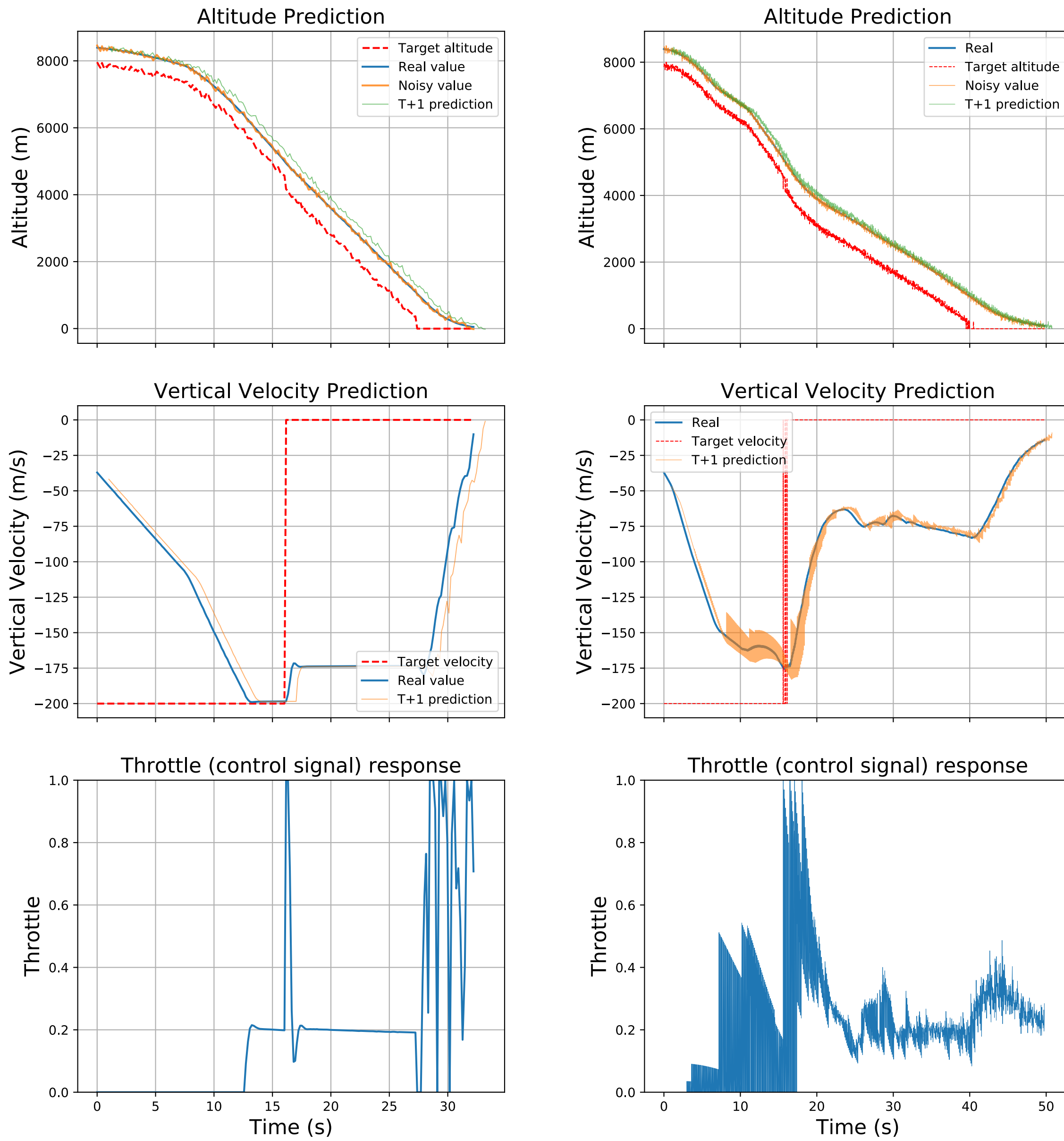
(a) Newtonian Model

(b) ML Models

Figure 3.8: Last moments of spaceship landing results from 8000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise.

We observed similar results to 3000 m landing in our simulation. The last few hundred meters of the landings were affected by the noise spikes. Since the last few hundred meters were disturbed, we visualized that part of the landing in figure 3.8.

In the Newtonian model, controller failure happened at the 32nd second. Simulated noise caused a spike in the data. According to the controller, the target was reached and disengaged. Controller disengagement caused the spaceship to free-fall from 80 m. In the ML models, controller failure happened at the 52nd second. Similar to the Newtonian model simulation, simulated noise caused a spike in the data, disengagement of the controller, and the free-fall from 70m. Similar to 3000 m flight, noise in the altimeter caused noisy predictions of both altitude and velocity for both models. The whole flight simulation with control signal values visualized in figure 3.9.

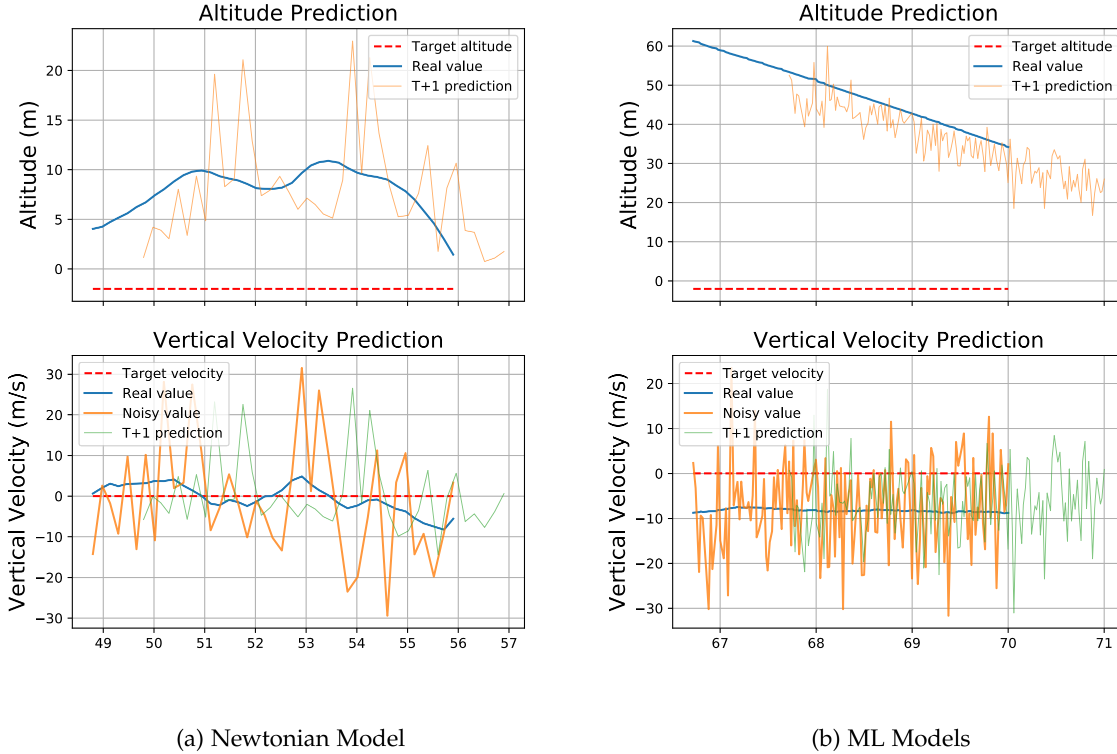


(a) Newtonian Model

(b) ML Models

Figure 3.9: Spaceship landing results from 8000 meters with two models. In both landings, the altimeter is simulated with a standard deviation of 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

8K Landing Results with Speedometer Noise



(a) Newtonian Model

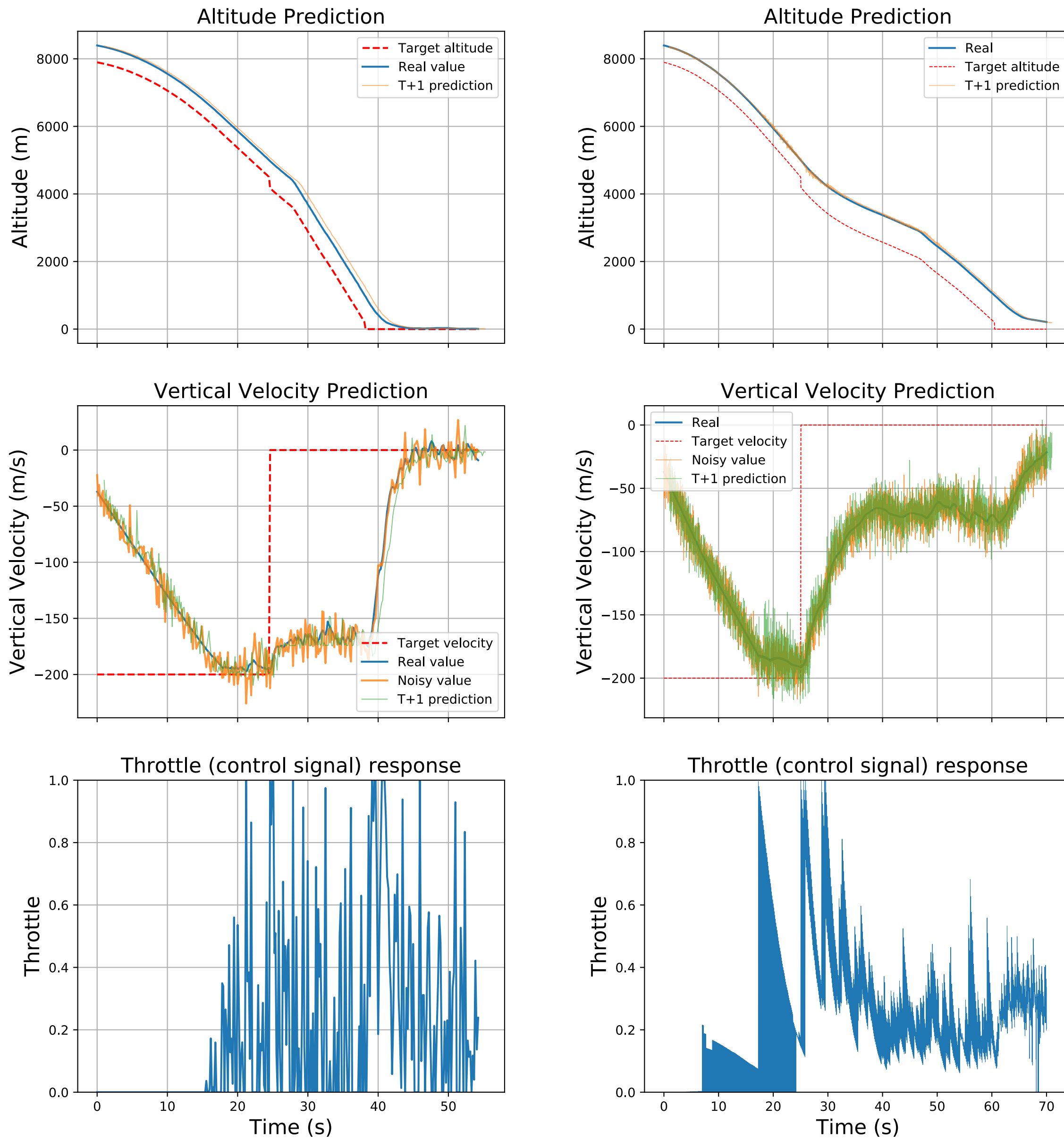
(b) ML Models

Figure 3.10: Last moments of spaceship landing results from 8000 meters with two models. The speedometer is simulated with a standard deviation of 10 m/s Gaussian noise.

In our 8K speedometer noise simulation, we observed similar results to our 3000 m simulation. We visualized the last moments of the simulation that the crash happened in figure 3.10.

In the Newtonian model, the spaceship floats at 10 m altitude. The controller reached the velocity target faster because of noise spikes, but it did not reach the altitude target. This unstabilized the system, including the yaw and pitch controller. Finally, the angle between the ground and the spaceship changed, the controller could not stabilize the spaceship and crash. In ML models, the spaceship velocity target was reached around -10 m/s. Similar to Newtonian model results, the system unstabilized, pitch and yaw changed and started to fall with a constant vertical speed.

We visualized the whole landing process and included the control signal in figure 3.5. Noise in the sensor caused noise in altitude and vertical velocity prediction for both models. Compared to altitude noise simulation, altitude prediction was affected less for both models.

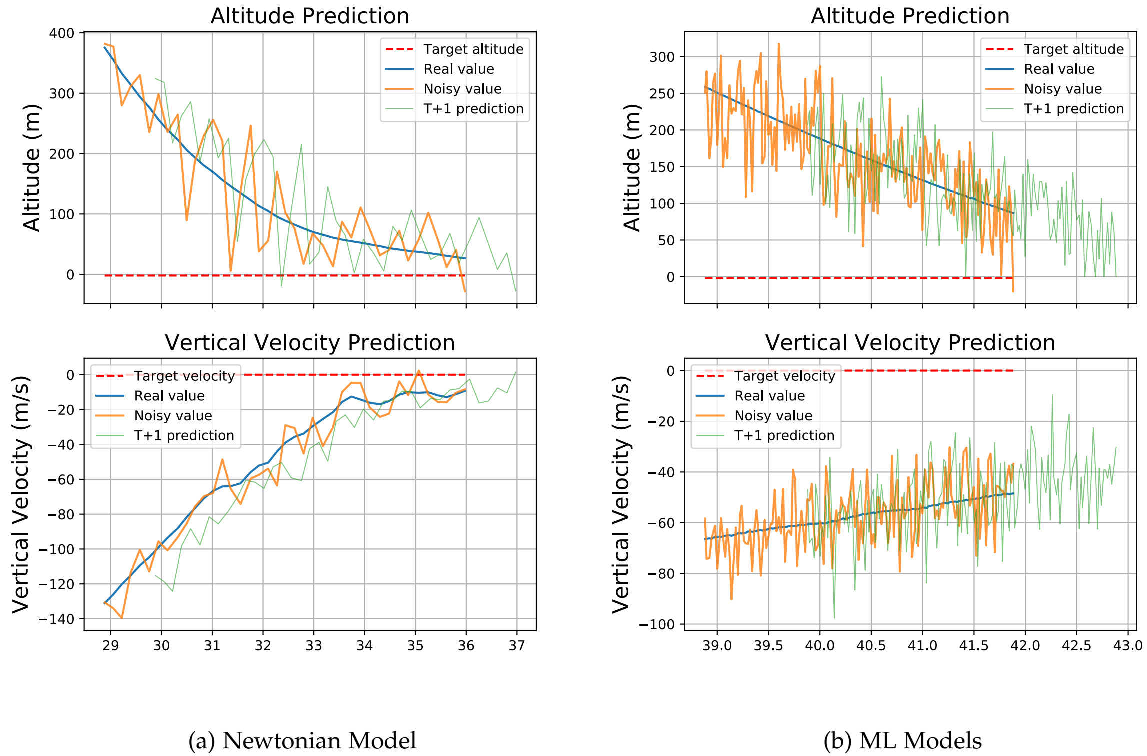


(a) Newtonian Model

(b) ML Models

Figure 3.11: Spaceship landing results from 8000 meters with two models. In both landings, speedometer is simulated with a standard deviation of 10 m/s Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

8K Landing Results with Altimeter and Speedometer Noise



(a) Newtonian Model

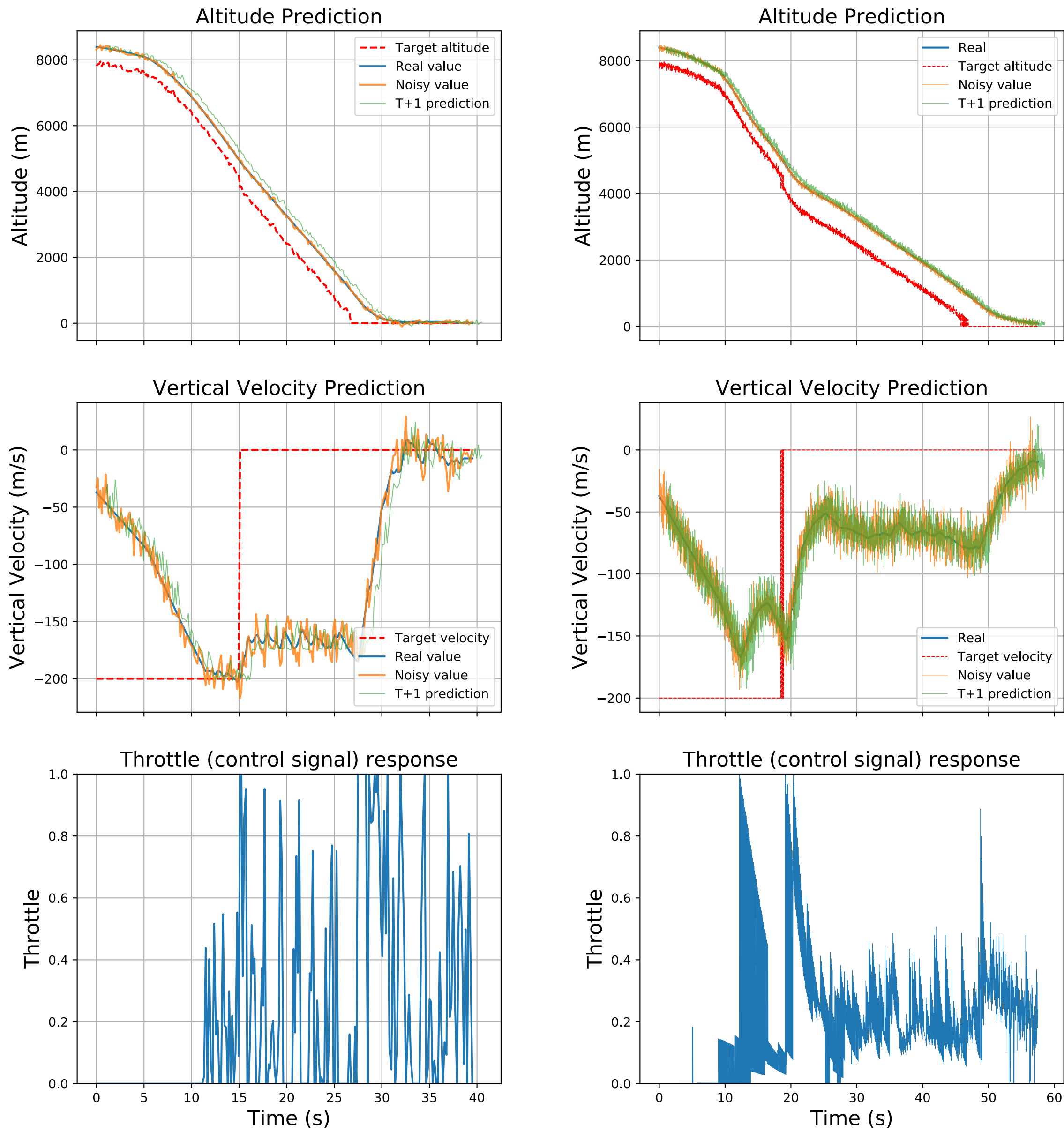
(b) ML Models

Figure 3.12: Last moments of spaceship landing results from 8000 meters with two models. The speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise

We observed similar results to 3000 m landing in our simulation. Noise spike disturbed the flight near the landing. we visualized that part of the landing in figure 3.12. These landing results were similar to landing with only altimeter noise.

In the Newtonian model, at 10 m altitude, simulated altimeter noise caused a spike in the data, the altimeter sent the value of -10 m to the MPC and this caused the controller to disengage and the free-fall resulting in a crash. In the ML models, similar to the Newtonian model, simulated altitude noise caused a spike in the data at 80 m from the ground. Again, according to the controller, the target was reached and this caused the controller to disengage. This caused the spaceship to free-fall from that altitude without any controls.

Similar to 3K landing, the main difference between this simulation and only the altimeter noise simulation is predictions. In both models, velocity and altitude predictions were affected. The whole simulation process with including the control signal visualized in figure 3.13.



(a) Newtonian Model

(b) ML Models

Figure 3.13: Spaceship landing results from 8000 meters with two models. In both landings, speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3.3.2 Landing Simulation Results with Failing Sensors

For this part of the simulation, we have created failures in the altimeter and speedometer. Since model predictions for the altitude and velocity depend on each other, firstly, we failed each one sensor separately and after, both sensors to observe the differences. All of the landings failed simulation rounds.

We failed the sensors on purpose to give constant output at some altitude levels. Altimeter and speedometer failure is simulated to output constant 0 (zero) value, and we observed the results. The summary of landing results can be seen in table 3.3 and table 3.4. Finally, we illustrated detailed simulations by comparing two models under faulty sensor data conditions.

Landing From	Altimeter Fault At	Speedometer Fault At	Result
3000 m	1000 m	-	Floating at 500 m
3000 m	-	1000 m	Crash while accelerating
3000 m	1000 m	1000 m	Crash while decelerating
8000 m	3000 m	-	Floating at 2800 m
8000 m	-	3000 m	Crash while accelerating
8000 m	3000 m	3000 m	Crash while decelerating

Table 3.3: Naive MPC simulation results with faulty sensors.

Landing From	Altimeter Fault At	Speedometer Fault At	Result
3000 m	1000 m	-	Floating at 600m
3000 m	-	1000 m	Crash while accelerating
3000 m	1000 m	1000 m	Crash while decelerating
8000 m	3000 m	-	Floating at 1600m
8000 m	-	3000 m	Crash while accelerating
8000 m	3000 m	3000 m	Crash while accelerating

Table 3.4: ML models simulation results with faulty sensors.

3.3.2.1 3000 m Landing Simulation Results

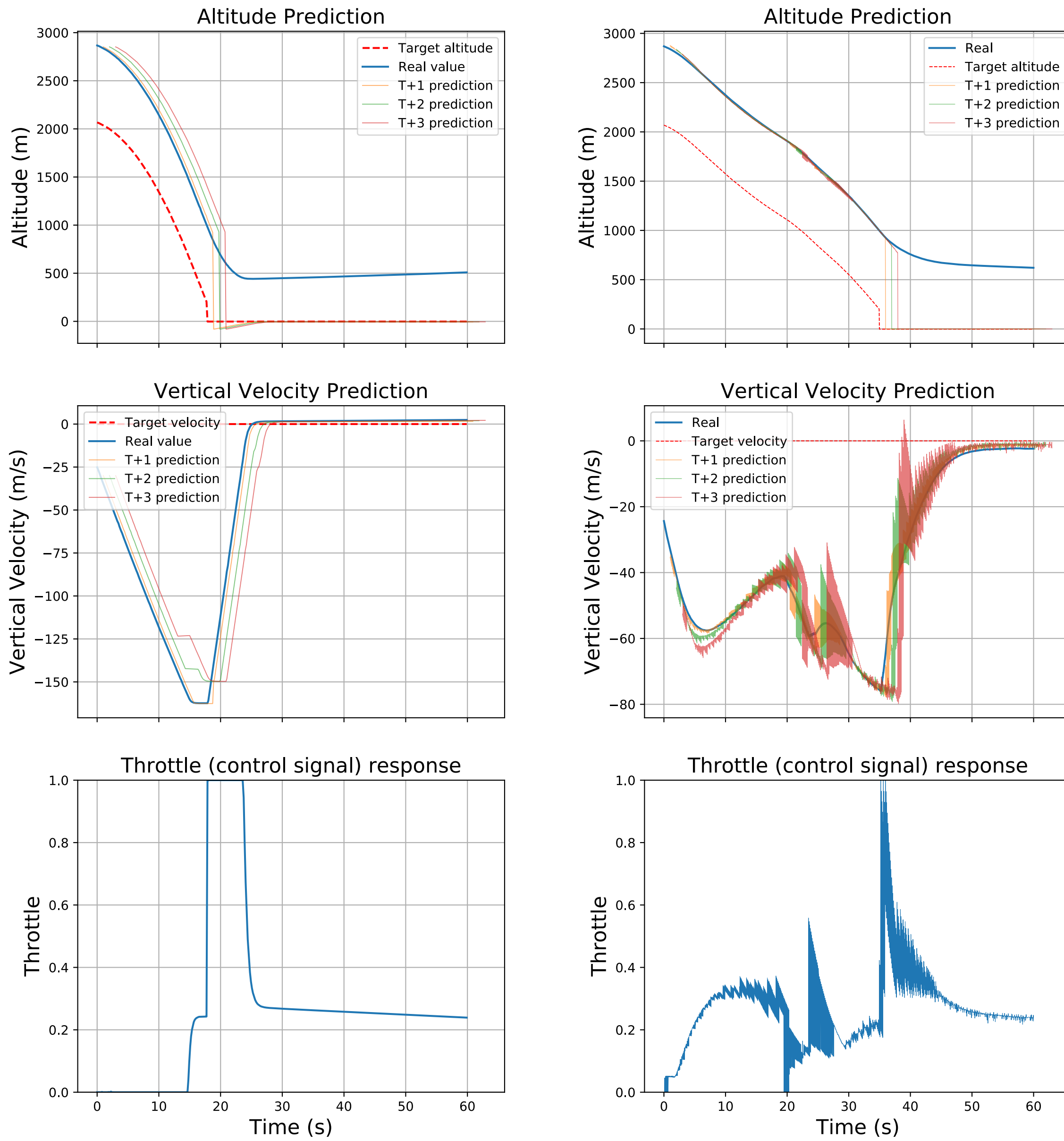
3K Landing Results with Failing Altimeter

In this part of the simulation, we have failed the altimeters for both controllers and observed the results. Failure simulation was done by giving constant zero (0) output for the altimeter. We failed the altimeters when the spaceship reached 1000 m altitude for both controller landing scenarios. Finally, total flight simulation data for both models are visualized in figure 3.14.

In our naive MPC simulation round, we observed predictable results. The Newtonian model predictions for both altitude and vertical velocity were robust and stable until the failure. The altimeter failure happened at the 18th second of the flight, and the model predictions were immediately affected at this point. As expected, The Newtonian model predictions changed to zero (0) after the altimeter failure. The controller assumes that it reached its target altitude, thus, completed the landing process. After satisfying the altitude target, the control function also tries to reach its velocity target. The controller maximized the throttle to reach its velocity target at the 25th second. The spaceship started to slow down at 1000 m altitude and balanced itself at 500 m after reaching the velocity target. Starting from the 26th second of the landing, the controller reached both velocity and altitude targets and maintained the current state.

In our ML models simulation round, we observed similar results to the Newtonian model flight simulation. ML models' predictions for both altitude and vertical velocity were again robust and stable until the failure. But, compared to the Newtonian model, the vertical velocity predictions and the control signal of ML models were noisier. The reason for the velocity prediction noise is the nature of the predictor for this model. The altimeter failure happened at the 37th second of the flight, and at this point, the model predictions were affected. The ML models' predictions changed to zero (0) after the altimeter failure. Again, the controller assumed that it reached its target altitude. After satisfying the altitude target, the control function also tried to get to its velocity target. The controller maximized the throttle in a short time to reach its velocity target at the 37th second. The spaceship started to slow down at 1000 m altitude and balanced itself at 600 m after reaching the velocity target. Starting from the 47th second of the landing, the controller reached both velocity and altitude targets and maintained the current state.

The total flight period was the main difference between the two models. The ML models' flight was longer than the Newtonian model. The reason for that is the difference between the cost functions between these two controllers. The Newtonian model includes the throttle value in cost function in contrast to the ML model.



(a) Newtonian Model

(b) ML Models

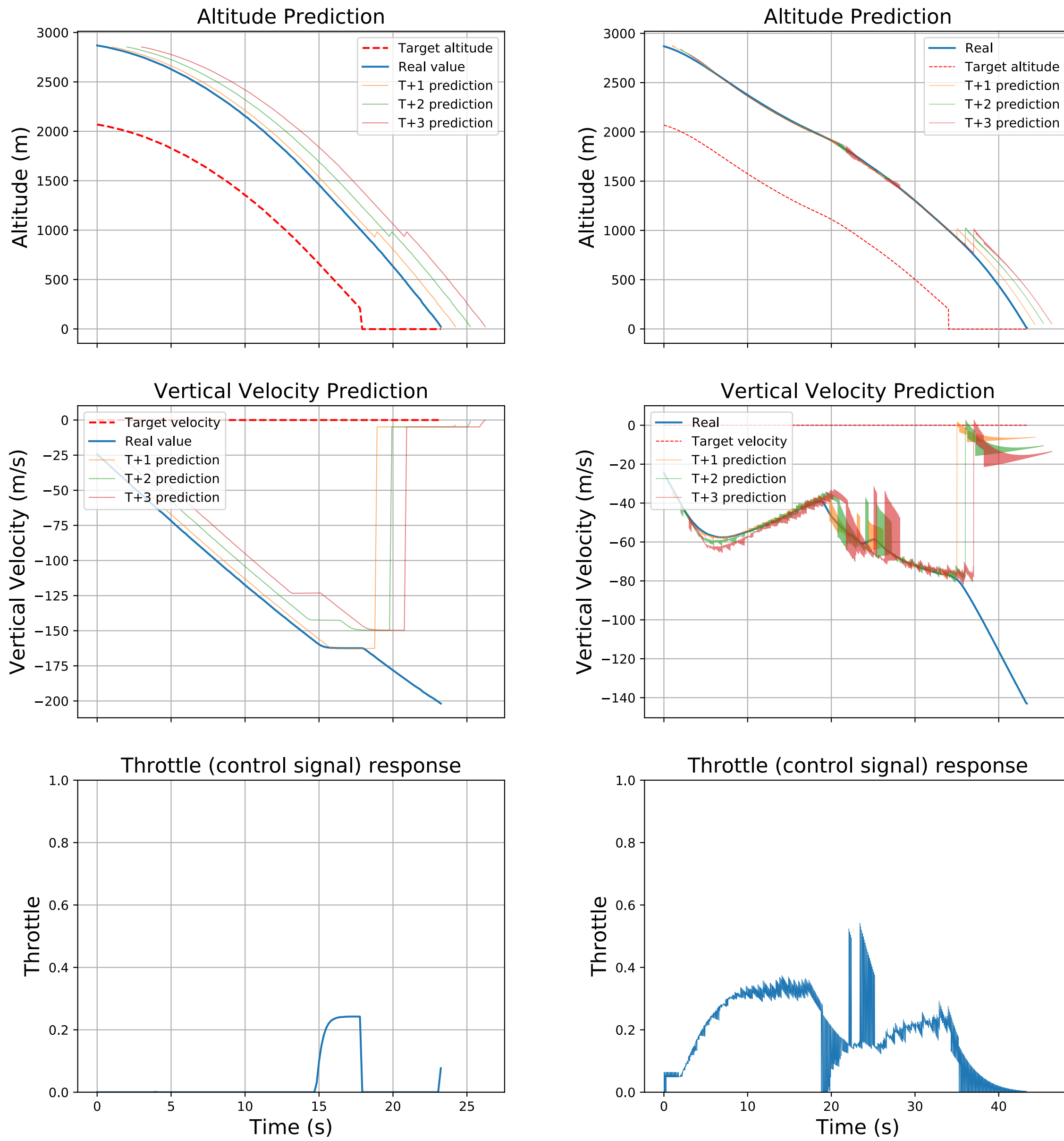
Figure 3.14: Spaceship landing results from 3000 meters with two models. In both landings, altimeter failure happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3K Landing Results with Failing Speedometer

In this part of the simulation round, we have failed the speedometers for both controllers and observed the results. Failure simulation was done by giving constant zero (0) output for the speedometer. We failed the speedometers when the spaceship reached 1000 m altitude for both controller landing scenarios. Finally, total flight simulation data for both models are visualized in figure 3.15.

In our naive MPC simulation round, we observed different behavior than the altimeter failure simulation results. Until the failure, the Newtonian model predictions for both altitude and vertical velocity were steady. The speedometer failure happened at the 18th second of the flight, and the model predictions were immediately affected at this point. As expected, The Newtonian model vertical velocity predictions changed to zero (0) after the speedometer failure. The controller estimates it has achieved its target velocity, further, altitude predictions increased. The cause of an increase in altitude prediction is the effect of vertical velocity on altitude predictions. When vertical speed is low, the predicted altitude is high in the next step. The controller activated the throttle at the 15th second and tried to maintain a regular landing. After 3 seconds, the failure happened, and at this point, the controller deactivated the throttle, predicting the current speed is close to zero (0). The cost function is responsible for this behavior; vertical velocity has a greater impact than altitude in the cost function, therefore the cost function had already a small value. The controller did not minimize the cost function further and this caused no output signal (throttle) until the crash. Finally, the spaceship free-fall starting from 1000 m altitude and crashed while accelerating.

In our ML models simulation round, we observed similar results to the Newtonian model flight simulation. Until the failure, the ML models' forecasts for both altitude and vertical velocity were robust and reliable. However, as compared to the Newtonian model, the ML models' vertical velocity estimates and control signal were noisier again. The speedometer failure happened at the 34th second of the flight, and at this point, the model's velocity prediction was zero (0). Again, the controller assumed that it reached its target speed. From this point of the flight similar to the Newtonian model chain of events happened. Altitude prediction was higher than the previous step because velocity is zero (0). The throttle was cut off slowly by the controller since the cost was already low, thus, this caused the spaceship to free-fall from 1000 m altitude. Finally, the spaceship crashed to the ground while it was accelerating.



(a) Newtonian Model

(b) ML Models

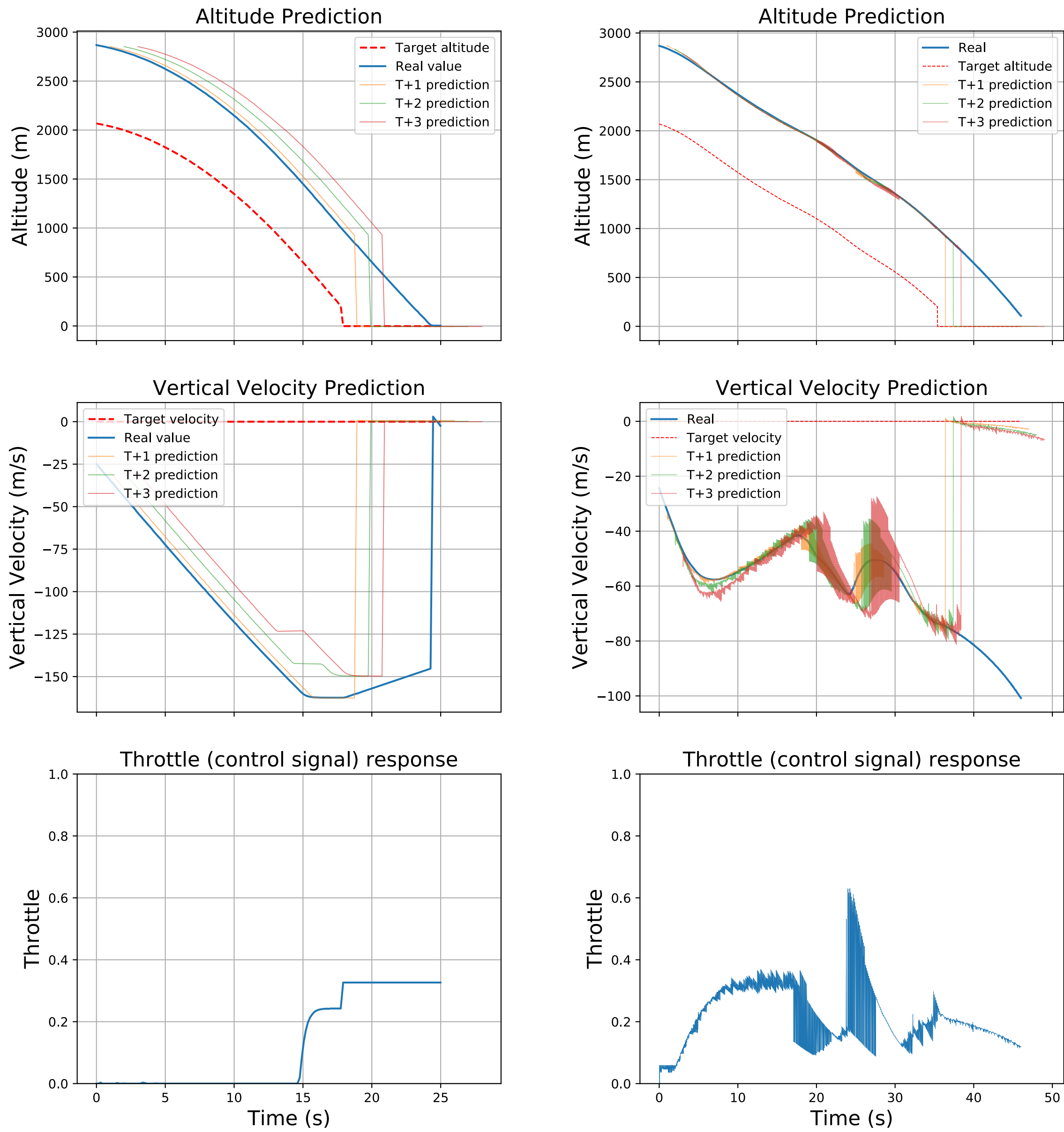
Figure 3.15: Spaceship landing results from 3000 meters with two models. In both landings, speedometer (vertical velocity) happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

3K Landing Results with Failing Altimeter and Speedometer

In this part of the simulation round, we have failed the altimeter and speedometer for both controllers and observed the results. Failure simulation was done by giving constant zero (0) output for both the speedometer and the altimeter. We failed both of the sensors when the spaceship reached 1000 m altitude for both controller landing scenarios. Finally, total flight simulation data for both models are visualized in figure 3.16.

In our naive MPC simulation round, we observed that the Newtonian model predictions for both altitude and vertical velocity were steady. Sensor failures happened at the 18th second of the flight, and the model predictions were immediately affected at this timestamp. As expected, both altitude and velocity predictions changed to zero (0) after failures. At the point of failure, both of the predictions satisfied their target values, and this caused the controller to stop optimizing. Just before the failures, the throttle value is optimized for the value of 0.32 to continue its regular landing. However, the controller was passive from that point and the throttle remained the same as before (the spaceship throttle value does not change without additional input). This thrust was enough to slow down the spaceship, and velocity decreased from -170 m/s to -140 m/s. But, the spaceship without any controls, crashed to the ground at the 24th second of the flight while deaccelerating.

In our ML models simulation round, we again observed similar results to the Newtonian model flight simulation. Until the failure, the ML models' predictions for both altitude and vertical velocity were robust. In addition, compared to the Newtonian model, the ML models' vertical velocity estimates and control signal were noisier again. The speedometer and the altimeter failures happened at the 36th second of the flight, and at this point, the model's velocity and altitude predictions were near zero (0). Again, the controller assumed that it reached its target speed and altitude. As same as the Newtonian model simulation, the controller stopped optimizing the cost function. The main difference between this simulation and the Newtonian model was that the ML model was slower at the point of failure because of more frequent thrust inputs. However, at the time of failures, the current thrust was not enough to decelerate the spaceship, and finally, the spaceship crashed to the ground while it was accelerating.



(a) Newtonian Model

(b) ML Models

Figure 3.16: Spaceship landing results from 3000 meters with two models. In both landings, speedometer and altimeter failures happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

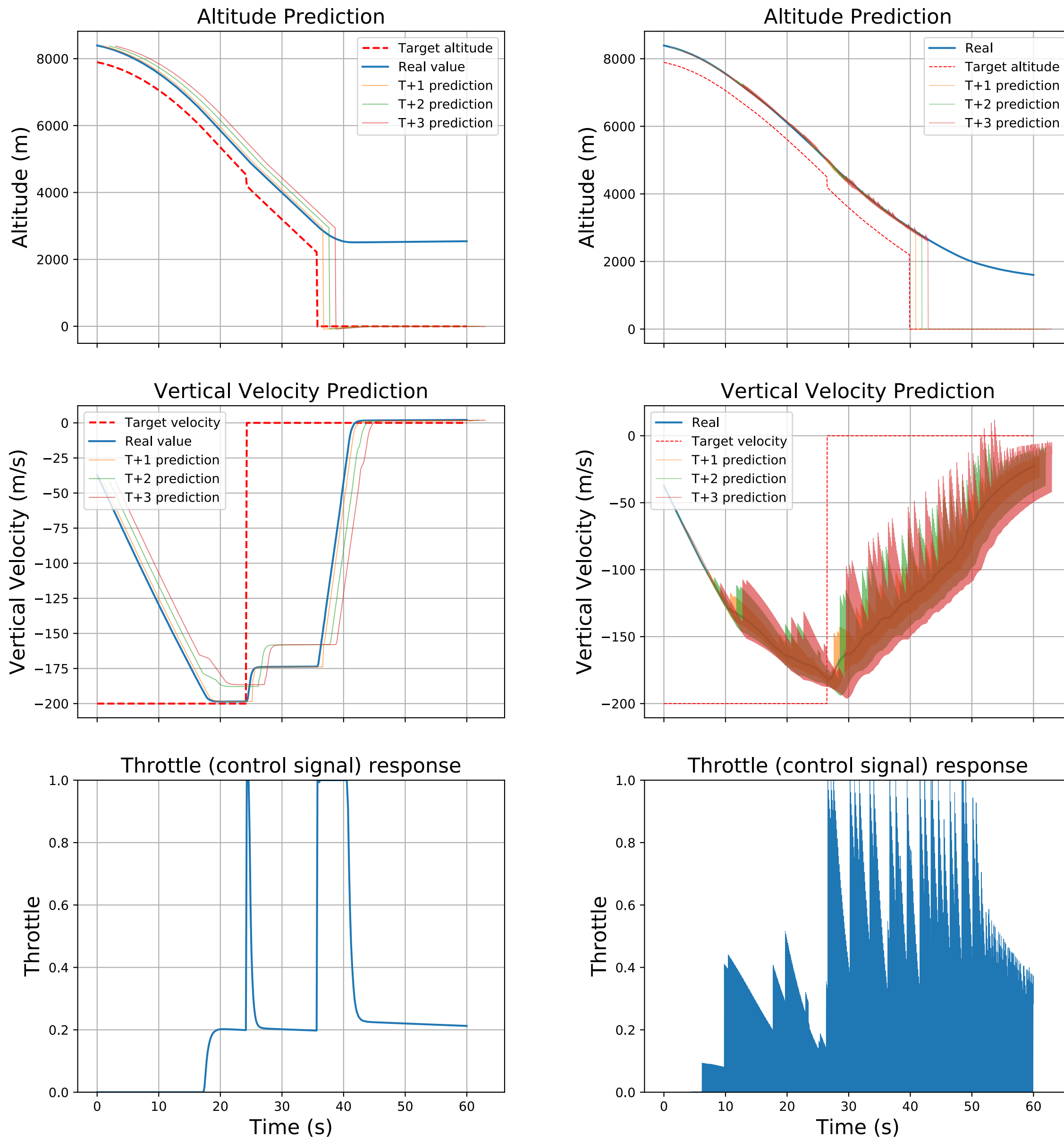
3.3.2.2 8000 m Landing Simulation Results

8K Landing Results with Failing Altimeter

In this part of the simulation, we have failed the altimeters for both controllers and observed the results. Failure simulation was done by giving constant zero (0) output for the altimeter. We failed the altimeters when the spaceship reached 1000 m altitude for both controller landing scenarios. Finally, total flight simulation data for both models are visualized in figure 3.17.

In our naive MPC simulation round, we observed similar to 3K flight results. The Newtonian model predictions for both altitude and vertical velocity were robust and stable until the failure. The controller maximized the throttle when the target velocity has changed at the 24th second of flight and the landing continued. The altimeter failed at the 37th second of the flight, and the model forecasts were instantly impacted and caused the model predictions to change to zero (0) after the altimeter failure. The controller estimated it has achieved its target altitude and has so completed the landing operation. After reaching the altitude target, the control function attempted to achieve the velocity target as well. The controller increased the throttle to achieve the desired velocity at the 38th second. After attaining the velocity target, the spaceship began to slow down at 3000 m altitude and balanced itself at 2800 m. The controller attained both velocity and altitude targets and maintained the present state from the 42nd second of the landing.

We observed comparable findings to the Newtonian model flight simulation in our ML models simulation round. ML models' predictions for both altitude and vertical velocity were again robust enough to land safely until the failure. However, as compared to the Newtonian model, ML models' vertical velocity estimates and control signal were noisier. The velocity prediction was much noisier when compared to the 3K flight and when the vertical velocity target had changed, predictions became much noisier. In addition, in contrast to the Newtonian model, vertical velocity was linear and constantly slowing. The altimeter failed in the 41st second of the flight, and the model projections were disrupted at this moment. Following the altimeter failure, the ML models' forecasts became zero (0). The controller assumed it had reached its target altitude. After satisfying the altitude target, the control function also tried to reach its velocity target as soon as possible in a linear fashion. The spaceship started to slow down and balanced itself at 1600 m after reaching the velocity target. The controller met both velocity and altitude targets and maintained the current state starting at the 59th second of the landing.



(a) Newtonian Model

(b) ML Models

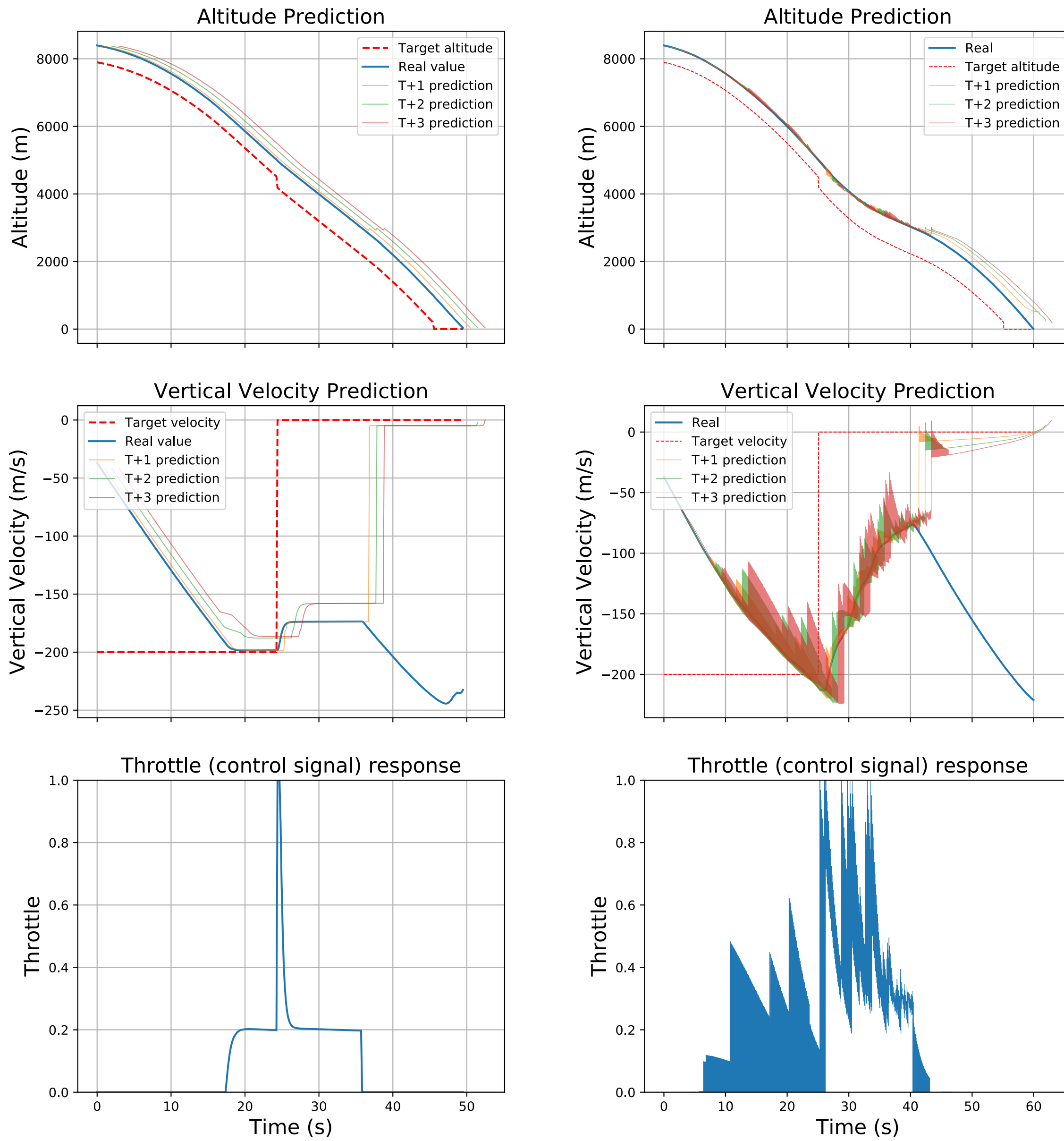
Figure 3.17: Spaceship landing results from 8000 meters with two models. In both landings, altimeter failure happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

8K Landing Results with Failing Speedometer

We failed the speedometers for both controllers in this section of the 8K landing simulation run and observed the result. The speedometer was given a steady zero (0) output to simulate failure. For both controller landing scenarios, the speedometers failed after the spaceship reached 3000 m altitude. Finally, figure 3.18 depicts the overall flight simulation data for both models.

We observed different behavior in our naive MPC simulation round than the altimeter failure simulation results. The Newtonian model's predictions for altitude and vertical velocity were stable until the failure. At the 24th second of the landing, the vertical velocity target has changed and throttle output was maximized by the controller and until the time of failure, the spaceship maintained a constant speed. The altimeter failed on the 38th second of the flight, causing the model forecasts to be affected immediately. After the speedometer failed, the Newtonian model vertical velocity predictions changed to zero (0), as expected. The controller estimates it has achieved its target velocity, similar to the 3K landing scenario, and altitude projections were higher than a usual flight. The effect of vertical velocity on altitude estimations is the reason for an increase in altitude prediction. The controller deactivated the throttle at the time of failure, predicting that the current speed was close to zero (0). The cost function is responsible for this behavior; vertical velocity has a bigger impact on the cost function than altitude, hence the cost function had a minimal value. The cost function was not further minimized by the controller, resulting in no output signal (throttle) until the crash. Finally, the spaceship began a free fall from a height of 3000 meters and crashed as it accelerated.

We noticed similar findings in our ML model flight simulation as in our Newtonian model flight simulation. Until the failure, the ML models' forecasts for both altitude and vertical velocity were stable for a regular landing process. At the 25th second of landing, the vertical velocity target has changed to zero(0). Thus, throttle output was increased by the controller linearly and until the time of failure and the spaceship decelerated to its target speed. The ML models' vertical velocity estimates and control signal were noisier than the Newtonian model. The speedometer failed in the 42nd second of the flight, and the model's velocity prediction was zero at that time (0). The controller assumed it had reached its targeted speed again. A chain of events occurred from this point in the flight, comparable to the Newtonian model. Because velocity is zero (0), the altitude projection was higher than the prior stage. Because the cost was already minimal, the controller gradually reduced the throttle, causing the spaceship to free-fall from 3000 meters altitude. Finally, the spaceship collided with the ground as it was speeding up.



(a) Newtonian Model

(b) ML Models

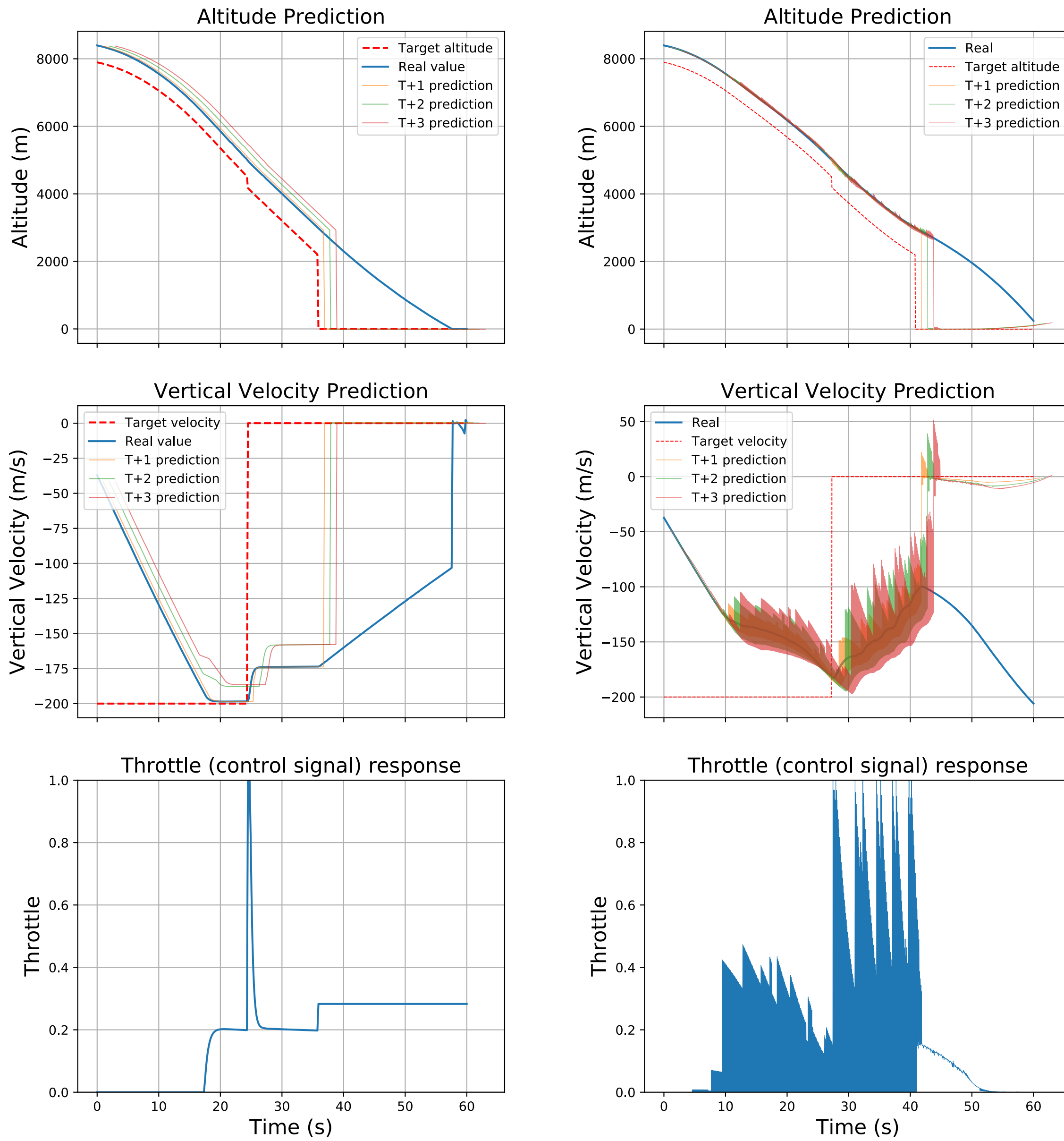
Figure 3.18: Spaceship landing results from 8000 meters with two models. In both landings, speedometer (vertical velocity) happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

8K Landing Results with Faulty Altimeter and Speedometer

We failed the altimeter and speedometer for both controllers in this section of the 8K landing simulation round and observed the result. Both the speedometer and the altimeter were given constant zero (0) output to simulate failure. For both controller landing scenarios, we failed both sensors when the spaceship reached 3000 m altitude. Finally, figure 3.16 depicts the overall flight simulation data for both models.

We noticed that the Newtonian model predictions for both altitude and vertical velocity were accurate in our Newtonian MPC simulation run. At the 36th second of the flight, sensor failures occurred, and the model predictions were instantly altered. After failures, both altitude and velocity forecasts were set to zero(0), as expected. Both predictions satisfied their target values at the point of failure, causing the controller to stop optimizing. The throttle value was optimized at a value of 0.3 just before the failures to require the plane to land safely. However, the controller became inactive at that time, and the throttle stayed unchanged. The spaceship's velocity dropped from -175 m/s to -110 m/s as a result of this throttle. However, the spaceship, which had no controls, crashed to the ground while decelerating at the 57th second of the flight.

We found analogous findings in our ML model flight simulation as we observed in our Newtonian model flight simulation. Until the failure, the ML models' altitude and vertical velocity estimates were accurate. Furthermore, the ML models' vertical velocity estimates and control signal were noisier than the Newtonian model. At the 42nd second of the flight, the speedometer and altimeter failed, and the model's velocity and altitude predictions were close to zero (0). Again, the controller assumed that it reached its target speed and altitude. As same as the Newtonian model simulation, the controller stopped optimizing the cost function and the control signal vanished slowly. Because of the more frequent thrust inputs, the ML model was slower at the point of failure than the Newtonian model. However, the present thrust was insufficient to slow the spaceship during the failures, and the spaceship eventually crashed to the ground while accelerating.



(a) Newtonian Model

(b) ML Models

Figure 3.19: Spaceship landing results from 8000 meters with two models. In both landings, speedometer and altimeter failures happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs.

4 Discussion

We have tested many failing and noisy sensors in our landing simulations. There are methods to solve these problems in practice. For noise in sensors, there are filtering methods such as Kalman Filter. There are applications in literature to incorporate the Kalman filter into MPC [30]. The Kalman filter is an algorithm that estimates unknown variables by using a set of data observed over time that contains noise and other imperfections. R. E. Kalman proposed it in 1960 [31], and it has since become a common approach for an optimal estimate. Because of its advantages of real-time, fast, efficient, and powerful anti-interference, the Kalman filter has been widely used in orbit computation, target tracking, and navigation, such as calculations of spacecraft orbit, tracking of maneuvering targets, and GPS positioning [32].

We have approached the noise in sensor problem with a Kalman Filter solution. Because of the time constraints, we could not finish and conclude this part of the project, but, we are sharing our current results in figure 4.1. In this figure, we have visualized filtered altimeter noise from one of our simulation rounds. Since Kalman Filter requires a model of a system, we have used our existing model and extended the model with environment and measurement noise. We have tested our model on offline flight data results and saw promising results. However, since we have used an offline method and Kalman Filter adds another complexity to our critical system, online usage of this method is debatable and requires extensive simulations.

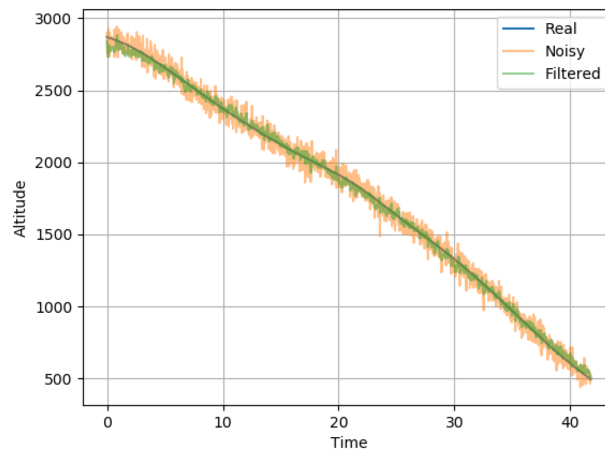


Figure 4.1: Filtered altitude sensor noise results. Kalman Filter is used for filtering process.

For our failing sensor simulations, finding a solution is harder. The trivial solution is implementing a hardware redundancy, but this may not be possible in every dynamic system. While robust MPC solutions [12] exist to mitigate the effects of external disturbances, there are no clear MPC strategies to account for the model mismatch. Sensor faults are a source of modeling error that can be avoided by applying fault-tolerant control (FTC) techniques.

Firstly, we need to generate robust residuals to detect faults. We can design a robust filter or an estimator that generates residuals reactive to faults. There are methods for this purpose such as fault detection filters, observer-based methods, and Kalman filter-based methods [26]. But, fault isolation is not guaranteed, thus, residuals should distinguish between faults. In our simulation case, it can be an altimeter failure, speedometer failure, or both. We can incorporate a filter and can check for differences between the predicted model state and the observed state.

The next step is determining if there is any fault in the system and the type of the fault. The trivial approach can be deciding a fault occurred if the value of a residual exceeds a threshold. In our case, it can be an unexpectedly big difference between the current altitude and measured altitude, so we can assume that the altimeter is faulty.

The final step of our method is reconfiguration. It is important to determine the best control actions after a failure and its detection. So, we need to reconfigure the controller online in response to the failures. One common approach is the multiple-model implementation for this problem and it has applications in MPC [33]. Thus, we can also employ this approach to our problem. We can create a set of different models to describe the system under normal operation and fault conditions. We will also employ a controller designed for each model specifically. In addition, a switching mechanism will be designed to determine the model of the system (faulty or not) and select the corresponding controller. In our case, we will have a model for altimeter failure, a speedometer failure, and both. We will detect the fault, isolate it and finally change our model and controller corresponding to that fault.

We have talked about modeling the system, but these models can also be replaced by reinforcement learning techniques. This way, robustness can be achieved with the same methods that are mentioned.

5 Conclusion

We simulated numerous landing scenarios with faulty sensors using two different Model Predictive Controllers. We compared the results and visualized our findings. In general, we found out that controllers with the Newtonian model and the ML models can land successfully when system information is perfectly available. However, when there are disturbances in the system, both of these controllers fail to land the spaceship.

We have implemented new features for controllers to introduce noisy and failing sensors to the system. We created similar landing scenarios that Atukalp[6] and Ganbarov[5] did in their earlier work. Our noisy landings showed that both of these controllers are capable of continuing to land until the critical moment of touchdown. However, since controllers depend on perfect data, they have failed to finish the landing because of spikes in the noise.

In our failing sensor simulations, we have created scenarios to observe and compare every combination of sensor failures. We have failed the altimeter, the speedometer, and two of the sensors together and explained the results. We observed different results for every failure type. We concluded that both of the controllers were not able to handle missing or wrong data in the system as expected, and landing simulations ended in various failures.

In future work, FDI methods can be implemented to detect and isolate a disruption in the system. In noisy sensors case, there are that methods can be used to filter the noise, such as Kalman Filter, to stabilize the system. In our failure case, adaptive predictive controller methods can be used when there is a failure to change the model or the cost function. Finally, models can be extended by reinforcement learning to include noisy and faulty scenarios.

Bibliography

[1] L. Z. Zhang Bojun and L. Gang. "High-Precision Adaptive Predictive Entry Guidance for Vertical Rocket Landing". In: *Journal of Spacecraft and Rockets* (2019).

[2] S. Di Cairano and H. Tseng. "Driver-assist steering by active front steering and differential braking: Design, implementation and experimental evaluation of a switched model predictive control approach". In: (2010), pp. 2886–2891.

[3] S. Gros, R. Quirynen, and M. Diehl. "Aircraft control based on fast non-linear MPC and multiple-shooting". In: (2012), pp. 1142–1147.

[4] M. F. Torsten Trimborn Lorenzo Pareschi. "Portfolio optimization and model predictive control: A kinetic approach". In: *Discrete and Continuous Dynamical Systems - B* 24.11 (2019), pp. 6209–6238.

[5] A. Ganbarov. *Autonomous spaceship navigation and landing using Model Predictive Control*. Technical University of Munich, Munich, Germany, 2020.

[6] K. Atukalp. *Automated Feature Selection and Learning of a Spaceship Model for Model Predictive Control*. Technical University of Munich, Munich, Germany, 2021.

[7] J.-j. Xue, Y. Wang, H. Li, X.-f. Meng, and J.-y. Xiao. "Advanced Fireworks Algorithm and Its Application Research in PID Parameters Tuning". In: *Mathematical Problems in Engineering* 2016 (Jan. 2016), pp. 1–9.

[8] K. R. Muske and J. B. Rawlings. "Model predictive control with linear models". In: *Aiche Journal* 39 (1993), pp. 262–287.

[9] G. A. Michèle Arnold. *Model Predictive Control of energy storage including uncertain forecasts*.

[10] C. E. García, D. M. Prett, and M. Morari. "Model predictive control: Theory and practice—A survey". In: *Automatica* 25.3 (1989), pp. 335–348.

[11] M. Yousuf, H. Al-Duwaish, and Z. Hamouz. "PSO Based Nonlinear Predictive Control of Single Area Load Frequency Control". In: (Aug. 2021).

[12] A. Bemporad and M. Morari. *Robust model predictive control: A survey*. Ed. by A. Garulli and A. Tesi. London: Springer London, 1999, pp. 207–226.

[13] J. B. Rawlings, D. Q. Mayne, and M. M. Diehl. *Model Predictive Control: Theory, Computation and Design*. Nob Hill Publishing, Feb. 2019. URL: http://www.nobhillpublishing.com/mpc-paperback/index-mpc.html.

[14] P. J. Campo and M. Morari. "Robust Model Predictive Control". In: *1987 American Control Conference* (1987), pp. 1021–1026.

[15] D. A. Copp and J. P. Hespanha. "Nonlinear output-feedback model predictive control with moving horizon estimation". In: (2014), pp. 3511–3517. DOI: 10.1109/CDC.2014.7039934.

[16] U. Yüzgeç, Y. Becerkli, and M. Türker. "Dynamic Neural-Network-Based Model-Predictive Control of an Industrial Baker's Yeast Drying Process". In: *Neural Networks, IEEE Transactions on* 19 (Aug. 2008), pp. 1231–1242. DOI: 10.1109/TNN.2008.2000205.

[17] D. Mayne. "Control of Constrained Dynamic Systems". In: *European Journal of Control* 7.2 (2001), pp. 87–99. ISSN: 0947-3580. DOI: https://doi.org/10.3166/ejc.7.87-99. URL: https://www.sciencedirect.com/science/article/pii/S0947358001711417.

[18] D. Mayne. "Robust and Stochastic MPC: Are We Going In The Right Direction?" In: *IFAC-PapersOnLine* 48.23 (2015). 5th IFAC Conference on Nonlinear Model Predictive Control NMPC 2015, pp. 1–8. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2015.11.255. URL: https://www.sciencedirect.com/science/article/pii/S2405896315025392.

[19] W. Langson, I. Chryssochoos, S. Raković, and D. Mayne. "Robust model predictive control using tubes". In: *Automatica* 40.1 (2004), pp. 125–133. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2003.08.009. URL: https://www.sciencedirect.com/science/article/pii/S0005109803002838.

[20] D. Mayne, M. Seron, and S. Raković. "Robust model predictive control of constrained linear systems with bounded disturbances". In: *Automatica* 41.2 (2005), pp. 219–224. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2004.08.019. URL: https://www.sciencedirect.com/science/article/pii/S0005109804002870.

[21] S. Lucia, A. Tătulea-Codrean, C. Schoppmeyer, and S. Engell. "Rapid development of modular and sustainable nonlinear model predictive control solutions". In: *Control Engineering Practice* 60 (2017), pp. 51–62. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2016.12.009. URL: https://www.sciencedirect.com/science/article/pii/S0967066116302970.

[22] S. Thangavel, R. Paulen, and S. Engell. "Multi-stage NMPC using sigma point principles". In: *IFAC-PapersOnLine* 53.1 (2020). 6th Conference on Advances in Control and Optimization of Dynamical Systems ACODS 2020, pp. 386–391. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.06.065. URL: https://www.sciencedirect.com/science/article/pii/S2405896320300847.

[23] G. Franceschini and S. Macchietto. "Model-based design of experiments for parameter precision: State of the art". In: *Chemical Engineering Science* 63.19 (2008). Model-Based Experimental Analysis, pp. 4846–4872. ISSN: 0009-2509. DOI: https://doi.org/10.1016/j.ces.2007.11.034. URL: https://www.sciencedirect.com/science/article/pii/S0009250907008871.

[24] S. Subramanian, S. Lucia, S. A. Baradaran Birjandi, R. Paulen, and S. Engell. "A Combined Multi-stage and Tube-based MPC Scheme for Constrained Linear Systems". In: *IFAC-PapersOnLine* 51.20 (2018). 6th IFAC Conference on Nonlinear Model Predictive Control NMPC 2018, pp. 481–486. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.11.043. URL: https://www.sciencedirect.com/science/article/pii/S2405896318327010.

[25] R. Isermann and P. Ballé. "Trends in the application of model-based fault detection and diagnosis of technical processes". In: *Control Engineering Practice* 5.5 (1997), pp. 709–719. ISSN: 0967-0661. DOI: https://doi.org/10.1016/S0967-0661(97)00053-1. URL: https://www.sciencedirect.com/science/article/pii/S0967066197000531.

[26] I. Hwang, S. Kim, Y. Kim, and C. E. Seah. "A Survey of Fault Detection, Isolation, and Reconfiguration Methods". In: *IEEE Transactions on Control Systems Technology* 18.3 (2010), pp. 636–653. DOI: 10.1109/TCST.2009.2026285.

[27] R. Arunthavanathan, F. Khan, S. Ahmed, and S. Imtiaz. "An analysis of process fault diagnosis methods from safety perspectives". In: *Computers and Chemical Engineering* 145 (2021). DOI: https://doi.org/10.1016/j.compchemeng.2020.107197. URL: https://www.sciencedirect.com/science/article/pii/S0098135420312400.

[28] *Kerbal space program*. 2021. URL: https://www.kerbalspaceprogram.com/game/kerbal-space-program/.

[29] *Kerbal remote procedure call*. 2021. URL: https://krpc.github.io/krpc/.

[30] "Model Predictive Control meets robust Kalman filtering". In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 3774–3779. ISSN: 2405-8963.

[31] R. E. Kalman. "A new approach to linear filtering and prediction problems". In: *Journal of Fluids Engineering* 82.1 (1960), pp. 35–45.

[32] Q. Li, R. Li, K. Ji, and W. Dai. "Kalman Filter and Its Application". In: *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)* (2015), pp. 74–77.

[33] E. Camacho, T. Alamo, and D. M. de la Peña. "Fault-tolerant model predictive control". In: *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)* (2010), pp. 1–8.

List of Figures

2.1 Block diagram of PID controller in a feedback loop [7] 4
2.2 Receding horizon strategy of MPC [10] 5
2.3 MPC control loop [11] 6
2.4 Feedback MPC control loop [16] 8
2.5 Scenario tree representation of the uncertainty evolution of multi-stage MPC [24] 10
2.6 Hardware redundancy and analytical redundancy for FDI [26] 11
2.7 Fault detection and isolation methods [27] 12
2.8 Fault detection, isolation and reconfiguration scheme [26] 13
2.9 A screenshot of Kerbal Space Program game interface 14
2.10 Screenshot of kRPC mod interface 15

3.1 Spaceship and acting forces on it 16
3.2 Last moments of spaceship landing results from 3000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise. 20
3.3 Spaceship landing results from 3000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 21
3.4 Last moments of spaceship landing results from 3000 meters with two models. The speedometer is simulated with a standard deviation of 10 m/s Gaussian noise. 22
3.5 Spaceship landing results from 3000 meters with two models. In both landings, speedometer (vertical velocity) is simulated with a standard deviation of 10 m/s Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 23
3.6 Last moments of spaceship landing results from 3000 meters with two models. The speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise 24
3.7 Spaceship landing results from 3000 meters with two models. In both landings, speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 25
3.8 Last moments of spaceship landing results from 8000 meters with two models. In both landings, altimeter is simulated with a standard deviation of 50 meters Gaussian noise. 26

3.9 Spaceship landing results from 8000 meters with two models. In both landings, the altimeter is simulated with a standard deviation of 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 27
3.10 Last moments of spaceship landing results from 8000 meters with two models. The speedometer is simulated with a standard deviation of 10 m/s Gaussian noise. 28
3.11 Spaceship landing results from 8000 meters with two models. In both landings, speedometer is simulated with a standard deviation of 10 m/s Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 29
3.12 Last moments of spaceship landing results from 8000 meters with two models. The speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise 30
3.13 Spaceship landing results from 8000 meters with two models. In both landings, speedometer and altimeter are simulated with a standard deviation of 10 m/s and 50 meters Gaussian noise. The filling effect in ML model plots comes from fluctuations in noisy outputs. 31
3.14 Spaceship landing results from 3000 meters with two models. In both landings, altimeter failure happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 34
3.15 Spaceship landing results from 3000 meters with two models. In both landings, speedometer (vertical velocity) happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 36
3.16 Spaceship landing results from 3000 meters with two models. In both landings, speedometer and altimeter failures happened at 1000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 38
3.17 Spaceship landing results from 8000 meters with two models. In both landings, altimeter failure happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 40
3.18 Spaceship landing results from 8000 meters with two models. In both landings, speedometer (vertical velocity) happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 42
3.19 Spaceship landing results from 8000 meters with two models. In both landings, speedometer and altimeter failures happened at 3000 meters above the ground. The filling effect in ML model plots comes from fluctuations in noisy outputs. 44

4.1 Filtered altitude sensor noise results. Kalman Filter is used for filtering process. 45

List of Tables

3.1 Naive MPC simulation results with noisy sensors. 18
3.2 ML models simulation results with noisy sensors. 19
3.3 Naive MPC simulation results with faulty sensors. 32
3.4 ML models simulation results with faulty sensors. 32