Integration of Electrochromic Smart Windows in Building Automation Systems

MARCUS HULTMARK VAREJÃO

Stockholm December 2013

Abstract

Contents

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, 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 that has been developed during the last years is the submarine of the university, named PoseiBot. The development of this submarine robotic platform has been made in several phases through the years. In the latest phase, a Model Predictive Control (MPC) strategy was implemented to control the submarine achieving good results in terms of the control effort, error reduction and robustness. MPC is an advanced control technique based on solving an optimal control problem with a finite prediction horizon in each sample. 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 the submarine'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 quadrotors as a robotic platform for powerline inspection, where DETAILS ABOUT THE CONTROL STRATEGIES AND HOW IT IS IMPLEMENTED. 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.

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 some restrictions in some 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 ROS apply, and it also represented an opportunity to extend the usage of this tool within the group. FIX MEEE

1.2 Purpose

The purpose of this project is the creation of a ROS package to implement MPC strategies in different platforms 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 there is when quadratic programs arise.

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 [**?**]. 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 conducted, 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 in the development of the model, not in the tuning of the controller. If a proper model is developed, the tuning of the controller comes naturally from specification parameters. 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 [**?**], [**?**], [**?**] and [**?**] to mention some examples.

In [**?**], a linear MPC strategy is used to provide the a system of water dams an adequate flow of water required for the paper mills, 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 [**?**], the problem to solve is the trajectory tracking of a submarine 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 laptop performing the MPC calculations and sending back the control signals to the ROV. The data processing was done in MATLAB ®, and the acquisition and GUI was done in LabVIEW ™. Using this strategy, substantial improvements in comparison with traditional PID strategies were obtained in tracking performance and control effort.

In [**?**], 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 [**?**], ACADO is used to implement 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 [**?**], 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, tying different optimization algorithms or differential solvers might be of interest, and these parameters are not customizable if this class is used.

This was taken care of in [**?**]. 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 [**?**], [**?**], [**?**], [**?**], [**?**], [**?**] and [**?**]. 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 [**?**] 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 [**?**] 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 due to 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 chap:Introduction chapter, 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: chap: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, chap:Dynamic Modeling of a Quadrotor, 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, chap:Software Architecture and Implementation 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 set of feasible solutions that minimizes or maximizes a performance index. These can be subdivided in continuous and discrete, depending on the nature of the variables. Their standard mathematical formulation is described as follows:

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

There are three elements to identify in the standard formulation: the cost function, the equality constraints and the inequality constraints. Depending on the type of the cost function the problem can be linear or quadratic, which are the most common types.

2.1.2 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 $\alpha x + (1-\alpha)y \in S, \forall \alpha \in [0,1]$.

2.1.3 Feasibility

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 [**?**]. 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 past inputs and outputs and the future control signals to the model.
- 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 [**?**].

- 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).

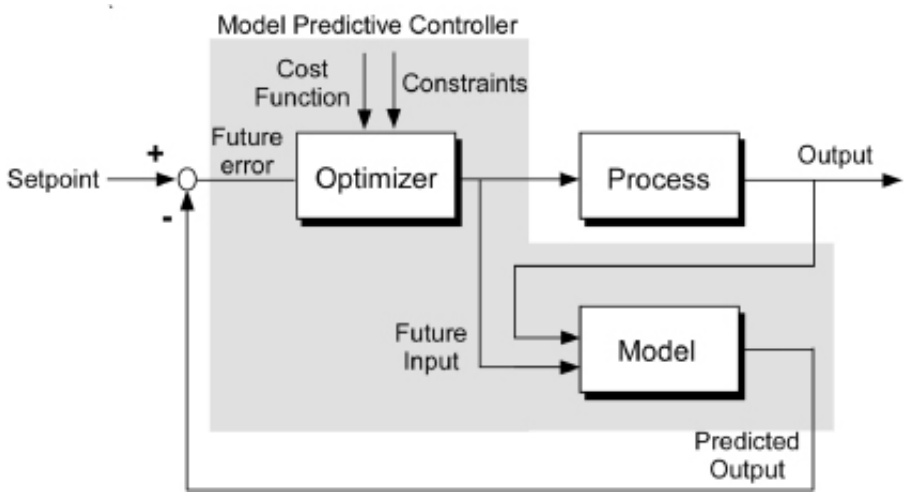

Figure 2.1: Block diagram of the basic elements in Model Predictive Control [**?**].

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 hanlding 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 [**?**] 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 of it 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:

$$y(k) = \sum_{i=1}^{N} h_i u(k-i) = H(z^{-1})u(k) \tag{2.2}$$

  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:

$$y(k) = \frac{B(z^{-1})}{A(z^{-1})}u(k) \tag{2.3}$$

  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 SISO systems.

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

$$\begin{cases} x(k+1) = Ax(k) + Bu(k) \\ y(k) = Cx(k) \end{cases} \tag{2.4}$$

  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 is straightforward to use for MIMO systems and the control law

will always be a linear combination of the state vector. A drawback might be that the calculations could get complicated if a state observer is needed, as sometimes the state selection doesn't have any physical meaning.

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

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:

$$V(u) = \sum_{i=1}^{N_p} [\hat{y}(k+i|k) - r(k+i)]^2 + \sum_{j=1}^{N_c} [\triangle u(k+i-1)]^2 \tag{2.5}$$

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 linear or of a different order. Most MPC implementations use quadratic objective functions because they are independant on the sign of the optimised variable 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 [**?**], which is of the following form:

$$V(u) = \frac{1}{2} \sum_{i=k_0}^{k_0+N_p-1} (y_k - y_{ref})' Q (y_k - y_{ref}) + (u_k - u_{ref})' R (u_k - u_{ref}) + \frac{1}{2} (y_{k_0+n_p} - y_{ref})' P (y_{k_0+n_p} - y_{ref}) \tag{2.6}$$

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 [**?**] 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:

$$\underline{U} \leq u \leq \bar{U} \tag{2.7}$$

$$\underline{A} \leq G\mathbf{x} \leq \bar{A} \tag{2.8}$$

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. A brief description of both will be presented here for informative purposes and finally, the method used in [**?**] will be described.

- **Active Set Methods.** In order to explain how these methods work, a definition for an *active set* must be introduced. An optimization problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & V(x) \\ \text{subject to} \quad & f_i(x) \leq b_i,\ i = 1,\ldots,m. \end{aligned} \tag{2.9}$$

  is defined by the objective function and by the set of constraints. This set defines the *feasible region*, which is the set of all x that satisfy the constraints, and thus are considered in the search for the optimal solution. Given x in the feasible region, a constraint f is active in x if $f_i(x) = 0$ and inactive if $f_i(x) > 0$. The *active set* at x comprises all the constraints that are active at that point [**?**].

  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:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & J(x) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x} + c^T\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \geq \mathbf{b} \text{ (inequality constraint)} \end{aligned} \tag{2.10}$$

  G is symmetric and positive semidefinite, A is a *mxn* and b are defined by

$$A = [a_i]_{i\in\Gamma} \quad b = [b_i]_{i\in\Gamma} \quad \Gamma = 1,2,\ldots,m$$

  Based on the KKT conditions one can say that if a given x^* is a solution of 2.10, there is a Lagrange multiplier vector λ^* such that the following conditions are satisfied for $(x,\lambda) = (x^*,\lambda^*)$.

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

If a new variable vector $y = Ax - b$ is introduced in the system, the conditions can be rewritten as follows.

$$\begin{aligned} Gx - A^T\lambda + d &= 0, \\ Ax - y - b &= 0, \\ y_i\lambda_i &= 0, \quad i = 1,2,\ldots,m, \\ (y,\lambda) &\geq 0. \end{aligned} \tag{2.14}$$

Which are correspondent with the KKT conditions for linear programming problems [**?**]. 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

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.

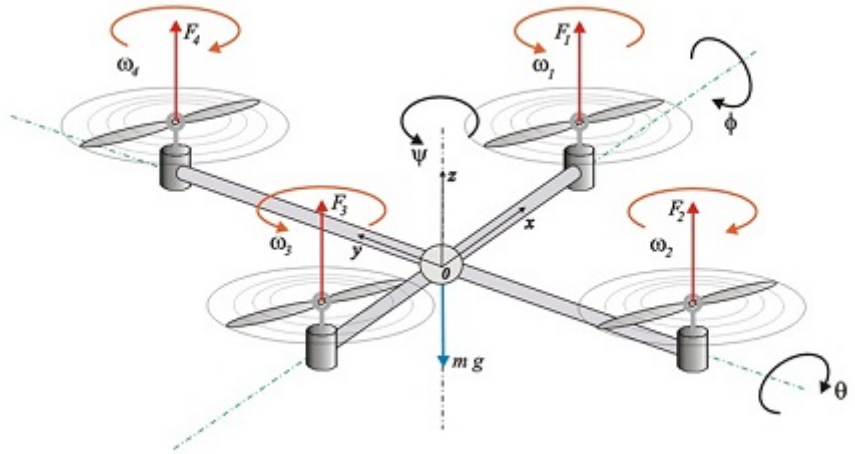

Figure 3.1: Functioning scheme for the quadrotor dynamics. (Taken from http://www.pupin.rs/RnDProfile/research-topic28.html)

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 [**?**], 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 [**?**], 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 [**?**] 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, it is mentioned that 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 earth 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 $\mathbf{I} \in \mathbb{R}^{3\times3}$ 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 [**?**] 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 **??**. 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 [**?**]. 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 [**?**] 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 [**?**], 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 as a 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.

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 [**?**].

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 a more realistic simulation.

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 X 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).

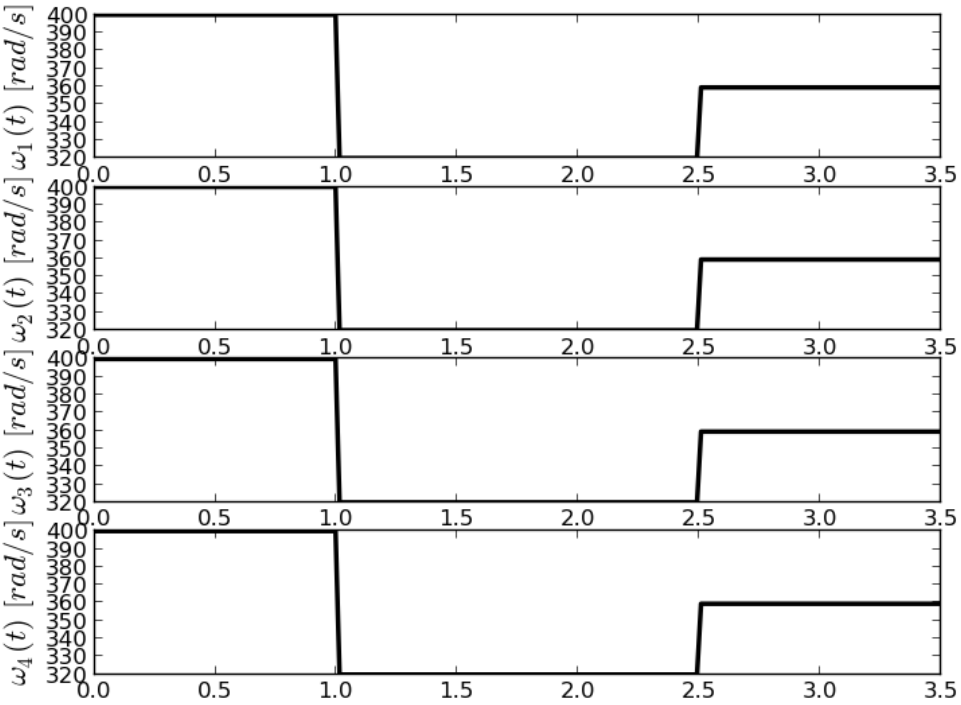

Figure 3.2: Inputs to generate an upward motion of the quadrotor.

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.

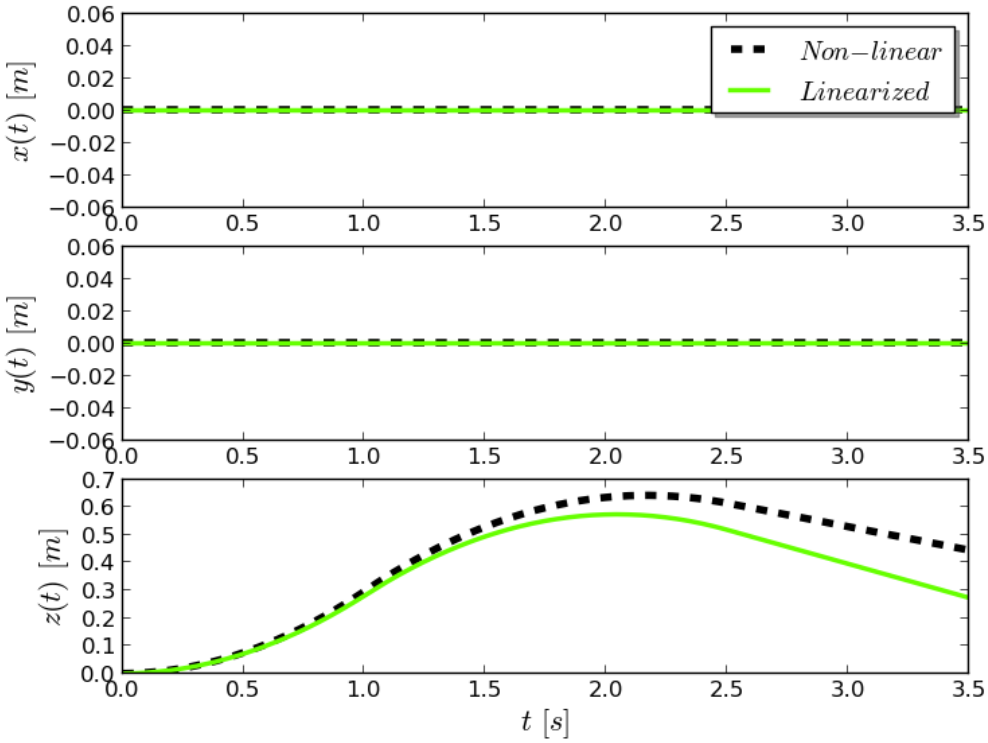

Figure 3.3: Resulting positions of the simulated systems for the given inputs.

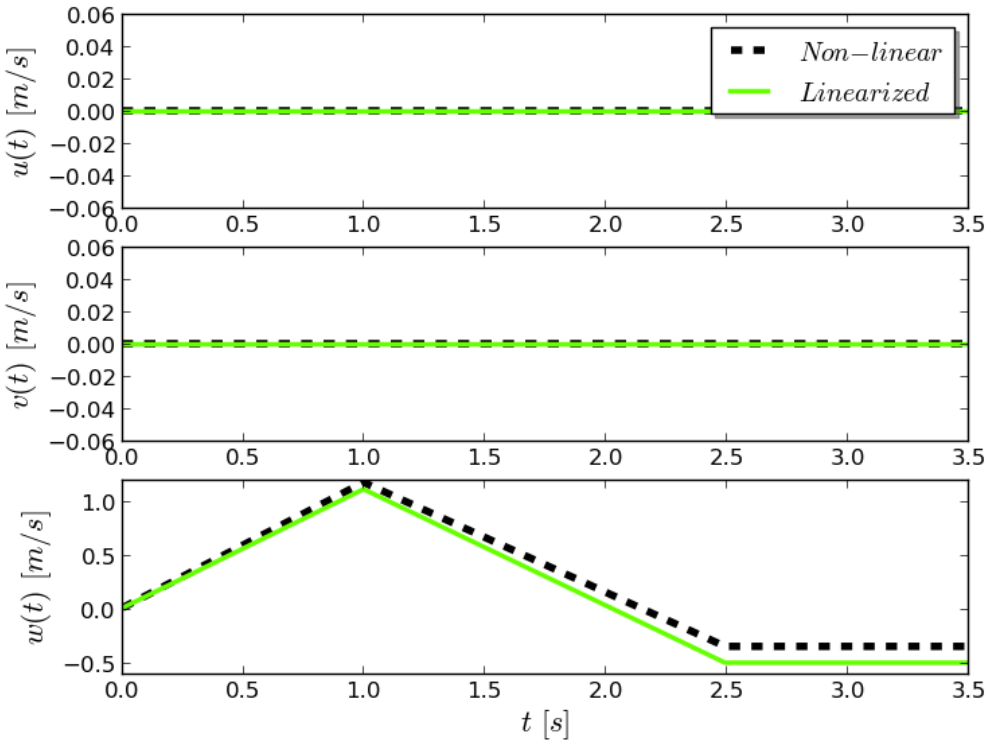

Figure 3.4: Resulting velocities of the simulated systems for the given inputs.

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 **??**, 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.

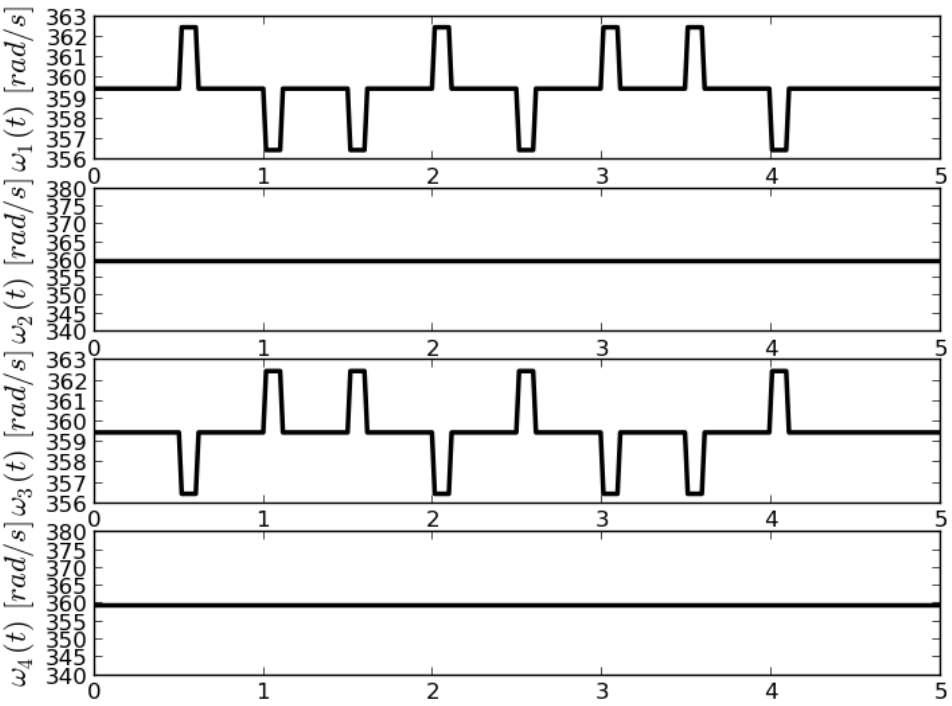

Figure 3.5: Rotor speed inputs to generate a lateral movement along the X axis on the quadrotor.

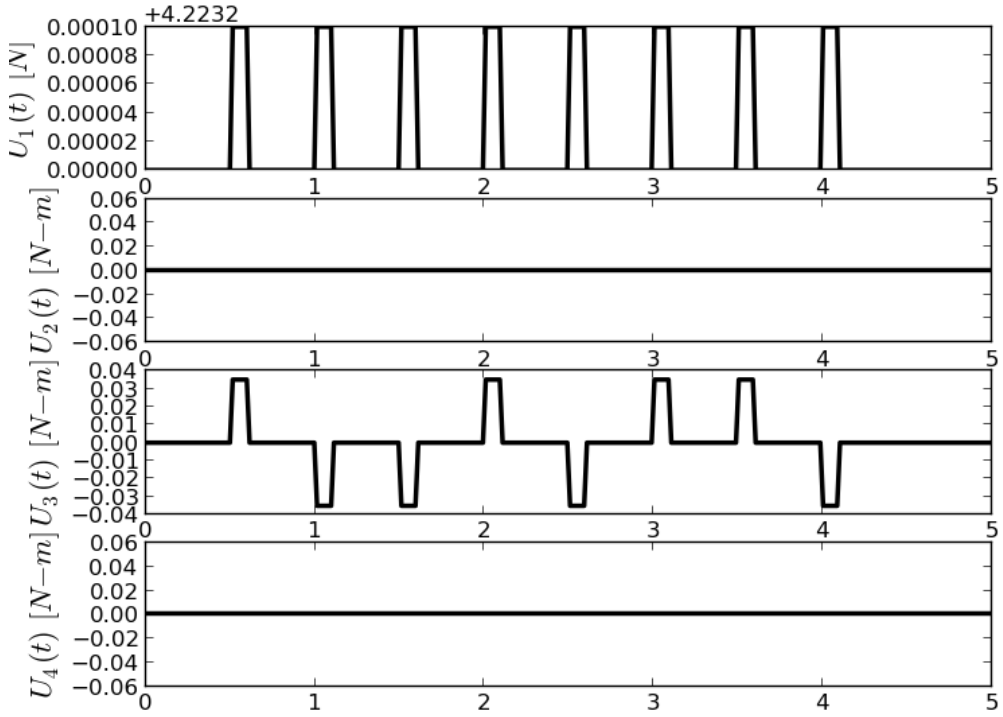

Figure 3.6: Rotor speed inputs from Figure 3.5 mapped into forces and torques acting in the quadrotor frame.

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. It is important to see that 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.

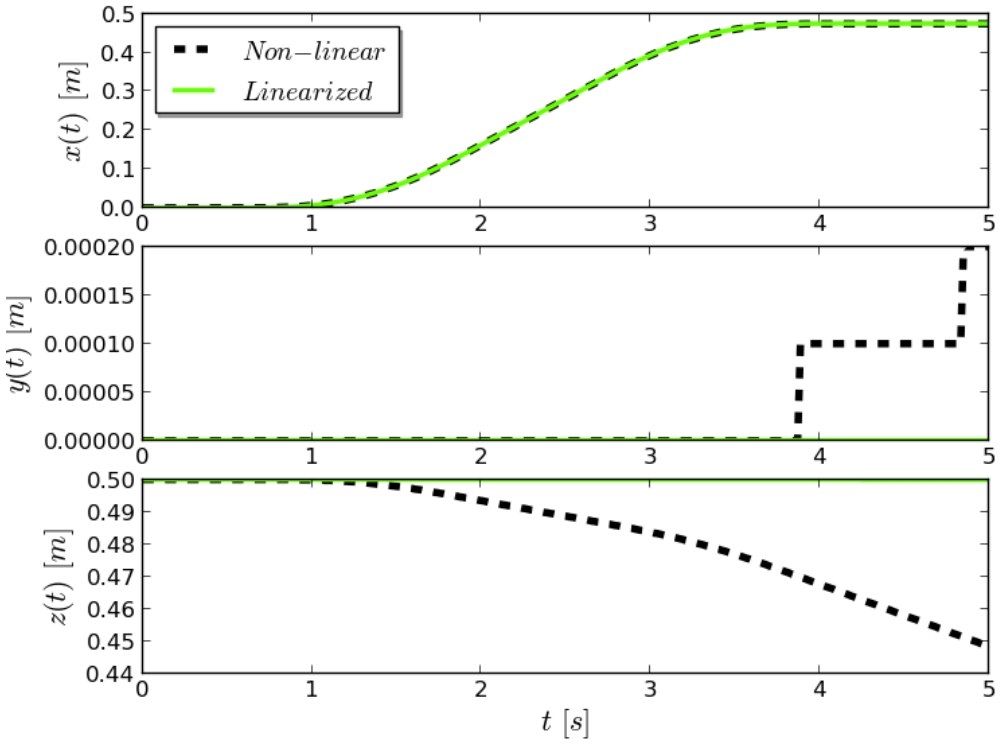

Figure 3.7: Resulting positions of the simulated systems for the inputs shown in Figure 3.5.

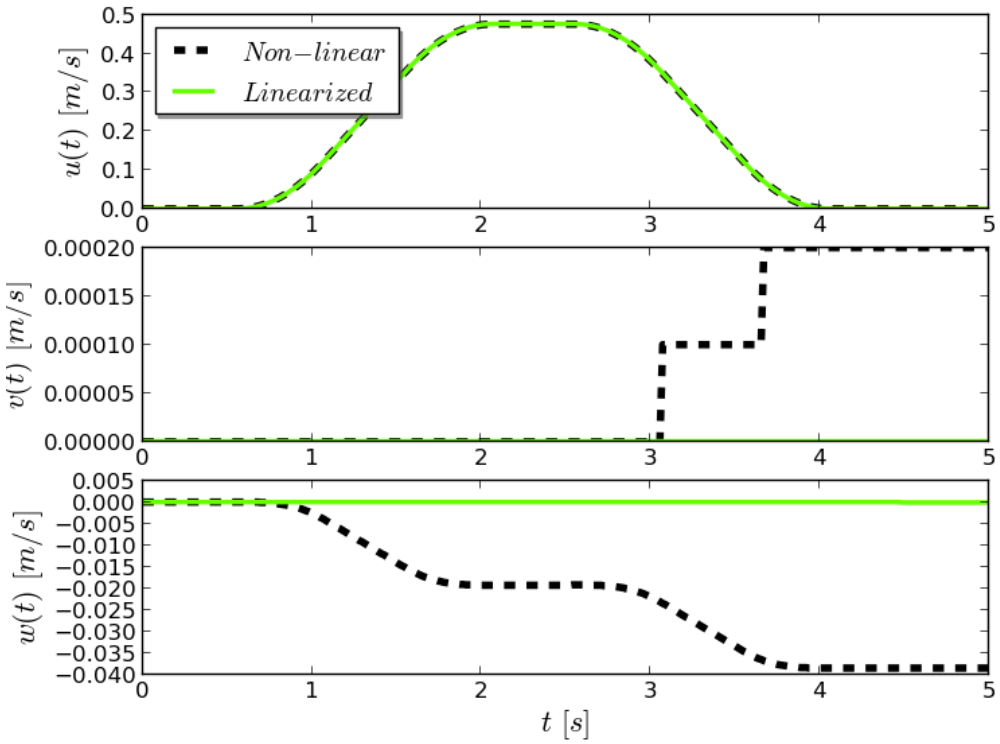

Figure 3.8: Resulting velocities of the simulated systems for the inputs shown in Figure 3.5.

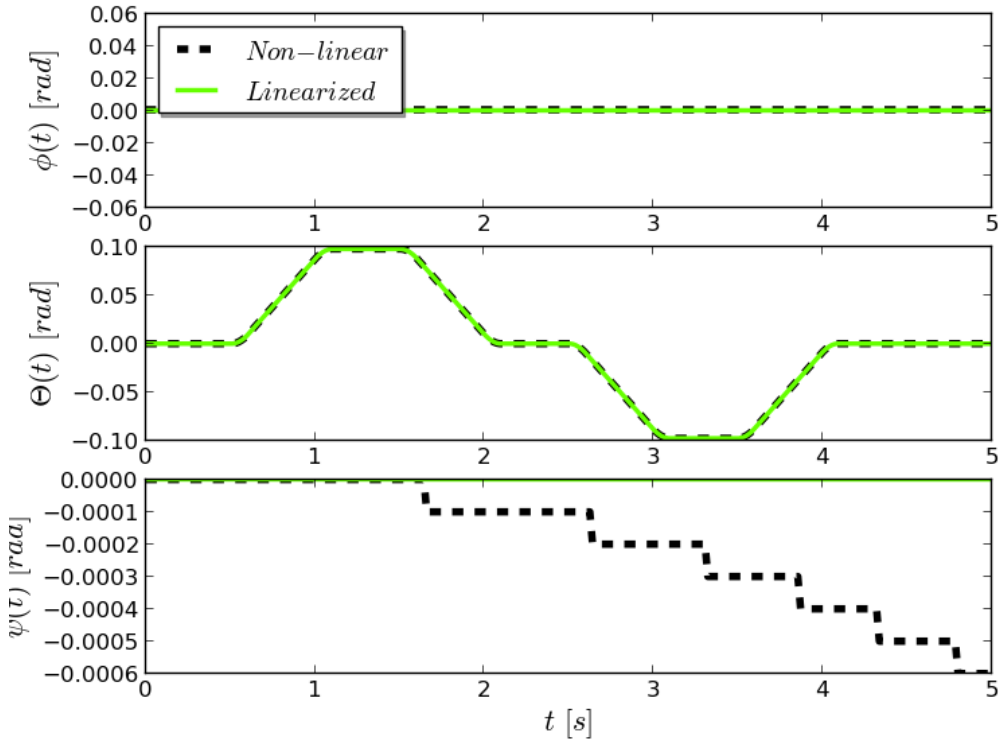

Figure 3.9: Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.5.

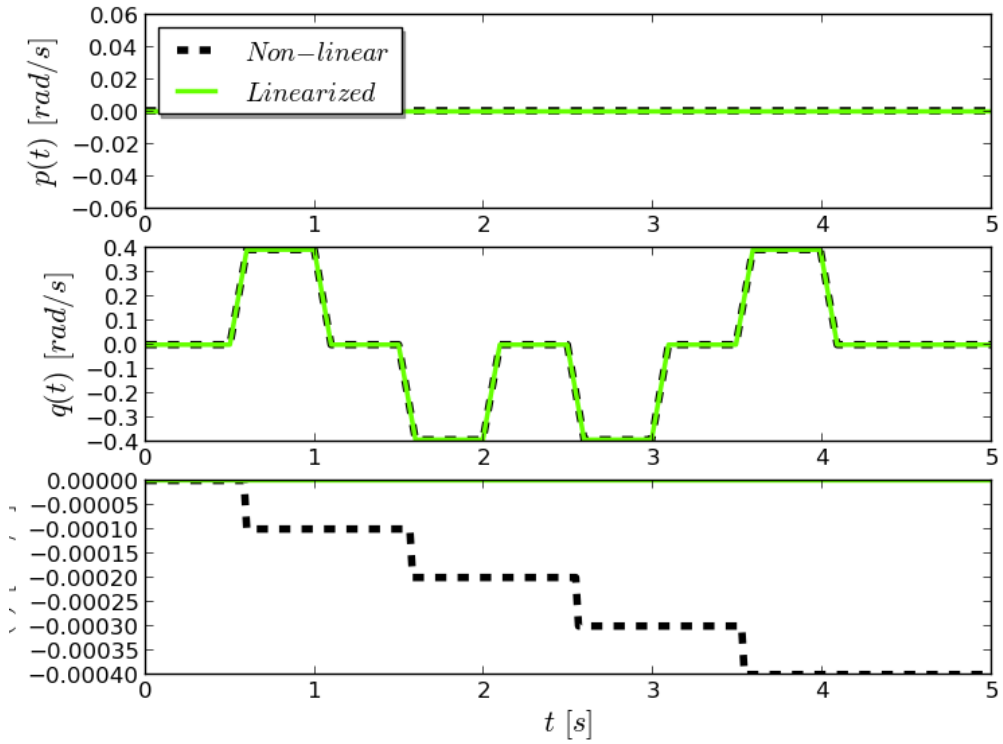

Figure 3.10: Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.5.

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.

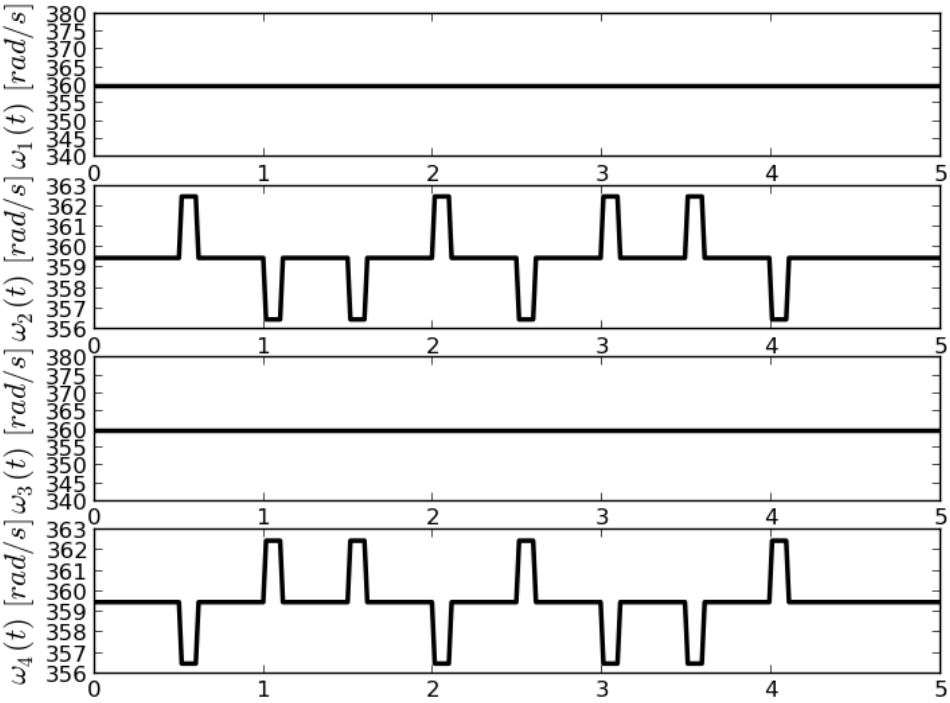

Figure 3.11: Rotor speed inputs to generate a lateral movement along the Y axis on the quadrotor.

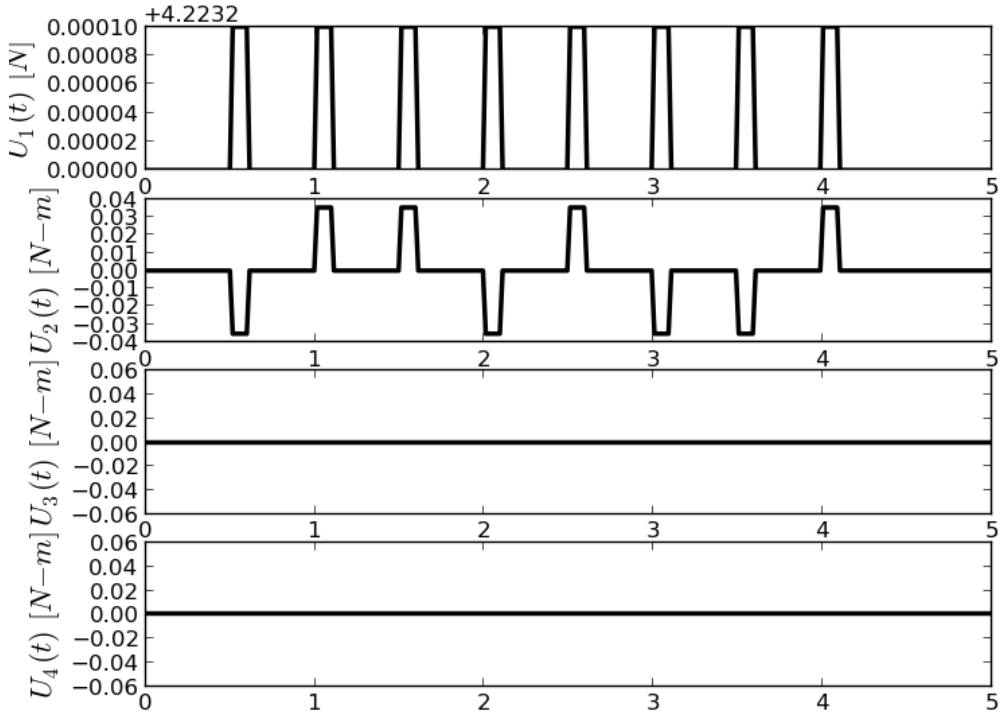

Figure 3.12: Rotor speed inputs from Figure 3.11 mapped into forces and torques acting in the quadrotor frame.

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.

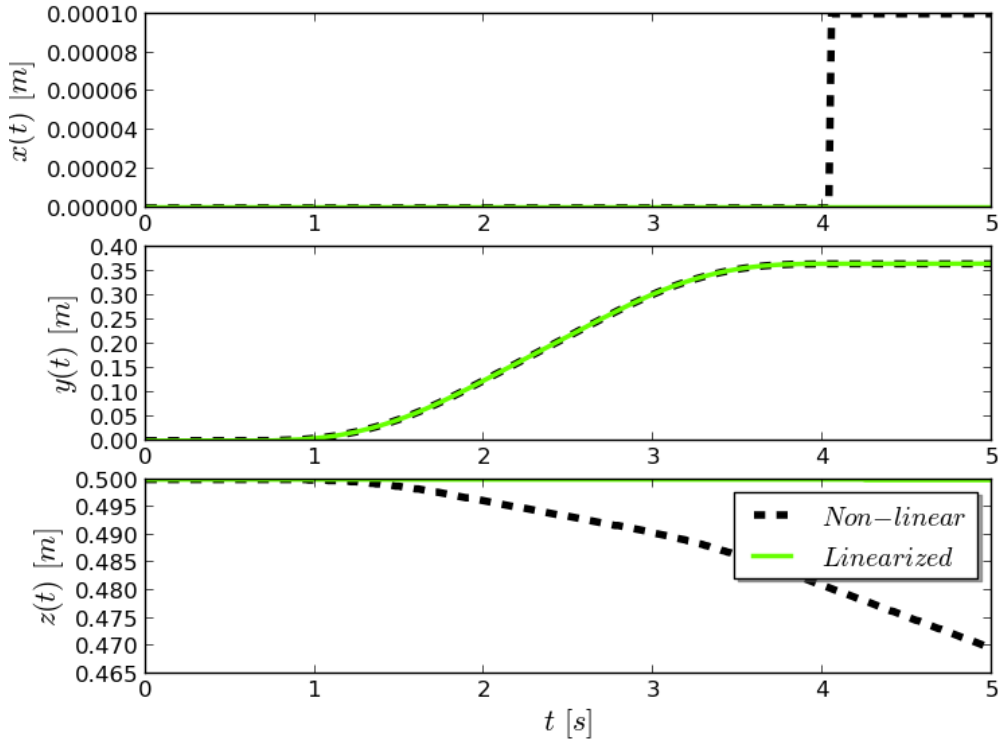

Figure 3.13: Resulting positions of the simulated systems for the inputs shown in Figure 3.11.

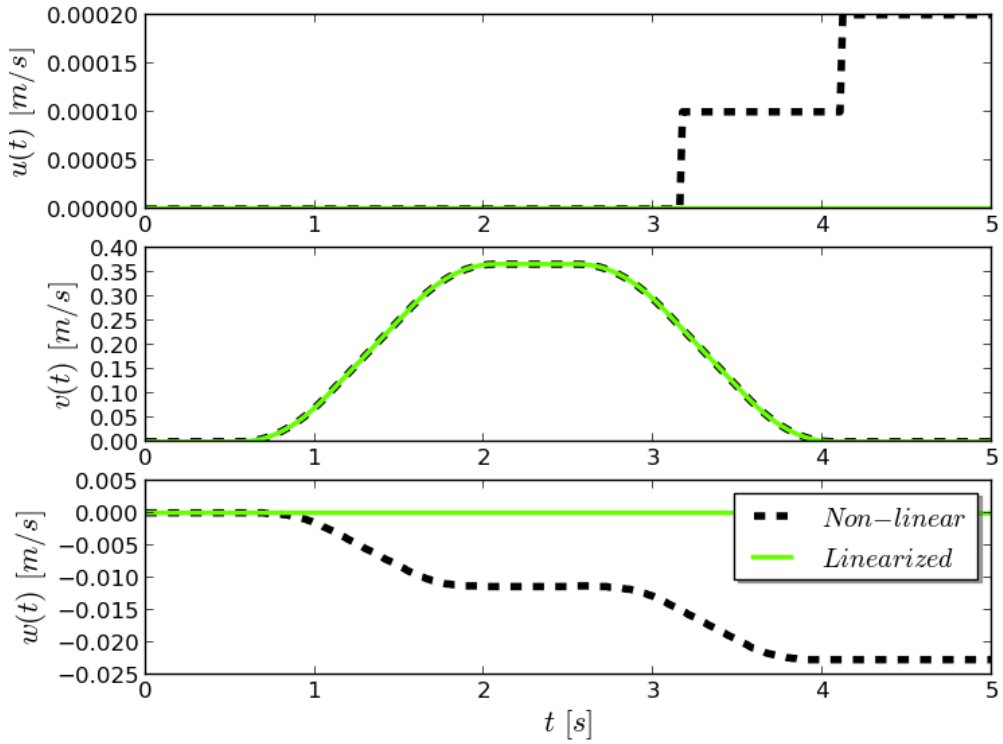

Figure 3.14: Resulting velocities of the simulated systems for the inputs shown in Figure 3.11.

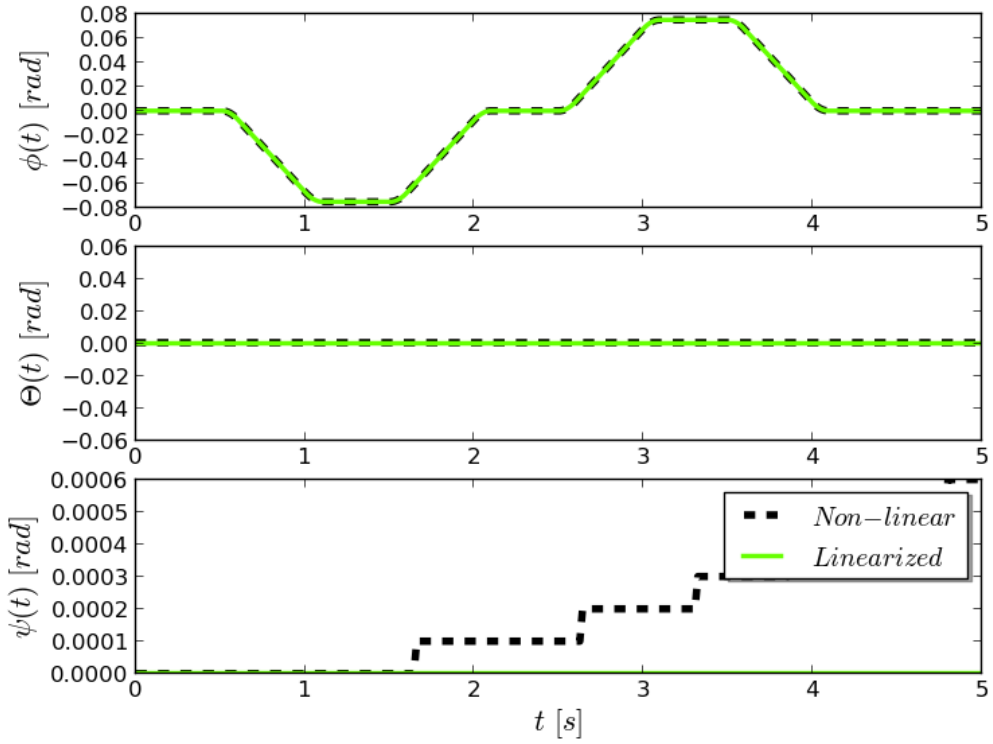

Figure 3.15: Resulting Euler angles of the simulated systems for the inputs shown in Figure 3.11.

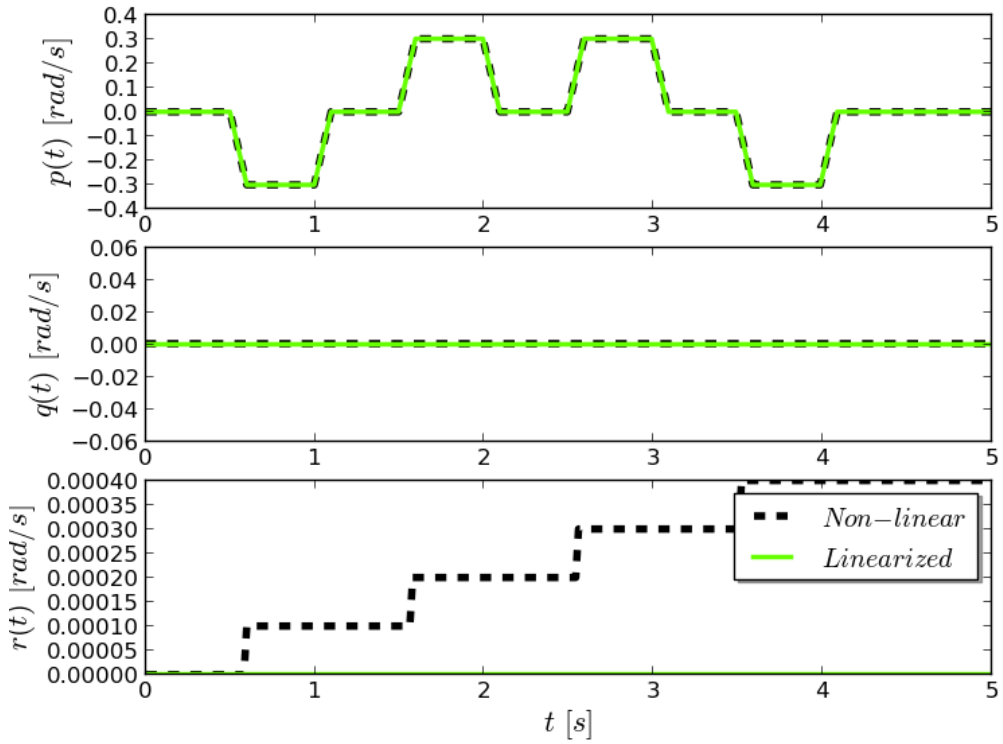

Figure 3.16: Resulting angular velocities of the simulated systems for the inputs shown in Figure 3.11.

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) [**?**].

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

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.

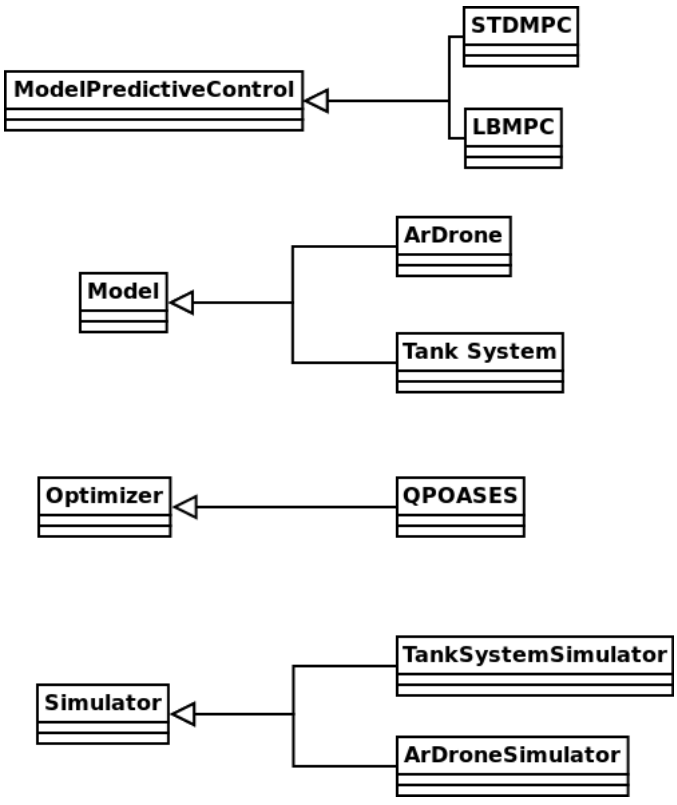

Figure 4.1: Inheritance relationship between the interface and the implementation classes.

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

- **Expandability.** This software is thinked of so 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

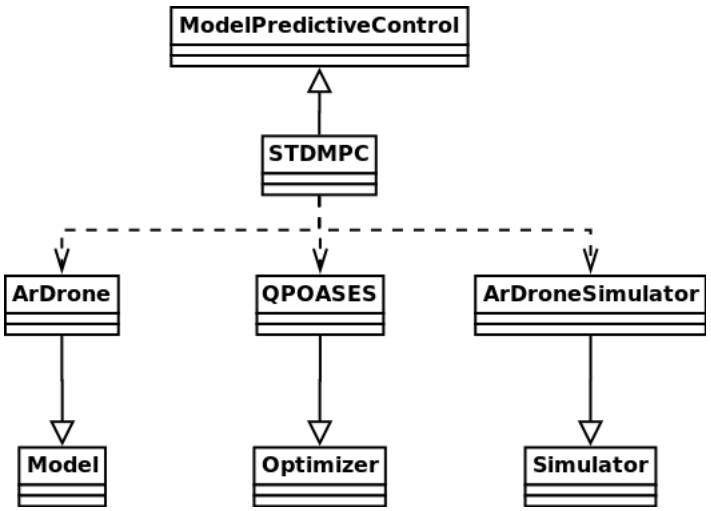

Figure 4.2: Implementation example of the class hierarchy for the ArDrone case.

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 2.2 and 2.3.

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 a 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 thinked 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.

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 in that particular class.

- resetMPC function: this function takes a pointer to an object of the Model, Optimizer and Simulator classes and sets the into the ModelPredictiveControl object, therefore providing the class with all the information and methods from these. 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 longs as they are provided.
- initMPC function: in this function all the parameters required to define the problem are read and all the initial calculations are performed. The parameters are read from ROS's parameter server, which can be modified without compiling afterwards using configuration files written in YAML format [**?**]. This allows quick modifications without the time consuming compilation process.

- updateMPC function: in 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.

4.4 The Parameter Server

5 Discussion and Conclusions

5.1 Discussion

5.2 Conclusions

6 Recommendations and Future Work

6.1 Recommendations

6.2 Future Work