A library on the Robot Operating System (ROS) for Model Predictive Control implementation

RENE DIAZ

KTH Industrial Engineering and Management

Master's Degree Project
Stockholm, Sweden December 2014

KTH Industrial Engineering and Management

A Library on the Robot Operating System (ROS) for Model Predictive Control implementation

Rene Diaz

Master of Science Thesis MMK 2014:90 MDA 465
KTH Industrial Engineering and Management
Machine Design
SE-100 44 STOCKHOLM

KTH Industrial Engineering and Management

Master of Science Thesis MMK 2014:90 MDA 465

A library on the Robot Operating System (ROS) for Model Predictive Control implementation

Rene Diaz

Approved:	Examiner:	Supervisor:
2014-11-14	Lei Feng	Bengt Eriksson
	Commissioner:	Contact person:
	Universidad Simón Bolívar	Carlos Mastalli

Abstract

Model Predictive Control is a receding horizon control technique that is based on making predictions in the future for a determined number of steps, using a model of the system to be controlled. This thesis report is centered around Model Predictive Control (MPC) and its application. In this thesis, there are two main goals: firstly, is the development of a software structure that uses the properties of Object Oriented Programming (OOP) and the Robot Operative System (ROS) to ease the use of MPC applications. Secondly, the use and verification of the capabilities of MPC controllers in plants with fast dynamics, such as the quadrotor. A linearized model of the quadrotor is developed for the controller to perform the predictions, and the non-linear version is used to make a numerical simulator to test the application. The MPC software structure works as it successfully integrates information from the classes that represent the model and optimization method to solve the quadratic problem. The resulting MPC controller shows a good response when following simple trajectories in the presence of simulated noise. However, when more complex trajectories are used, a considerable offset from the reference is obtained. Such behavior mostly caused by the use of a very limited model, which demonstrates the considerable sensibility of the controller to the accuracy of the used model.

KTH Industriell teknik och management

Examensarbete MMK 2014:90 MDA 465

A library on the Robot Operating System (ROS) for Model Predictive Control implementation

Rene Diaz

Approved:	Examiner:	Supervisor:
2014-11-14	Lei Feng	Bengt Eriksson
	Commissioner:	Contact person:
	Universidad Simón Bolívar	Carlos Mastalli

Sammanfattning

Model Predictive Control är en vikande horisont styrteknik som är baserad på att göra förutsägelser i framtiden för ett bestämt antal steg, med användning av en modell av systemet som skall styras. Denna avhandling rapport är centrerad kring Model Predictive Control (MPC) och dess tillämpning. I denna avhandling, finns det två huvudsakliga mål: för det första, är utvecklingen av en programvara struktur som använder egenskaper objektorienterad programmering (OOP) och Robot Operativ System (ROS) för att underlätta användningen av MPC-program. För det andra, användning och kontroll av funktionerna i MPC regulatorer i system med snabba dynamik, såsom quadrotor. En linjär modell av quadrotor är utvecklad för styrenheten att utföra de förutsägelser, och den icke-linjära versionen används för att göra en numerisk simulator för att testa programmet.
MPC mjukvara struktur fungerar som det framgångsrikt integrerar information från de klasser som representerar modellen och optimering metod för att lösa andragrads problemet. MPC regulatorn visar en god respons när man följer enkla banor i närvaro av simulerad brus. Emellertid, när mer komplexa banor används, en avsevärd förskjutning från referens erhålles. Ett sådant beteende oftast orsakas av användningen av en mycket begränsad modell, vilket visar på betydande känslighet styrenheten för riktigheten av den använda modellen.

Contents

List of Figures **vi**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 1
1.3 Bibliographic Revision . . . 2
1.4 Thesis Outline . . . 4

2 Model Predictive Control **5**
2.1 Introduction . . . 5
2.1.1 Optimization problem . . . 5
2.1.2 Quadratic programming problem . . . 5
2.1.3 Active and Inactive Constraints and Sets . . . 5
2.1.4 Feasibility . . . 6
2.1.5 Convexity . . . 6
2.1.6 Karush-Kuhn-Tucker Conditions . . . 6
2.2 Model Predictive Control Theory . . . 7
2.2.1 Model . . . 8
2.2.2 Objective Function . . . 9
2.2.3 Constraints . . . 10
2.2.4 Optimization . . . 10
2.2.5 Horizons . . . 11
2.3 Summary . . . 11

3 Dynamic Modeling of a Quadrotor **12**
3.1 Theoretical Derivation of the Quadrotor Model . . . 12
3.2 Verification of the Model . . . 15
3.2.1 Upward motion along the Z axis . . . 16
3.2.2 Lateral movement along the X axis . . . 18
3.2.3 Lateral movement along the Y axis . . . 19
3.3 Summary . . . 26

4 Software Architecture and Implementation **27**
4.1 Robot Operative System (ROS) . . . 27
4.1.1 Nomenclature . . . 27
4.2 Library overview . . . 28
4.3 Interface classes . . . 29
4.3.1 Model . . . 29
4.3.2 Optimizer . . . 30
4.3.3 Simulator . . . 31
4.3.4 ModelPredictiveControl . . . 31
4.4 Summary . . . 35

5 Results and Discussion **36**
5.1 Tank system simulations . . . 36
5.1.1 Simulation settings and parameters . . . 36
5.1.2 Results . . . 37
5.1.3 Discussion . . . 38
5.2 Quadrotor simulations . . . 38
5.2.1 Simulation settings and parameters . . . 38
5.2.2 Results . . . 39
5.2.3 Discussion . . . 42

6 Conclusions and Future Work **50**
6.1 Future Work . . . 51

Bibliography **52**

A Appendix: MPC Parameters **54**
A.1 Tank System . . . 54
A.1.1 State Space matrices . . . 54
A.1.2 Weight matrices . . . 54
A.2 Quadrotor . . . 54
A.2.1 State Space matrices . . . 54
A.2.2 Weight matrices . . . 55

List of Figures

2.1 Block diagram of Model Predictive Control (taken from Camacho and Bordons [6]). . . 8

3.1 Functioning scheme for the quadrotor dynamics. (Taken from http://www.pupin.rs/RnDProfile/research-topic28.html) . . . 12
3.2 Inputs to generate an upward motion of the quadrotor. . . . 16
3.3 Resulting positions of the simulated systems for the given inputs. . . . 17
3.4 Resulting velocities of the simulated systems for the given inputs. . . . 17
3.5 Rotor speed inputs to generate a lateral movement along the X axis on the quadrotor. . . 18
3.6 Rotor speed inputs from Figure 3.5 mapped into forces and torques acting in the quadrotor frame. . . . 19
3.7 Resulting positions of the simulated systems for the inputs shown in Figure 3.5. . . . 20
3.8 Resulting velocities of the simulated systems for the inputs shown in Figure 3.5. . . . 20
3.9 Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.5. . . . 21
3.10 Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.5. 21
3.11 Rotor speed inputs to generate a lateral movement along the Y axis on the quadrotor. . . 22
3.12 Rotor speed inputs from Figure 3.11 mapped into forces and torques acting in the quadrotor frame. . . . 23
3.13 Resulting positions of the simulated systems for the inputs shown in Figure 3.11. . . . 24
3.14 Resulting velocities of the simulated systems for the inputs shown in Figure 3.11. . . . 24
3.15 Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.11. . . 25
3.16 Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.11. 25

4.1 Inheritance relationship between the interface and the implementation classes. . . . 28
4.2 Implementation example of the class hierarchy for the ArDrone case. . . . 29
4.3 Flowchart for the *resetMPC* function. . . . 32
4.4 Flowchart for the *initMPC* function. . . . 33
4.5 Flowchart for the *updateMPC* function. . . . 34

5.1 Representation of the tank system. . . . 36
5.2 Voltage inputs calculated for the tank system by the MPC controller. . . . 37
5.3 Reference level and actual level of the simulated tanks. . . . 38
5.4 Trajectory reference and actual trajectory positions of the simulated platform. . . . 40
5.5 Trajectory reference and actual trajectory orientations (ψ) of the simulated platform. . . 40
5.6 Linear velocities of the simulated platform. . . . 41
5.7 Angular velocities of the simulated platform. . . . 41
5.8 Control signals generated by the MPC strategy. . . . 42
5.9 Trajectory reference and simulated trajectory positions of the platform with the disturbance model. . . . 43
5.10 Trajectory reference and simulated trajectory orientations (ψ) of the platform with the disturbance model. . . . 44
5.11 Linear velocities of the simulated platform with the disturbance model. . . . 44
5.12 Angular velocities of the simulated platform with the disturbance model. . . . 45

5.13 Control signals generated by the MPC strategy with the disturbance model. 45
5.14 Positions of the simulated quadrotor with the square trajectory. 46
5.15 Linear velocities of the simulated quadrotor with the square trajectory. 47
5.16 Calculated inputs from the MPC to the simulated quadrotor when using the square trajectory. 48
5.17 Visualization of the control trajectories of the simulated platform. 49

List of Acronyms

AI Artificial Inteligence

GUI Graphical User Interface

IDL Interface Definition Language

IMU Inertial Measurement Unit

KKT Karush-Kuhn-Tucker

LAN Local Area Network

MIMO Multiple Inputs, Multiple Outputs

MPC Model Predictive Control

OOP Object Oriented Programming

PI Proportional Integral

PID Proportional Integral Derivative

QP Quadratic Problem

ROS Robot Operating System

ROV Remotely Operated Vehicle

SISO Single Input, Single Output

SLAM Simultaneous Localization And Mapping

WLAN Wireless Local Area Network

YAML YAML (Yet Another Markup Language) Ain't Markup Language

1 Introduction

1.1 Background

The Mechatronics Research group at Simon Bolivar University in Caracas, Venezuela is focused on the development of solutions based on the integration of knowledge in the fields of Automatic Control, Computer Science, Artificial Inteligence (AI) and Robotics, Electronics and Mechanics. This development is made on a project based strategy, with projects coming from both industry and academia. One of the projects is focused on the development of underwater inspection using the underwater robot developed in the group called PoseiBot. The development of this underwater robotic platform has been made in several phases through the years. In the latest phase, a Model Predictive Control (MPC) strategy was implemented by Molero et al. [16] to control the submarine in order to achieve accuracy control in PoseiBot in terms of the control effort, error reduction and robustness. MPC is commonly applied to large systems with slow dynamics, but recently with the increase of computational power and the development of new algorithms that are more efficient, systems with faster dynamics are being targeted to be controlled by predictive methods. This was implemented through communication of PoseiBot's microcontroller to a remote computer out of the water using serial wired communication to a flotation device that communicates wirelessly with the remote computer. In the computer, signal acquisition and data processing was done via LabVIEW ™and MATLAB®, respectively.

Another project developed in the group is the usage of helicopter models as a robotic platform for powerline inspection. Due to the wind conditions around the powerlines to be inspected, a robust control algorithm is required to assure a safe operation of the quadrotor while maneuvering around the lines. There is a strong interest on using instead the quadrotor available in the research group instead of the helicopter because of the increased stability.

Based on the requirements set by the aforementioned projects, there has been an increasing interest in advanced control techniques and specially MPC applications for both platforms. MPC has been proven as an efficient tool for solving multivariable control problems that might be difficult to decouple in plants that might have restrictions in the variables and might even be nonlinear. MPC can handle all of these requirements satisfactorily while being optimal in the solution, which is important in cases where resources are limited.

But in order to provide a solution that could fit both applications in a relatively quick way, some standardization and abstraction is required in the solution. That is where the benefits of using Robot Operative System (ROS) apply, and it also represented an opportunity to extend the usage of this tool within the group.

1.2 Purpose

The purpose of this project is the creation of a ROS package that will provide a framework to implement MPC in a standard and abstract way . The standard characteristic is necessary to get a package that is easy to use without needing to know how it works internally. The abstraction required comes from the fact that the software must work equally good independently of the platform that is being controlled. Of course, there are limitations on how much abstraction can be obtained, since every application will

require the development of a process model for the package to use. However, the goal is to use the properties of ROS to achieve this.

To reach this goal, the first activity to do will be an extensive bibliographic revision about MPC and its varieties, either theoretically and implemented in different systems. Another topic included in this revision is quadratic programming, since the interest is to apply MPC with constraints. When the MPC problem is not constrained the control law can be calculated exactly, but when constraints are added the solution must be obtained numerically, and that is when quadratic programs arise. This happens for linear systems and/or linearized systems, which are the object of interest in this phase of the project.

After this phase, the focus will be the design of the organization and development of the package. This is an important phase of the project because a proper design will allow a modular organization of the functionality, i.e. the nodes in the package will be enabled to be used in different combinations without altering the way the software works. The development is carried out in an iterative way, so the code can be tested and improved in each iteration.

The third phase consists in the creation of a demonstrative platform to use it as an overall test for the package. This includes the creation of a Model and Simulator classes for such system. The model used is kept simple to ease the validation of the results. The chosen system for this purpose is a water recirculation system with two tanks, that is used in the Automatic Control courses. This will save the modeling work, since this is a well-known plant.

At this point, the MPC package will be already running properly, and then the time to try it in a relevant platform comes. The modeling of the quadrotor platform will be performed to use it with the MPC package and perform simulated tests in trajectories of interest. This phase may require several tests in order to characterize and obtain the properties of the quadrotor if there is no relevant work available about it. The model also requires a validation process for itself to prove that it works in an adequate manner.

1.3 Bibliographic Revision

Even though MPC has been proven since long time ago to be applicable for different types of plants and processes, it took some time until the industry embraced it as the powerful tool it is. One of the first attemps to show the pros and cons of MPC is described by Richalet in [21]. In this paper, the benefits of implementing MPC are addressed from an industrial point of view, as well as the differences in the approach required to apply it in a proper way. The diversity of applications for MPC is also a topic in this paper: two cases are considered, one with slow dynamics systems and one in a system with quick dynamics; being able to handle both satisfactorily. An important conclusion from this paper is that the difference in application compared to traditional control techniques is that the effort is centered on the development of the model, not in the tuning of the controller. If a proper model is developed, the tuning of the controller consists on a proper choice of the horizons and weight matrices. On the other side, this requires a higher level of training for the staff in charge of the system.

In order to take advantage of this new engineering approach for the application of this technique, there have been several attempts to provide a platform to ease the control and focus on the modeling work. Most implementations in research are implemented using MATLAB® as in [8], [16], [12] and [11] to mention some examples.

In [8], a linear MPC strategy is used to provide a system of water dams an adequate flow of water required for the paper mills while maintaining the water levels among some defined boundaries and optimizing the use of it. In this thesis report it is easy to see practically the point that was made before: a good part

of the work is done in the development of a suitable model, afterwards the tuning of the MPC strategy is reduced to the tuning of the weight matrices, the prediction and control horizons and the size of the control time step. The MPC technique in this report is performed in MATLAB ®, using a quadratic cost function and state estimation via Kalman. In this case, the system dynamics are not so fast, so the computational power provided by MATLAB ® is enough to solve the problem within the sampling time restrictions.

In [16], the problem to solve is the trajectory tracking of a underwater Remotely Operated Vehicle (ROV). In this case, the MPC formulation used is a particular one because the constraints in the control and state variables are translated to the cost function directly using penalty functions. In this way, each constraint has a penalty cost associated that goes into the objective function. The implementation used a combination of wired and wireless technology for the data sending/receiving process, which was sent to a remote computer performing the MPC calculations and sending back the control signals to the ROV. The data processing was done in MATLAB ®, and the acquisition and Graphic User Interface (GUI) was done in LabVIEW ™. Using this strategy, substantial improvements in comparison with traditional Proportional-Integral-Derivative (PID) strategies were obtained in tracking performance and control effort.

In [12], MPC is used to control a turbocharged diesel engine. Several models are used to get the predictions: one simple linear model which lead to very good results; and a linear model evaluated in several operation points, forty five (45) to be precise. This switching of linear models makes it difficult to assure stability between operation points. To get a good performance, integral action was required.

In [11], ACADO is used to implement a MPC in a submarine ROV model. ACADO is a toolkit for automatic control and dynamic optimization. However, in this report a successful implementation of the MPC using this toolkit was not achieved, therefore a simulated MPC was implemented using Simulink ®. The linearized models of the submarine were shown to not be enough for a proper trajectory tracking, specially when going far from the operation points. In this thesis, it is to highlight the use of ROS for communication purposes, particularly to use the drivers developed for the XBox controller to add them to the teleoperation system. This is one of several advantages of using ROS for these purposes: open source code reuse to ease the addition of hardware to the system.

One disadvantage of MPC implementations using MATLAB ® and Simulink ® is that in cases where it is applied in unmanned vehicles, it makes the platform dependant on the communication with a remote computer. When applied in mobile autonomous platforms, the usual way used to perform the calculations is via C/C++ code deployed in single board computers. When this is done, it is even more convenient to have a way to reuse code for different MPC applications and focus more on obtaining a good model, and the later tuning required. The following papers have been focused on finding the way to make a standard MPC implementation for these cases.

In [15], the approach was to create a generalized class to solve MPC and dynamic optimization problems, using the *BzzMath* library to perform the calculations of the differential equations that describe the models. To use the class, the user must define only the differential system defined in the model and the objective function required to minimize, avoiding any struggle with numerical issues with the integration of the differential system and/or the minimization process. The class is designed for C++, but it has support for FORTRAN users as well. The inner architecture of the class is built in a intuitive way: the differential system provides information to the objective function, which is user defined and also accepts economical scenarios in case they are required. The combination of the model, the configurations and the economical scenarios combine altogether in the objective function. Then this objective function is passed to an optimization algorithm which minimizes the objective function and provides the results. However, depending on the application, trying different optimization algorithms or differential solvers

might be of interest, and these parameters are not customizable if this class is used.

In [22] the aforementioned interest in being able to customize the MPC problem was addressed. The approach taken here is towards the same goal, but instead of providing a generalized class, the proposal is to provide a whole library to deal with the different scenarios when formulating an MPC problem, and exploting the properties of Object Oriented Programming (OOP) to easily change the classes in the structure to fit the required problem. For example, the different varieties of linear models are dealt with by means of inheritance, where each type of linear model class inherits its properties from the base linear model class, easing the implementation and adding specific functionality tailored for each kind of model in particular. The library is based on the donlp2 solver, but there are ways to add another solver. This allows to customize the MPC and use the solver and model that fits best to each particular case.

Regarding the modeling of the quadrotor, there has been a lot of work done previously on this kind of platform in modeling and in control techniques applied to it [3], [5], [13], [19], [23], [24] and [14]. Most of the master thesis reports studied had the same goal in common: modeling, identification and control of the quadrotor platform. Therefore, the modeling work done was straightforward and the identification of parameters was taken from previous reports. The control techniques applied in most of these reports are classic PID structures, since the control is one of three major activities of the content, however in [1] a switching MPC approach is taken using several linearized models around different operation points. The switching is ruled by the Roll and Pitch angles, and the system is constrained to operate in a certain range of angles that define the operation points. This implementation uses an optical flow device to estimate the planar motion movements, then the velocities in the XY plane are estimated through a couple of 2 state Extended Kalman Filters. The system is proven to be able to perform very well in indoor conditions.

In [4] the MPC strategy is extended by means of adaptive or learning techniques that improve the model of the system continuosly using online data. The performance of the model is included in the cost function bounded by a nominal model, and includes a modeling error in the optimization constraints. The advantage of this is that even in the learning algorithm fails and the model is not improved, the system is kept within safety limits because of its inclusion in the cost function. The outcome of this work is remarkable, as the updating of the model is fast enough to allow the quadrotor to perform meaningful tasks that require speed and precision, in this case, catching a ball.

1.4 Thesis Outline

In the Introduction, the context of the project is presented, where the objectives of this project are defined and the state-of-the-art in the corresponding fields of knowledge are presented.

In Chapter 2, titled Model Predictive Control, a review of this advanced control method is introduced and specific information about each element that is involved in MPC is described.

In Chapter 3, the theoretical foundation used to develop a model for the quadrotor platform is described, and the implementation of the model is performed and validated.

In Chapter 4, the proposed library is described in detail: how it is organized, what does it include or not, what can be done with it and how does it work.

Chapter 5 presents the results of testing in the two different simulated systems that were developed,

Chapter 6 contains the conclusions derived from this thesis and,

Chapter 7 indicates the recommendations and/or future work efforts to be made with this project.

2 Model Predictive Control

2.1 Introduction

In this section the basic mathematical concepts required to understand the quadratic problem that arises in each iteration of Model Predictive Control will be briefly explained.

2.1.1 Optimization problem

An optimization problem consists of finding a solution within a feasible set that minimizes or maximizes a performance index. This problem can be subdivided in continuous and discrete, depending on the nature of the variables. The standard mathematical formulation is described as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & V(x) \\ \text{subject to} \quad & f_i(x) \leq b_i,\ i = 1,\ldots,m. \\ & h_i(x) = b_i,\ i = 1,\ldots,n. \end{aligned} \tag{2.1}$$

There are three elements to identify in this structure: the cost function $V(x)$, the equality constraints and the inequality constraints. Depending on the definition of the cost function and the constraints, the optimization can be linear or quadratic, which are the most common types encountered in real life applications.

2.1.2 Quadratic programming problem

A quadratic programming problem or quadratic program (QP) is a special case of an optimization problem where the cost function is a quadratic function and the constraints are linear. Given the variable and gradient vector correspondingly $\mathbf{x}, \mathbf{g} \in \mathbb{R}^n$, both column vectors and Q a symmetric matrix of size $n \times n$, the quadratic problem can be formulated as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & V(x) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{x}^T g \\ \text{subject to} \quad & A\mathbf{x} \geq \mathbf{b} \text{ (inequality constraint)} \end{aligned} \tag{2.2}$$

Where A is a matrix that represents the group of constraints and it is of size $m \times n$, b is a column vector of size m that contains the limits in the constraints, where m is the number of constraints. Equality constraints can be arranged into the A and b matrices with some manipulation.

2.1.3 Active and Inactive Constraints and Sets

Given an optimization problem such as 2.1, an inequality constraint $F_i(x) \leq b_i$ can be defined as *active* at a point $\tilde{x}$ in the feasible set when $F_i(\tilde{x}) = b_i$, and *inactive* otherwise. Equality constraints are active in all its solutions that are as well contained in the region of the allowed possible solutions.

Therefore one can define the *active set* at a point $\tilde{x}$ as the combination of the region that satisfy the equality constraints and inequality constrains that become active at $\tilde{x}$. Correspondingly, the region that doesn't satisfy these conditions is called the *inactive set*.

2.1.4 Feasibility

When constraints are included in an optimization problem, they define the region where the possible solution exists. The resultant region is defined as the *feasible* set. The algorithms used to solve the problem require an initial point, which depending on the specific application may be contained in the feasible set or not. The feasibility problem therefore consists in finding a feasible solution (if existent) regardless of the objective function.

2.1.5 Convexity

A set of points $S \in \mathbb{R}^n$ is a convex set if the straight line connecting any two points that belong to S lies completely inside S. In a more formal definition, for any two points $x, y \in S$, the following is true $\alpha x + (1-\alpha)y \in S, \forall \alpha \in [0,1]$.

This property is important because if this holds, the assumptions made for linear programming can be extended to cover convex optimization problems, and therefore be solved with fast and reliable methods that already exist for solving linear optimization problems.

2.1.6 Karush-Kuhn-Tucker Conditions

The Karush-Kuhn-Tucker conditions (a.k.a. (KKT) conditions) are a set of first order conditions that must be satisfied by the solution of nonlinear programming problems in general. It is formulated as an extension of the Lagrange multipliers method to solve optimization problems with equality constraints, as the KKT conditions applies for problems with constraints formulated as equalities and inequalities. These conditions are seldom used to solve the optimization problem directly, instead, they are verified in iterative methods. Before introducing the formal definition of the Karush-Kuhn-Tucker conditions, it is necessary to first define the Lagrangian for an optimization problem.

For the given form of the optimization problem shown in 2.1, the Lagrangian $\mathbf{L}$ is defined as the cost function plus penalty functions that take the constraints into account. The λ vector is used for the equality constraints and the μ vector for the inequalities.

$$\mathbf{L}(\mathbf{x}, \lambda, \mu) = V(x) + \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}) + \sum_{i=1}^{n} \mu_i f_i(\mathbf{x}) \tag{2.3}$$

If x^* is a solution of 2.1 that satisfies some regularity conditions, then there are vectors λ^* and μ^* such that the following conditions are satisfied:

- Stationarity

  $$\text{Minimization} \nabla_x V(\mathbf{x}) + \sum_{i=1}^{m} \nabla_x \lambda_i h_i(\mathbf{x}) + \sum_{i=1}^{n} \nabla_x \mu_i f_i(\mathbf{x}) = 0$$

  $$\text{Maximization} \nabla_x V(\mathbf{x}) + \sum_{i=1}^{m} \nabla_x \lambda_i h_i(\mathbf{x}) - \sum_{i=1}^{n} \nabla_x \mu_i f_i(\mathbf{x}) = 0$$

- Equality constraints

  $$\nabla_\lambda V(\mathbf{x}) + \sum_{i=1}^{m} \nabla_\lambda \lambda_i h_i(\mathbf{x}) + \sum_{i=1}^{n} \nabla_\lambda \mu_i f_i(\mathbf{x}) = 0$$

- Complementary slackness condition

$$\mu_i f_i(\mathbf{x}) = 0, \forall i = 1, \ldots, n$$

$$\mu_i \geq 0, \forall i = 1, \ldots, n$$

The stationarity conditions come from the derivation of the cost function to find the stationary points used in basic unconstrained optimization problems. The derivatives with respect to the λ coefficients (Lagrange multipliers) of the cost function lead to equations that restrict the solution of the problem to satisfy the equality constraints. However, due to the fact that the Lagrangian also depends on μ, the system is not determined. The complementary slackness conditions arises from the fact that if the optimal solution x^* satisfies $f(\mathbf{x}) \leq 0$, its contribution to the cost function is null, and one can set its corresponding μ_i coefficient to zero. If the solution is at the border of the constraint, $f(\mathbf{x}) = 0$. In both cases, the condition $\mu_i f_i(\mathbf{x}) = 0$ holds. The μ_i coefficients can take any value, but they are 0 when $f(\mathbf{x}) \leq 0$, and in the other case, $f(\mathbf{x}) = 0$, they can only be positive, since the gradient respect to x of $f_i(\mathbf{x})$ and $V(\mathbf{x})$ are opposed in direction.

In the particular case of a quadratic programming problem as the ones that arise in Model Predictive Control, the problem becomes a convex optimization problem when the cost function is convex. A convex quadratic cost function is the one where the Hessian matrix Q from 2.2 is positive semi-definite, the inequality constraints are convex functions and the equality constraints are linear. In the case of a convex quadratic programming program, the KKT conditions not only are necessary but also sufficient conditions for optimality.

2.2 Model Predictive Control Theory

Model Predictive Control (MPC) is an advanced control technique developed in the late 70's within the chemical industry. The basic idea in this algorithm is to use a model of the process or system to be controlled in order to predict and optimize future process behaviour. MPC can take into account restrictions in the input variables as well as the states/controlled variables [10]. This leads to solving an optimization problem in each sample, which demands high computing power in order to achieve this for small sampling times and deterministic operation.

There are several types of MPC controllers, but the basic steps of the algorithm are kept in each one of them, summarized as follows:

- Prediction: at each time instant, the model is used to get the predicted outputs of the process for as many time instants in the future as the prediction horizon states. These predictions depend on the current state of the plant and the optimizer solutions for the control signals, given a certain prediction horizon.
- Optimization: the future control signals are calculated by either minimizing a cost index or maximizing a performance index that takes into account the error between the predicted outputs and the reference trajectory (or an approximation to it) and the control effort. This objective function to be optimized is usually a quadratic function, since it assures global minimum or maximum values, smoother control signals and more intuitive response to parameter changes [10].
- Shifting: the control signal for the current time instant t is sent to the system and the next control signals in the future are rejected, since the output in the instant $t+1$ is already known, and the prediction step is repeated with this new value and the information is brought to the following time step. Then the control signal corresponding to time instant $t+1$ will be calculated at time $t+1$ (which is different from calculating the control signal for time $t+1$ at time t, due to the new information).

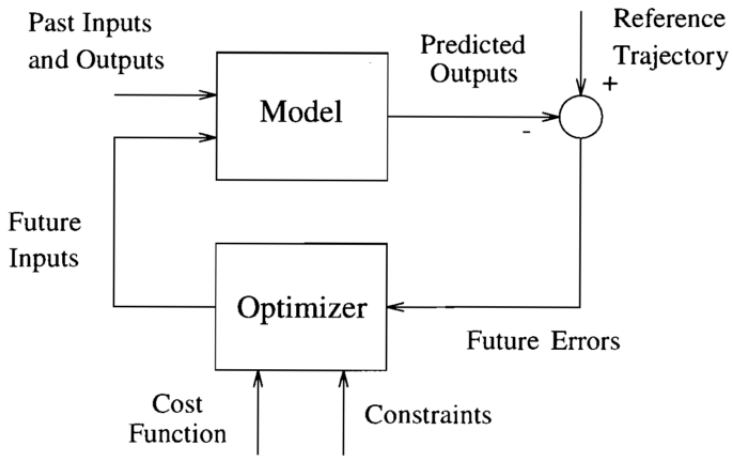


Figure 2.1: Block diagram of Model Predictive Control (taken from Camacho and Bordons [6]).

MPC controllers have some advantages compared to other control techniques, like the ones stated below:

- It can be used to control a great variety of processes, including ones that have long delay times or non-minimum phase or even unstable ones.
- In particular versions of MPC, it can be tuned to compensate for dead times.
- The strategy is easily extendable to the multivariable case.
- It can explicitly include constraints either on the control signal or the states/controlled variables.
- It introduces feed-forward in a natural way to compensate for measurable disturbances.

However, due to the characteristics of the technique, some drawbacks also arise:

- The main disadvantage to mention about MPC is the computing power required to solve the optimization problem in a suitable amount of time on each sampling instant, specially when the controlled systems have very fast dynamics. When the size of the problem is small, this can be solved by calculating all the possible control laws offline and then perform the look-up at runtime. However, the size of the optimization problem is also depending on the prediction horizon, which is a variable parameter considered for the tuning of the controller. This makes computational power an important aspect to consider in the implementation of MPC. Nowadays industrial computers shouldn't have problems handling this kind of processing, but since these are also used for several other functions, deterministic operation is important to preserve.
- Another disadvantage of MPC is that the technique is dependant on the process model. The algorithm itself is independent of the model, but the quality of the control signal obtained through the optimization problem is strongly dependant on good predictions coming from the model. However, the integration of statistical methods of learning in recent works [4] has been proven to solve this problem, without further modeling work.

2.2.1 Model

Being of such importance to the performance of MPC controllers, the process model should be precise enough to capture all the important dynamics but simple enough to keep the optimization problem at a decent size, thus saving computational time when solving. Since MPC is not a unique strategy, different implementations may vary in the type of models used. However most implementations of MPC make

use of one of these types of models: Transient Response, Transfer Function or State Space models. A brief description of each is presented in this section.

- **Transient Response.** Due to its simplicity, it is probably the most used kind of model in industry. To derive this kind of model, known inputs are fed to the real system or process and the outputs are measured. The most common inputs used for these experiments are impulse and step inputs, so in each corresponding case they are better known as impulse and step response models. The inputs and outputs are related by the following truncated sum of N terms:

$$y(k) = \sum_{i=1}^{N} h_i u(k-i) = H(z^{-1})u(k) \tag{2.4}$$

  where $H(z^{-1})$ is a polynomial of the backward shift operator, z^{-1}. For a model coming from such a simple experiment, the information that can be obtained is of great help to the understanding of the system: influenced variables by the input, time constants of the system and general characteristics can be determined from transient models. The fact that no previous knowledge of the system is required is an advantage for unknown processes. As a drawback, usually a lot of parameters are required as N is usually a big number for these models.

- **Transfer Function.** The transfer function model is obtained through the quotient of the Laplace transforms of the inputs and the outputs. When expressed in discrete time, these polynomials are a function of the backward shift operator, as stated below:

$$y(k) = \frac{B(z^{-1})}{A(z^{-1})} u(k) \tag{2.5}$$

  These models give a good physical insight and the resultant controller is of a low order and compact. However, to develop a good transfer function model some information of the system is required beforehand, more specifically about the order of the polynomials. Also, it is best suited for single variable systems, also known as Single Input, Single Output (SISO) systems.

- **State Space.** The states of the system are described as a linear combination of the previous states and inputs, and the output as a mapping of the states. The general form of state space models is as follows:

$$\begin{cases} \mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{u}(k) \\ \mathbf{y}(k) = C\mathbf{x}(k) \end{cases} \tag{2.6}$$

  Where A is the system matrix, B is the input matrix and C is the output matrix. The big advantage of this type of model is that it straightforward to use for multivariable systems or Multiple Input, Multiple Output (MIMO), and the control law will always be a linear combination of the state vector.

For this thesis, the state space model representation is used. This selection allows an easier way to handle models of different sizes; also the recursive structure of the predictions from the model allows more compact formulations of the quadratic problem, as developed by [7].

2.2.2 Objective Function

The objective function takes into account the error between reference trajectory and measured states, as well as the change in the control effort. The optimization process will give the values of the control signal u that minimizes the values of this objective function. A basic form of this objective function would be the following:

$$V(u) = \sum_{i=1}^{N_p} [\hat{y}(k+i|k) - r(k+i)]^2 + \sum_{j=1}^{N_c} [\triangle u(k+i-1)]^2 \tag{2.7}$$

Where the term $\hat{y}(k+i|k)$ represents the estimated output from the model calculated at time k for any sample in the horizon, $r(k+i)$ is the reference trajectory desired for the process (which is usually known in applications such as robotics) and the term $\triangle u(k+i-1)$ is the control effort. This objective function is quadratic, but it can also be of a different order. Most MPC implementations use quadratic objective functions because most metrics are quadratic, e.g. euclidian metric, and they assure that a global minimum is reached. Some implementations of MPC use an approximation to the real reference trajectory which parameters can be tuned in order to adjust to fast tracking or smooth response.

Parameters N_p and N_c are the prediction and control horizon respectively. These can be set to the same number although it is not a necessary condition. The definition of these parameters define when it is of interest to consider the different errors. This allows the objective function to be flexible for systems with dead-times or non-minimum phase. Also these errors may have different relevance in the control of the system, therefore this objective function might include or not weight matrices in order to ponder differently the errors.

In this thesis, the objective function used is pre-defined by the optimization solver qpOASES, presented by [7], which is of the following form:

$$V(u) = \frac{1}{2} \sum_{i=k_0}^{k_0+N_p-1} (\mathbf{y}_k - \mathbf{y}_{ref})' Q (\mathbf{y}_k - \mathbf{y}_{ref}) + (\mathbf{u}_k - \mathbf{u}_{ref})' R (\mathbf{u}_k - \mathbf{u}_{ref}) + \frac{1}{2} (\mathbf{y}_{k_0+n_p} - \mathbf{y}_{ref})' P (\mathbf{y}_{k_0+n_p} - \mathbf{y}_{ref}) \tag{2.8}$$

In this equation, three sources of error are noticeable: output errors (or depending on the process, state errors), input errors and terminal cost errors. Each source has its respective weight matrix: Q, R and P, respectively. In this implementation, the prediction horizon and the control horizon have been merged into the same number N_p, simplifying the function.

2.2.3 Constraints

Constraints are limitations in the values of the variables that are considered in the open loop optimization problem formulated in MPC. The explicit consideration of constraints is translated into an increase in computational complexity, as the solution of the problem can only be obtained through numerical methods. Usually, constraints in the inputs are due to limitations of the actuators interacting with the process, and constraints in the outputs or in the states come from safety or operational limits in the process itself or the sensors present in the system.

In [7] a distinction is made between bounds and constraints, where the bounds are the limit values for the variable being optimised (the control signal u) and the constraints are expressions to define the limitations of the outputs and states, which are mapped through the matrix G as follows:

$$\underline{\mathbf{U}} \leq \mathbf{u} \leq \bar{\mathbf{U}} \tag{2.9}$$

$$\underline{\mathbf{A}} \leq G\mathbf{x} \leq \bar{\mathbf{A}} \tag{2.10}$$

2.2.4 Optimization

There are two main type of algorithms used commonly to solve quadratic problems like the ones that arise in MPC: active set methods and interior point methods.

- **Active Set Methods.**

  The aim of an active set method is to find an optimal active set, which will make it possible to use equality constrained QP solving techniques (solving the KKT system). The algorithm will start making a guess of this optimal active set, and if it misses, gradient and Lagrange multipliers will be used to improve the initial guess. The final solution of the problem will lie in the proximity of the borders of the feasible region.

- **Interior Point Methods.** Given the following quadratic problem in the standard form, shown in 2.2, based on the KKT conditions one can say that if a given $\mathbf{x}^*$ is a solution of 2.2, there is a Lagrange multiplier vector λ^* such that the following conditions are satisfied for $(\mathbf{x}, \lambda) = (x^*, \lambda^*)$.

$$\begin{aligned} Q\mathbf{x} - A^T\lambda + \mathbf{g} &= 0, \\ A\mathbf{x} - \mathbf{b} &\geq 0, \\ (A\mathbf{x} - \mathbf{b})_i\lambda_i &= 0, \quad i = 1, 2, \ldots, m, \\ \lambda &\geq 0. \end{aligned} \tag{2.12}$$

  If a new variable vector $\mathbf{y} = \mathbf{A}\mathbf{x} - \mathbf{b}$ is introduced in the system, the conditions can be rewritten as follows.

$$\begin{aligned} Q\mathbf{x} - A^T\lambda + \mathbf{g} &= 0, \\ A\mathbf{x} - \mathbf{y} - \mathbf{b} &= 0, \\ \mathbf{y}_i\lambda_i &= 0, \quad i = 1, 2, \ldots, m, \\ (\mathbf{y}, \lambda) &\geq 0. \end{aligned} \tag{2.14}$$

  Which are correspondent with the KKT conditions for linear programming problems [17]. If we assume that we are only working with a convex objective function and feasible region, these conditions are necessary but also sufficient to assure the existence of such pair, and therefore the solution of the system 2.13 solves the quadratic problem.

2.2.5 Horizons

In MPC, the horizons define how far is the controller making a prediction in the future. The typical horizons in MPC are the prediction horizon N_p, and the control horizon, N_c. The prediction horizon has the basic function described previously: defining the prediction length. The selection of N_p is a matter of trial and error but it is highly dependant on the dynamics of the plant being considered. The prediction must be able to take into account the settling time of the system under a disturbance but also must be kept small enough so that it doesn't increase unneccessarily the size of the quadratic problem to solve. In order to have a better way to tune this, the control horizon N_c tells the MPC how much solutions it must calculate, independently of the size of the prediction horizon, keeping the computations at the minimum.

2.3 Summary

In this chapter, the mathematical foundation for MPC has been described, in order to get a better understanding of the optimization problem ocurring at each step. Also, MPC is introduced, together with its basic steps and characteristics. The different elements in MPC are explained in separate items to cover the individual functions and importance of each within the algorithm.

3 Dynamic Modeling of a Quadrotor

This chapter describes the modeling efforts made to develop a suitable representation of the system for the MPC library developed, based on the physical phenomena responsible for the quadrotor's operation. To start, the theoretical derivation of the quadrotor model is exposed. To verify the correct behavior of the derived model, a section is dedicated to the verification tests and their respective results. A global summary is included at the end of the chapter to compile and discuss the achievements made.

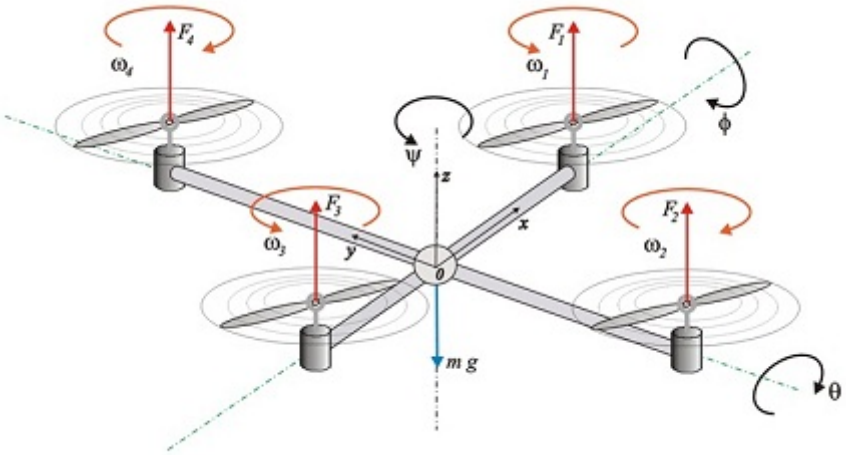


Figure 3.1: Functioning scheme for the quadrotor dynamics. (Taken from http://www.pupin.rs/RnDProfile/research-topic28.html)

A quadrotor has all six degrees of freedom in space and it has four actuators, which makes it an underactuated system. This means that two of the degrees of freedom must be controlled by means of regulating the other four in a proper manner. The four rotors are coupled, so the motion in the different directions is controlled by the difference in angular speed of the pairs of rotors. In Figure 3.1, the rotors are numbered so the pairs are defined in the directions of the axes. In order to move in the X axis, an imbalance must be made between the forces exerted by rotors 1 and 3, thus meaning a difference of angular speed in these rotors. It works the same way with the Y axis, as rotors 2 and 4 must be imbalanced as well to generate a motion in this direction. Notice that the changes in roll (ϕ), pitch (θ) angles are required to generate the motion in the X or Y directions. In order to move in the Z axis, all four rotors shall act in the same direction, therefore all four rotors must increase or decrease their angular speed. To perform a yaw (ψ) movement, the imbalance comes from both pairs, this is, rotors 2 and 4 rotate with a different angular speed than 1 and 3, since they rotate in opposite directions to cancel the rotating forces from the pairs and enable the quadrotor to *hover* or standing still in the air.

3.1 Theoretical Derivation of the Quadrotor Model

All known physical phenomena is used in order to obtain the theoretical model, however, the model can be as extensive as desired: in [14], the rotor aerodynamics and the concepts related to blade theory (drag and lift coefficients) are taken into consideration, however, other authors have chosen to reduce the system to use the model for control design, since these models depend on aerodynamic forces and torques,

which are subjected to disturbances caused by winds and turbulence. In [9], a more detailed study of the aerodynamic effects present in the quadrotor is made, but this kind of modeling is out of the scope of this report. In [3] Bouabdallah et Al. state that the main physical effects present in the quadrotor system are mentioned and theoretically formulated, from which we can mention aerodynamic effects, inertial counter torques, gyroscopic effects, gravity effects and friction. However, the main effects to include for a simple model should be the gyroscopic effects of the rigid body rotation in space and the effects of the four propeller's rotation.

Let A and B denote two coordinate frames, where A is fixed to the ground and B is fixed to the quadrotor body in its gravity center. The relation between these two coordinate frames is defined by an homogeneous transformation given by the Euler angles, that in our case are the same angles used to determine the orientation of an airbourne vehicle: roll (ϕ), pitch (θ) and yaw (ψ). The transformation between A and B is defined by a rotation matrix given by the aforementioned angles and a translation vector measured from A to B as follows:

$$\begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix} = \mathbf{R}_\mathbf{A}^\mathbf{B} \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix} + \mathbf{t}_\mathbf{A}^\mathbf{B} \tag{3.1}$$

Where $\mathbf{R}_\mathbf{A}^\mathbf{B}$ and $\mathbf{t}_\mathbf{A}^\mathbf{B}$ are the rotation matrix from A to B and the translation vector from A to B, respectively. The rotation matrix is defined as follows:

$$\mathbf{R}_\mathbf{A}^\mathbf{B} = \begin{bmatrix} \cos(\psi)\cos(\theta) & \cos(\theta)\sin(\psi) & \text{-}\sin(\theta) \\ \cos(\psi)\sin(\phi)\sin(\theta) - \cos(\phi)\sin(\psi) & \cos(\phi)\cos(\psi) + \sin(\phi)\sin(\psi)\sin(\theta) & \cos(\theta)\sin(\phi) \\ \sin(\phi)\sin(\psi) + \cos(\phi)\cos(\psi)\sin(\theta) & \cos(\phi)\sin(\psi)\sin(\theta) - \cos(\psi)\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix} \tag{3.2}$$

And the translation vector is simply defined as the position vector from the inertial frame A to the body frame B. This rotation and translation together define a homogeneous transformation $\mathbf{T}$ as in equation 3.3.

$$\mathbf{T}_\mathbf{A}^\mathbf{B} = \begin{bmatrix} \mathbf{R}_\mathbf{A}^\mathbf{B} & \mathbf{t}_\mathbf{A}^\mathbf{B} \\ 0 & 1 \end{bmatrix} \tag{3.3}$$

If $\mathbf{v} \in A$ is the velocity of the body frame B expressed in A, $\Omega \in B$ is the rotational velocity of the angular frame B with respect to A, expressed in B, m is the quadrotor's mass and $\mathbf{I} \in \mathbb{R}^{3\times 3}$ is the inertia matrix expressed in the body fixed frame B; a Newton's second Law of Motion force and torque balance, together with the kinematic relations between the frames lead to the following formulation:

$$\begin{aligned} \dot{\mathbf{t}} &= \mathbf{v} \\ m\dot{\mathbf{v}} &= mg\hat{\mathbf{k}_\mathbf{A}} + R\mathbf{F} \\ \dot{R} &= R\Omega \times \mathbf{v} \\ \mathbf{I}\dot{\Omega} &= -\Omega \times \mathbf{I}\Omega + \tau \end{aligned} \tag{3.5}$$

When the system is expanded as scalar equations, the resultant is a 12 equation system as follows:

$$
\begin{aligned}
\dot{x} &= u \\
\dot{y} &= v \\
\dot{z} &= w \\
\dot{u} &= \frac{1}{m}(\cos\psi\sin\theta\cos\phi + \sin\psi\sin\phi)\mathbf{F} \\
\dot{v} &= \frac{1}{m}(\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi)\mathbf{F} \\
\dot{w} &= \frac{1}{m}(\cos\theta\cos\phi)\mathbf{F} - g \\
\dot{\phi} &= \dot{p} + q\sin\phi\tan\theta + r\cos\phi\tan\theta \\
\dot{\theta} &= q\cos\phi - r\sin\phi \\
\dot{\psi} &= q\sin\phi\sec\theta + r\cos\phi\sec\theta \\
\dot{p} &= \frac{(I_{yy} - I_{zz})}{I_{xx}}qr - \frac{J_R\Omega}{I_{xx}}q + \frac{l}{I_{xx}}\tau_\phi \\
\dot{q} &= \frac{(I_{zz} - I_{xx})}{I_{yy}}pr + \frac{J_R\Omega}{I_{yy}}p + \frac{l}{I_{yy}}\tau_\theta \\
\dot{q} &= \frac{(I_{xx} - I_{yy})}{I_{zz}}pq + \frac{1}{I_{zz}}\tau_\psi
\end{aligned}
\tag{3.7}
$$

Where $\mathbf{F}$ and τ are vectors expressed in B that represent all the external forces and torques made by the aerodynamics of the rotors. These aerodynamic effects have been studied in [9] in depth, although this approach is unpractical in a robotics context. Nevertheless, some aerodynamics will be covered in this section to get a model that interacts at actuator level, this is, uses properties of the actuators as states.

For this equations system, the natural selection for the states is to choose linear and angular positions and velocities, since the derivatives of these are given in the left hand side of the set of equations 3.7. It is important to notice that these angular velocities are not equal to the derivative of the Euler angles in the mobile frame B. The derivative of the Euler angles is a discontinuous function. On the other side, the angular velocities in the mobile frame p, q, r are directly measurable through the Inertial Measurement Unit of the quadrotor, as it's actually done. From these measurements, the Euler angles are calculated [19]. Therefore we obtain a system with 12 states, as follows:

$$
\mathbf{X} = \begin{bmatrix} x & y & z & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix} \tag{3.8}
$$

As of right now, the model takes as input the upward force coming from the combination of the thrust of the four rotors, and the three torques in space that command the orientation and angular velocities of the quadrotor frame. These quantities are difficult to measure and knowing the relationship between the rotational speeds of the actuators and these forces and torques, it is easier to think in a model where the inputs are the angular velocities of the rotors. The thrust provided by a single rotor is given by [14] according to momentum theory:

$$
T_i = C_T \rho A_{r_i} r_i^2 \bar{\omega}^2 \tag{3.9}
$$

Where C_T is the thrust coefficient for that rotor blade geometry and profile, ρ is the density of air, A_{r_i} is the rotor disk area and ω is the angular velocity. In order to perform a simpler and more practical identification, a simplified lumped-parameters model can be determined by static thrust experiments, as shown in equation 3.10 :

$$
T_i = c_T \bar{\omega}^2 \tag{3.10}
$$

For the case of the quadrotor, the forces and torques in space that influence the system can be decomposed in terms of the rotational speeds as follows:

$$
\begin{aligned}
\mathbf{F} &= c_T\bar{\omega}_1^2 + c_T\bar{\omega}_2^2 + c_T\bar{\omega}_3^2 + c_T\bar{\omega}_4^2 \\
\tau_\phi &= dc_T(\bar{\omega}_2^2 - \bar{\omega}_4^2) \\
\tau_\theta &= dc_T(\bar{\omega}_3^2 - \bar{\omega}_1^2) \\
\tau_\psi &= c_Q((\bar{\omega}_2^2 + \bar{\omega}_4^2) - (\bar{\omega}_1^2 + \bar{\omega}_3^2))
\end{aligned}
\tag{3.12}
$$

Obtaining this simplified model experimentally has the advantage that the c_T coefficient includes the drag effect on the airframe that is induced due to the airflow caused by the rotor. For this model of the quadrotor, more detailed identification experiments were performed by [24], from which the resulting parameters are taken.

Due to the transformation matrix, the previously presented model of the system is non-linear, and since the solver requires a linear representation, a linearization process is required. This linearization process is done using the truncated approximation to Taylor series around a certain linearization point. The performance of this model will decrease when going away from this operation point, and therefore for the purposes of MPC, the accuracy of the predictions in these cases will not be as good. There are several ways to improve this, like selecting operation points distributed around the known regions of operation of the model, so it will change the information being used depending on its current state. Another alternative is to use a variable operation point that is changed in every single iteration so the model is always in the operation point. This strategy has the downside that it implies that the system matrices will be recalculated on every iteration, consuming computational resources. The operation point used for the validation tests is presented below. This point corresponds to the quadrotor in a hover state at a determined height z correspondent with the desired height of the test.

$$
\mathbf{X}^* = \begin{bmatrix} 0 & 0 & z & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

Since the objective of this thesis is not focused on modeling, a simple model with a static operation point will be used for demonstration together with the MPC software. This will have its effect on the performance of the MPC, but this effect can be reduced in some level by increasing the prediction horizon, at the cost of generating a bigger quadratic problem to solve. However, the computer running this simulation has enough processing power to handle it, but this should be highly regarded when performing this task on an onboard computer.

3.2 Verification of the Model

It is important to clarify that the following work consists only in verification, without validation. When the model is verified, its behavior is assesed to be correct, according to the described by the equations that rule the quadrotor's dynamics. However, it is not validated, in the sense that it is not proven that the model describes the behavior of the real platform. This work was not performed because the available AR Drone quadrotor takes as inputs velocity commands to an unknown control scheme programmed in the quadrotor's processing unit. Unveiling the inner control scheme in the quadrotor and bypassing it is a task that for time reasons is not performed, and therefore, the model will be simulated with the parameters identified in [24].

$$
\mathbf{X} = \begin{bmatrix} x & y & z & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix} \tag{3.13}
$$

There will be two simulations performed: one with the linearized model and one with the whole non-linear model of the platform. This is useful for the application in the MPC, where it is required to have one linear model for the platform to perform the predictions and set the quadratic problem, and the simulator for the platform, that will be represented by the full non-linear model in order to have more realistic simulations.

The following experiments show the outputs of the systems when a predefined set of inputs is given. These inputs are designed to obtain a specific movement pattern characteristic of the quadrotor in order to be able to analyze the outputs and assess the correctness of the outputs.These movements are the following: upwards motion along the Z axis, lateral and frontal movement around the X, Y axes respectively, and yaw rotation.

3.2.1 Upward motion along the Z axis

In order to generate the thrust to elevate the quadrotor, the rotors must be all operating at the same speed, above the equilibrium rotational speed (which is around 360 rad/s).

$$\mathbf{X} = \begin{bmatrix} x & y & z & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix} \tag{3.14}$$

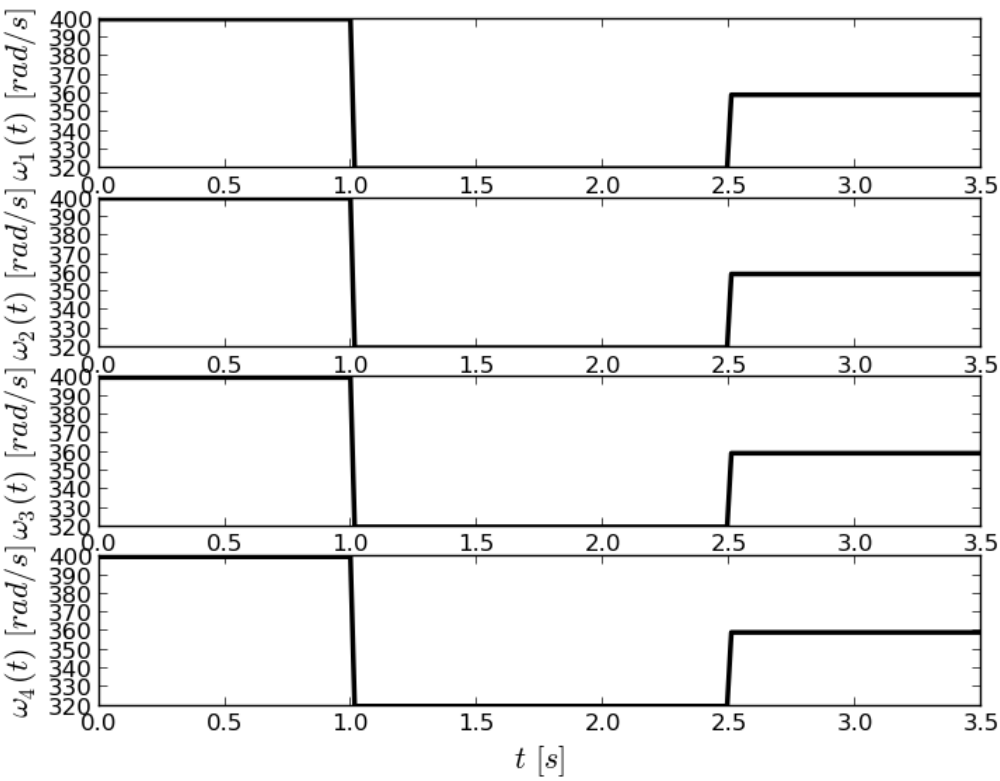


Figure 3.2: Inputs to generate an upward motion of the quadrotor.

In Figure 3.2, the used input signal to the model can be observed. The desired motion is an initial elevation of the quadrotor followed by a descent to some point. Therefore the signal follows this pattern. The resulting plots show the response of the simulated systems.
The plots for the Euler angles and the angular velocities are not shown because in this movement the angular variables are not changing. The behavior obtained is the expected in both cases, with the differences between them being caused by the linearization process. We can see that as the linear model goes further away from the linearization point the performance decreases in comparison to the non-linear model that is used as a performance reference. It is to notice that this is a simulation that doesn't take into account the ground. In Figure 3.4, the velocity goes to negative values because the time that the signal goes under the equilibrium rotational speed is bigger than the time that the thrust is active, and when the equilibrium is achieved, the model keeps the negative velocity. The behavior is correct, but the absence of ground makes room for this kind of details that might confuse when verifying.

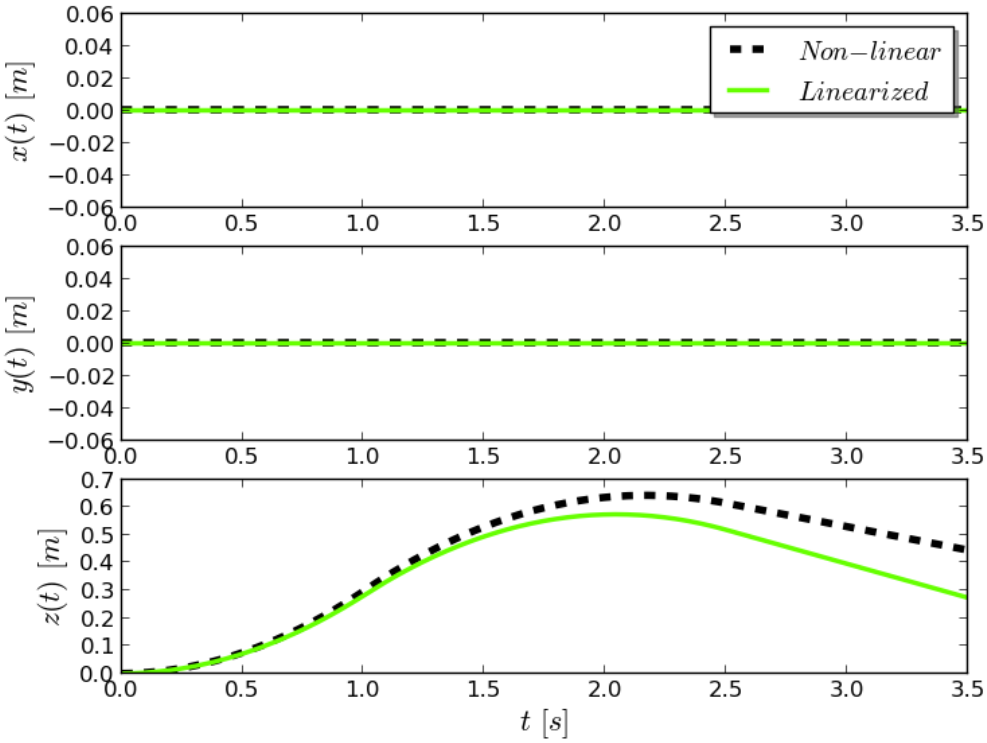


Figure 3.3: Resulting positions of the simulated systems for the given inputs.

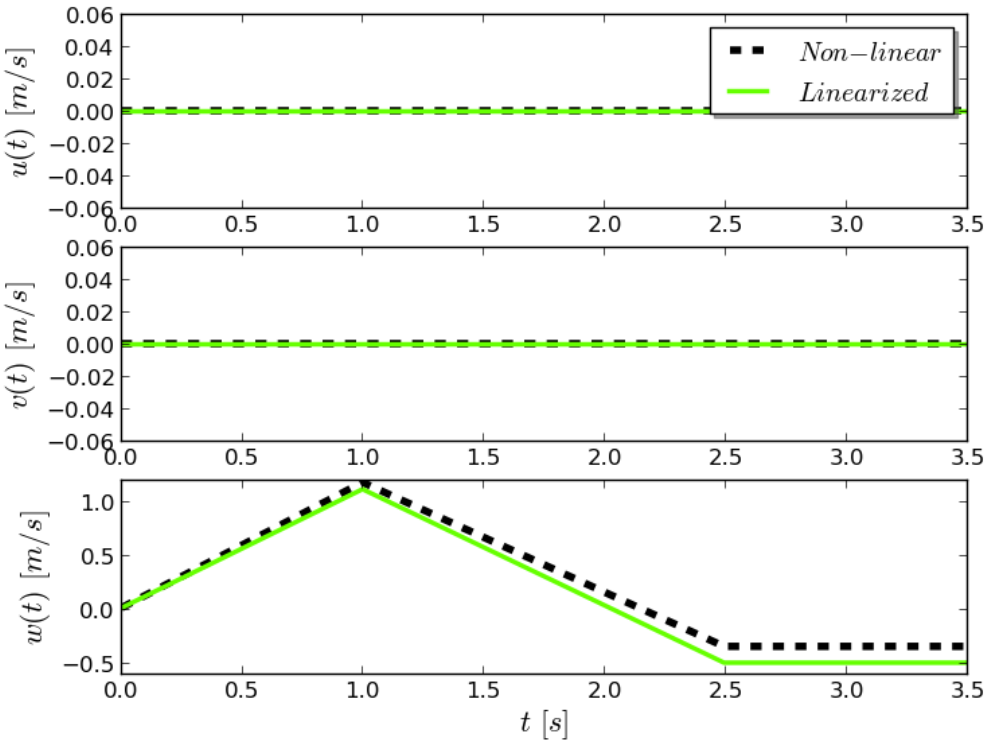


Figure 3.4: Resulting velocities of the simulated systems for the given inputs.

3.2.2 Lateral movement along the X axis

To move the quadrotor in the XY plane, a difference in the rotor speed between the pair (1,3) must be produced as seen from equation 3.12, so the pitch torque increases and tilts the frame sideways. The design goal for the input signal was to imitate the inner controller loop in the quadrotor, since the modeling only considers the physical equipment without any control, and this system configuration is open-loop unstable. However, because of this, the signal must restore the system to its equilibrium state. The signal was obtained in an empirical way, based on the linearized and non-linear expressions that describe the system.

The resultant signal is a collection of pulses in opposite directions to counter each other and create the restoration effect. To have a better understanding of the effect of the signal on the model, the mapping between rotor speeds and thrust and torques acting on the quadrotor frame has been made and plotted and added below.

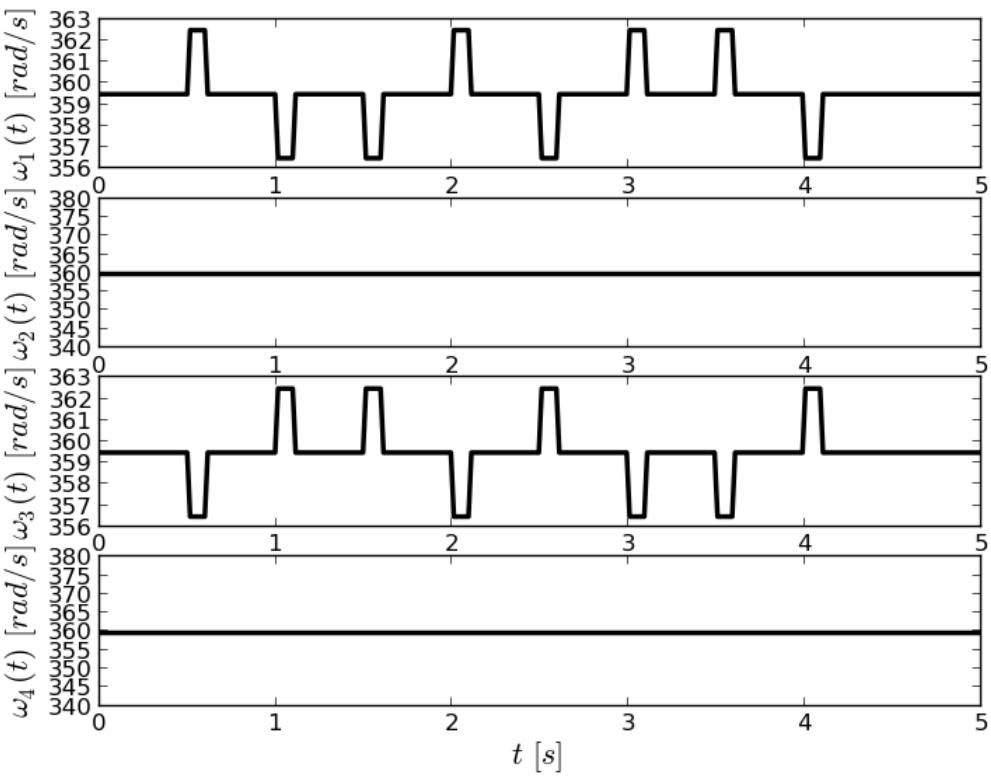


Figure 3.5: Rotor speed inputs to generate a lateral movement along the X axis on the quadrotor.

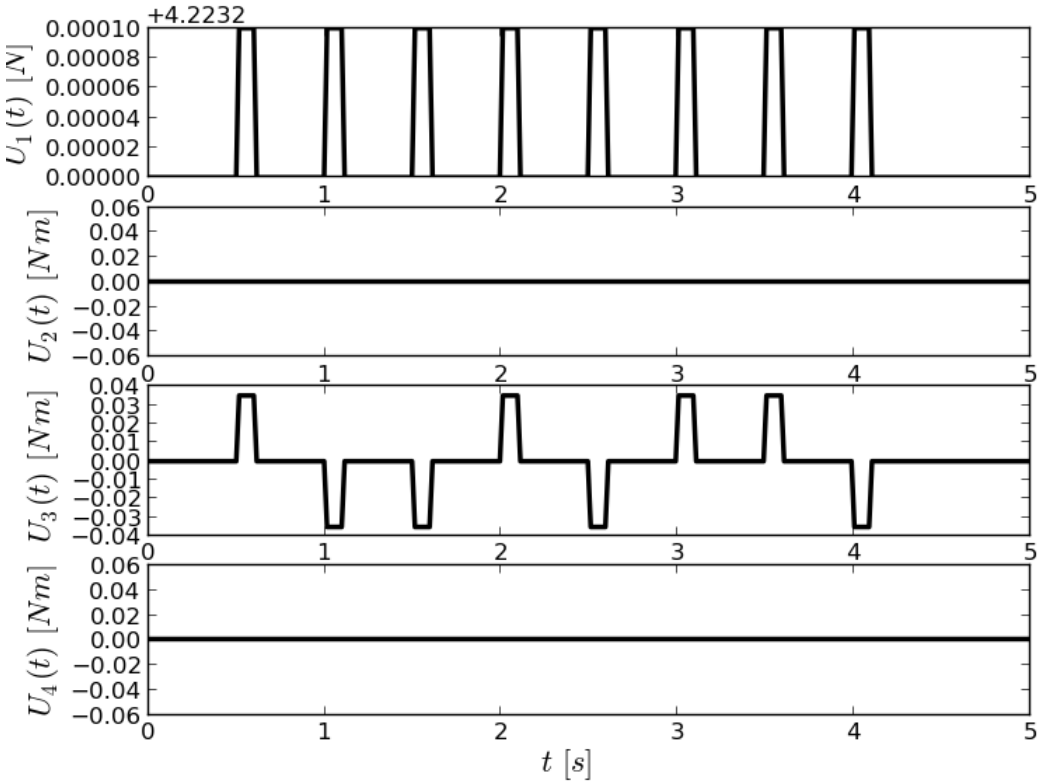


Figure 3.6: Rotor speed inputs from Figure 3.5 mapped into forces and torques acting in the quadrotor frame.

In this simulation the quadrotor is starting from a determined height of 0.5 meters. The resulting behavior of the model is quite satisfactory for this particular control signal. The difference between the linearized and the non-linear model is barely noticeable because the main non-linearities are introduced by the roll and pitch angles, which are kept in a very small range. Therefore, one could consider that $\cos(\alpha) \approx 1$ and $\sin(\alpha) \approx 0$. The torque inputs seen in Figure 3.6 are the ones calculated without the linearization process. Therefore, when a difference between a pair of rotors is stablished, there is also a slight change in the thrust, because the difference is squared. However, this difference is too small to influence the platform. This is not noticeable in the linearized output torque inputs. Another observation to highlight is that any movement of the quadrotor in any direction of the XY plane will decrease a little bit the Z coordinate because the thrust is redistributed for lateral movement.

3.2.3 Lateral movement along the Y axis

The same input signals designed for the previous test are used in this case, only that they are applied in the (2,4) pair of rotors to switch axes.

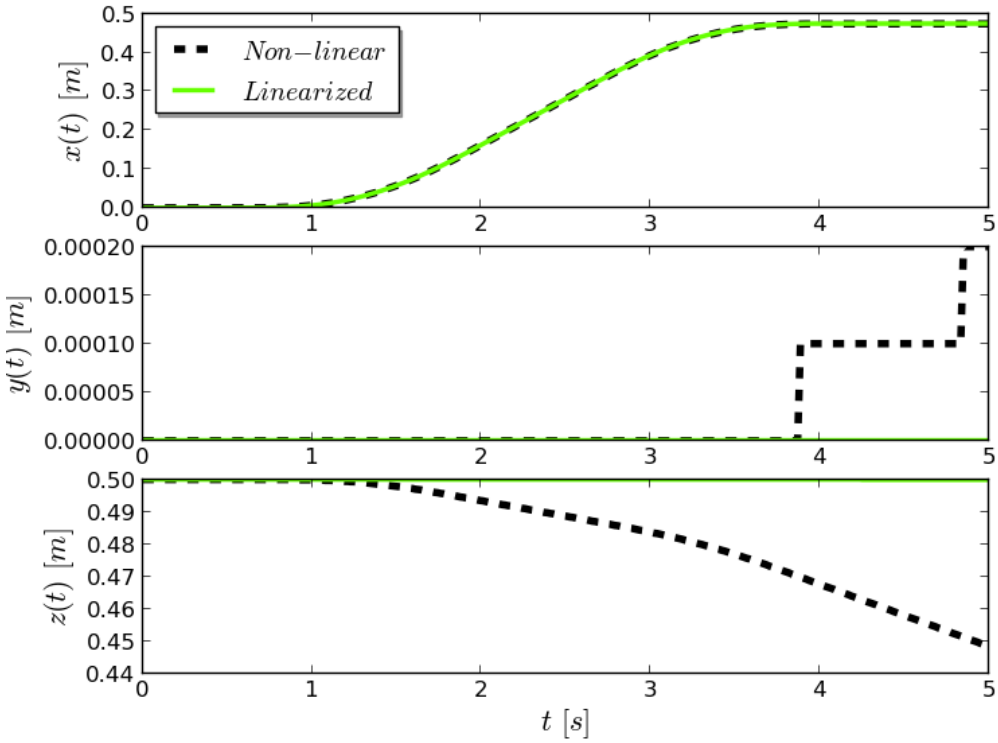


Figure 3.7: Resulting positions of the simulated systems for the inputs shown in Figure 3.5.

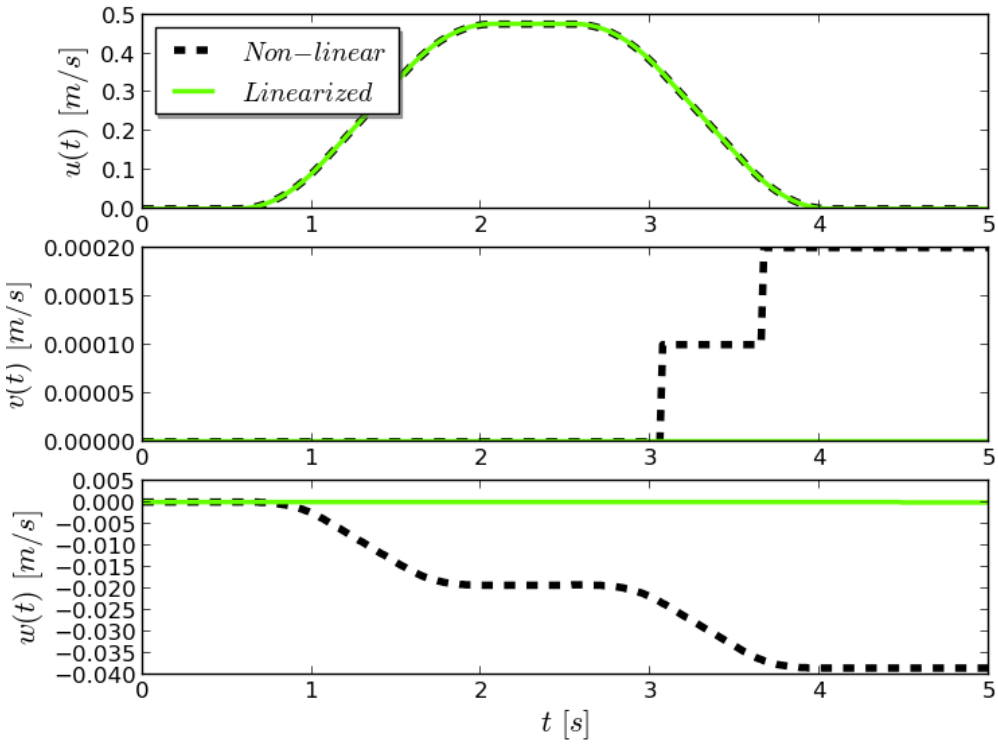


Figure 3.8: Resulting velocities of the simulated systems for the inputs shown in Figure 3.5.

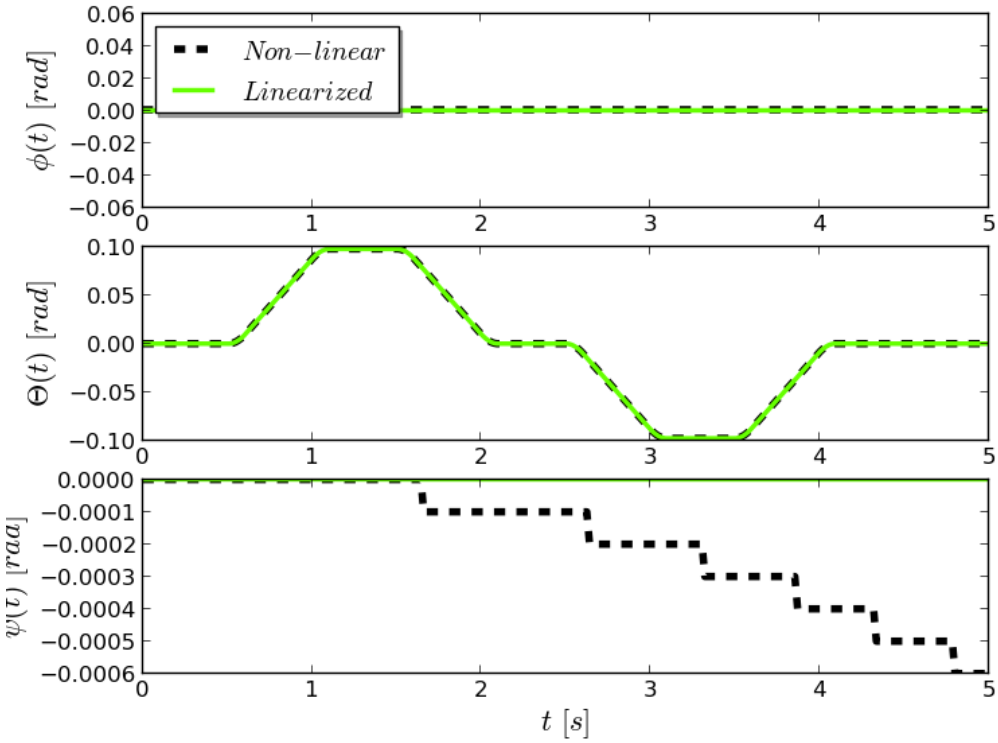


Figure 3.9: Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.5.

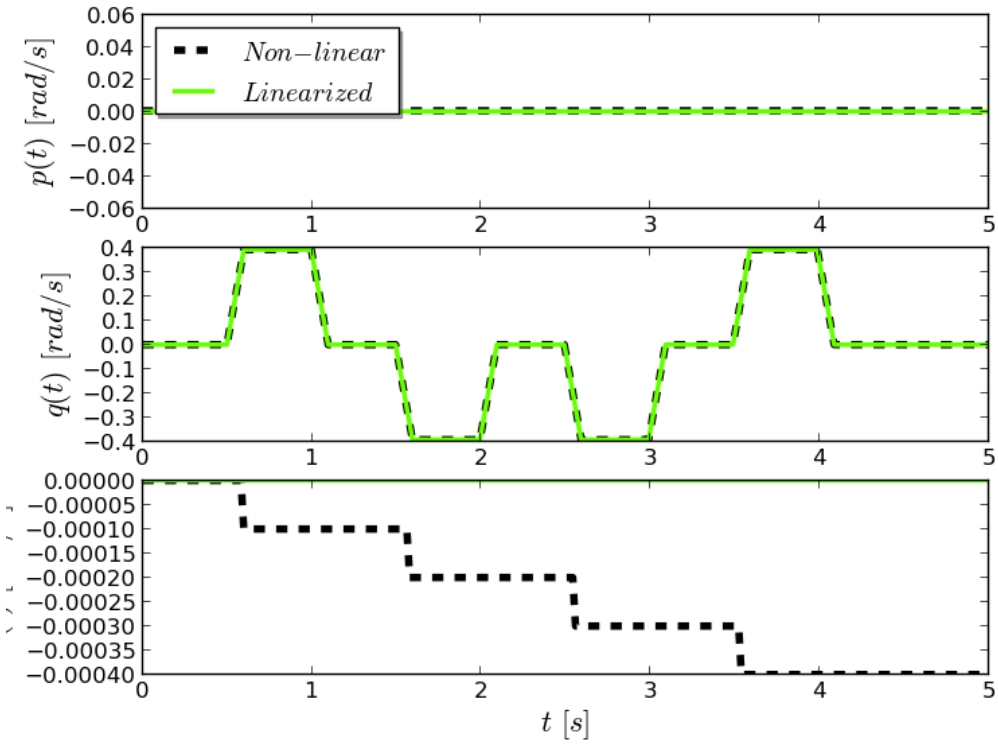


Figure 3.10: Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.5.

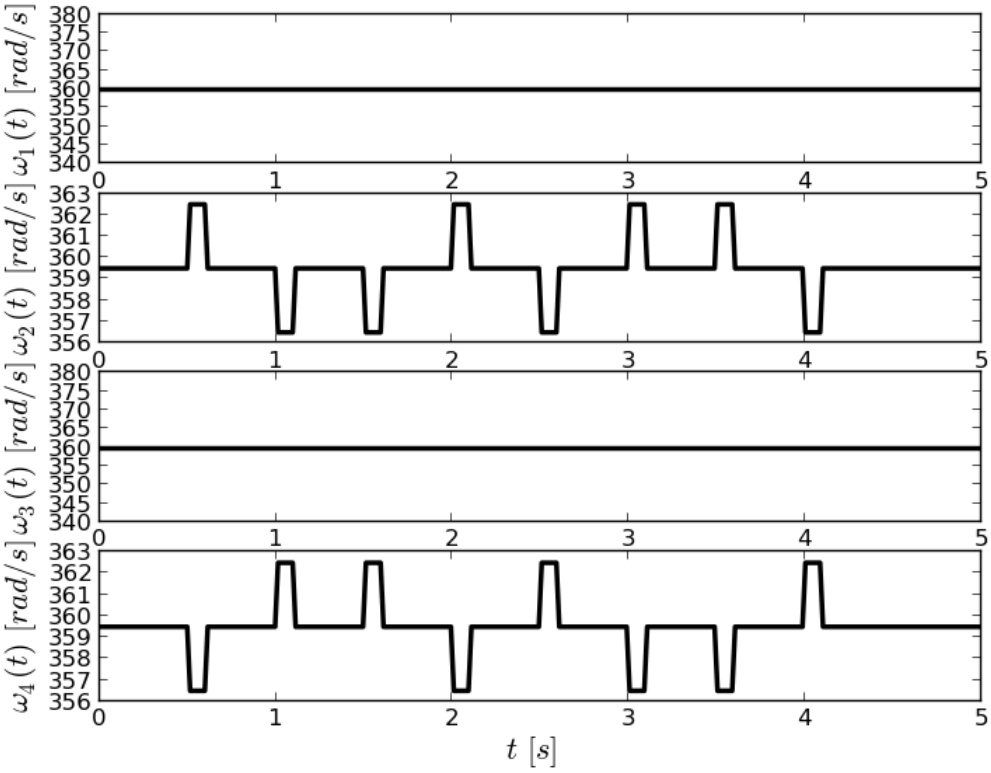


Figure 3.11: Rotor speed inputs to generate a lateral movement along the Y axis on the quadrotor.

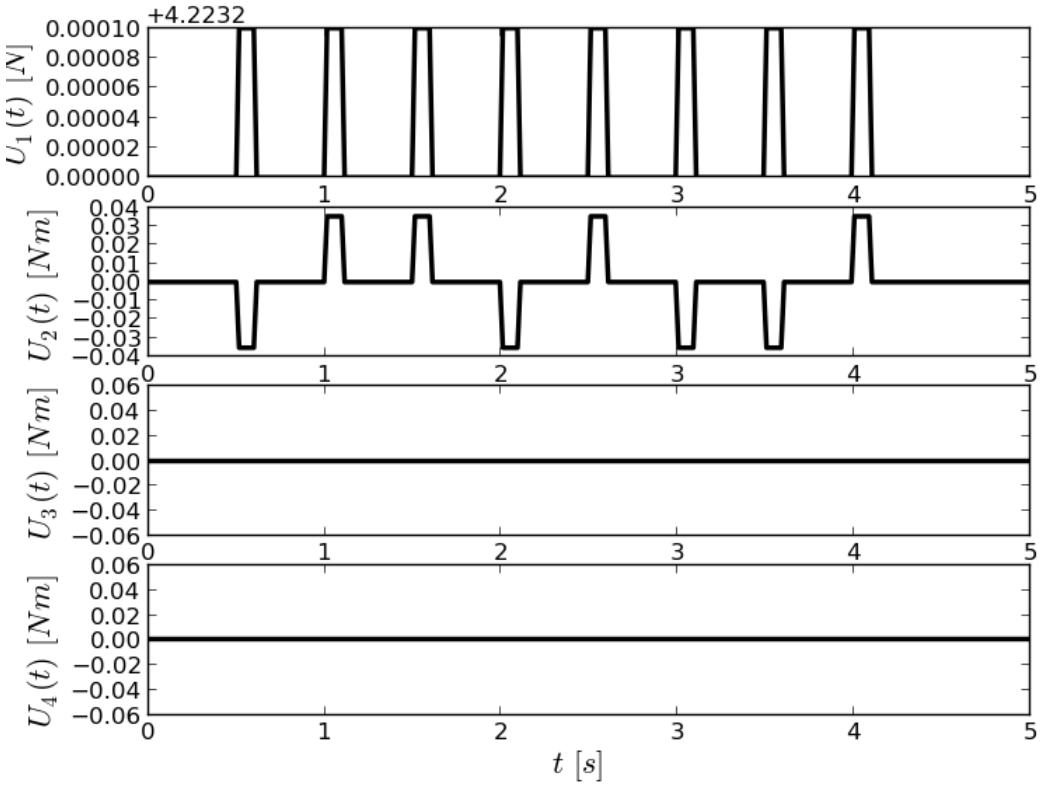


Figure 3.12: Rotor speed inputs from Figure 3.11 mapped into forces and torques acting in the quadrotor frame.

The resulting outputs have the same properties as the ones observed for the movement along the X axis: a descent in the Z coordinate caused by the coupling of the thrust force and very similar behavior between the linearized and non-linear model. In Figure 3.13 one can notice that the total displacement in the Y axis is a little bit less in this direction, because this direction is sideways and the protective hull of the quadrotor has a bigger cross section area along this axis.

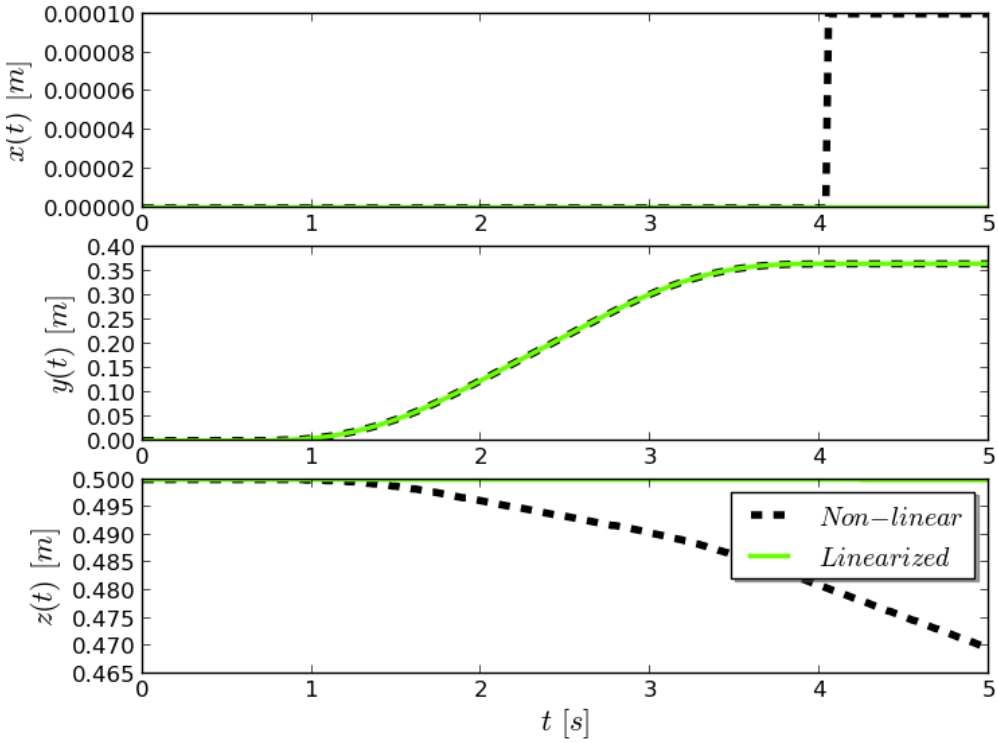


Figure 3.13: Resulting positions of the simulated systems for the inputs shown in Figure 3.11.

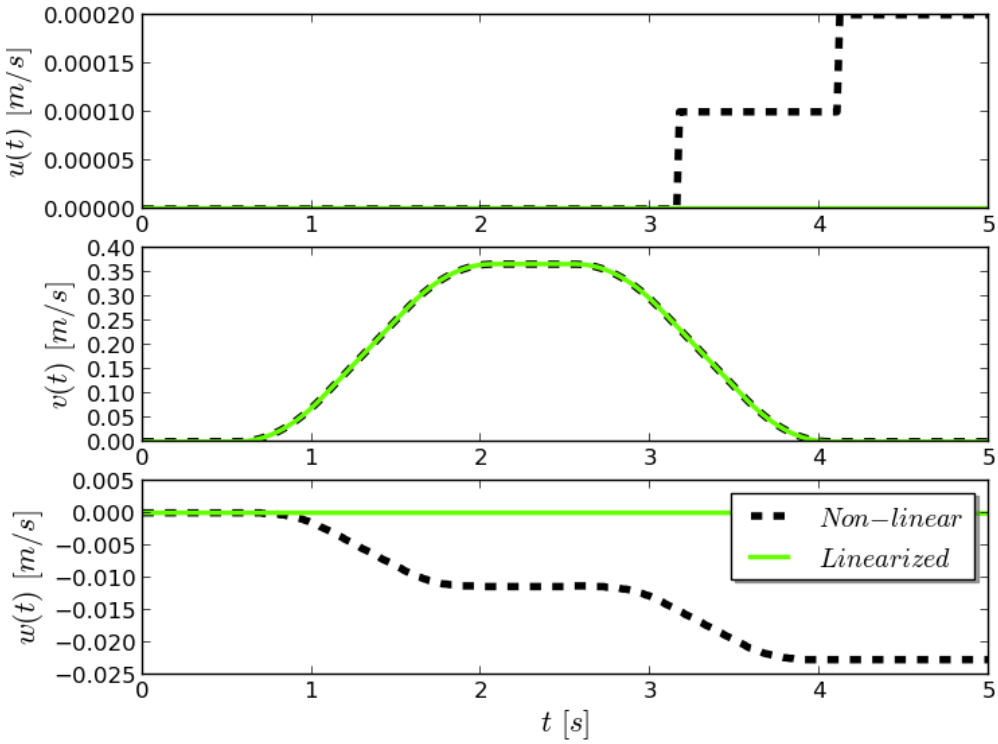


Figure 3.14: Resulting velocities of the simulated systems for the inputs shown in Figure 3.11.

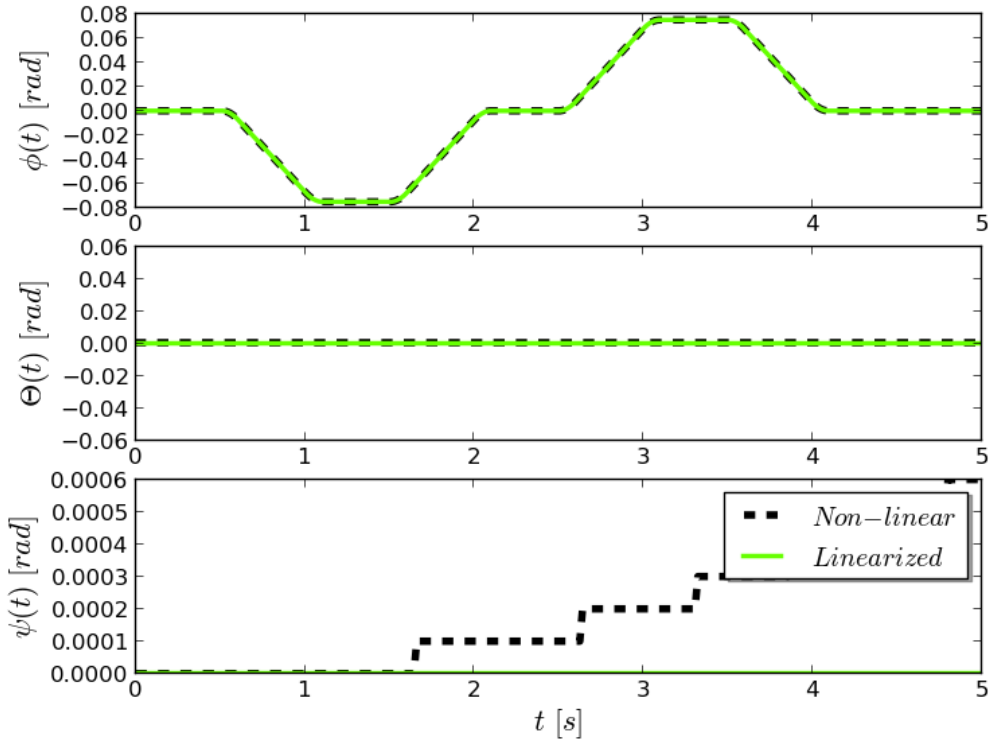


Figure 3.15: Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.11.

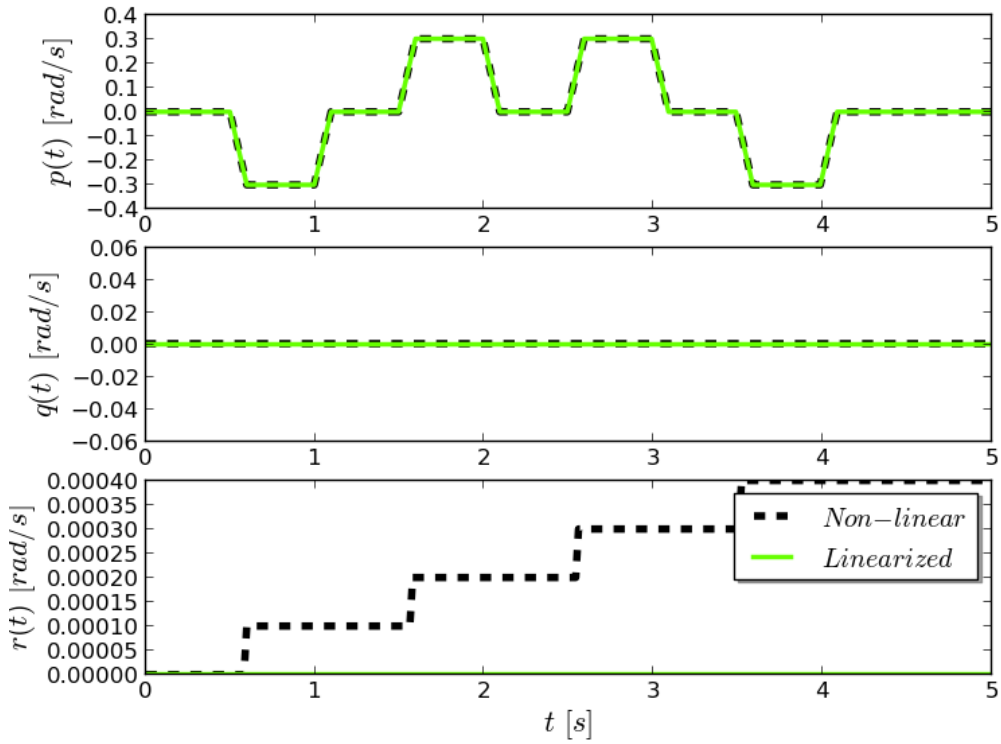


Figure 3.16: Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.11.

3.3 Summary

In this chapter a linear model of the quadrotor has been derived to integrate with the MPC software solution. This model is derived from the theoretical description of the physical phenomena responsible for the movement of the quadrotor. The quadrotor being modelled, Parrot's AR-Drone, has an inner control loop that includes the user. This is not taken in consideration in the derivation of this model. The model takes the angular speeds of the rotors as inputs and provides the quadrotor's position and yaw angle as outputs. The selection of the outputs correspond to the states being controlled, as it will be addressed later in the report. The model is linear due to a linearization process based on Taylor's series approximation, centered around a single operation point. The resulting model behaves good for small variations from this operation point. The input signals designed for the verification tests take into account that the model is open-loop unstable, so a restoring effect was required in order to obtain results that are easier and more intuitive to analyze.

4 Software Architecture and Implementation

4.1 Robot Operative System (ROS)

In robotics, the amount of code required to get functional robots is quite big, since the coding goes from a driver level of the components to more higher level AI algorithms and routines. Usually, it is difficult to write code that can be adapted to several platforms or robots, and the persons in charge of this may have different choices of languages, depending on the expertise. These reasons make the integration of the different applications required to get the robot up and running a challenging work.

ROS is a response to these necessities, and the result is an meta-operative system designed to build a framework that provides an abstract communication layer between robotic applications in order to minimize integration efforts and reuse code. This will allow to use the same program with different platforms, accross different languages and different levels of the software, simplifying a lot the work required to develop an experimental set and making it possible to expand easily the experiments. ROS can also run in a distributed way, so that different processes can run in different computers accross a Local Area Network (LAN) [18].

The ROS architecture is built in a peer-to-peer topology, where a number of processes can be running in a single host or in several hosts and communicate to each other. The coordination of the communication tasks is done by a *master* process that can run in any computer in the network. The ability to run several nodes in different languages is achieved by a language-neutral Interface Definition Language (IDL), that specifies each field of the message for the code generators and compilers of each language to generate an implementation native to the correspondent language.

4.1.1 Nomenclature

ROS functionality can be distributed in the following elements:

- **Nodes** A node is any individual process in the system that performs a computational task. Nodes can be organized in a nested way, so a node can consist of several smaller nodes with distributed functionality. ROS can create a visual interface to see the organization of the currently running nodes with simple commands in the terminal.

- **Messages** A message is the way that ROS nodes communicate between each other. It is a strictly typed data structure defined as a short text file for the compiler to interpret. A message can have primitive type like integers, doubles and floats; as well as other previously defined messages in a nested fashion.

- **Topics** A node sends a message through topics. A topic is the channel that ROS provides to send and receive messages. A topic is defined by a string, such as "*navigation*". When a node sends a message, it is said that the node has "*published*" a message to a certain topic, and when it receives a message it does it by "*subscribing*" to a certain topic.

- **Services** In case of requiring syncronous communication between nodes, ROS offers services. A service consists of three elements: a string that defines the name and two strictly typed messages, one for the request and one for the response. Unlike topics, only one node can advertise a particular service with a unique name.

- **Parameter Server** This is a very useful resource that ROS provides. It stores variable values of different types and used for different purposes while ROS is up and running. Some of these variables can be parameters for the running applications or configuration settings. These parameters can be loaded each time through a *launch* file and a configuration file written in (YAML) format, which is an acronym for YAML (Yet Another Markup Language) Ain't Markup Language [2]. The launch file initiates ROS, specifies the nodes that should be running for a particular application, and loads the parameter server with the information provided by the YAML file. One big advantage of this way of launching nodes is that the configuration file does not need to be compiled each time, so one can change parameters without the downtime used compiling.

4.2 Library overview

A quick description of the library can be made from Figure 4.1 and Figure 4.2. In Figure 4.1, the inheritance relationship between the interface classes and each particular instance of those classes can be seen directly here, without any functional relationship being displayed. The base classes are implemented with pure virtual functions or as interfaces; this is, they only exist through instances that inherit their methods and attributes. The interface classes cannot exist by themselves. The methods and attributes implemented in each particular application are the same, but particular implementation is defined in the derived objects.

The functional relation between the classes is displayed in Figure 4.2. A particular instance of the ModelPredictiveControl interface class is responsible of solving the MPC problem, which unifies the functionality of the Model, Optimizer and Simulator (if any simulator is to be used; if not, the information is shared to the platform via ROS topic) classes.

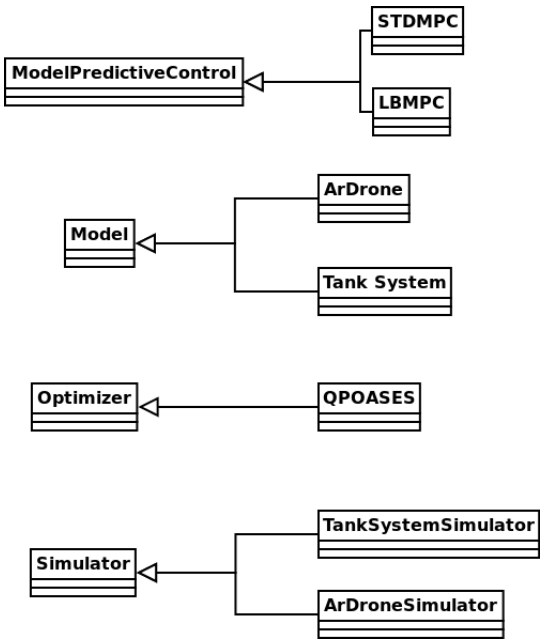


Figure 4.1: Inheritance relationship between the interface and the implementation classes.

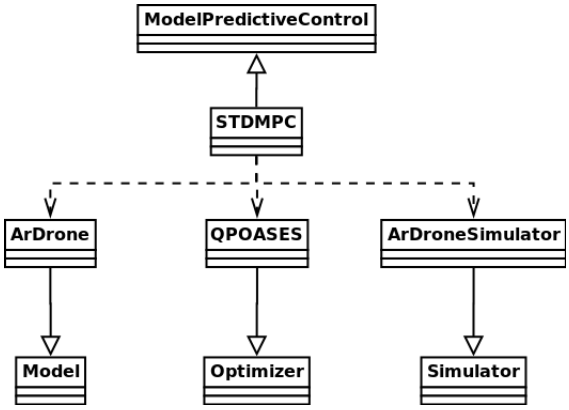


Figure 4.2: Implementation example of the class hierarchy for the ArDrone case.

The main requirements for this library at the moment of its design were the following:

- **Expandability.** This software is conceived in a way that users would know how to adapt the software easily to their specific applications with a small amount of modifications of the base program. The aim of the group is to be able to develop more functionality based on the foundational blocks of code that are created in this version. Since there are a lot of variants of the MPC algorithm, the goal is to be able to provide this options for the end user.

- **Modularity.** The software is designed to work with different quadratic programming solvers, different models and different simulators, just by creating derived classes from the bases that are provided. In this way, the possible combination of solvers, platforms and simulators is increased allowing for result verification and/or adapting the possible combinations to obtain the best performance for a particular application. This is also useful in order to allow to reuse code, since once a derived class for an application is completed it is ready to use in any other application available, which is why this feature is closely related to the previously mentioned expandability.

4.3 Interface classes

In this section is described in detail how each class works and integrates with each other and how they are built to achieve this goal.

4.3.1 Model

The Model base class is used to provide the information of the dynamics of the desired model to the MPC solver. The model is provided as a linear state space model, as follows:

$$\begin{cases} \dot{\mathbf{x}} &= A\mathbf{x} + B\mathbf{u} \\ \mathbf{y} &= C\mathbf{x} + D\mathbf{u} \end{cases} \tag{4.1}$$

So far, the only quadratic program solver that has been used provides support for models expressed in this form. If further development is continued, the aim is to provide support for the types of models that are mentioned in section 2.4 and 2.5.

The functionality of this class can be summarized in three simple actions: *compute*, *set* and *get*. The Model class computes the system matrices, gets the number of states, inputs and outputs; and sets the states and inputs of the corresponding model. The header file provides the structure to build up upon, so that the user can define in the source file the specific information from the desired process model to be used and how these functions are being implemented.

- computeDynamicModel function: this function takes the address of three Eigen matrix objects of the desired size and assigns the system matrices to those addresses as global variables for further use, returning a boolean variable to indicate success or failure. The matrices can be simply defined if the elements of each are known, or they can be calculated and discretized online when the function runs. These two ways are shown in the two different systems implemented in this thesis: in the case of the tank system, the matrices are just entered in the function so they are filled when the function runs; in the ARDrone quadrotor case, the matrices are calculated around the desired linearization point and discretized everytime the functions runs. The linearization and discretization methods are defined by the user, as long as a model with the previously mentioned structure is obtained.

- getStatesNumber function: this function takes no inputs and returns an integer with the number of states defined by the user in the source file. The states number is stored as a global variable for further use.

- getInputsNumber function: the same functionality as the previous get function, but with the number of inputs to the model.

- getOutputsNumber function: the same functionality as the previous get function, but with the number of outputs of the model.

- setStates function: this function takes an array of doubles as input and returns a boolean variable indicating success or failure at termination of the routine. The function assigns the elements of the array to the state vector array, so it is used for updating states at the end of the control loop.

- setInputs function: this function has the same functionality as the previous set function, but with the inputs to the model.

As seen in Figure 4.2, the Model interface class is one of the three classes that must be instantiated to be provided to an instance of the ModelPredictiveControl class to solve the MPC problem. The benefit of using an interface class is that any particular derived object can be instantiated as a pointer to the base class. This allows an easier integration between objects.

4.3.2 Optimizer

This interface class is designed to provide a way to integrate the selected quadratic programming solver to the MPC framework. Due to the variability of solvers and options available in each, this class was kept as simple as possible, to make it easier to adapt the functionality of the quadratic programming softwares and their individual options. The idea is to be able to adapt several solvers that use different methods to solve the quadratic problem depending on the structure of the problem and the particular application.

The integration of any solver is thought of as a *wrapper* or a frame to build upon the original functionality of the solver. The approach is to try to solve the quadratic problem including methods as simple and intuitive as possible in the class, so the details of the use of the specific solver are not required by the end user. To achieve that goal, the methods implemented were the following:

- init function: this function reads all the parameters required for the initialization and setting of the variables of the solver from ROS' parameter server. This function should be overloaded for each solver in particular.

- computeOpt function: in this function the quadratic problem is solved. Several solvers require a "*cold*" start run to get better approximations and improve the speed of the calculation, as is the case with qpOASES. In this particular case, it means that there are two different functions, one for the "*cold*" start and one for the "*hot*" start. The function recognizes this difference and launches

the appropiate methods in each case. The results are stored in a protected global variable for the class. The exceptions thrown by the solver are also made easier to read and interpret for the end user, through messages that are visible in the terminal window.

- getOptimalSolution function: this function is made to be used outside of the scope of the optimizer class instance, to be able to obtain the optimal solution, since they are protected inside the class.
- getConstraintNumber function: this function is made to be used outside the scope of the optimizer class instance, to provide the number of constraints wherever needed.
- getVariableNumber function: this function is made to be used outside the scope of the optimizer class instance, to provide the number of variables involved in the quadratic problem wherever needed.

The methods of this class can be extended in the case that other software need more specific functionality. These are the functions that work with the chosen solver for this thesis, but there is a great interest of expansion in this direction to allow the user to decide which optimization algorithm suits best the application.

4.3.3 Simulator

This class is a frame for methods to simulate numerically a determined system. It works as a black-box: given some inputs and a sampling time, it delivers outputs. Since the idea is to replicate numerically as accurately as possible the behavior of the real system, the equations used in the simulator are the ones that provide the closest results to the outputs of the real system. Thus, in the quadrotor case, the equations being used to simulate are the ones from the non-linear system. The functionality of this class is quite small, and it is all detailed in one single function:

- simulatePlant function: as described before, this function comprehends the whole functionality of the class. It takes as inputs arrays of double representing the current states, the control inputs and the sampling time; and it delivers as outputs the array of states for the next iteration.

4.3.4 ModelPredictiveControl

This is the main class that unifies all the functionality of the previous ones to actually solve the quadratic problem posed by each MPC iteration. In the end, the methods from this class are the main methods to be used by the end user in the MPC implementation. This class also allows developers to work in their own variety of MPC, as they are a lot of varieties existing and also being developed. For example, there is currently ongoing work on developing a class to implement Learning Based Model Predictive Control, in which statistical learning algorithms are used to "learn" the non-linearities and imperfections of the model and continue updating it as time goes by, thus allowing to reduce the initial modeling effort to obtain a model that is as accurate as possible. The class developed in this project solves the standard Model Predictive Control problem.

All of these classes would have to adapt their main functionality to the following three methods to keep the ease of use of the library. Any additional functionality should be added within one of these three methods. A flowchart for each method will be provided to facilitate the understanding of each method.

- resetMPC function: this function takes a pointer to an object of the Model and Optimizer classes and sets them into the ModelPredictiveControl object, therefore providing the class with all the information and methods from these.This is used to obtain information about the model, and the optimization problem, also stored in the Parameter Server as configuration files written in YAML format. The benefit from this is that the information in the Parameter Server can be modified and used in the program without the need to recompile.

  Here is where the benefits of using interface classes comes into play: because of this, the pointers to the aforementioned objects can be instantiated as interface classes despite what object it is, as long as they are derived from them. This makes the implementation in the main function easily interchangeable between systems, solvers and simulators as long as they are provided.

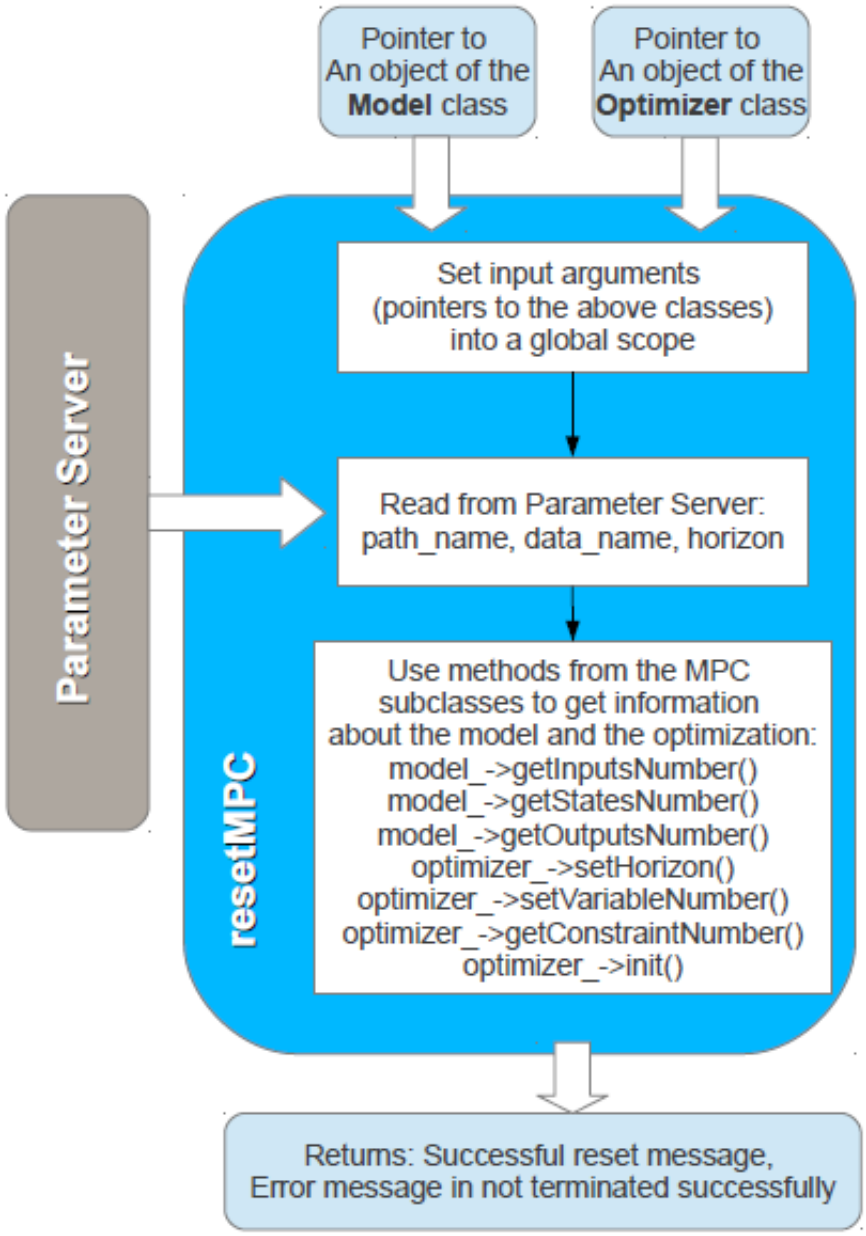


Figure 4.3: Flowchart for the *resetMPC* function.

- initMPC function: in this function the variables that will store critical information of the MPC will be initialized. The state-space representation is taken from the Model class provided and the extended A and B matrices are calculated. The extended matrices refer to the matrices that result from the reorganization of all the recursive expressions for the system at any time in the future until the number of steps specified by the prediction horizon [7]. Then the function reads all the remaining information required to set the MPC problem from the parameter server.

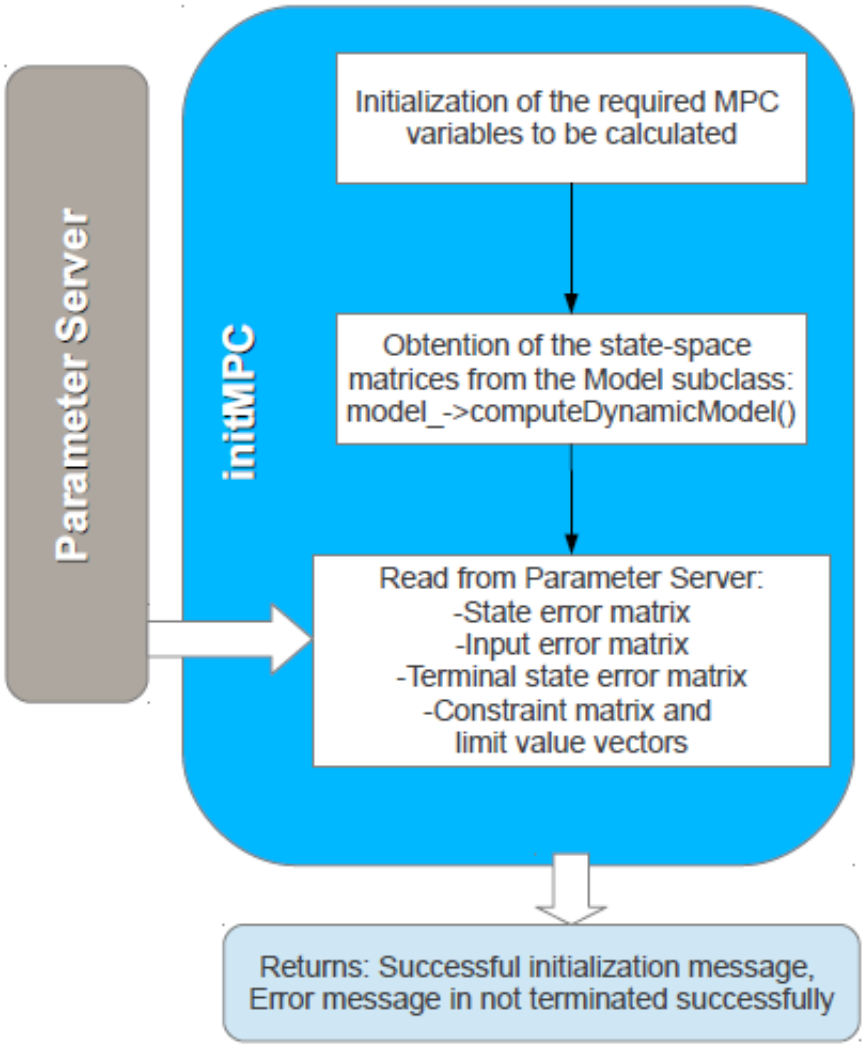


Figure 4.4: Flowchart for the *initMPC* function.

- updateMPC function: this function is where the actual MPC problem is solved. The function takes the current state of the system and the desired reference for the system as inputs, and uses all the previously computed variables and parameters required to assemble the quadratic problem, then it summons the functionality from the optimizer class in order to solve the problem and provide a solution.

  Due to the linearization process, the model is now expressed in terms of variations around the linearization point given in the previous chapter. Therefore, the calculation of the control signal comprises two steps: a steady state solution, which will be the control signal required to keep the

platform in its current state; and the variation, that will be calculated by the MPC controller. This is done in the software by a Jacobi decomposition to solve the $Ax = b$ system, where $b = x_{ref}(A - \mathbb{I})$. This comes from the manipulation of the state space representation of the system to solve for u_{ref} assuming that $x_{ref} = x_{ref+1}$, since the system is supposed to stay in the same state.

Then the function computes the Hessian matrix and the gradient vector stated in equation 2.2 to use the optimizer functionality and solve the MPC problem.

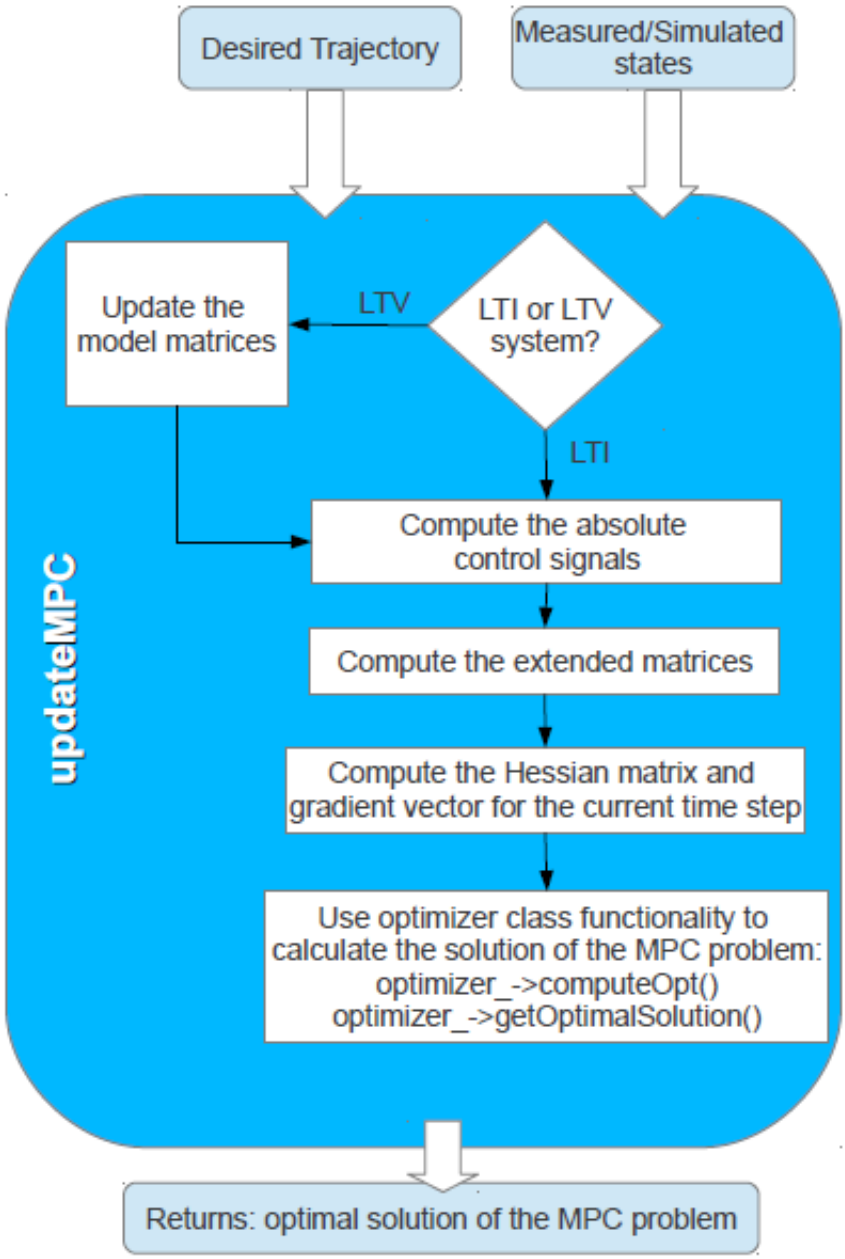


Figure 4.5: Flowchart for the *updateMPC* function.

4.4 Summary

In this chapter, the proposed architecture for the MPC library has been described. The resulting library is a useful guide to build MPC applications as it allows the integration of several quadratic programming solvers, models and simulators (if developed) while providing easy and transparent communication via ROS. The main design goals for this proposal were the modularity and expandability of the code. By using properties from OOP such as inheritance and interface classes, the library allows easy code reuse and quick exchangeability of different instances of each of the classes. This makes the library a good platform to perform comparisons between different QP solvers and make the best choice for a particular application, for example.

5 Results and Discussion

This chapter presents and analyzes the results of the implementation of the developed MPC framework on two different benchmarks. The first benchmark is the simulator for a well known tank system present here at the Control labs. This system was developed as a small test platform of the software before using it on the quadrotor simulator. The second benchmark is the simulator for the quadrotor, which comes from the modeling effort made in chapter 3. For each case, the structure to present the results will be divided in three sections: Simulation settings and parameters, Results and Discussion.

5.1 Tank system simulations

5.1.1 Simulation settings and parameters

The tank system's equations were provided with their correspondent identified parameters. The system is made of two tanks that are connected by gravity: one tank provides the inflow for the other and from there to a water deposit. There is also a small pump, its operation voltage represents the input to the system. The states of the system are the two tank level, of which the desired output is the level of the second tank. The system is depicted below as in Figure 5.1:

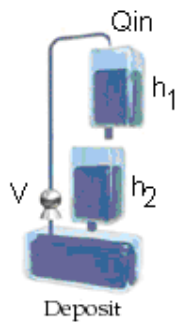


Figure 5.1: Representation of the tank system.

The constraints on the system arise from the size of the tanks, which limit the level of water that the system can handle. The constraints for the levels of the tanks, H_1 and H_2 respectively, and the input voltage are shown below:

$$0 \leq H_1 \leq 25[cm]$$
$$0 \leq H_2 \leq 28[cm]$$
$$0 \leq V \leq 10[V]$$

Regarding the tuning of MPC, the parameters available to adjust are the horizons and the weight matrices in the cost function. The selection of a proper prediction horizon for any MPC application is dependant on the dynamics of the system that is being controlled. This choice is of great influence in the size of the optimization problem to solve, and therefore in the computational power required to provide deterministic operation. If the horizon is too short, the prediction will not give information about future control signals and might create unstability in the controller [8]. On the other hand, if the horizon is too long, the optimization problem to solve could be too large to solve in each time sample, depending on the

timing requirements for each case. The selected prediction horizon of 10 samples at a rate of 100 Hz was obtained starting from small values, where the optimization problems were mostly unfeasible, and increasing until feasibility was achieved, as well as a decent control signal. Further tuning is made with the weight matrices. The weight matrices are also selected in a trial and error fashion.

The specifications for the simulation of the MPC on the tank system is presented below.

Prediction horizon	10 samples at a frequency of 100 Hz
Weight matrices	Specified in Appendix A
Input constraints	$0 \leq V \leq 10[V]$
State constraints	$0 \leq H_1 \leq 25[cm]$ $0 \leq H_2 \leq 28[cm]$
Processor	Intel ® Core™2 Duo @ 3.0 GHz

5.1.2 Results

The results of the simulation of the tank system under the MPC controller are shown below.

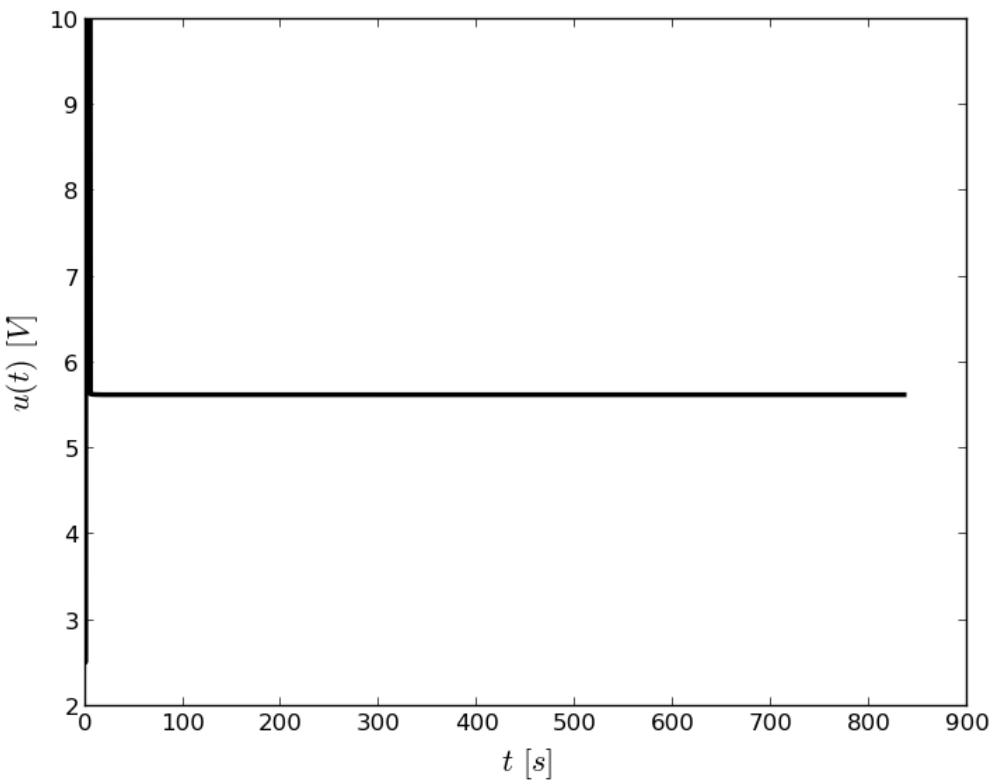


Figure 5.2: Voltage inputs calculated for the tank system by the MPC controller.

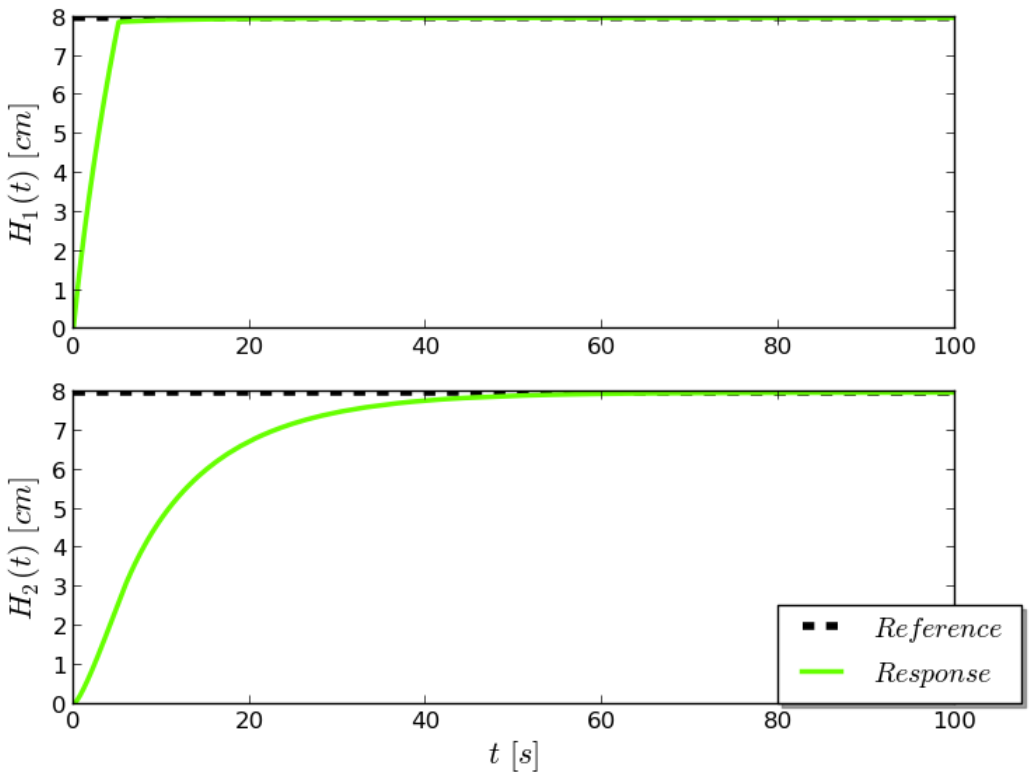


Figure 5.3: Reference level and actual level of the simulated tanks.

5.1.3 Discussion

Directly from Figures 5.2 and 5.3, it is easy to see the controller acting on the input voltage and its influence on the tank states. Since this is a regulation problem for a relatively simple process, the MPC's features are not giving any advantage in comparison with standard PID controllers. However, due to the receding horizon the controller can introduce a feed forward action in a more natural way and can react better to disturbances than a classic controller. In Figure 5.2 the input constraint takes effect, limiting effectively the input voltage to the pump to 10 volts. It is important to notice that the output variable, H_2, takes some time to get to the required setpoint because the speed at which it can be filled is limited by Torricelli's principle for fluids flowing out under the sole influence of atmospherical pressure from a tank. However, level H_1 does rise quickly as the pump sends enough water to fill the tank to the point where the maximum flow of water out of it is achieved.

5.2 Quadrotor simulations

5.2.1 Simulation settings and parameters

Since the quadrotor has only four actuators to move in six degrees of freedom, it is by definition, an underactuated system. Hence, there will be only four variables that will be controllable. The choice made was to control the x, y, z positions and the orientation around the z axis (ψ). It is important to mention that the MPC is not sending directly the input signals to the motors. The reason for this is that the model that is being used to perform the calculations is a linearized model, therefore the state and input variables represent deviations around an operation point. The MPC is instead controlling these linearized input variables and sending that to modify the initial operation point. The operation point being used is calculated by simple force balance in the hover condition, to obtain the required angular

speed of the rotors to keep the quadrotor static in the air, which is approximately 360 radians per second. That way, the control signal vector going to the quadrotor simulator is built as follows:

$$\mathbf{u} = \underbrace{\bar{\mathbf{u}}}_{\text{operation point}} + \underbrace{\Delta\mathbf{u}}_{\text{controlled by MPC}}$$

In order to adjust to this, the constraints in the input were also modified, through a displacement of the operation range for this variation of the angular speed. The original operation range is taken from the experiments realized by Sun [24], which is $130 \leq \omega_i \leq 500$, in radians per second. In order to be congruent with that range, the variation of the angular speed is limited between $-230 \leq \Delta\omega_i \leq 140$ around the operation point, again in radians per second.

In this particular case, the tuning of the MPC is also influenced by the model, since the one being used for prediction is a linearized version. A very long prediction in the future might mean moving too far away from the operation point, which will cause erratic predictions due to nonlinearities, additionally to the increase of the computation time. Also, these erratic prediction may lead the controller to unfeasibility and therefore unstability. In this system that is not stable in open loop, this could mean crashing the quadrotor. The selected prediction horizon of 30 samples at a rate of 120 Hz is long enough to give good predictions ahead for the system while keeping the problem at a reasonable size. This result was obtained by trial and error.

With the weight matrices, the procedure is also made in a trial and error fashion. A good initial guess for the quadrotor model is taken from previous implementation parameters [4]. The simulation is performed in a computer with an Intel ® Core™2 Duo running at 3.0 GHz. The complete specifications for the simulation are presented in the following table.

Prediction horizon	30 samples at a frequency of 120 Hz
Weight matrices	Specified in Appendix A
Input constraints	-230 $\leq \Delta\omega_i \leq 140[rad/s]$
State constraints	-5 $\leq V_x \leq 5[m]$ -5 $\leq V_y \leq 5[m]$ 0 $\leq z \leq 5[m]$
Processor	Intel ® Core™2 Duo @ 3.0 GHz

The trajectory reference used to test the MPC in the quadrotor simulator is based in simple decoupled movements in each one of the controlled directions, done one at a time. The movement consists in four stages: elevation from the floor, change in orientation (a change in the yaw angle), rotating back to the original orientation, and movements in the X and Y axes, in that order.

5.2.2 Results

The results of the simulation with the mentioned parameters and settings are shown below:

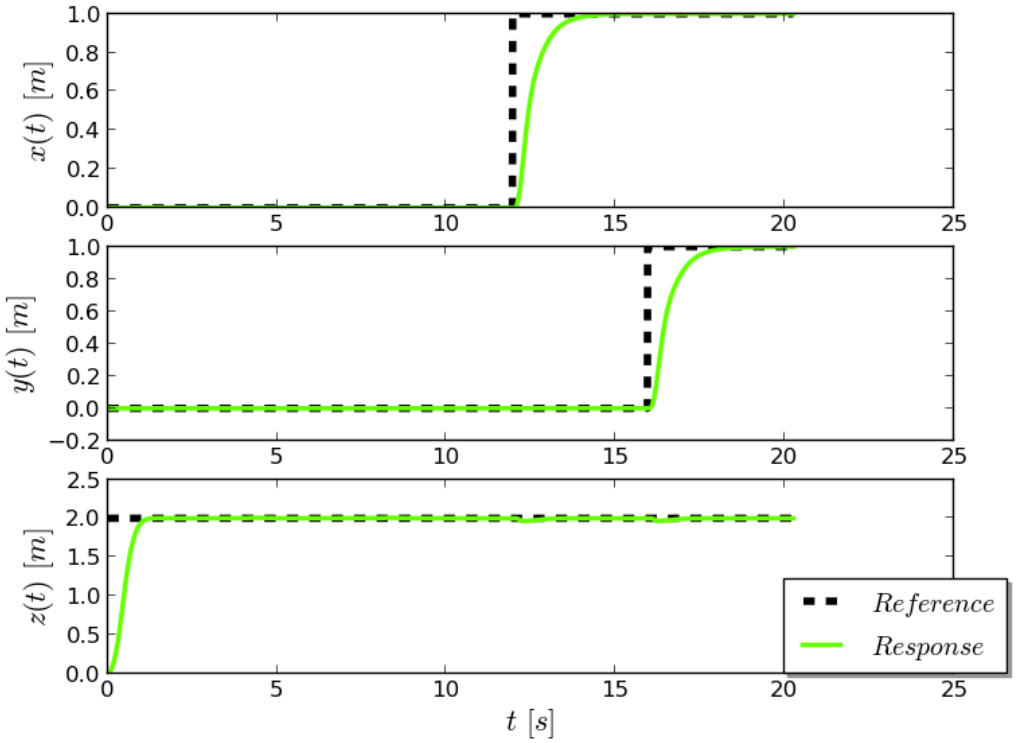


Figure 5.4: Trajectory reference and actual trajectory positions of the simulated platform.

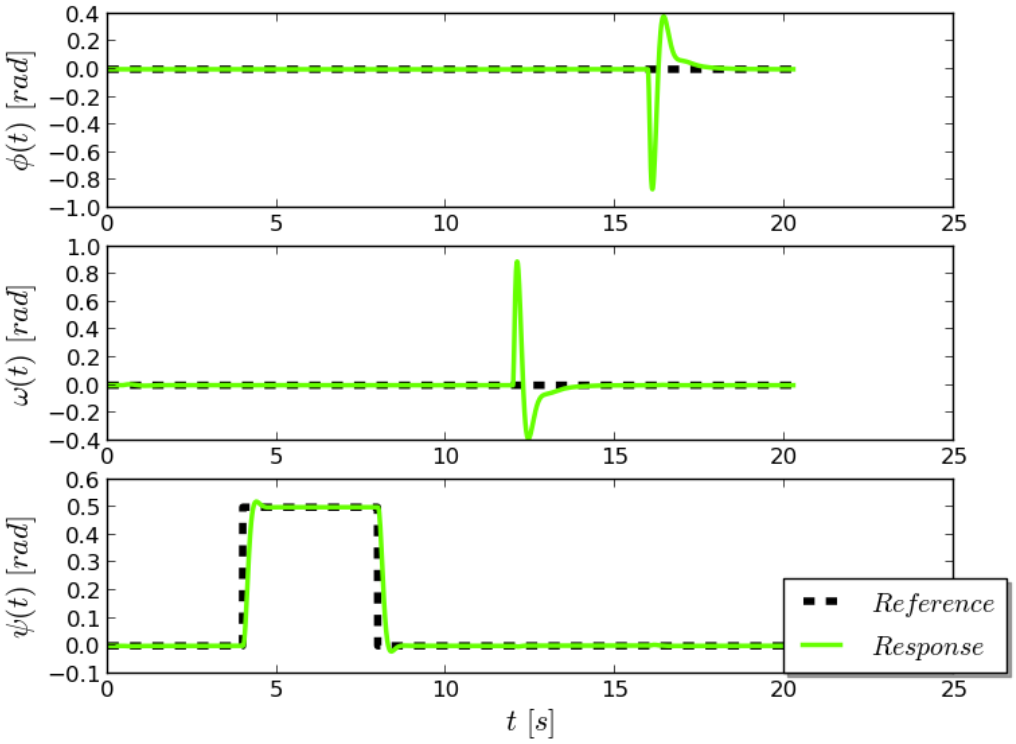


Figure 5.5: Trajectory reference and actual trajectory orientations (ψ) of the simulated platform.

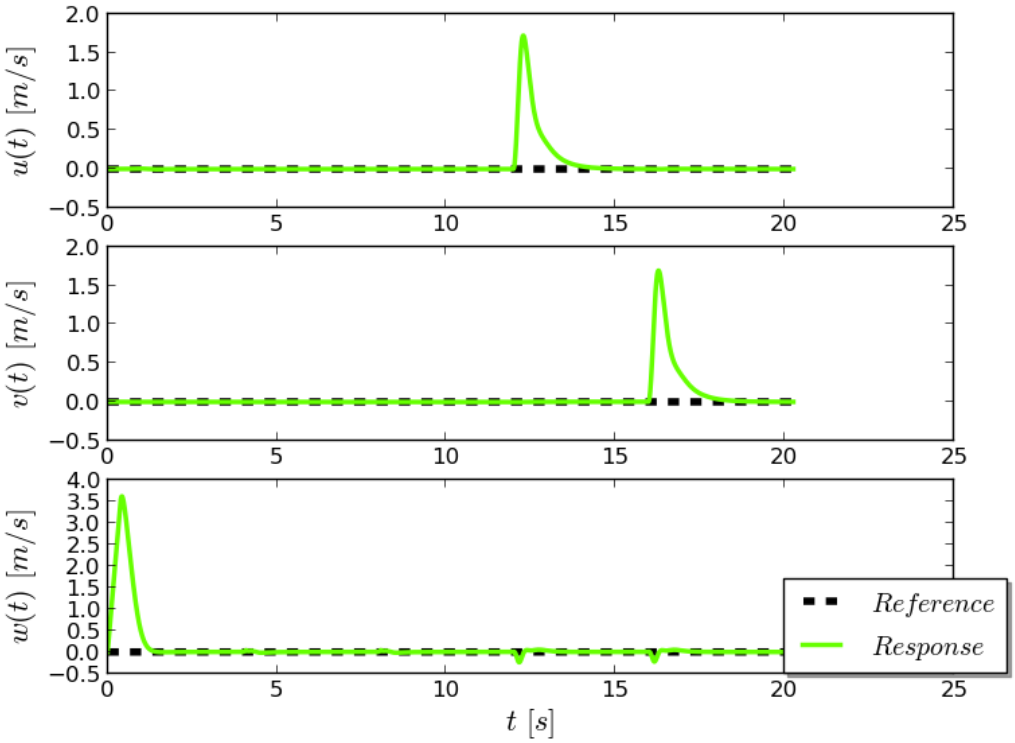


Figure 5.6: Linear velocities of the simulated platform.

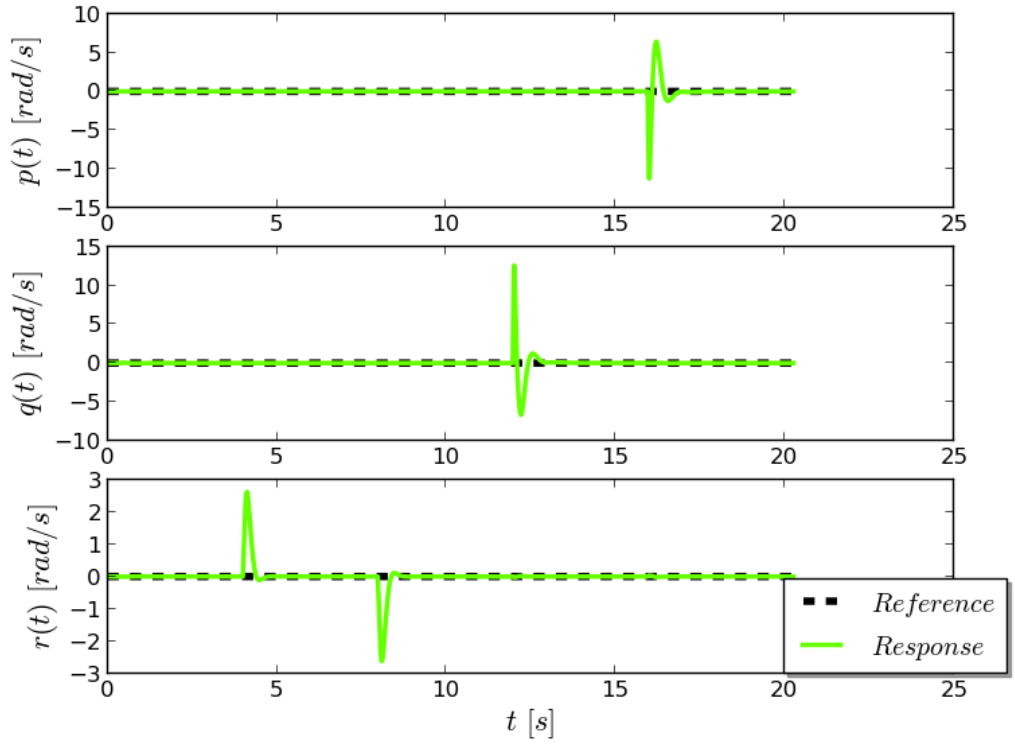


Figure 5.7: Angular velocities of the simulated platform.

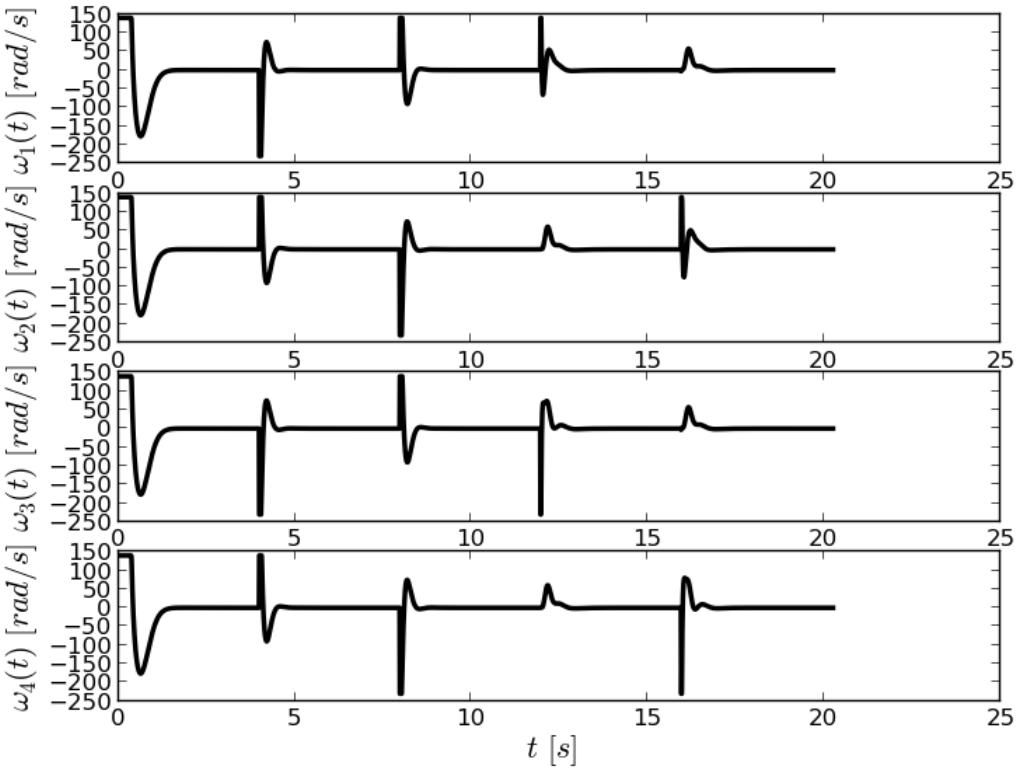


Figure 5.8: Control signals generated by the MPC strategy.

5.2.3 Discussion

The simulations show the performance of the MPC strategy in the simulated quadrotor. Figures 5.4 and 5.5 show the position and orientation response of the system against step input signals on each controlled variable. The response shown presents a smooth behavior, no overshoot and a small settling time correspondant to a fast system like the quadrotor. The prediction horizon used, $N_p = 30$, has a right balance in speed to allow a fast response without overshoot.

The initial set of restrictions considered was bigger than the one used in the simulations shown, since there were restrictions considered on the roll and pitch angles (ϕ and θ) taken in order to assure the stability of the quadrotor. Normally, this would be addressed by a hovering Proportional-Integral (PI) control loop in the original AR-Drone control architecture. However, the consideration of the constraints on these angles lead to unfeasibility in the quadratic programming problem, so the correspondant restrictions were relaxed, and eventually dropped. The stability of the platform was addressed by strenghtening the restrictions on the velocities in the x and y axes, since these velocities are dependant on the roll and pitch angles.

The actual version of the library does not include functions to cope with unfeasibility. This must be considered in further stages of development in order to provide a better operation under the presence of more constraints. Some suggested strategies to cope with unfeasibility are referred by Camacho and Bordons [6]: an initial strategy consists of dropping the state constraints at the initial portion of the horizon in order to make the problem feasible; another way is to create soft constraints from the hard constraints stated initially, and then adding a term in the cost function to penalize constraint violation [16]. Feasibility is important to assure closed-loop stability. One could also use the solution of the unconstrained problem when unfeasibility appears, but this will not guarantee stability.

Figure 5.8 shows the resultant inputs to the simulator, produced by the optimization solver. As it is shown, the control inputs are only responding to the requirements of the simulation trajectory, hence the regular pattern in the signals. The simulator does not take into account sensor noise or environmental perturbations that can arise such as crossed winds that might require more drastic control inputs to the motors.

To make a verification of the disturbance rejection capabilities of the controller, a test using a disturbance model with a uniform distribution, bounded in a defined range was performed. This disturbance model can not be considered as white gaussian noise that would arise from the sensing instruments, but it works for the aforementioned purpose. In a future work, tests with a disturbance model of the sensor noise and environmental effects on the quadrotor can be performed. These results in a more realistic simulation scenario, would give a better insight of the behavior of the controller in a future implementation on the drone. The bounds of the disturbances were defined to be in the $[0, 0.01]$ range for each state. This affects differently some states, but in general it introduces a slight disturbance without destabilizing the platform. The results of the aforementioned tests are shown below from Figure 5.9 to 5.13.

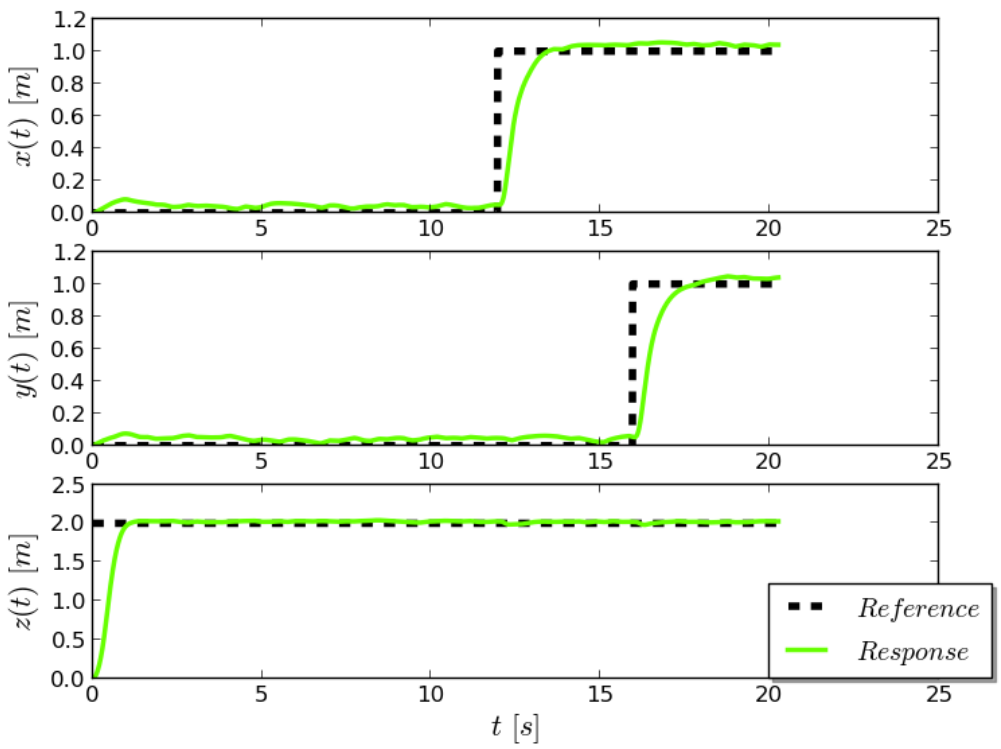


Figure 5.9: Trajectory reference and simulated trajectory positions of the platform with the disturbance model.

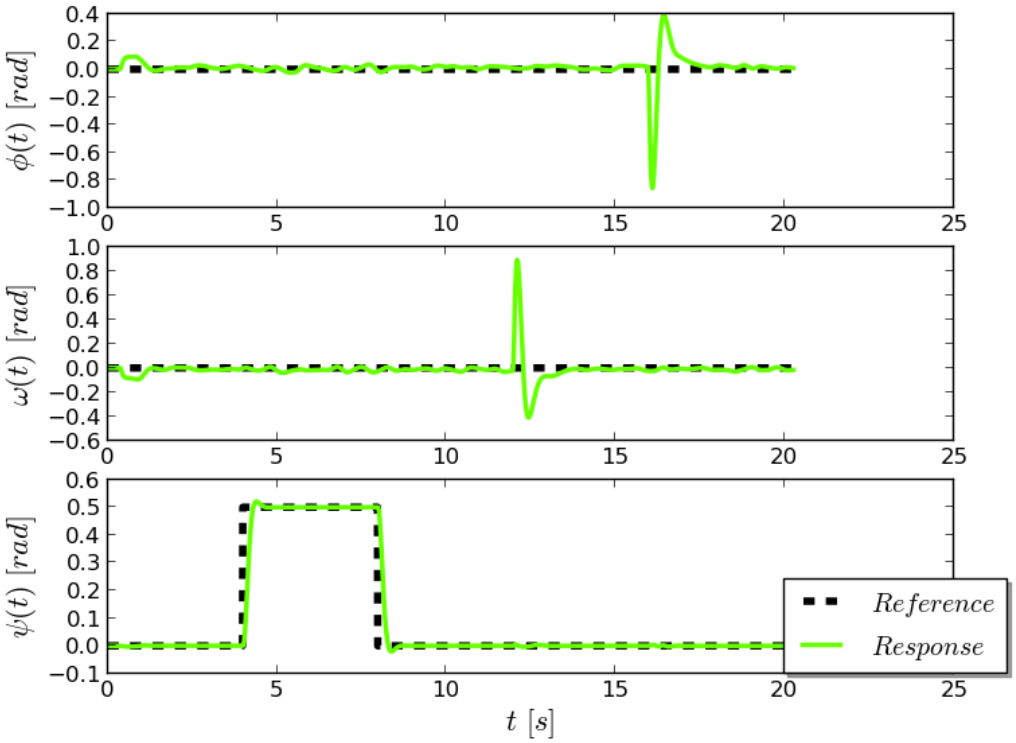


Figure 5.10: Trajectory reference and simulated trajectory orientations (ψ) of the platform with the disturbance model.

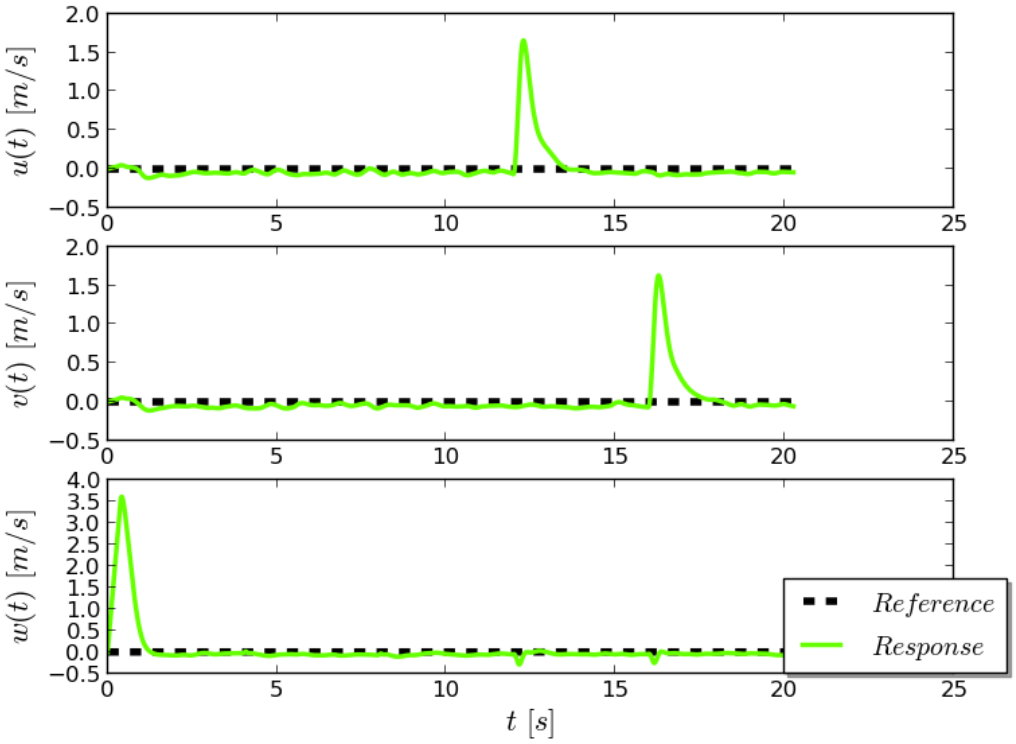


Figure 5.11: Linear velocities of the simulated platform with the disturbance model.

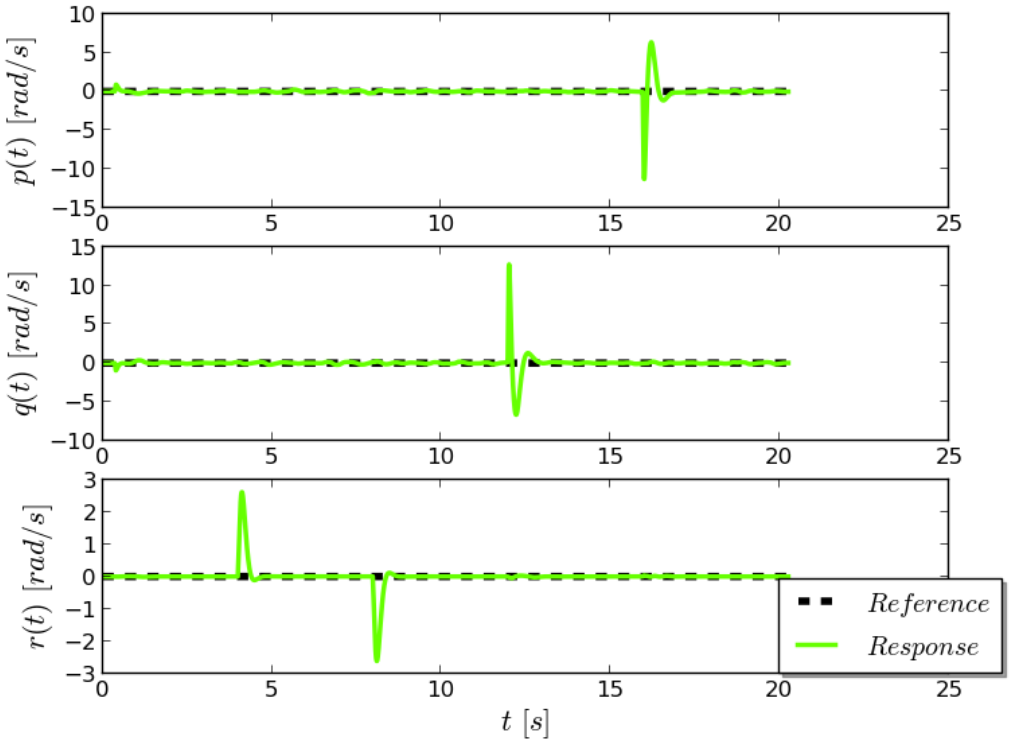


Figure 5.12: Angular velocities of the simulated platform with the disturbance model.

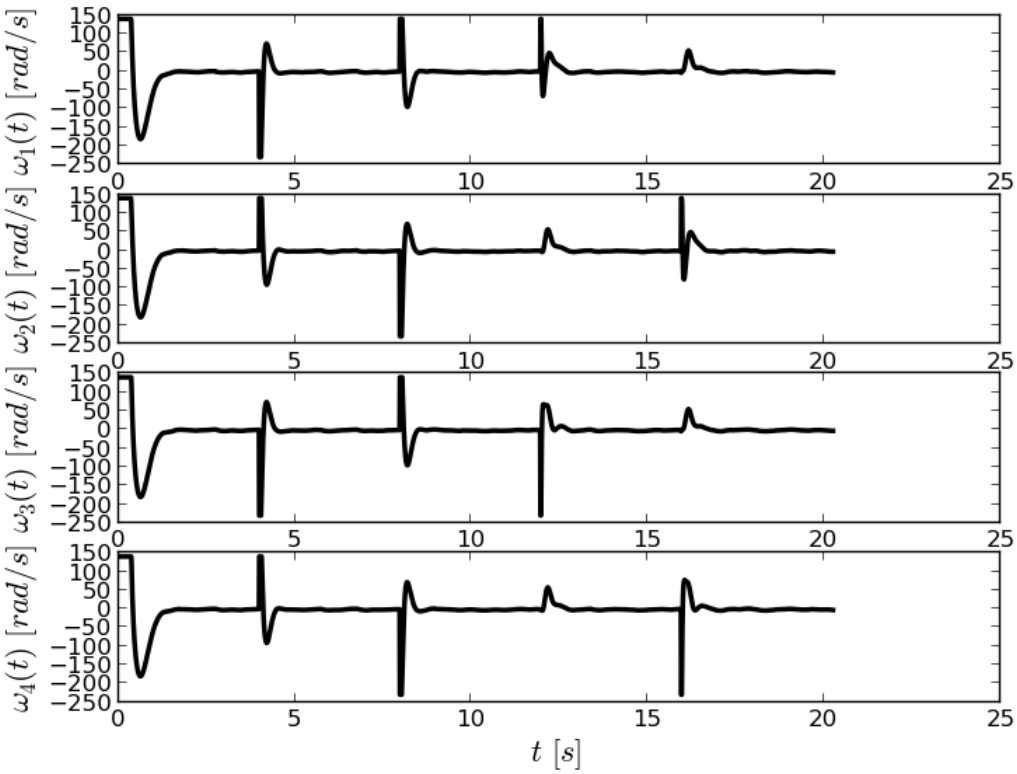


Figure 5.13: Control signals generated by the MPC strategy with the disturbance model.

The effect of the disturbances is noticeable in Figure 5.13. With the same disturbance magnitude, there is a slight overshoot in the X and Y coordinates as Figure 5.9 shows, indicating that these are the most sensitive states, since these are critical for the stability of the platform. Despite that, the controller manages to keep the simulated drone stable. This simulation is not performed in real-time, therefore, these results are used to give insight about the proper functionality of the controller instead of the performance. However, the qpOASES solver has a method to limit the computation time of the optimization problem solution to implement it on real-time applications. This allows to obtain suboptimal solutions that can be sufficiently accurate to be implemented in the simulator and also in the real platform. These time cutting methods are not used in this report, but this remains as a topic of interest for an implementation on the quadrotor. In these simulations the computation times for both tests (with and without disturbances) had similar timing, so disturbance rejection seems to not affect this particular topic. The time per iteration in each test is shown below.

Test	**Average Time per MPC iteration**
Without disturbances	0.0331 [s]
With disturbances	0.0337 [s]

The simulator was tested in a second trajectory, which is more representative of the kind of motion that the drone would be required to perform in real life applications. It consists in following a square reference of 2 meters of width, one in the horizontal XY plane and one in the vertical XZ plane. The results using this reference are shown below.

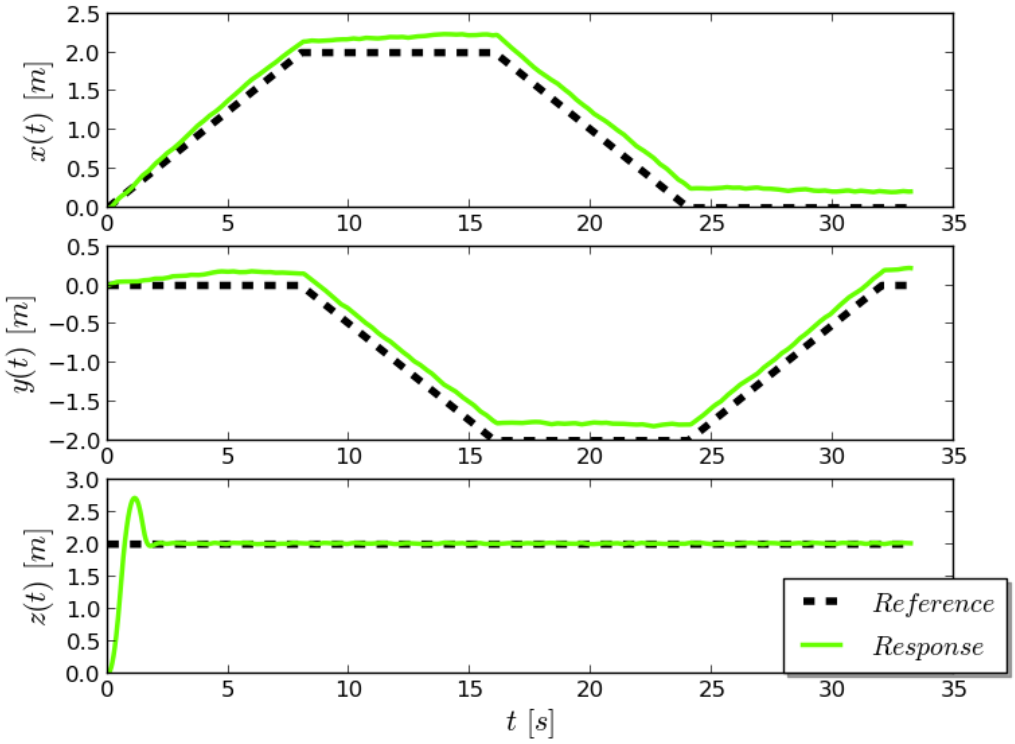


Figure 5.14: Positions of the simulated quadrotor with the square trajectory.

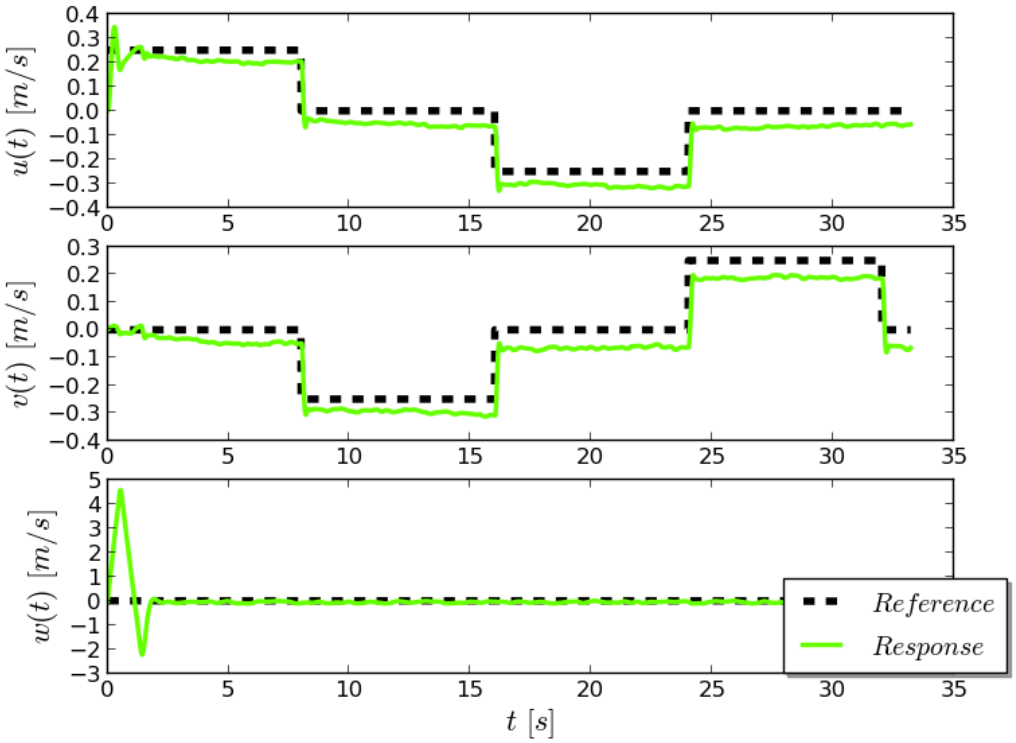


Figure 5.15: Linear velocities of the simulated quadrotor with the square trajectory.

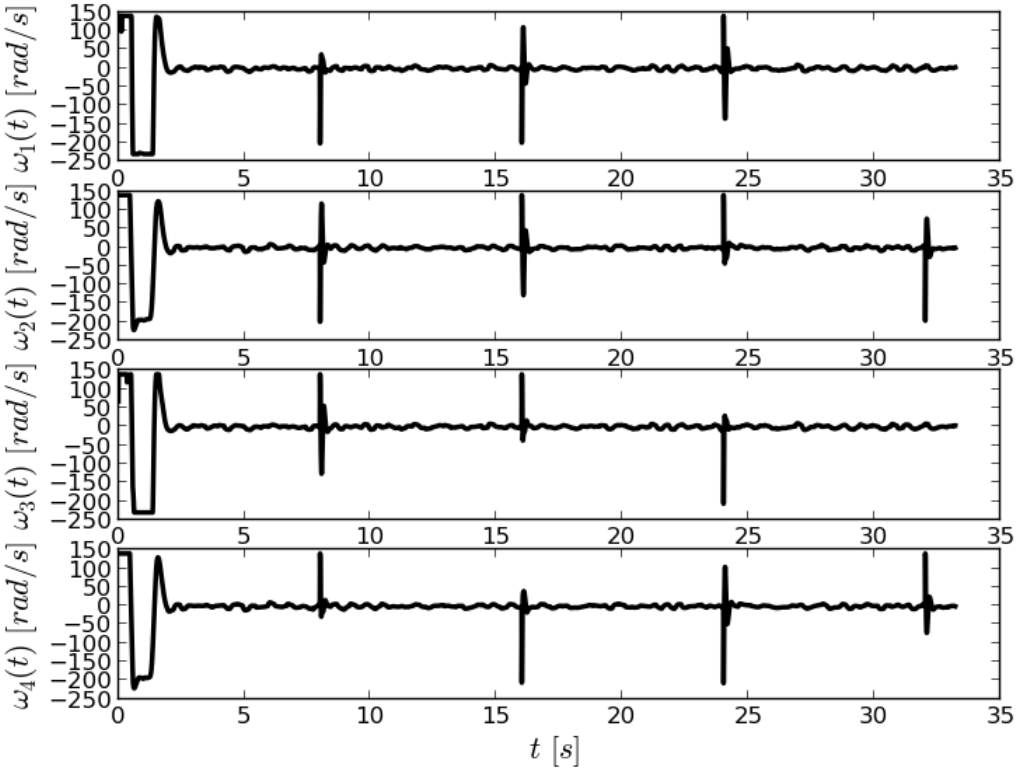
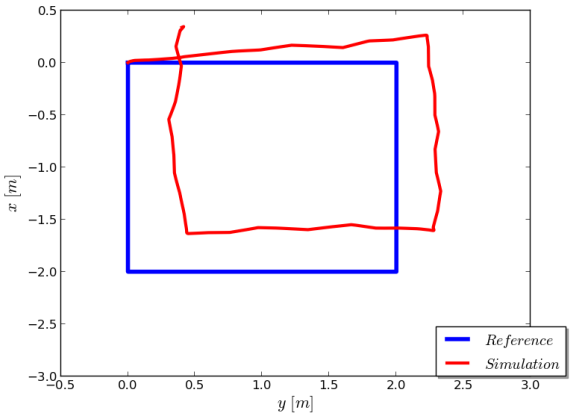
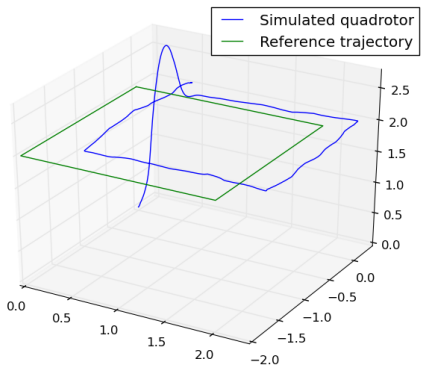


Figure 5.16: Calculated inputs from the MPC to the simulated quadrotor when using the square trajectory.

To improve the visualization of these results, the positions are translated into a 2D and 3D space.



(a) Trajectory of the simulated quadrotor in the XY plane.



(b) Trajectory of the simulated quadrotor in a three dimensional space.

Figure 5.17: Visualization of the control trajectories of the simulated platform.

The previous plots show that the controller performance is not satisfactory. An increasing offset from the desired path appears in the simulation. One reason for this is that the MPC's performance is highly dependant on the accuracy of the model. The model used for this controller is a linear state space representation around the hover point for the quadrotor, which changes immediately after the take off. As the model goes further away from the linearization point, the accuracy of the model decreases respect to the non-linear model, and even more from the real quadrotor. This affects directly the performance of the controller, as the behavior predicted by the model in the MPC drifts away from the simulated behavior of the quadrotor. As model uncertainty increases, it is possible to deal with this situation increasing the restrictions on the system and their correspondent ranges.

However, this approach can easily lead to unfeasibility, as the system becomes overconstrained. In order to improve this, several variants of the MPC algorithm have been studied and implemented to increase its robustness against imperfections in the model and stronger sources of disturbance. Tube MPC [20], is one of these efforts to increase the robustness of the MPC algorithm when uncertainty is present. This variation uses feedback to assure that the states are contained in a tube of possible trajectories. Another fact that must be taken into account to analyze the results, is that the model used in the controller does not count for disturbances. This was not done in this report, but it is of interest to measure the influence of the disturbance model in the development of a model for MPC controllers and their performance.

6 Conclusions and Future Work

In this project, a C++ library has been developed in order to work as a quick backbone and template to implement MPC applications, throughout the different classes and methods described in Chapter 4. The software qpOASES was successfully integrated to the proposed MPC architecture, with some limitations, since the method to achieve hard real-time solving never worked in this particular case. Nevertheless, the proposed architecture uses OOP in order to allow the integration of different quadratic programming solvers and making them available and interchangeable once they are integrated with one call at the main controller function. By the same principles, the models and simulators (if developed) can be exchanged quickly. The integration with the ROS environment allows quick connectivity with the growing collection of devices and drivers available in the ROS development community, transparent communication between devices and the opportunity to integrate the controller to a ROS network of devices via LAN or Wireless LAN (WLAN).

Two models, and therefore two controllers were built: the tank system, which was made from previous identification efforts made here in the Control Lab; and the quadrotor, which did require modeling work. However, the model was only verified and not validated, since no parameter identification experiments were performed. Instead, the parameters were taken from Sun [24] (2012), in which the identification efforts for the same quadrotor were made. The result was a linearized state space representation of the quadrotor around the hover point. The validation experiments showed that the model developed for the quadrotor simulator performed satisfactorily as it can be seen in Figures 3.2 to 3.16. From these modeling activities, numerical simulators were also developed as testing platforms for both controllers.

Each controller was tested in different situations: the tank system simulator was used to regulate to a setpoint and the quadrotor simulator was used to follow a determined trajectory. In both cases, the controllers achieved good time responses, with no overshoot or steady state error. When a disturbance model was considered in the quadrotor simulation, an offset was observed between the reference and actual trajectories of the platform. There are two possible reasons for this problem: the process model developed does not consider a model for disturbances, and/or the model used in the MPC calculation is a linearized model around the hover point, from which the simulation moves away continuously. For a fast plant like the quadrotor, a low time response is very important to keep airbourne stability. The tuning of the weight matrices was done by trial and error. Despite that, the tuning procedure was relatively fast, as the weight matrices were obtained in a small number of attempts. In the quadrotor case the difficulty increased, due to the size of the problem and the bigger amount of variables involved. This is why MPC controllers require more modeling work than tuning, which is the inverse situation from classic PID controllers.

In the case of MPC controllers for plants with a high number of states such as the quadrotor, the selection of the constraints plays a key role in the solution of the optimization problem that arises in each step. In order to guarantee stability, the active set considered must be feasible first, and an excess on the number of constrained variables or the selection of unrrealistic ranges of operation for some variables might lead to unfeasible problems. Camacho and Bordons [6] book addresses several methods to deal with unfeasibility. In the case of the quadrotor controller, the number of constrained variables was reduced. The strategy to use will depend on the variables involved and the plant that is being controlled, as some constraints can't be redefined in some cases for safety and/or operational reasons.

6.1 Future Work

In the current report, the MPC proposal is only tested in the simulators that were developed. This was due to several hardware concerned issues that were out of the scope of the project to solve. However, the implementation of the MPC in the real quadrotor has always been the main goal, but some of the proposals made here require a special focus in order to succeed. The following work proposals come from ideas that were attempted, but were unsuccessful during the time lapse of this project.

To have a better look at the simulations, Gazebo was tested for these purposes, but the model of the quadrotor that it has includes the internal controller of the drone. A next step on this project could be to explore the use of the MPC over the internal controller of the drone and not directly on the hardware.

In the real quadrotor, the states come from the integration of the feedback from the different sensors present in the drone: the Inertial Measurement Unit (IMU), the sonar altitude sensor and the cameras. One of the ideas is to use a program made by students at the Technological University of Munich that uses Simultaneous Localization And Mapping (SLAM) on the information from the frontal camera, an Extended Kalman filter for state estimation and a PID controller to steer the quadrotor from visual feedback. The point is to use the MPC over the PID controller taking directly the feedback produced by the program via ROS topics to feed the MPC. This was attempted in this project, but the estimated states had an error due to a problem in the calibration process of the drone.

Bibliography

[1] Kostas Alexis, Christos Papachristos, George Nikolakopoulos, and Anthony Tzes. Model predictive quadrotor indoor position control. *2011 19th Mediterranean Conference on Control & Automation (MED)*, pages 1247–1252, June 2011.

[2] Oren Ben-kiki, Clark Evans, and Brian Ingerson. YAML Ain ' t Markup Language (YAML ™) Working Draft 2004-12-28, 2004.

[3] S. Bouabdallah, P. Murrieri, and R. Siegwart. Design and control of an indoor micro quadrotor. *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, pages 4393–4398 Vol.5, 2004.

[4] Patrick Bouffard. On-board Model Predictive Control of a Quadrotor Helicopter: Design, Implementation, and Experiments. Technical report, University of California at Berkeley, 2012.

[5] Tommaso Bresciani. Modelling , identification and control of a quadrotor helicopter. Master's thesis, Department of Automatic Control, Lund University, 2008.

[6] E. F. Camacho and C. Bordons. *Model Predictive Control*. Springer, 1999.

[7] H. J. Ferreau. An online active set strategy for fast solution of parametric quadratic programs with applications to predictive engine control. Master's thesis, University of Heidelberg, 2006.

[8] F Gabrielsson. Model Predictive Control of SkeboåWater system. Master's thesis, Kungliga Tekniska Högskolan, 2012.

[9] Gabriel Hoffmann, Haomiao Huang, Steven Waslander, and Claire Tomlin. Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment. *AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 1–20, August 2007.

[10] Morten Hovd. A brief introduction to Model Predictive Control. Course notes for Optimization and Control (TTK4135) at Norwegian University of Science and Technology, 2004.

[11] Patrik Johansson and Jacob Bernhard. Advanced control of a remotely operated underwater vehicle. Master's thesis, Institutionen för systemteknik Department of Electrical Engineering vid Tekniska Högskolan vid Linköpings Universitet, 2012.

[12] Ida Kristoffersson. Model predictive control of a turbocharged engine. Master's thesis, Kungliga Tekniska Högskolan, 2006.

[13] Marcelo De Lellis and Costa De Oliveira. Modeling , Identification and Control of a Quadrotor Aircraft. Master's thesis, Czech Technical University in Prague, 2011.

[14] R. Mahoney, V. Kumar, and P. Corke. Multirotor aerial vehicle: Modeling, estimation, and control of quadrotor. *IEEE Robotics Automation Magazine*, 19(3), 2012.

[15] F. Manenti, G. Buzzi-Ferraris, I. Dones, and H. A. Preisig. Generalized class for nonlinear model predictive control based on bzzmath library. In *Icheap-9: 9th Conference on Chemical and Process Engineering*, 2009.

[16] A. Molero, R. Dunia, J. Cappelletto, and G. Fernandez. Model predictive control of remotely operated underwater vehicles. *IEEE Conference on Decision and Control and European Control Conference*, pages 2058–2063, 2011.

[17] J. Nocedal, S.J. Wright, and S. M. Robinson. *Numerical Optimization*. Springer, 1999.

[18] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS : an open-source Robot Operating System. In *Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, 2009.

[19] Guilherme Vianna Raffo. Modelado y control de un helic optero quadrotor. Master's thesis, Universidad de Sevilla, 2007.

[20] S. Rakovic, B. Kouvaritakis, C. Cannon, C. Panos, and R. Findeisen. Parameterized Tube Model Predictive Control. Technical report, Oxford University, 2010.

[21] J. Richalet. Industrial applications of model based predictive control. *Automatica*, 29(5):1251–1274, September 1993.

[22] M. Rosendo Dalte, W. Lages Fetter, and J. A. Vasconcelos Alves. A library tailored for real-time implementation of model predictive control. In *9th International Symposium on Robot Control (SYROCO'09)*, 2009.

[23] A. L. Salih, M. Moghavvemi, H. A. F. Mohamed, and K. S. Gaeid. Modelling and PID controller design for a quadrotor unmanned air vehicle. *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, pages 1–5, May 2010.

[24] Y. Sun. Modeling, Identification and Control of a Quad-Rotor Drone using Low Resolution Sensing. Master's thesis, University of Illinois at Urbana-Champaign, 2012.

A Appendix: MPC Parameters

A.1 Tank System

A.1.1 State Space matrices

The equations of the tank:

$$\begin{aligned} A_t \frac{dH_1}{dt} &= \beta V - A_d \sqrt{2gH_1} \\ A_t \frac{dH_2}{dt} &= A_d \sqrt{2gH_1} - A_d \sqrt{2gH_2} \end{aligned} \tag{A.1}$$

After a linearization process:

$$\begin{bmatrix} \dot{H}_1 \\ \dot{H}_2 \end{bmatrix} = \begin{bmatrix} -\frac{A_d\sqrt{2g}}{2A_t\sqrt{H_1*}} & 0 \\ \frac{A_d\sqrt{2g}}{2A_t\sqrt{H_1*}} & \frac{-A_d\sqrt{2g}}{2A_t\sqrt{H_2*}} \end{bmatrix} \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} + \begin{bmatrix} \frac{\beta}{A_t} \\ 0 \end{bmatrix} \mathbf{V} \tag{A.2}$$

Substituting the values of β = 3.96, $A_t = 15.52[\text{cm}^2]$, $A_d = 0.178[\text{cm}^2]$, and discretizing with Backward Euler approximation and a sampling period of 0.01 seconds:

$$\begin{bmatrix} \dot{H}_1 \\ \dot{H}_2 \end{bmatrix} = \begin{bmatrix} 0.9992 & 0 \\ -0.000803 & 1.001 \end{bmatrix} \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} + \begin{bmatrix} 0.002551 \\ 0 \end{bmatrix} \mathbf{V} \tag{A.3}$$

A.1.2 Weight matrices

State error weight matrix:

$$\mathbf{P} = \begin{bmatrix} 100.0 & 0 \\ 0 & 200.0 \end{bmatrix}$$

Input Error weight matrix

$$\mathbf{Q} = 0.001$$

Terminal State weight matrix:

$$\mathbf{R} = \begin{bmatrix} 1000.0 & 0 \\ 0 & 1000.0 \end{bmatrix}$$

A.2 Quadrotor

A.2.1 State Space matrices

From the theoretical derivation made in chapter 3, and discretizing using Backward Euler approximation and a sampling period corresponding to 120 Hz:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0.0083 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0.0083 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0.0083 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0.081423 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -0.081423 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0.0083 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0.0083 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0.0083 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1.1325e-04 & 1.1325e-04 & 1.1325e-04 & 1.1325e-04 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -8.365e-03 & 0 & 8.365e-03 \\ 0.0109 & 0 & -0.0109 & 0 \\ -3.6788e-04 & 3.6788e-04 & -3.6788e-04 & 3.6788e-04 \end{bmatrix}$$

A.2.2 Weight matrices

The definite matrices chosen for the simulations of the quadrotor are presented as follows:
State Error weight matrix:

$$\mathbf{P} = \begin{bmatrix} 25.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 25.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 30.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.001 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 30.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}$$

Input Error weight matrix:

$$\mathbf{Q} = \begin{bmatrix} 0.00001 & 0 & 0 & 0 \\ 0 & 0.00001 & 0 & 0 \\ 0 & 0 & 0.00001 & 0 \\ 0 & 0 & 0 & 0.00001 \end{bmatrix}$$

Terminal State weight matrix:

$$\mathbf{R} = \begin{bmatrix} 30.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 30.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 120.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 120.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}$$