Contents

List of Figures

Chapter 1

Introduction

Unmanned Aerial Vehicles (UAV) are today widely used in ground observation, and by equipping them with different sensors they can be used in different situations. While the use of UAVs eases many cases of ground observation, there are some difficulties related to the attitude of the aircraft. When the sensor is attached directly to the aircraft the sensor will be coupled with the UAVs states, so what is captured by the camera depends on the angles of the UAV. Figure 1.1 illustrates how what is captured by the camera changes with the roll angle ϕ.

A common solution to decouple the sensor from the UAV states is to attach the sensor to a gimbal which will counteract the movements of the UAV. While this is a good solution for decoupling, it raises some new issues regarding its weight and size. As one of the benefits of UAVs is their small size, the gimbal can quickly be too big and heavy for the UAV, and it may make the aerodynamics of the aircraft less effective. This usually leads to increased fuel consumption.

This paper will investigate methods that will ensure precise ground observation when a camera attached directly to the aircraft is used to observe both curved and piecewise linear paths on ground level, while also avoiding the extra costs associated with a gimbal. This will be accomplished by finding an optimal path that minimizes the deviance between what is to be observed and what is captured by the camera. The optimal path will be calculated by an offline intervalwise Nonlinear Model Predictive Control (NMPC) algorithm before the flight commences [1]. The control method is developed with the usage of a hyperspectral pushbroom camera that is fixed to the UAV in mind.

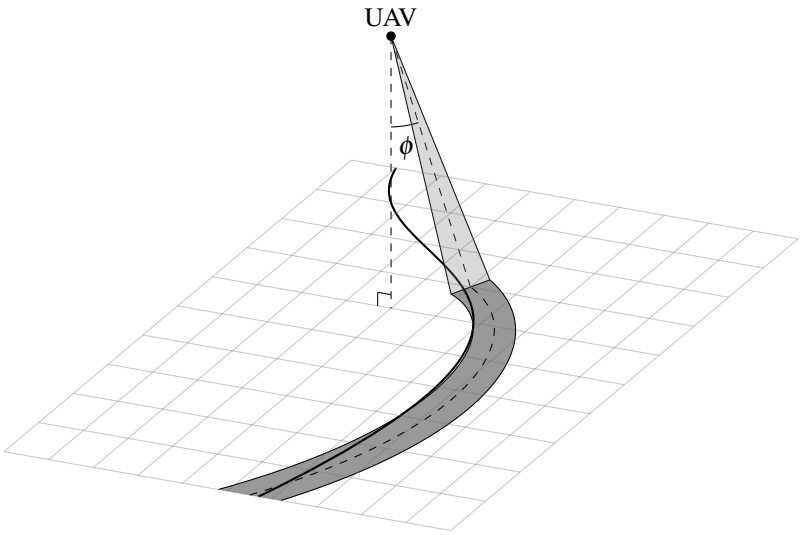


Figure 1.1: An illustration of the issues related to UAV operation with fixed sensors.

1.1 Related Work

The most common method to decouple the UAV attitude states from the sensor today is to equip the aircraft with a gimbal, which results in easy UAV operation without losing track of the features that is to be observed. Since the gimbal angles have limited range, features can be lost from the camera field of view (FOV) for some combinations of aircraft positions and attitudes. Previous solutions to this problem include generating trajectories that ensures that the gimbal angles are able to cover the features of interest [2], or putting constraints on the roll angle and altitude of the UAV [3].

A simpler solution to avoid lateral movements of the FOV is to change the UAV course by using the rudder instead of the ailerons. The rudder deflection creates a yawing moment which causes the aircraft to change course [4]. This type of controller is referred to both as a Rudder Augmented Trajectory Correction (RATC) controller [4] and a skid-to-turn (STT) controller [5]. Results show that the performance of these controllers are comparable to conventional controllers using roll to change course, and that errors in the images is greatly reduced [4] [5] [6].

While the controllers offer a solution to the control problem that reduces the errors in the images, they do not ensure that the features of interest stay inside the sensors FOV. To ensure that they stay within the FOV, Jackson has developed an optimization algorithm that minimizes the error of the sensor footpring [7]. Jakcson's solution differs from the solution proposed in this paper as his solution uses a simplified model of the UAV to calculate the optimal path online, while this paper focuses on using a more precise model to find an optimal solution before the flight is initiated.

Jackson presents a path planner that aims to minimize the error between the target on the ground, and the footprint of a camera fixed to a UAV by using a Nonlinear Model Predictive Controller (NMPC). The NMPC is compared to a PID and a sliding-mode controller that seek to follow the same path. Simulations of the three controllers show that of the three, the PID controller had the biggest crosstrack error. The results of the simulation of the NMPC and the sliding-mode controller showed that the two had comparable performance. The NMPC controller was able to find a near optimal solution with the performance characteristics of a real-time application.

One important point made by Jackson is that perfect tracking of a ground path with a fixed camera is not possible when using the roll angle to change the course of the aircraft. A controller that attempts to solve this problem would be unstable beacause of the depence the camera position has on the roll angle. When the path turns right and the aircraft rolls right to follow this path, the camera position will move left, away from the path. This only applies to controllers that attempt a perfect tracking of the path, so that a near-perfect tracking is still achievable.

1.2 Hyperspectral Imaging

The control method developed in this paper will be developed with the use of a fixed hyperspectral, pushbroom sensor in mind. A hyperspectral sensor/camera allows for accurate detection of different types of material from the UAV by sensing the wavelength of the received light.

1.2.1 Description

Hyperspectral imaging uses basics from spectroscopy to create images, which means that the basis for the images is the emitted or reflected light from materials [8]. The amount of light that is reflected by a material at different wavelengths is determined by several factors, and this makes it possible to distinguish different materials from each other. The reflected light is passed through a grate or a prism that splits the light into different wavelength bands, so that it can be measured by a spectrometer.

When using a hyperspectral camera for ground observation from a UAV, it is very likely that one pixel of the camera covers more than one type of material on the ground. This means that the observed wavelengths will be influenced by more than one type of material. This is called a composite or mixed spectrum [8], and the spectra of the different materials are combined additively. The combined spectra can be split into the different spectra that it is build up of by using noise removal and other statistical methods which will not be covered here.

1.2.2 UAV Ground Observation

Hyperspectral imaging is already being used for ground observation from UAVs. Its ability to distinguish materials based on spectral properties means that it can be used to retrieve information that normal cameras are not able to. For example in agriculture it can be used to map damage to trees caused by bark beetles [9], or it can be used to measure environmental properties, for example chlorophyl fluorescense, on leaf-level in a citrus orchard [10].

Systems for ground observation with hyperspectral cameras can be very complex, which often leads to heavy systems. In [11], a lightweight hyperspectral mapping system was created for the use with octocopters. The purpose of the system is to map agricultural areas using a spectrometer and a photogrammetric camera, and the final takeoff weight of the system is 2.0 kg. The resolution of the final images made it possible to gather information on a single-plant basis, and the georeferencing accuracy was off by only a few pixels.

1.3 Contributions

1.4 Thesis Outline

Chapter 2

Kinematics

What is captured by the camera, the camera footprint, is dependent on the attitude angles of the aircraft when the camera is fixed to the aircraft body. In this section a model for calculating the camera footprint on the ground assuming flat earth will be presented, as well as the necessary UAV states for this thesis.

2.1 Reference Frames

Three different reference frames will be used to describe the kinematics of the UAV: the *body frame*, *North East Down* (NED) frame and *Earth Centered Earth Fixed* (ECEF) frame. The transformations given here can be found in Fossen [12].

The body frame, denoted $\{b\}$, is attached to the UAV and is used to describe the attitude and velocity of the aircraft. The NED frame, denoted $\{n\}$, is used to locally describe the position of the UAV using Cartesian coordinates. The position of the camera footprint will be given in the NED frame, based on the attitudes in the body frame.

While the body and NED frame are local frames that are useful to describe the UAVs attitude, speed and position, a different frame is needed to express the location of the UAV in a global perspective. For this the ECEF frame, denoted $\{e\}$, is used. In the ECEF frame position is often represented using Cartesian coordinates with the origin at the earth center, but in this case the position will be represented by longitude, latitude and height.

2.1.1 Transformations

Between NED and Body

The body frame and the NED frame are related by rotation matrices, one for each of the attitude angles. The transformation to the NED frame from the body frame is given by the rotation matrix $\mathbf{R}_b^n$:

$$\mathbf{R}_b^n(\mathbf{\Theta}_{nb}) = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi c_\phi s_\theta \\ s_\psi c_\theta & c_\psi c_\phi + s_\phi s_\theta s_\psi & -c_\psi s_\phi + s_\theta s_\psi c_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{2.1}$$

where c and s are the cosine and sine trigonometric functions of the angle in subscript, respectively. The transformation from the NED frame to the body frame can be found by taking the inverse of the transformation matrix $\mathbf{R}_b^n(\mathbf{\Theta}_{nb})$.

Between NED and ECEF

The transformation between ECEF and NED frames when the position is given using longitude, latitude and height is also given by a rotation matrix $\mathbf{R}_n^e(\mathbf{\Theta}_{en})$. However, it is the velocity vectors in each frame that are related by the rotation matrix. The rotation matrix is composed by two rotations about the latitude l and longitude μ:

$$\mathbf{R}_n^e(\mathbf{\Theta}_{en}) = \begin{bmatrix} -c(l)s(\mu) & -s(l) & -c(l)c(\mu) \\ -s(l)s(\mu) & c(l) & -s(l)c(\mu) \\ c(\mu) & 0 & -s(\mu) \end{bmatrix}. \tag{2.2}$$

The transformation between the velocity vectors in the ECEF and NED frame can be written as:

$$\dot{\mathbf{p}}_{b/e}^e = \mathbf{R}_n^e(\mathbf{\Theta}_{en})\dot{\mathbf{p}}_{b/e}^n. \tag{2.3}$$

Since this transformation represent velocities, a reference position must be known when transforming positions. Since NED is a local frame, the position in NED $\mathbf{p}^n$ will be given as a displacement from the reference ECEF position $\mathbf{p}_0^e$. The relation between position in NED and ECEF can therefore be written as [13]:

$$\mathbf{p}^e - \mathbf{p}_0^e = \mathbf{R}_n^e(\mathbf{\Theta}_{en})\mathbf{p}^n. \tag{2.4}$$

2.2 UAV Model

In this thesis the linearized UAV model presented by Beard & McLain [14] will be used as the prediction model for the path planner that is presented in chapter 3. In this section the UAV states used in this model will be presented, as well as the linearization method.

2.2.1 UAV States

The position of the UAV will be given using the North East Down (NED) coordinate frame denoted $\{n\}$:

$$\mathbf{p}_{b/n}^{n} = \begin{bmatrix} p_N \\ p_E \\ p_D \end{bmatrix} = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix}, \tag{2.5}$$

while the global location will be given using the Earth Center Earth Fixed (ECEF) frame, denoted $\{e\}$, represented by longitude and latitude:

$$\mathbf{\Theta}_{en} = \begin{bmatrix} l \\ \mu \end{bmatrix}. \tag{2.6}$$

Following the notation used in [14], the velocities of the UAV will be given in the body frame denoted $\{b\}$:

$$\mathbf{V}_{g}^{b} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{2.7}$$

The attitude $\mathbf{\Theta}_{nb}$ of the UAV will be given as Euler-angles, with the corresponding angular velocities $\dot{\mathbf{\Theta}}_{nb}$:

$$\mathbf{\Theta}_{nb} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \quad \dot{\mathbf{\Theta}}_{nb} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}. \tag{2.8}$$

Euler angles are used over quaternions in this paper because the optimization is to be run offline so that computation time is not a critical measure. Even though Euler angles do suffer from gimbal lock while quaternions don't [14], the UAV will not be performing any high-angle maneuvers so that a gimbal lock should never occur.

The UAV model has four control inputs, namely the elevator, aileron, rudder, and throttle:

$$\mathbf{u} = \begin{bmatrix} \delta_e & \delta_a & \delta_r & \delta_t \end{bmatrix}^\mathsf{T}. \tag{2.9}$$

The complete nonlinear state space equations for the UAV model is given in appendix A.

2.2.2 Linearizing the Model

In order to ease computational load the *linear decoupled model* of a UAV, presented by Beard & McLain [14], will be used in this thesis. While a linear model is not able to fully describe the motions of an aircraft, it is valid around the *trimmed state* of the aircraft. An aircraft in its trimmed state will be able to maintain a straight level flight without any change in the control input, and since the UAV in this thesis is not expected to perform any high-angle maneuvers that puts it far away from the trimmed state, the linear model will be valid. An aircraft in the trimmed state satisfies the following equation:

$$\dot{\mathbf{x}} = f(\mathbf{x}^*, \mathbf{u}^*) = 0. \tag{2.10}$$

Linearization is performed by adding perturbations to the trimmed state solution, and the linearized states $\bar{\mathbf{x}}$ represent the perturbations away from the trimmed state [15]. The linearized state is defined as $\bar{\mathbf{x}} = \mathbf{x} - \mathbf{x}^*$, where $\mathbf{x}^*$ is the trimmed state.

The states of an aircraft are highly *coupled*, meaning that they affect each other. This greatly increases the complexity of finding an optimal solution of the model since a change in one variable has effect on more than one state. For this reason the model is also decoupled into lateral and longitudinal models, where the states in one of the models does not affect the states in the other model. This simplification is done by removing terms that has a very small effect on the state, as these effects are easily controlled by the control systems [14]. The lateral and longitudinal states are given as:

$$\begin{aligned} \dot{\mathbf{x}}_{lat} &= \begin{bmatrix} v & p & r & \phi & \psi \end{bmatrix}^\mathsf{T}, \mathbf{u}_{lat} = \begin{bmatrix} \delta_a & \delta_r \end{bmatrix}^\mathsf{T} \\ \dot{\mathbf{x}}_{lon} &= \begin{bmatrix} u & w & q & \theta & h \end{bmatrix}^\mathsf{T}, \mathbf{u}_{lon} = \begin{bmatrix} \delta_e & \delta_t \end{bmatrix}^\mathsf{T}. \end{aligned} \tag{2.11}$$

2.3 Camera Footprint

The camera footprint is coupled with the three attitude angles given in $\boldsymbol{\Theta}_{nb}$. The position of the camera footprint will be calculated using forward kinematics, and an illustration of how the roll ϕ and pitch θ affects the camera position is shown in Figure 2.1.

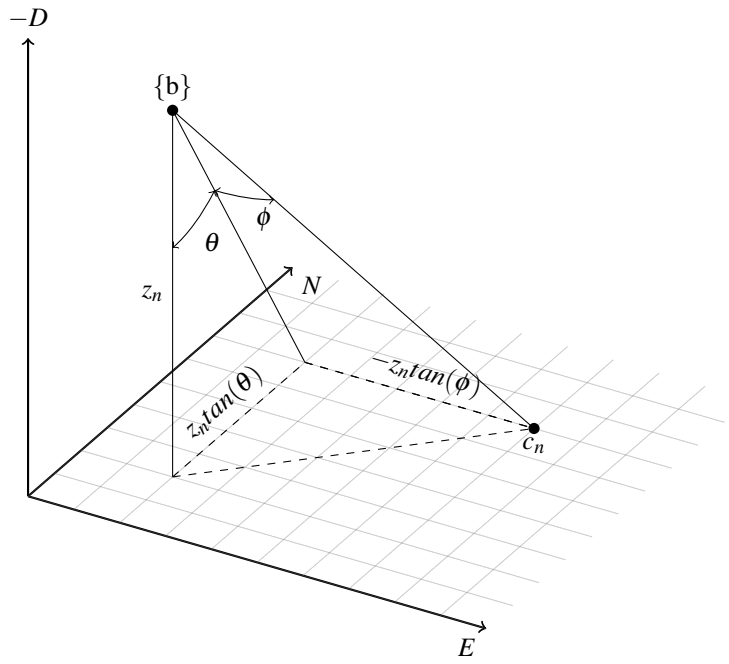


Figure 2.1: Illustration of how the aircraft attitude influence the camera position.

2.3.1 Centre Position

The attitude of the UAV is given in the body frame $\{b\}$ and the height z_n is given in the NED frame $\{n\}$, and the model presented here assumes flat earth. The position of the footprint centre point $\mathbf{c}_b^b$ in the body frame $\{b\}$ can be expressed as the distance from the center of the body frame, the UAV, caused by the angles ϕ and θ:

$$\mathbf{c}_b^b = \begin{bmatrix} c_{x/b}^b \\ c_{y/b}^b \end{bmatrix} = \begin{bmatrix} z_n tan(\theta) \\ -z_n tan(\phi) \end{bmatrix}. \tag{2.12}$$

The coordinates of the camera position in $\{n\}$ can be found by rotating the point $\mathbf{c}_b^b$ with respect to the aircraft heading ψ, and by translating the rotated point to the aircrafts position in the $\{n\}$ frame. The rotation matrix for rotating with respect to the heading is given as:

$$\mathbf{R}_{z,\psi} = \begin{bmatrix} cos(\psi) & -sin(\psi) \\ sin(\psi) & cos(\psi) \end{bmatrix}. \tag{2.13}$$

The final expression for the camera footprint centre position $\mathbf{c}^n$ in the $\{n\}$ frame then becomes:

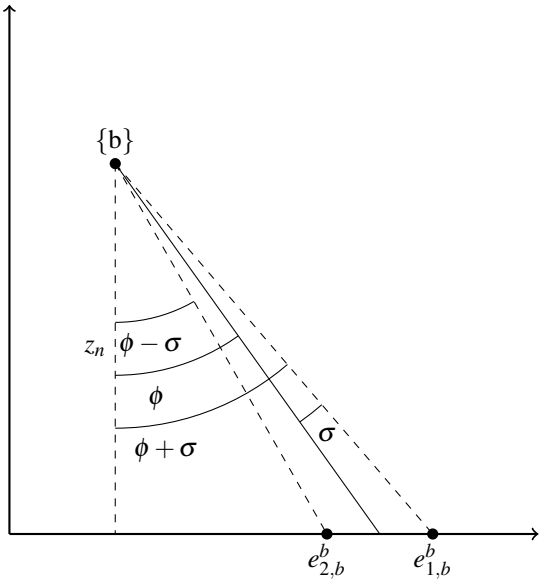


Figure 2.2: Illustration of how the field of view for a pushbroom sensor is calculated.

$$\begin{aligned} \mathbf{c}^n &= \mathbf{p}^n + \mathbf{R}_{z,\psi}\mathbf{c}^b_b \\ &= \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \mathbf{R}_{z,\psi} \begin{bmatrix} x^b_{x/b} \\ c^b_{y/b} \end{bmatrix} \end{aligned} \tag{2.14}$$

2.3.2 Edge Points

Since a hyperspectral pushbroom sensor captures images in a line, the centre point of the camera footprint does not express the entire area that is captured by the sensor. The edge points of the camera footprint are calculated with respect to the sensor's field of view, as shown in figure 2.2. These points, **e**, can be found by altering (2.12):

$$\mathbf{e}^b_{1,b} = \begin{bmatrix} z_n tan(\theta) \\ -z_n tan(\phi + \sigma) \end{bmatrix}, \quad \mathbf{e}^b_{2,b} = \begin{bmatrix} z_n tan(\theta) \\ -z_n tan(\phi - \sigma) \end{bmatrix}. \tag{2.15}$$

The final expression for the cameras edge points then becomes:

$$\mathbf{e}^n_i = \mathbf{p} + \mathbf{R}_{z,\psi}\mathbf{e}^b_{i,b}. \tag{2.16}$$

Chapter 3

Model Predictive Control

Model Predictive Control (MPC) is a term used to describe control methods that uses knowledge about the process to calculate the future control inputs to the system in order to follow a reference trajectory [16]. In this chapter the equations for an *offline intervalwise MPC* that seeks to minimize the distance between the camera centre point and the ground path that is to be observed will be given. A linear state space-model for the UAV will be used to predict the future states and control inputs.

3.1 MPC Method

The MPC strategy can be broken down into three tasks [16]:

1. Predict the future outputs of the process for the given prediction horizon using past inputs to the process and the past measured states of the process, and by using the future control signals.
2. Optimize an objective function in order to determine the future control signals that follows a given reference trajectory as closely as possible.
3. Apply the optimal control signals to the process, and measure the resulting output so that it may be used to calculate the next prediction horizon in the first task.

In short MPC problems are made up of three elements: Prediction model, objective function and constraints. The prediction model represents the model of the process that is to be controlled, and will in this case consist of the differential equations for the states of the UAV. The objective function is the function that is to be minimized by the optimization algorithm, in this case this will be the distance from the camera centre point to the desired ground path together with some of the UAV states that will give a stable flight. The objective function represents the difference between the reference

trajectory that the UAV is to follow and the current states of the UAV. The constraints are used to limit the values that either states or control inputs can take, and can prevent solutions that are not physically feasible.

A common mathematical formulation of the three elements that make up the optimization problem is shown in (3.1) [17]. $f(x)$ represents the objective function that is subject to equality and inequality constraints respectively. In this thesis the differential equations describing the UAV model will be implemented as equality constraints, while the inequality constraints will be used to define the ranges the control inputs must be within.

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ \text{s.t} \quad & c_i(x) = 0, i \in \varepsilon, \\ & c_i(x) \geq 0, i \in I. \end{aligned} \tag{3.1}$$

3.2 Discretization

The model and optimization problem will be written on continuous time form, which means that it has to be discretized in order to be solved. The method used to discretize the problem plays a big role in how the problem is solved, and a common method to use for nonlinear programs (NLP) is the *direct multiple-shooting* method. Direct discretization methods can be explained as "first discretize, then optimize", which allows for easier treatment of inequality constraints [18]. One of the major benefits by using the direct multiple-shooting method is that all shooting nodes are initialized with the result from the previous iteration [19].

In short, the direct multiple-shooting method starts by computing a discretized control trajectory for a finite time interval. Independently, the Ordinary Differential Equations (ODE) of the optimization problem is solved one time for every timestep of the discretized control trajectory. Simultaneously, an integral of a cost function is computed, which is the reason why the direct multiple-shooting method is also called a *simultaneous* method.

$$\begin{aligned} \min_{\mathbf{s},\mathbf{q}} \quad & \textstyle\sum_{i=0}^{N-1} F_i(\mathbf{s}_i, \mathbf{q}_i) + E(\mathbf{s}_N) \\ \text{s.t} \quad & \mathbf{x}_0 - \mathbf{s}_0 = 0 \\ & \mathbf{x}_i(t_{i+1}; \mathbf{s}_i, \mathbf{q}_i) - \mathbf{s}_{i+1} = 0, \quad i = 0, ..., N-1 \\ & h(\mathbf{s}_i, \mathbf{q}_i) \leq 0, \quad i = 0, ..., N \end{aligned} \tag{3.2}$$

The direct multiple-shooting method can be described by the NLP shown in (3.2) [19]. In the equation the objective function F is the result of integrating the cost function, and $\mathbf{s}$ and $\mathbf{q}$ are the optimization variables for the states and controls respectively. $\mathbf{s}$ and $\mathbf{q}$ are introduced in order to ensure that the solution for the time interval is tied to the initial values. E is the end term of the objective function.

3.3 Offline Intervalwise MPC

The control problem in this thesis will be solved by using an offline intervalwise MPC to generate an optimal path that will reduce the image error when using a fixed camera to survey a ground track. The generated path is intended to be tracked by the autopilot on the UAV that will perform the survey, with the intention of optimally surveying the ground path.

3.3.1 Offline MPC

An *offline MPC* means that the initial state of the MPC is not a measurement of the UAV states, but rather the result of a simulation of the UAV. This means that the result from the prediction model used in the MPC will act as the physical system, and the outputs of the model will be fed back as inputs to the MPC for every iteration. The equations of the offline MPC are the same as the ones for the online version.

Rawlings & Mayne [20] refers to this kind of problem as a *deterministic problem* since there is no uncertainty in the system. A feedback loop in this kind of system is not needed in principle, since it does not present any new information. They also state that the resulting control action from an MPC for a deterministic system is the same as the control action from a *receding horizon control law* (RHC), which is another kind of predictive control.

3.3.2 Intervalwise MPC

Since this is a deterministic system, it is possible to perform the entire path optimization over one long optimization horizon. However, the computational load of using one long optimization horizon is heavier than the load of using several, shorter optimization horizons. For this reason an *intervalwise MPC* will be used. The term intervalwise has been introduced by Kwon & Han [1] to describe a type of receding horizon controller that implements the same strategy.

Commonly an MPC is used to optimize the model over a given *horizon*, where the initial states are given. After the optimization has finished, the first timestep of the optimization is returned and applied to the system, before a measurement of the system is performed. The new measurements are given as initial states for the next horizon, and so on.

The principle is the same for an intervalwise MPC. However, instead of only returning the first timestep, an *interval* of timesteps are returned, and the last timestep of the interval is used as intial states for the next optimization horizon. This way the number of MPC iterations is reduced, and the increased complexity by having long optimization horizons is avoided. Figure 3.1 shows how timesteps, intervals and horizons relate to each other. Since the MPC developed here is an offline MPC, the timesteps of each interval is stored as the result.

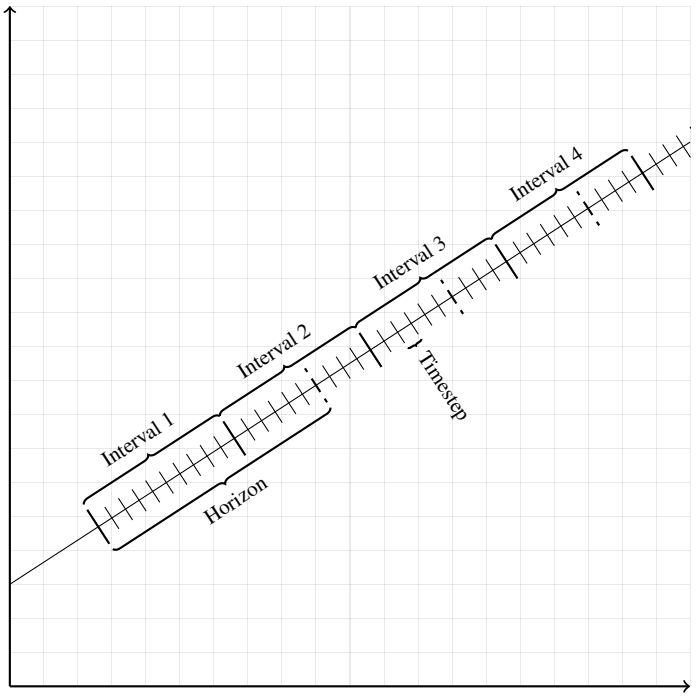


Figure 3.1: How intervals and horizons relate in an Intervalwise MPC.

3.4 Objective Function

The main objective of the MPC developed in this thesis is to minimize the cross track error between the centre point of the camera footprint and the ground path that is to be observed. This, together with other objectives, will be defined in the objective function of the optimization problem. In this section a way of formulating the objective function, least-squares, will be described, and how the objective function for an MPC is formulated to express the optimization horizons.

3.4.1 Least-Squares Problem

In many applications the objective function is formulated as a least-square (LSQ) problem. LSQ is a form of regression where the distance between a measurement and a known model is computed. In this case the known model is the reference signals, and the distance between the current states and the reference signal is calculated as a LSQ problem. The general mathematical formulation for LSQ is [17]:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) = \frac{1}{2}\sum_{j=1}^{m} |\phi(x,t_j) - y_j|. \tag{3.3}$$

In (3.3) r_j is called the residual function, which represents the distance between the measurement y_j taken at time t_j, and the model ϕ. In the optimization problem the residual function is what the algorithm seeks to minimize by selecting the parameters x that gives the lowest possible value of the residual function r_j.

In order to have a reference model that the measurements can be compared to, the desired values will be associated with timepoints. This means that the optimization algorithm will at given timepoints compare the current values of x to the value of the reference model at the same time. A visual representation of this is shown in Figure 3.2.

3.4.2 MPC Objective Function

The objective function is where the goal of the optimization is expressed, together with the optimization horizon of the problem. Typical goals of the optimization is to follow a predefined trajectory or reference signal while reducing the control inputs used. This can be expressed as follows [16]:

$$J(N_1,N_2,N_u) = \sum_{j=N_1}^{N_2} \delta(j)[\hat{y}(t+j|t) - w(t+j)]^2 + \sum_{j=1}^{N_u} \lambda(j)[\Delta u(t+j-1)]^2. \tag{3.4}$$

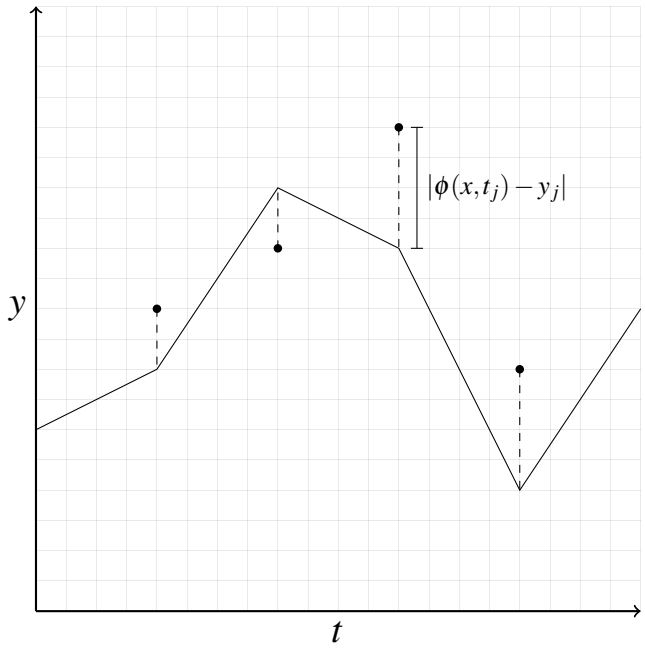


Figure 3.2: The distance between the measurements, represented by dots, and the model, represented by a line.

The first term of (3.4) represents the costs from the states of the model, and the second term represents the cost of the control effort. In the first term $\hat{y}$ is the value of the prediction model, which is compared to the desired trajectory w. In the second term the changes in control Δu is expressed. The change in control is used instead of the value of the control signal itself, since the steady state of the control signal may differ from zero. δ and λ are weighting variables which are used to tune the MPC. The three different N coefficients defines the horizon over which the states and the control effort should be optimized. The optimization horizon for states and control effort can be different, but they will stay the same for this problem.

3.5 Problem Definition

$$\begin{aligned} \min_{\mathbf{x},\Delta\mathbf{u}} \quad & \mathbf{J}_{i+k} = \tfrac{1}{2}\textstyle\sum_{j=i}^{j+N}[(\mathbf{y}(\mathbf{x}_j)-\mathbf{y}_{d,j})^\intercal \mathbf{Q}(\mathbf{y}(\mathbf{x}_j)-\mathbf{y}_{d,j})+(\Delta\mathbf{u}_j)^\intercal \mathbf{R}(\Delta\mathbf{u}_j)] \\ \text{s.t} \quad & \mathbf{x}^{low} \leq \mathbf{x}_j \leq \mathbf{x}^{high} \\ & \mathbf{u}^{low} \leq \mathbf{u}_j \leq \mathbf{u}^{high} \\ & \Delta\mathbf{u}^{low} \leq \Delta\mathbf{u}_j \leq \Delta\mathbf{u}^{high} \\ & \dot{\mathbf{x}}_{j+1} = f(\mathbf{x}_j,\mathbf{u}_j) \end{aligned} \tag{3.5}$$

The equations for the full optimization problem is shown in (3.5). The objective function uses the same setup as shown in (3.4), but in matrix form. Each of the three components of the problem definition will be described in detail in the following sections.

3.5.1 Prediction Model

The linear decoupled UAV model presented in chapter 2 will be used as the prediction model for the MPC. The model is associated with the following states and control inputs:

$$\mathbf{x} = \begin{bmatrix} p_N & p_E & h & u & v & w & \phi & \theta & \psi & p & q & r \end{bmatrix}^\intercal \tag{3.6a}$$

$$\mathbf{u} = \begin{bmatrix} \delta_e & \delta_a & \delta_r & \delta_t \end{bmatrix}^\intercal. \tag{3.6b}$$

The prediction model relates to the equality constraints of equation 3.1 in the form of differential equations. As explained in the previous chapter the control rates $\Delta\mathbf{u}$ will be used as control inputs in the optimization problem:

$$\Delta\mathbf{u} = \begin{bmatrix} \Delta\delta_e & \Delta\delta_a & \Delta\delta_r & \Delta\delta_t \end{bmatrix}^\intercal. \tag{3.7}$$

The control surfaces $\mathbf{u}$ are calculated from the rates $\Delta\mathbf{u}$ through integration:

$$\dot{\mathbf{u}} = \Delta\mathbf{u}. \tag{3.8}$$

3.5.2 Objective Function

The objective function **J** will be minimized over the entire optimization horizon, which consists of N timesteps. The current timestep for the entire optimization problem is denoted i, while the current timestep within the current optimization horizon is denoted j. Since this is an intervalwise MPC, as described in section 3.3, all states within in the interval will be stored, and the interval consists of k timesteps. If the number of intervals needed to cover the entire path is L, the result will contain kL timesteps.

The first term of the objective function calculates the distance between the UAV states and the reference trajectory. The vector $\mathbf{y}_d$ is the *measurement vector*, which is the references for the states:

$$\mathbf{y}_d = \begin{bmatrix} c_{xd} & c_{y_d} & h_d & u_d \end{bmatrix}^\mathsf{T}. \tag{3.9}$$

The function $\mathbf{y}(\mathbf{x})$ holds the current values for the optimization problem. While the height h and velocity u can be used as-is, the camera centre point $\mathbf{c}^n$ needs to be calculated using (2.14):

$$\mathbf{y}(\mathbf{x}) = \begin{bmatrix} p_N + h\cos(\psi)\tan(\theta) - h\sin(\psi)\tan(\phi) \\ p_E + h\sin(\psi)\tan(\theta) + h\cos(\psi)\tan(\phi) \\ h \\ u \end{bmatrix}. \tag{3.10}$$

In order to reduce the control effort for the optimization problem, the rate of change of the control inputs $\Delta\mathbf{u}$ will be minimized. Since all the control rates is to be compared to zero, no function is needed.

The matrices **Q** and **R** are the weighting matrices. They are diagonal matrices where each row represent one state or control rate. The higher the value in the row, the more value is given to the difference between the corresponding state or control rate and the reference trajectory while minimizing the objective function.

3.5.3 Constraints

The states and control inputs to the optimization problem are bounded by constraints to ensure that the values stays within ranges that are physically possible. The constraints $\mathbf{u}^{min}$ and $\mathbf{u}^{max}$ directly relates to the maximum deflection angle for the control surfaces, while the throttle is described as a proportion between zero and one. The same goes for the control rate constraints $\Delta\mathbf{u}^{max}$ and $\Delta\mathbf{u}^{min}$, as these as well are directly related to physical restrictions. It is worth noting that in addition to constraints, the control rates are included in the objective function, which seeks to minimize these variables.

When constraints are put on the optimization problem the complexity of the problem increases, which may make it more computational difficult to find a feasible solution.

For this reason the constraints put on the UAV states $\mathbf{x}^{min}$ and $\mathbf{x}^{max}$ will not be set to begin with, as it is assumed that the ”cheapest” way to fly the aircraft is the ”correct” way. However, if testing shows that the MPC finds solutions that shouldn't be feasible, constraints will be included to remove these solutions.

Chapter 4

MPC Implementation

The offline intervalwise MPC presented in chapter 3 will be implemented using C++ and the ACADO Toolkit [21]. The implementation conists of two main parts: the MPC algorithm that prepares the optimization problem, and the optimization solver that will use the ACADO Toolkit to solve the optimization problem.

This chapter will describe the ACADO Toolkit, how to use it and how it works; as well as the MPC algorithm. An overview of what information the modules share is shown in figure 4.1.

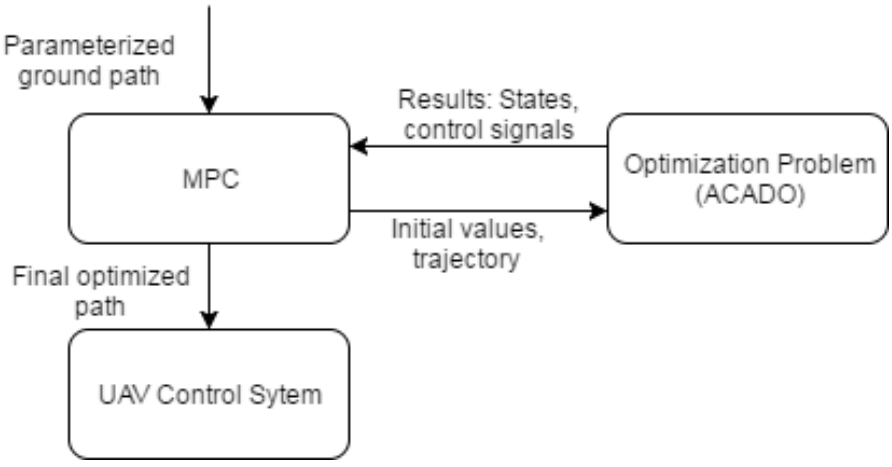


Figure 4.1: An overview of what information the modules share.

4.1 ACADO toolkit

The ACADO Toolkit [21] is an open-source toolkit that supports several different methods for solving optimization problems. The toolkit provides methods to solve four different classes of optimization problem: Optimal control problems, multi-objective optimization and optimal control problems, parameter and state estimation problems, and model predictive control.

Even though the toolkit will be used to create an MPC in this paper, the optimal control problems (OCP) class will be used to solve the optimization problem. The reason for this is that between each iteration of the MPC algorithm a new trajectory must be generated, and the MPC problem class does not have the functionality needed to do this.

4.1.1 Runge-Kutta Method

The Runge-Kutta method is a form of *numerical integrator* that can be used to solve differential equations, and is used by the ACADO toolkit to integrate the prediction model. The method is based on the Euler method, which is a very simple method for numerical integration.

The ACADO toolkit provides algorithms for *explicit* Runge-Kutta methods [21], where explicit means that the method calculates the state of the system at a later time based on the current state. The Runge-Kutta method calculates the later state of the system by calculating several approximations of the derivative of the system. The current state of the system, together with a linear combination of the approximated derivatives, gives the next state of the system [15]. For a system on the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y},t)$, the Runge-Kutta method can be mathematically expressed as:

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_n + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j, t_n + c_i h), i = 1,...,\sigma \tag{4.1a}$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\sum_{j=1}^{\sigma} b_j\mathbf{k}_j \tag{4.1b}$$

where t_n is the current time, $\mathbf{y}_n$ is the output of the system at the current time, h is the step size and σ is the number of approximations of the derivative that is calculated. a, b and c are parameters for the specific Runge-Kutta method used, and c must satisfy $0 < c < 1$.

Systems may consist of dynamics of different time constants, and the difference between the constants may have a big impact on how fast an explicit method can perform the computations. While they don't have a formal definition, systems that explicit methods compute poorly are referred to as *stiff systems* [22]. The poor performance of explicit method comes from the stability of these methods, as the stability is dependent

on the step size. This means that for systems with dynamics of varying time constants, a very small stepsize has to be chosen in order for the method to remain stable. For stiff systems *implicit* methods have a better performance and faster computation time.

4.2 Implementing the Optimization Problem

Since ACADO offers a symbolic way of implementing the optimization problem the equations from chapter 3 can be implemented as they stand. The 12 UAV states $\mathbf{x}$ are implemented as differential states, as well as the four control surfaces $\mathbf{u}$. The control rate $\dot{\mathbf{u}}$ is defined as the control states of the problem, and are linked to the control states through (3.8). The model of the UAV is implemented using the differential equations for each state.

ACADO offers a symbolic way of defining the objective function as an LSQ as well. By defining what states are included in the LSQ and their weights, ACADO automatically minimizes the objective function. The reference model can either be a constant value or a time varying variable. When a path is to be tracked, the path needs to be given with related timepoints.

Lastly, the constraints are also implemented using the symbolic syntax, by simply assigning max-min values for the variables that are subject to constraints. The initial value of the states is also set using the same syntax.

4.2.1 Nonlinear Prediction Model

Initially, effort was made to implement the nonlinear model presented by Beard & McLain [14] as the prediction model in the optimization problem. This would have given more precise results as the nonlinear model is a closer representation of the real UAV. Since the nonlinear model also includes the effect wind has on the aircraft, the path could be optimized with the knowledge about the wind conditions as well. The level of calculation needed for the nonlinear model is significantly higher; however, since this implementation is intended to run offline before the flight occurs, computation time is not a critical concern.

Achieving stable flight within the optimization problem with the nonlinear model on the other hand, turned out to be a difficult task that was far from trivial. This is somewhat due to the nonlinearity, but also to the high coupling between states in the model. The coupling causes changes in one state to affect many other states, which results in a much more complex problem. Several different algorithm and solver settings in ACADO was tested, as well as different objective functions and weighting of these functions. After many attempts the decision to use the linear model instead was made, largely due to this being a project with limited time available.

4.2.2 Nonlinearity

The ACADO toolkit is written for nonlinear optimization problems, and using it together with a linear optimization will give a correct solution, but the computation time for a linear problem will be longer than it needs to be because of extra overhead related to nonlinear algorithms [Cite sourceforge?].

For this reason, using it with a linear prediction model may seem odd. However, the cost function used in this problem is not linear. This is because of both the calculations for the position $\mathbf{p}_N$ and $\mathbf{p}_E$ is represented by nonlinear equations, and the equations to calculate the camera footprint are nonlinear. If the timing demands for this optimization problem was more important, a solution may be to linearize the position equations as well as the equations used to calculate the camera footprint, and then implement the optimization problem using a toolkit that is made for linear problems.

4.3 MPC

The task of the MPC module is to supply the ACADO implementation with the information needed to perform the optimization, and also control the optimization algorithm so that the correct horizon is calculated, as well as storing the results in the correct order. The pseudocode for the MPC implementation is shown in algorithm 1 and 2 in appendix C.1.

4.3.1 Generating the Trajectory

The ground path that is to be observed is assumed to be *time independent*, meaning that it does not matter when a section of the path is captured by the camera. However, the function that minimizes a least-squares objective function that is provided with the ACADO toolkit requires that the path is given as values with associated time points.

In order to meet this requirement a time dependent path will be generated at the beginning of every iteration of the MPC. This will be done by making the assumption that the UAV will maintain its reference speed throughput the horizon. With this assumption the distance the UAV will travel between every timestep can be calculated, and based on this distance the desired position for the UAV at the next timestep can be found. Since the horizon is in the order of seconds and the predicted path is updated every iteration, this assumption will not lead to big errors. The principle behind the calculation is shown in Figure 4.2.

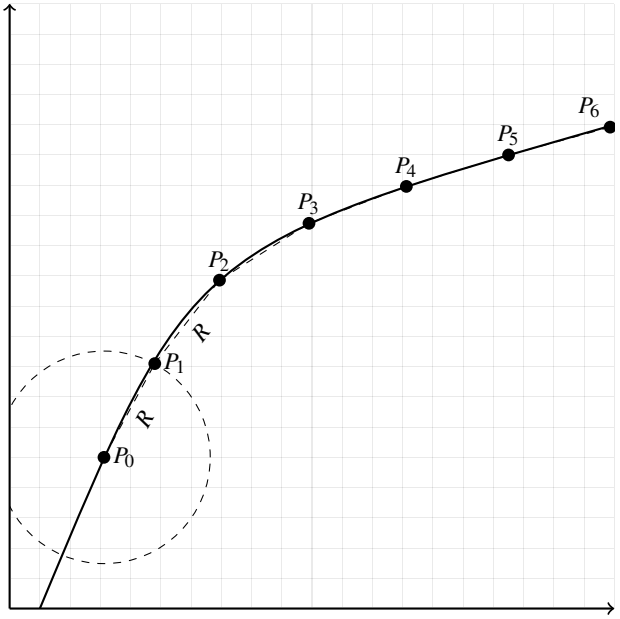


Figure 4.2: Calculating trajectory based on constant speed.

Chapter 5

Simulation Environment

5.1 Software In the Loop Testing

To test if the optimized path gives an improvement in ground observation, and if it is able to keep the observation path within the camera footprint at all times, software in the loop (SITL) testing will be performed. For this the applications Dune, ArduPilot and Neptus will be used. How they interact is shown in figure 5.1, and a short explanation of the three will be given in the following sections.

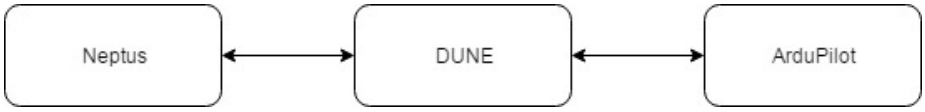


Figure 5.1: An overview of what information the modules share.

5.1.1 DUNE

DUNE Unified Navigation Environment is a part of the LSTS toolchain (Laboratrio de Sistemas e Tecnologia Subaqutica) which aims to provide a control architecture that will ease the control of unmanned air, ground surface and underwater vehicles [23]. DUNE is the software intended to run on board the unmanned vehicles, and provides sensor drivers, and navigation and control functionality. In this thesis DUNE will be used to turn the flight path into waypoints that can be sent to the autopilot.

The functionality needed to track the path is made as a *task* using the API provided by DUNE. The task receives information about the UAV states from the vehicle, and uses this information to provide meaningful input to the autopilot depending on the stage

of the operation. When generating waypoints from the path the task uses the principle of *Line-of-Sight* (LOS) guidance to find waypoints a given distance away from the vehicle.

5.1.2 ArduPilot

ArduPilot is an open-source autopilot, which supports several types of vehicles [24]. It provides functionality for SITL simulation by interfacing the flight dynamics model provided by JSBSim [25]. ArduPilot communicates with DUNE to receive control commands and send information about the UAV states.

5.1.3 Neptus

Neptus is also a part of the LSTS toolchain, and is used to execute the simulations, and generate logs after they are finished [26]. It provides a map interface to observe the simulations in real-time, and also displays information about the aircraft. It communicates with DUNE throught using IMC messages based on a control message set defined by the LSTS toolchain.

5.2 Finding Trim Conditions

The trim conditions, as described in section 2.2, play an important role when using a linear model as this model is linearized about these points. Since the trim conditions also represent a straight level flight they are good initial states and controls for the simulation and are feasible points optimization. In order to find the trim conditions a built-in Matlab function together wtih a Simulink model will be used and the procedure that is used, which will be summarized in this section, is described by Beard & McLain [14].

Matlab features a built-in function `trim` that calculates the trim conditions of a given Simulink model. The Simulink model must be set up with the four control signals as inputs, and the output of the model is the airspeed V_a, the angle of attack α and the sideslip β. The out states are chosen as they easily expresses a trimmed stable flight. The airspeed is set to the desired cruise speed, while the angle of attack and sideslip is set to zero for a straight level flight. The function uses initial guesses of the states and inputs, also the derivatives, and what the desired output is. The Simulink model used for this thesis was developed by Gryte for his master thesis [27].

5.3 Generating the Path

The MPC application expects a parameterized path that can be either linear or curved. The linear paths will be generated using parameterization, and the curved paths will be generated as Dubins paths. This will be done using Matlab, and the generated paths will be stored in a textfile containing only the points on the path.

One important matter when generating the paths is to ensure that the resolution of the path is high enough. How high the resolution needs to be depends on the speed of aircraft and the length of the timestep, and how little uncertainty is needed when generating the trajectory.

Dubins Path

A Dubins path is a path that consists of two circles connected by a straight line, and it has been proven that this is the shortest path between two vehicle configurations [28]. In order to generate the Dubins path the algorithms presented by Beard & McLain [14] will be used. A n illustration of a simple Dubins path is shown in figure 5.2.

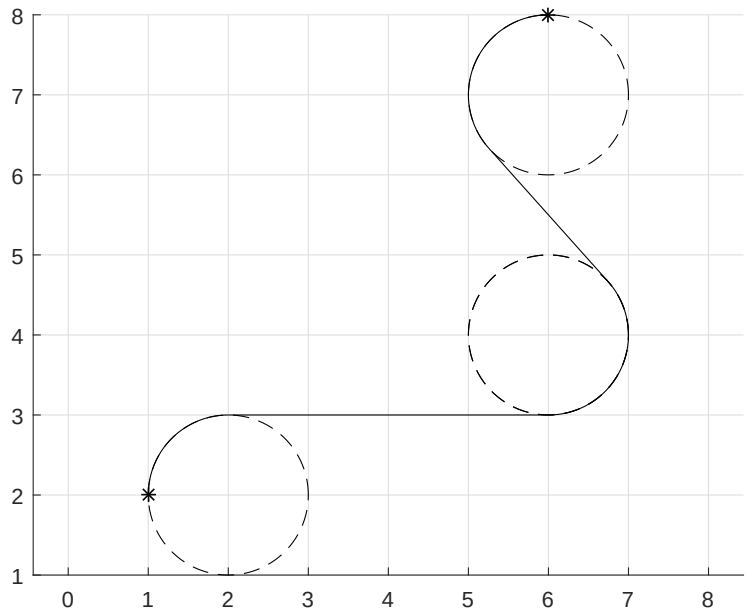


Figure 5.2: An illustration of a simple Dubins path.

Chapter 6

Optimized Paths (Working Title)

In this chapter an analysis of the optimized paths will be done. The MPC will be run on several different paths, and the discussion will focus on how the control problem is solved. Unless stated otherwise all of the optimizations have been performed at an altitude of 150m and a cruising speed of 25m/s.

6.1 Horizon Length

In order to determine what horizon length is needed to optimize the path, several paths were optimized with horizon lengths varying from 10 to 140 spaced by 10. The different horizon lengths were used to optimize a path consisting of a 45° turn, and both linear and curved paths were used. The results of these optimizations can be seen in Figures 6.1 and 6.2.

The results for the three paths are very similar; the longer horizon length, the better tracking of the path. The MPC starts turning earlier than the ground track to compensate for the shift in the camera position, and the aircraft straightens out from the turn later for the same reason.

For the shorter horizon lengths the MPC runs into problems because it hasn't planned far enough ahead when the turn begins. Since it does not look far into the future the aircraft is stille straight above the path when the turn begins. At that point it is too late to start altering the aircrafts position to ensure that the camera stays on the path, so the MPC uses a roll angle in the opposite direction to keep observing the path. This in turn leads to the aircraft turning left, worsening the situation and making the problem more and more difficult. In some cases this causes the roll angle to become very high, causing the MPC to lose control and the aircraft loses height.

Upon closer inspections it can be seen that when the horizon length reaches 90, there are no more big unwanted motions in the system. As the horizon length reaches 110 the optimized paths are almost identical, but lincreasing the horizon length still increases the accuracy of the path tracking. The optimization is time consuming, and as seen in Figure 6.3 the time it takes to optimize the paths increases exponentially. For this reason a horizon length of 110 will be used for the rest of the simulations, since this gives a accurate path tracking for a relatively short computation time.

6.2 Turns

In this section both linear and curved turns will be optimized, and an analysis will be performed on the results.

6.2.1 70° Turn

Curved 70° Turn

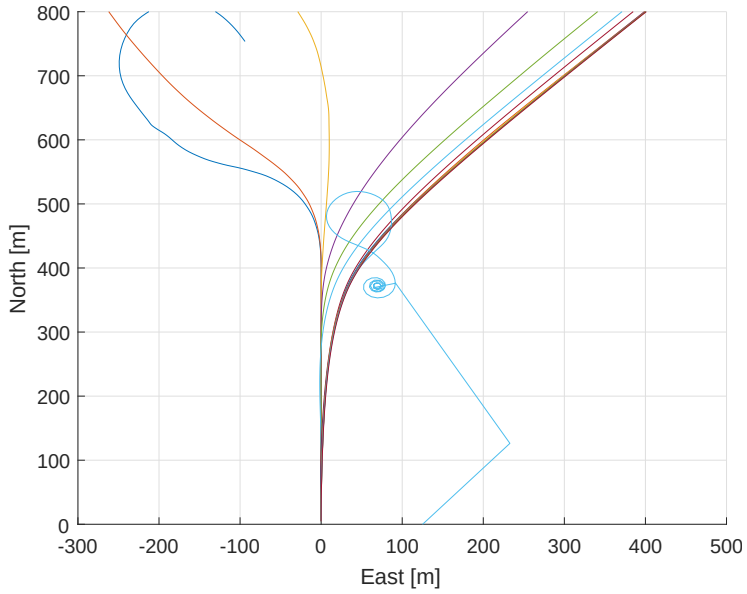
The optimized path of a 70° turn with a radius of 150m is shown in Figure 6.4. Even though the camera centre points deviates more from the desired path than during the 45° turn, the result is still a smooth path with no big unwanted motions.

Linear 70° Turn

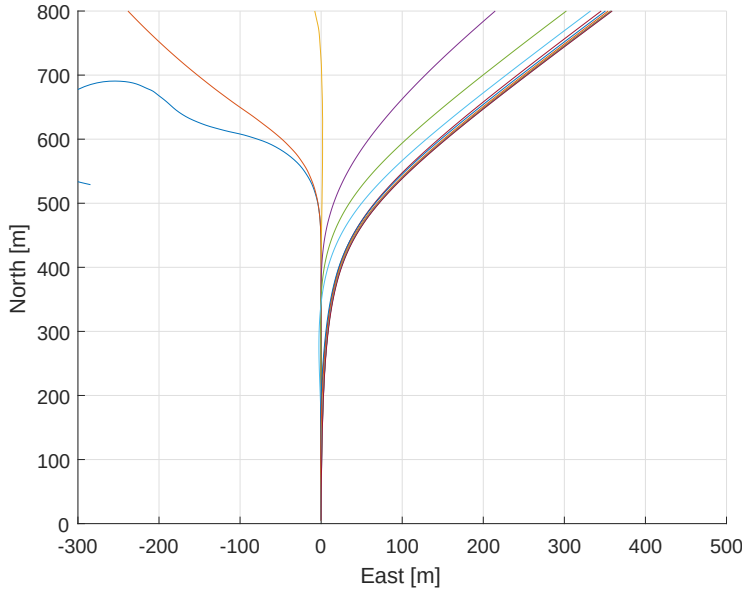
Optimizing a path containing a 70° linear turn returns a very different result than for the curved turn. As can be seen in Figure 6.5, the MPC is not able to achieve stable flight throughout the turn. The reason for this is similar to the reason why optimization with a short horizon length fails: the optimization tries banking the aircraft left to track the path, instead of cutting the corner. In this case the banking happens before the UAV have reached the corner as a result of the optimization ”foreseeing” that it cannot keep tracking the path.

In an attempt to achieve stable flight throughout the turn, the weighting on camera position in the objective function was reduced. The result of changing the weighting from 10^{-1} to 10^{-3} can be seen in Figure 6.6. This tuning results in a stable flight, but the path tracking is not as precise and smooth as for the 45° turn. In addition the resulting camera path consists of several loops. This occurs as a combination of both the roll angle and pitch angle changes at the same time.

A third attempt on a linear 70° turn was made, this time with a weighting on the camera position of 10^{-5}. As can be seen in Figure 6.7, this results in a stable flight. The path tracking on the other hand is poor. With this tuning, the weight on the camera position is so much lower than on the other states included in the objective function so that the MPC finds a path that do not track the path returns the lowest cost. The optimization

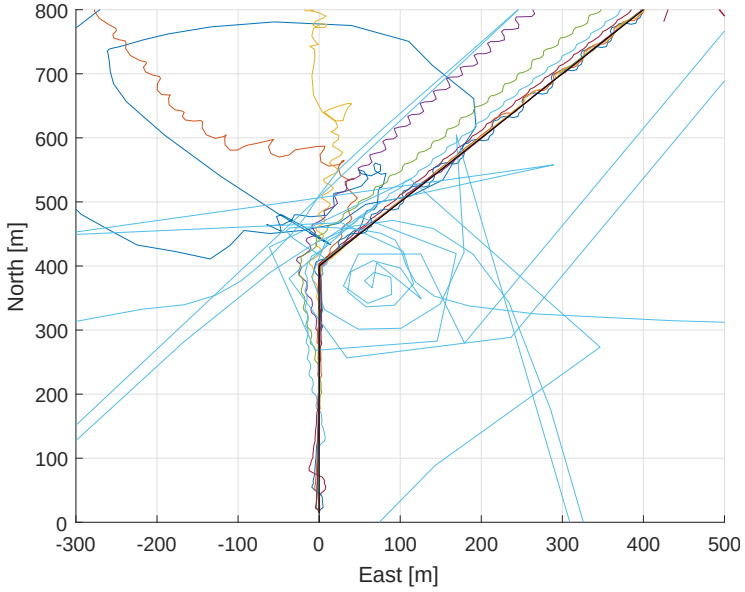


(a) Linear 45° turn.

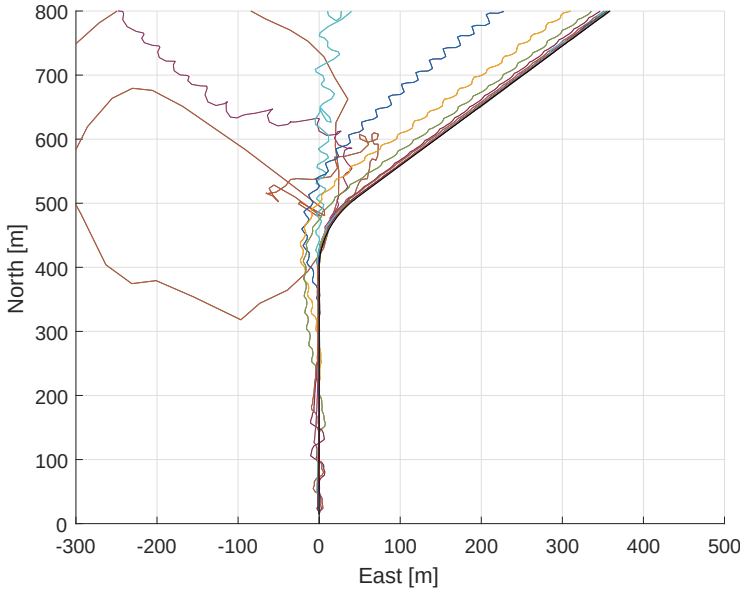


(b) Curved 45° turn with 200m radius.

Figure 6.1: The position of the UAV during the two turns with horizon lengths varying from 10 to 140.



(a) Linear 45° turn.



(b) Curved 45° turn with 200m radius.

Figure 6.2: The camera position during the two turns with horizon lengths varying from 10 to 140.

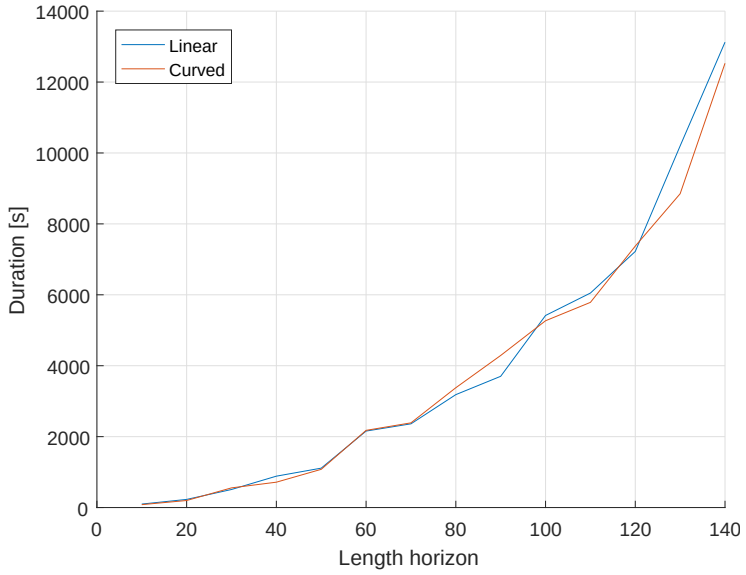
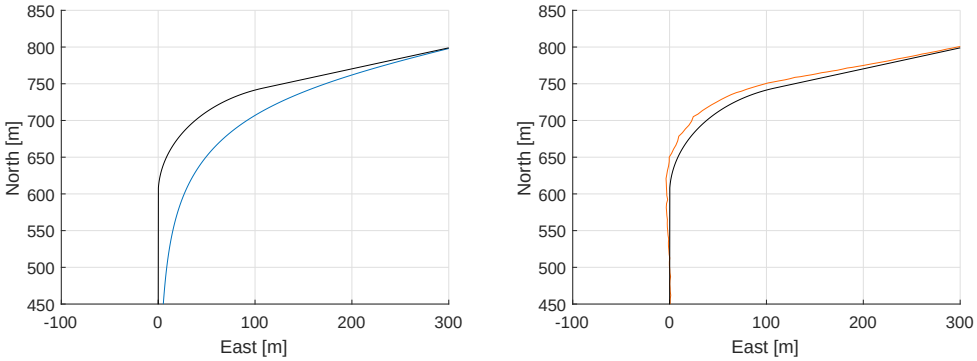


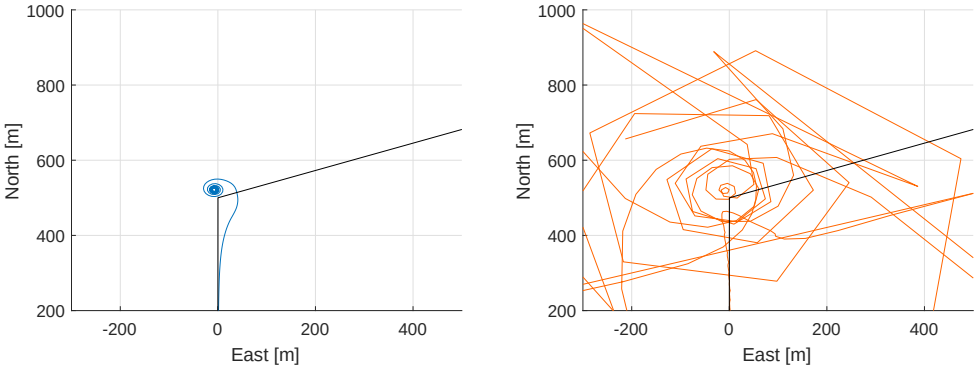
Figure 6.3: Duration of each optimization with different horizon length.



(a) UAV position.

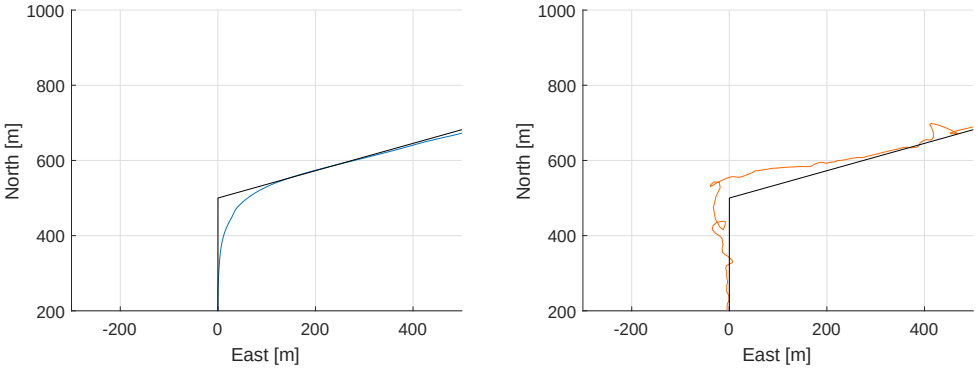
(b) Camera poistion.

Figure 6.4: Results of optimizing a curved 70° turn.



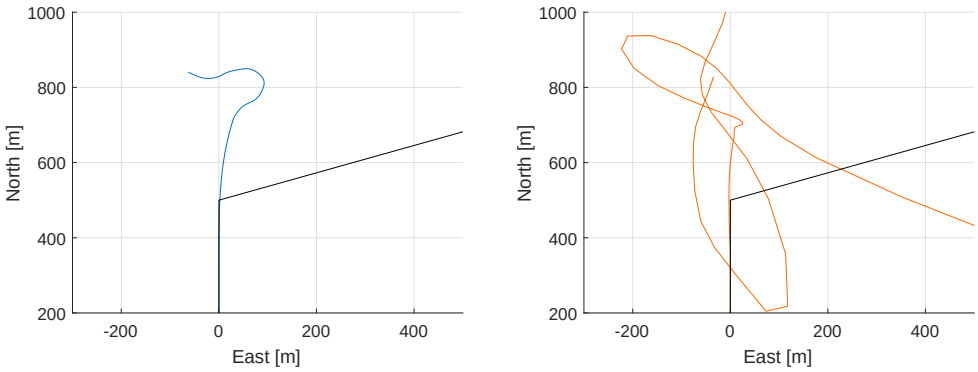
(a) UAV position. (b) Camera poistion.

Figure 6.5: Results of optimizing a linear 70° turn with 10^{-1} weight on camera position.



(a) UAV position. (b) Camera poistion.

Figure 6.6: Results of optimizing a linear 70° turn with 10^{-3} weight on camera position.



(a) UAV position.

(b) Camera poistion.

Figure 6.7: Results of optimizing a linear 70° turn with 10^{-5} weight on camera position.

still makes some attempts to track the path, but ends up ”swinging” the camera quickly past the path.

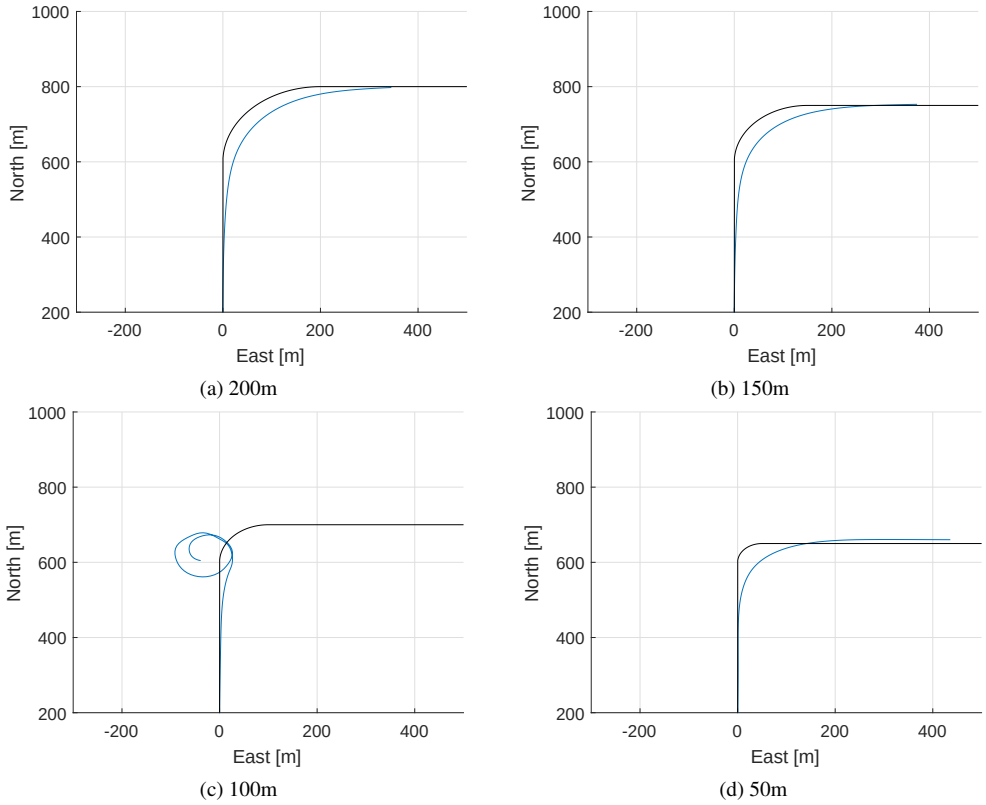
6.2.2 90° Turn

Curved 90° Turn

Figure 6.8 and 6.9 shows the optimized paths of a 90° path with different turn radii. The results show that with a radius of 200m, 150m, and 50m the optimization algorithm returns a smooth stable flight path. However, the camera position contains more nudges than for gentler turns, which happens because of the roll angle varying more. This can also be seen in Figure 6.10. When optimizing the turn with 50m radius the roll angle reaches almost 30°. It can also be seen that the sharper the turn the earlier the aircraft starts banking.

As expected the camera centre points deviates more from the original path than for gentler turns. After the 50m radius turn in Figure 6.9d, the camera position ends up with a significant deviance of 50m away from the desired path. While this most likely would correct itself if the path was simulated further, it is a result of how the cost function is weighted. The weight on the camera position is very low compared to the weight put on control rates, as well as speed and altitude, which means that the deviance from the path has a smaller cost than correcting it.

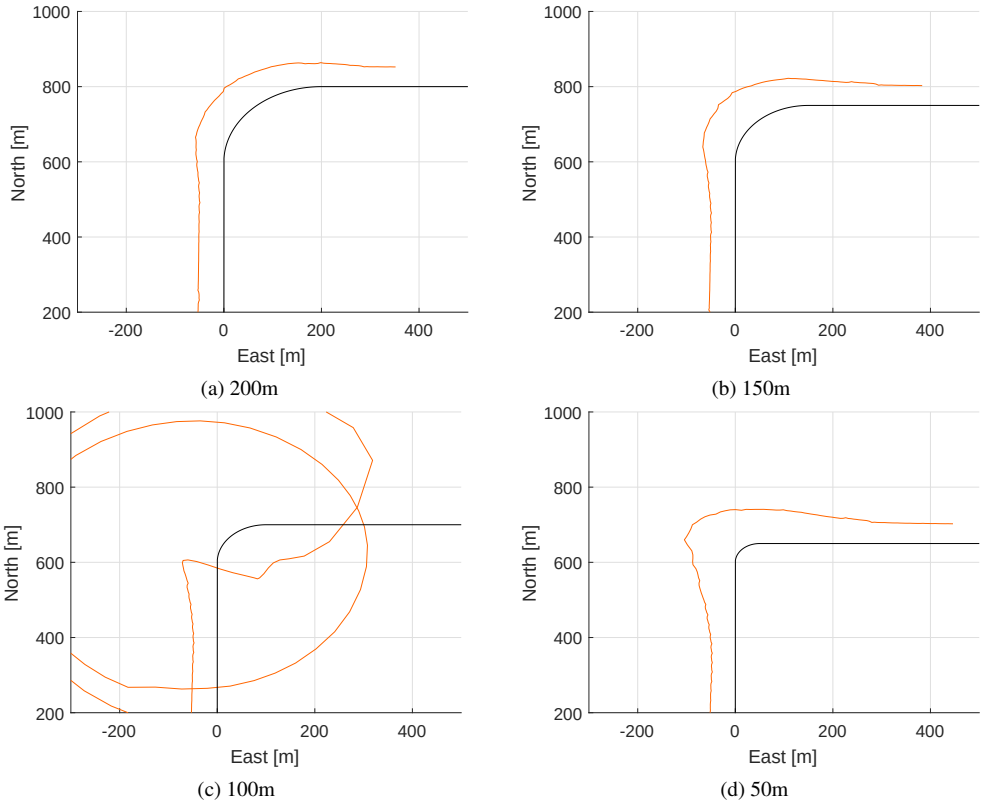
A more surprising result is seen in Figures 6.9c and 6.9c. Even though the MPC is able to optimize the turn with 50m radius, it is not able to return a stable flight path for the turn with 100m radius. The camera position path is similar to previous failures: instead of optimizing the UAV position to track the camera path it uses the bank angle to achieve a satisfying short-term solution.



(a) 200m (b) 150m

(c) 100m (d) 50m

Figure 6.8: The position of the UAV when optimizing a curved 90° turn with varying radius.



(a) 200m

(b) 150m

(c) 100m

(d) 50m

Figure 6.9: The position of the camera when optimizing a curved 90° turn with varying radius.

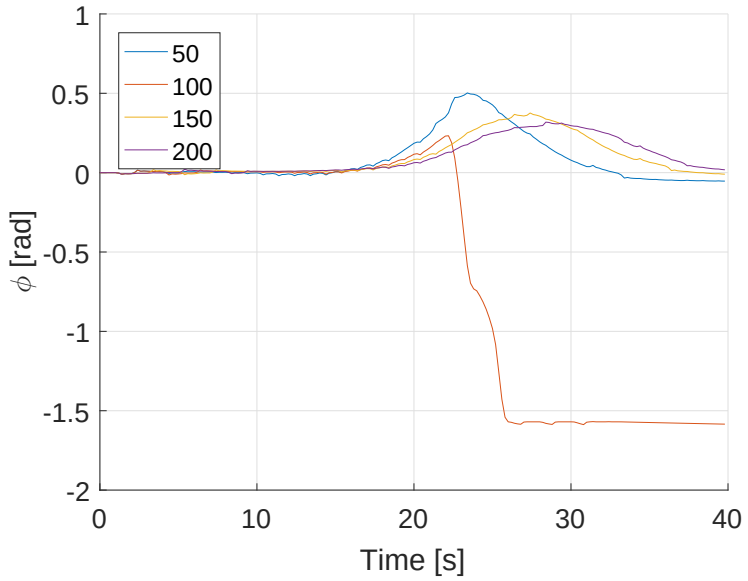


Figure 6.10: The roll angle ϕ during the 90° turns.

Linear 90° Turn

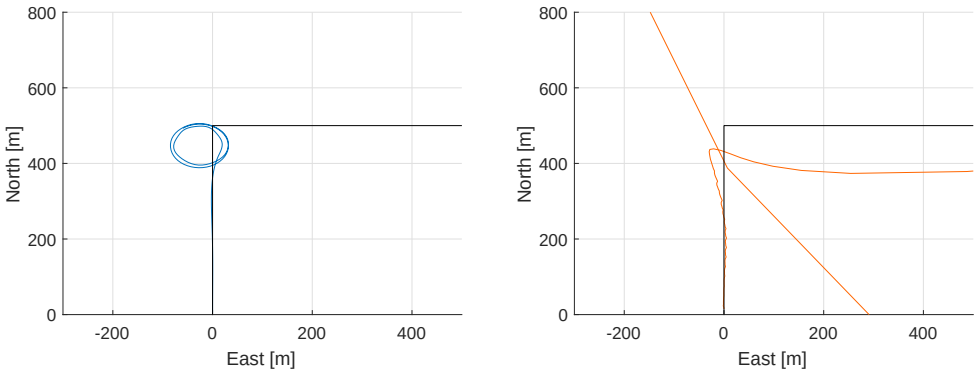
Since the MPC struggles to track a 70° corner, it is no surprise that it fails to track a 90° corner as seen in Figure 6.11. Once again the MPC ends up with solving a difficult path using roll instead of the position, which ends with a bad solution. Different weightings on the position was tested without success.

6.2.3 180° Turn

When optimizing a 180° turn the radius plays a big role, as can be seen in Figures 6.12 and 6.13. The figures show optimizations of six 180° turns with radius ranging from 300m to 50m.

Unsurprisingly, broader turns give better solutions. For the turn with 300m radius the tracking is very good. And while the tracking is not as good for the turns with 250m and 200m, they return a smooth stable path that puts the camera centre not too far away from the path. Notice in Figures 6.13b and 6.13c that the tracking not only is worse throughout the turn, there is also more overshoot at the end of the turn. The MPC most likely accepts this overshoot as the deviance from the camera path is a lower cost than using more control input to sharpen the turn. Figure 6.14 shows that the 250m and 200m turns already have higher roll angles, and opposed to the 300m turn they do not flatten out during the turn.

For the radii 150m, 100m, and 50m the result is not as good. For the turns with 150m



(a) UAV position.

(b) Camera poistion.

Figure 6.11: Result of attempting to optimize a linear 90° turn.

and 100m radius the result is similar to previous paths where the path is too sharp: the MPC uses roll to compensate for the sharp turn, which causes the aircraft to enter a spiral. In Figure 6.12e it may appear as the aircraft is about to recover and continue tracking the ground path, but the height plots show that at this point the aircraft is only about 20m above ground and still descending.

For the 50m turn in Figure 6.12f the MPC fails differently. About 250m before the turn even begins the aircraft drifts off to the left, the opposite direction of the turn. In Figure 6.13f it can be seen that the camera point is still close to the ground path. However, it is swinging rapidly from side to side a few times before it completely drifts off.

6.3 Effect of Height

As can be seen in (2.12), the effect the pitch and roll has on the camera position increases proportionally with the altitude of the aircraft. In Figure 6.15 a 45° curved turn has been optimized with the aircraft flying at different altitudes, namely 100m, 200m and 300m. While the altitude do not affect the path the MPC chooses to fly the aircraft, the effect is easily visible in the camera position shown in Figure 6.15b. When flying at an altitude of 100m the camera position takes the inner turn, while for 300m it greatly widens the turn.

6.4 Path

6.4.1 Two Opposite Turns

How the MPC takes advantage of subsequent opposite turns can be seen in Figure 6.16. When there is an opposite turn trailing the first turn, the optimized path cuts across the

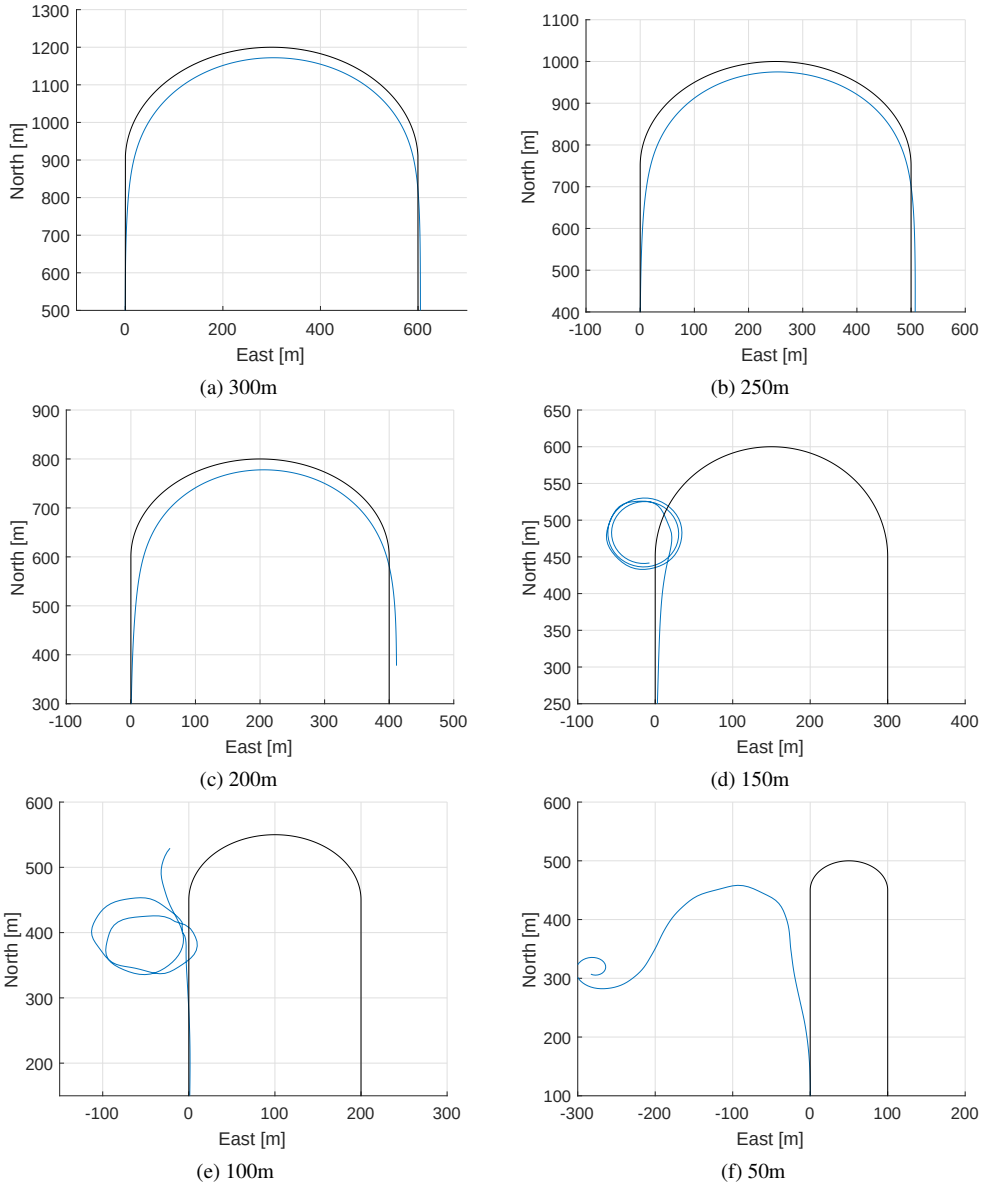
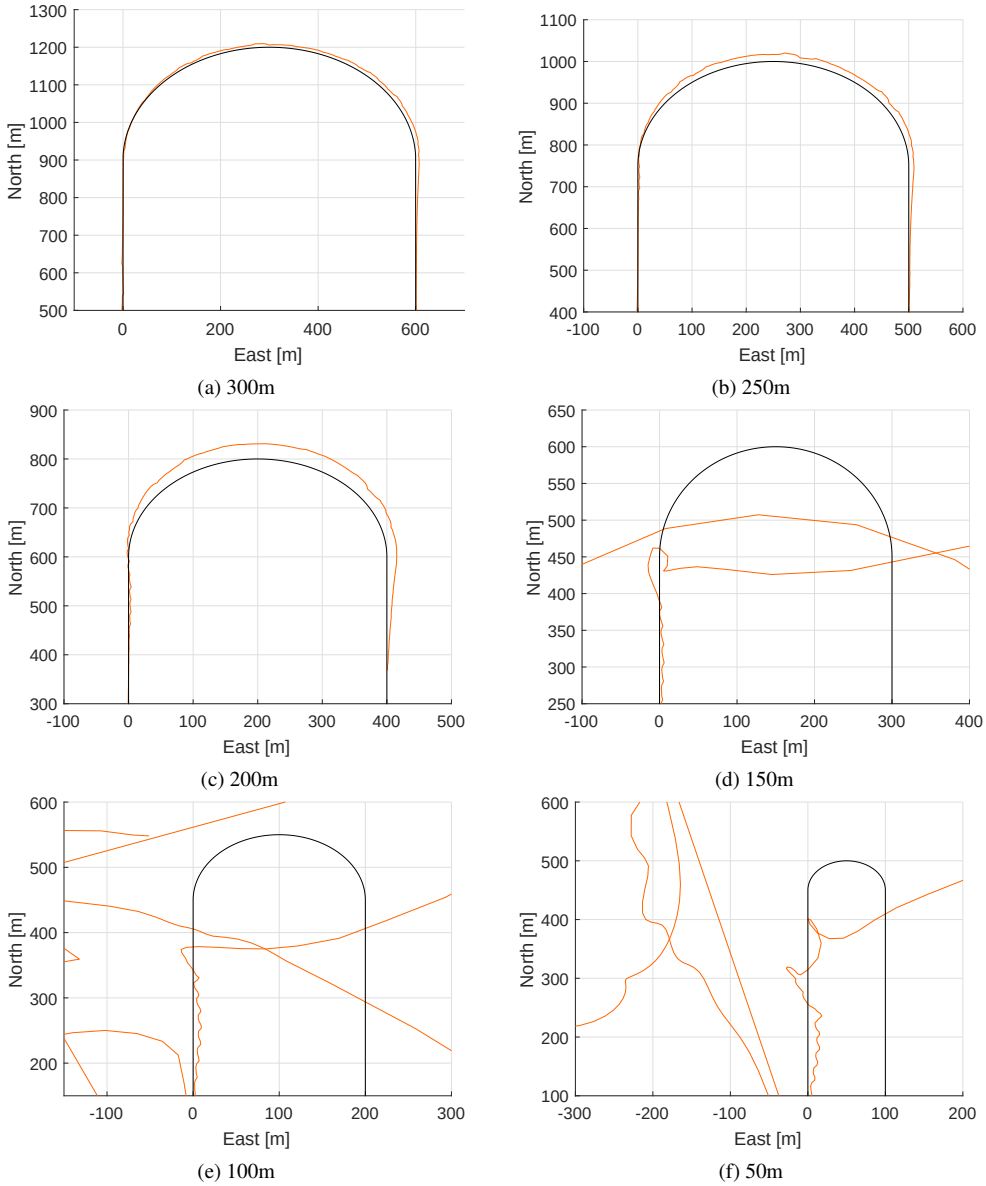


Figure 6.12: The position of the UAV when optimizing a curved 180° turn with varying radius.



(a) 300m (b) 250m

(c) 200m (d) 150m

(e) 100m (f) 50m

Figure 6.13: The position of the camera when optimizing a curved 180° turn with varying radius.

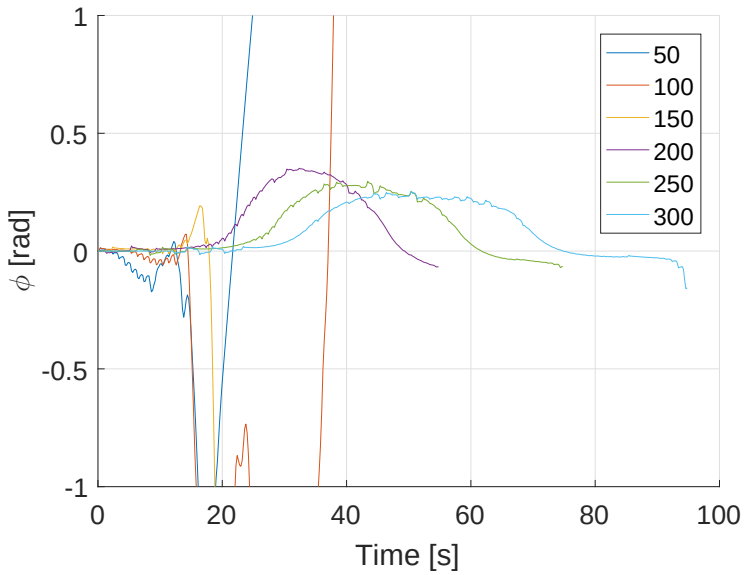
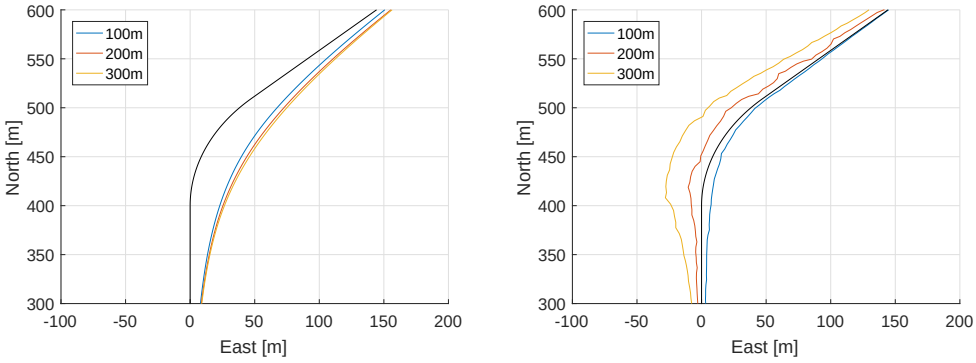


Figure 6.14: The roll angle ϕ during the 180° turns.



(a) UAV position

(b) Camera position

Figure 6.15: The UAV and camera position when tracking a 45° turn with 200m radius at different altitudes.

line connecting the two turns, while still keeping the camera on the observation path. This is also seen in Figure 6.16e that shows the heading angle throughout the turn. The heading angle never reaches 45° or 70°, because of cutting across the line.

6.4.2 Lawnmover Path

The result of attempting to optimize a lawnmover-pattern path is shown in Figure 6.17. The radius was set to be 250m, as the optimization of 180° path show that the MPC performs well on this turn, and the line connecting to archs is set to 500m.

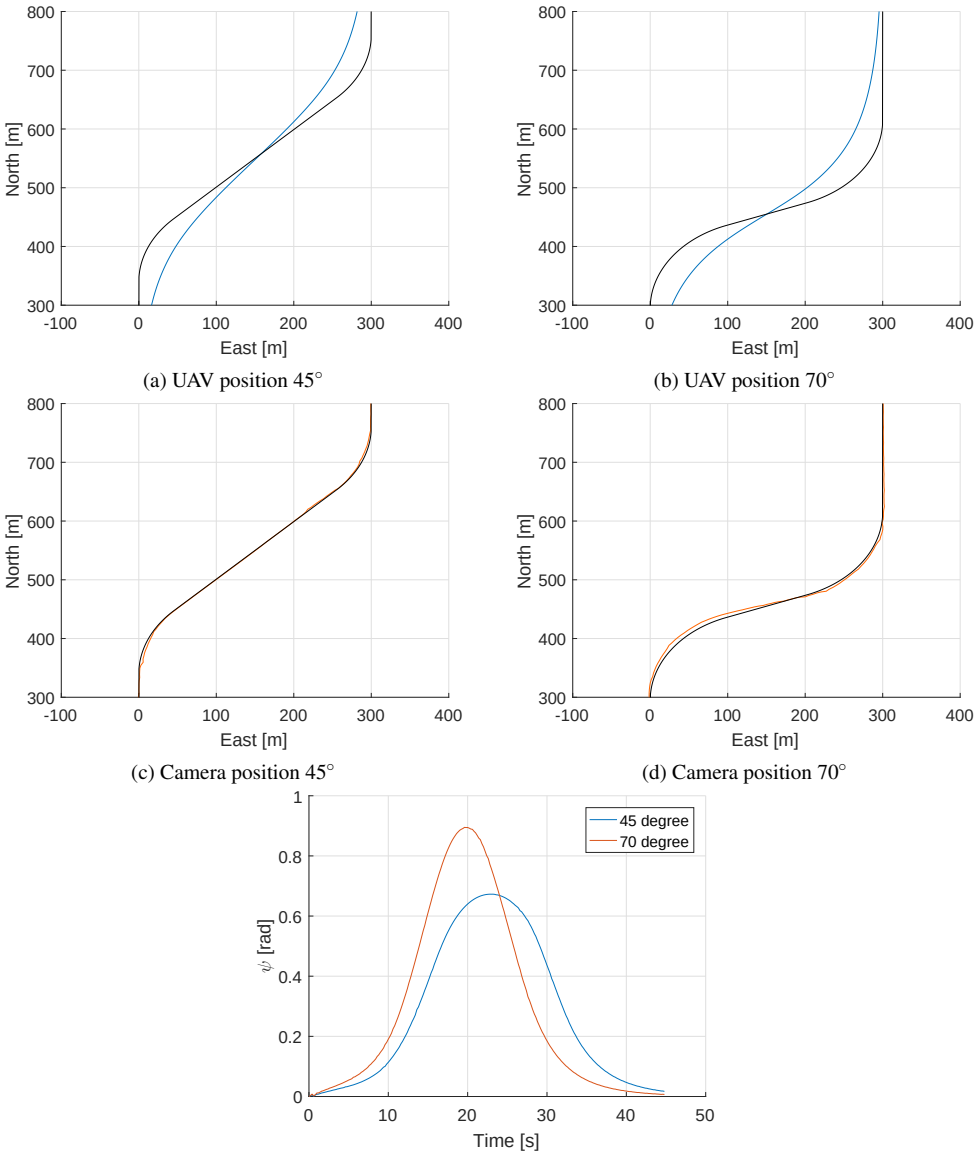
When the MPC optimizes the first 180° turn, it performs in a similar manner to the results in section 6.2.3: it takes the inner turn to begin with, and then widens the turn at the end in order to avoid changing the roll angle quickly. While it should have taken the advantage of the straight line to position the UAV right above the path so it could return to trimmed flight, it instead chooses continue flying next to the path with a constant roll angle to compensate for the offset from the path. It uses the rudder to compensate for the roll angle so it can travel at a constant course angle. The roll angle it uses to track the path while flying next to the path can be seen in Figure 6.17c.

As the UAV approaches the next 180° arch it is already on the inner side of the turn. However, instead of this being an advantage since it optimizes a 180° turn like this by taking the inner turn, it throws the UAV into an unstable right-turning spiral.

6.5 Reducing the Stepsize

For the results shown in this section a stepsize of 0.2s has been used for the MPC. This stepsize returns smooth stable paths for most of the curved paths, but do not work well with the sharper corners in the linear paths. Testing showed that by decreasing the timestep to 0.1s, the MPC manages to return a stable flight path for linear corners with a 70° angle. The MPC was still not able to return a stable flight path for 90° linear turns.

Figure **??** shows the result of optimizing a linear 70° turn with a stepsize of 0.2s. Even though the MPC returns a stable flight path, it can be seen that the camera contains many small oscillations. By closer inspection of Figure **??**, it can be seen that the roll angle of the aircraft oscillates throughout the entire flight.



(a) UAV position 45°

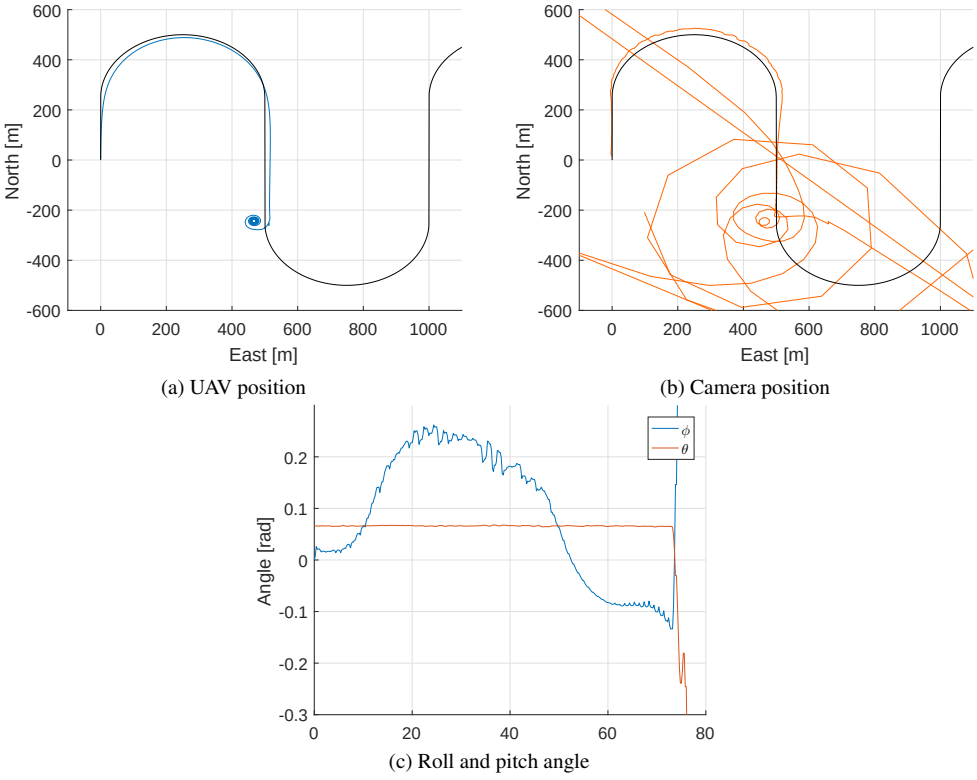
(b) UAV position 70°

(c) Camera position 45°

(d) Camera position 70°

(e) Heading angle

Figure 6.16: UAV position, camera position and heading angle during two subsequent turns of 45° and 70°.



(a) UAV position

(b) Camera position

(c) Roll and pitch angle

Figure 6.17: Result of attempting to optimize a lawnmover-pattern path with radius 250m.

Chapter 7

Simulating the Optimized Path

7.1 Curved Paths

7.2 Linear Paths

Chapter 8

Discussion

8.1 Oscillations (Working Title)

The results in section 6.5 show that when the stepsize is reduced, the oscillations in the states increase. This phenomenon is also present when using a stepsize of 0.2s, but because of the longer time between every re-initiation the effect is not as visible as for 0.1s.

A major contributor to generating these oscillations is the trajectory generator described in section 4.3.1, that assumes the UAV will maintain a fixed speed in order to calculate the distance the UAV will travel during one timestep. This assumption will introduce some inaccuracies, but the results show that during the optimization the speed did not vary much. However, this method of generating the trajectory introduces another inaccuracy, in that the path given as an input is discretized.

Since the reference path is discretized, it is not possible to always find a point up ahead that is exactly the distance the UAV will travel during one timestep away. For this reason, points with a distance away that is within a given range is accepted as the next waypoint. If this range is too big the inaccuracy will be too big, which will cause a spike in the reference model in the cost function. Because of this spike in reference the optimization algorithm will seek to follow the spike, which causes the oscillations in the states.

In Figure 8.1 the effect the acceptance range of waypoints has on the roll through a 45° linear turn is shown. When the acceptance range is ± 0.5m, the magnitude of the oscillations is bigger than for an acceptance range of ± 0.1m. A part from a spike in the wrong direction for the ± 0.5m signal, the two roll angles follow approximately the same trajectory.

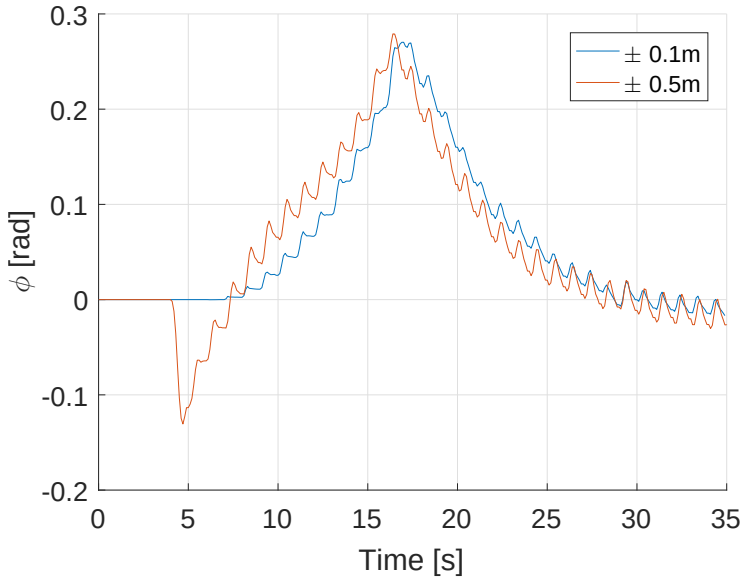


Figure 8.1: The roll angle of the aircraft during a linear 45° turn with different acceptance rates for waypoints.

8.2 Comment on Control Signals

8.3 Cost Function

Chapter 9

Conclusion

9.1 Future Work

- Improve how the trajectory is generated
- Linearize cost function - implement solver made for linear problems
- Implement nonlinear model
- Changin altitude

Appendices

Appendix A

Nonlinear UAV Model

The complete state space equations for the nonlinear model presented by Beard & McLain [14], that is the basis for the linearized model used as the prediction model in the MPC, is given here.

$$\begin{bmatrix} \dot{p}_n \\ \dot{p}_e \\ \dot{p}_d \end{bmatrix} = \begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ c_\theta c_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{A.1}$$

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{\mathsf{m}} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \tag{A.2}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & \frac{sin(\phi)}{cos(\theta)} & \frac{cos(\phi)}{cos(\theta)} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{A.3}$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} \Gamma_1 pq - \Gamma_2 qr \\ \Gamma_5 pr - \Gamma_6 (p^2 - r^2) \\ \Gamma_7 pq - \Gamma_1 qr \end{bmatrix} + \begin{bmatrix} \Gamma_3 l + \Gamma_4 n \\ \frac{1}{J_y} m \\ \Gamma_4 l + \Gamma_8 n \end{bmatrix} \tag{A.4}$$

f_x, f_y and f_z in (A.2) represent the forces in each direction in the body frame, and m represent the mass of the UAV. In (A.4) the Γ values represent the inertia of the UAV, and l, m, and n is the moments about the axes in the body frame.

Appendix B

ACADO Code

```
1  #include <acado_optimal_control.hpp>
2
3  #include "mpc_script.hpp"
4  #include "aerosonde_param.hpp"
5
6
7
8  ACADO::DMatrix optimize_path(
9                      ACADO::VariablesGrid path,
10                     ACADO::DVector X0,
11                     ACADO::DVector U0,
12                     ACADO::DVector DU0){
13
14   USING_NAMESPACE_ACADO
15
16
17   //_____________________________________________
18   /* Introduce Variables */
19
20   DifferentialState p_N, p_E, h;
21   DifferentialState u, v, w;
22   DifferentialState phi, theta, psi;
23   DifferentialState p, q, r;
24   DifferentialState elevator, aileron;
25   DifferentialState rudder, throttle;
26
27   Control d_elevator;
28   Control d_aileron;
```

```
29  Control d_rudder;
30  Control d_throttle;
31
32  IntermediateState p_N_dot, p_E_dot, p_D_dot;
33  IntermediateState Va, alpha, beta;
34  IntermediateState GAMMA, CHI;
35  IntermediateState cx, cy;
36
37  DifferentialEquation f;
38
39  // Lateral state-space model coefficients
40  double Yv, Yp, Yr, Yda, Ydr;
41  double Lv, Lp, Lr, Lda, Ldr;
42  double Nv, Np, Nr, Nda, Ndr;
43
44  // Longitudinal state-space model coefficients
45  double Xu, Xw, Xq, Xde, Xdt;
46  double Zu, Zw, Zq, Zde;
47  double Mu, Mw, Mq, Mde;
48  double C_L, C_D;
49  double C_X0, C_Xalp, C_Xde, C_Xq;
50  double C_Z0, C_Zalp, C_Zde, C_Zq;
51
52  // Misc
53  const double g = 9.81;
54  const double PI = 3.14;
55
56  Va = sqrt(u*u + v*v + w*w);
57  alpha = atan(w/u);
58  beta = asin(v/Va);
59
60
61
62  //-----------------------------------------
63  /* Trimmed State Variables */
64
65  const double u_trim = 25.0;
66  const double v_trim =  0.0;
67  const double w_trim =  0.0;
68
69  const double phi_trim   = 0.0;
70  const double theta_trim = 0.066;
71  const double psi_trim   = 0.0;
72
73  const double p_trim = 0.0;
74  const double q_trim = 0.0;
```

```
75  const double r_trim = 0.0;
76
77  const double elevator_trim = -0.15;
78  const double aileron_trim  = 0.0;
79  const double rudder_trim   = 0.0;
80  const double throttle_trim = 0.1;
81
82  const double Va_trim    = 25.0;
83  const double beta_trim  =  0.0;
84  const double alpha_trim =  0.0;
85
86  const double h_trim = 150.0;
87
88  //________________________________________
89
90  double gamma = J_x*J_z - J_xz*J_xz;
91  double gamma_1 = J_xz*(J_x-J_y+J_z)/gamma;
92  double gamma_2 = (J_z*(J_z-J_y)+J_xz*J_xz)/gamma;
93  double gamma_3 = J_z/gamma;
94  double gamma_4 = J_xz/gamma;
95  double gamma_5 = (J_z - J_x)/J_y;
96  double gamma_6 = J_xz/J_y;
97  double gamma_7 = ((J_x-J_y)*J_x+J_xz*J_xz)/gamma;
98  double gamma_8 = J_x/gamma;
99
100 double C_p0 = gamma_3*C_l0    + gamma_4*C_n0;
101 double C_pbeta = gamma_3*C_lbeta + gamma_4*C_nbeta;
102 double C_pp = gamma_3*C_lp    + gamma_4*C_np;
103 double C_pr = gamma_3*C_lr    + gamma_4*C_nr;
104 double C_pda = gamma_3*C_lda   + gamma_4*C_nda;
105 double C_pdr = gamma_3*C_ldr   + gamma_4*C_ndr;
106 double C_r0 = gamma_4*C_l0    + gamma_8*C_n0;
107 double C_rbeta = gamma_4*C_lbeta + gamma_8*C_nbeta;
108 double C_rp = gamma_4*C_lp    + gamma_8*C_np;
109 double C_rr = gamma_4*C_lr    + gamma_8*C_nr;
110 double C_rda = gamma_4*C_lda   + gamma_8*C_nda;
111 double C_rdr = gamma_4*C_ldr   + gamma_8*C_ndr;
112
113
114
115 //________________________________________
116 /* Force Coefficients: X and Z */
117
118 C_L = C_L0 + C_Lalp*alpha_trim;
119 C_D = C_D0 + C_Dalp*alpha_trim;
120
```

```
121   C_X0   = 0;
122   C_Xalp = - C_D*cos(alpha_trim)
123            + C_L*sin(alpha_trim);
124   C_Xde  = - C_Dde*cos(alpha_trim)
125            + C_Lde*sin(alpha_trim);
126   C_Xq   = - C_Dq*cos(alpha_trim)
127            + C_Lq*sin(alpha_trim);
128
129   C_Z0 = 0;
130   C_Zalp = - C_D*sin(alpha_trim)
131            - C_L*cos(alpha_trim);
132   C_Zde  = - C_Dde*sin(alpha_trim)
133            - C_Lde*cos(alpha_trim);
134   C_Zq   = - C_Dq*sin(alpha_trim)
135            - C_Lq*cos(alpha_trim);
136
137
138
139   //________________________________________
140   /* Lateral State-Space model coefficients */
141
142   Yv = ((rho*S*b*v_trim)/(4*mass*Va_trim))
143            * (C_Yp*p_trim + C_Yr*r_trim)
144      + ((rho*S*v_trim)/mass)
145            * (C_Y0 + C_Ybeta*beta_trim
146            + C_Yda*aileron_trim + C_Ydr*rudder_trim)
147      + ((rho*S*C_Ybeta)/(2*mass))
148            * sqrt(u_trim*u_trim + w_trim*w_trim);
149
150   Yp =  w_trim + ((rho*Va_trim*S*b)/(4*mass))*C_Yp;
151   Yr = -u_trim + ((rho*Va_trim*S*b)/(4*mass))*C_Yr;
152
153   Yda = ((rho*Va_trim*Va_trim*S)/(2*mass))*C_Yda;
154   Ydr = ((rho*Va_trim*Va_trim*S)/(2*mass))*C_Ydr;
155
156   Lv = ((rho*S*b*b*v_trim)/(4*Va_trim))
157            * (C_pp*p_trim + C_pr*r_trim)
158      + rho*S*b*v_trim*(C_p0 + C_pbeta*beta_trim
159            + C_pda*aileron_trim + C_pdr*rudder_trim)
160      + ((rho*S*b*C_pbeta)/2)
161            * sqrt(u_trim*u_trim + w_trim*w_trim);
162
163   Lp = gamma_1*q_trim + ((rho*Va_trim*S*b*b)/4)*C_pp;
164   Lr = -gamma_2*q_trim + ((rho*Va_trim*S*b*b)/4)*C_pr;
165
166   Lda = ((rho*Va_trim*Va_trim*S*b)/2)*C_pda;
```

```
167   Ldr = ((rho*Va_trim*Va_trim*S*b)/2)*C_pdr;
168
169   Nv = ((rho*S*b*b*v_trim)/(4*Va_trim))
170            *(C_rp*p_trim + C_rr*r_trim)
171      + rho*S*b*v_trim*(C_r0 + C_rbeta*beta_trim
172            + C_rda*aileron_trim + C_rdr*rudder_trim)
173      + ((rho*S*b*C_rbeta)/2)
174            * sqrt(u_trim*u_trim + w_trim*w_trim);
175
176   Np =  gamma_7*q_trim + ((rho*Va_trim*S*b*b)/4)*C_rp;
177   Nr = -gamma_1*q_trim + ((rho*Va_trim*S*b*b)/4)*C_rr;
178
179   Nda = ((rho*Va_trim*Va_trim*S*b)/2)*C_rda;
180   Ndr = ((rho*Va_trim*Va_trim*S*b)/2)*C_rdr;
181
182
183
184   //________________________________________
185   /* Longitudinal State-Space Model Coefficients */
186
187   Xu = ((u_trim*rho*S)/mass)*(C_X0 + C_Xalp*alpha_trim
188                                + C_Xde*elevator_trim)
189      - ((rho*S*w_trim*C_Xalp)/(2*mass))
190      + ((rho*S*c*C_Xq*u_trim*q_trim)/(4*mass*Va_trim))
191      - ((rho*S_prop*C_prop*u_trim)/mass);
192
193   Xw = -q_trim+((w_trim*rho*S)/mass)
194            *(C_X0 + C_Xalp*alpha_trim
195                   + C_Xde*elevator_trim)
196      + ((rho*S*c*C_Xq*w_trim*q_trim)/(4*mass*Va_trim))
197      + ((rho*S*C_Xalp*u_trim)/(2*mass))
198                  - ((rho*S_prop*C_prop*w_trim)/mass);
199
200   Xq  = -w_trim + ((rho*Va_trim*S*C_Xq*c)/(4*mass));
201   Xde = ((rho*Va_trim*Va_trim*S*C_Xde)/(2*mass));
202   Xdt = ((rho*S_prop*C_prop*k_motor
203                    *k_motor*throttle_trim)/mass);
204
205   Zu = q_trim + ((u_trim*rho*S)/mass)*(C_Z0
206            + C_Zalp*alpha_trim + C_Zde*elevator_trim)
207      - ((rho*S*C_Zalp*w_trim)/(2*mass))
208      + ((u_trim*rho*S*C_Zq*c*q_trim)
209                    /(4*mass*Va_trim));
210
211   Zw = ((w_trim*rho*S)/mass)*(C_Z0
212            + C_Zalp*alpha_trim + C_Zde*elevator_trim)
```

```
213     + ((rho*S*C_Zalp*u_trim)/(2*mass))
214     + ((rho*w_trim*S*c*C_Zq*q_trim)
215                     /(4*mass*Va_trim));
216
217   Zq = u_trim + ((rho*Va_trim*S*C_Zq*c)/(4*mass));
218   Zde = ((rho*Va_trim*Va_trim*S*C_Zde)/(2*mass));
219
220
221   Mu = ((u_trim*rho*S*c)/J_y)*(C_m0
222           + C_malp*alpha_trim + C_mde*elevator_trim)
223     -((rho*S*c*C_malp*w_trim)/(2*J_y))
224     +((rho*S*c*c*C_mq*q_trim*u_trim)
225                     /(4*J_y*Va_trim));
226
227   Mw = ((w_trim*rho*S*c)/J_y)*(C_m0
228             + C_malp*alpha_trim
229             + C_mde*elevator_trim)
230     +((rho*S*c*C_malp*u_trim)/(2*J_y))
231     +((rho*S*c*c*C_mq*q_trim*w_trim)
232                     /(4*J_y*Va_trim));
233
234   Mq  = ((rho*Va_trim*S*c*c*C_mq)/(4*J_y));
235   Mde = ((rho*Va_trim*Va_trim*S*c*C_mde)/(2*J_y));
236
237
238
239   //________________________________________
240   /* Position Differential Equations */
241
242   p_N_dot = cos(theta+theta_trim)
243             * cos(psi+psi_trim)*(u+u_trim)
244             + (sin(phi+phi_trim)
245             * sin(theta+theta_trim)*cos(psi+psi_trim)
246             - cos(phi+phi_trim)*sin(psi+psi_trim))
247             * (v+v_trim)
248             + (cos(phi+phi_trim)
249             * sin(theta+theta_trim)*cos(psi+psi_trim)
250             + sin(phi+phi_trim)
251             * sin(psi+psi_trim))*(w+w_trim);
252
253   p_E_dot = cos(theta+theta_trim)
254             * sin(psi+psi_trim)*(u+u_trim)
255             + (sin(phi+phi_trim)
256             * sin(theta+theta_trim)*sin(psi+psi_trim)
257             + cos(phi+phi_trim)
258             * cos(psi+psi_trim))*(v+v_trim)
```

```
259             + (cos(phi+phi_trim)
260             * sin(theta+theta_trim)*sin(psi+psi_trim)
261             - sin(phi+phi_trim)
262             * cos(psi+psi_trim))*(w+w_trim);
263
264   p_D_dot = -sin(theta)*u + sin(phi)*cos(theta)*v
265             + cos(phi)*cos(theta)*w;
266
267
268   f << dot(p_N) == p_N_dot;
269   f << dot(p_E) == p_E_dot;
270
271
272   //________________________________________
273   /* Lateral Differential Equations */
274
275   f << dot(v) == Yv*v + Yp*p + Yr*r
276               + g*cos(theta_trim)*cos(phi_trim)*phi
277               + Yda*aileron + Ydr*rudder;
278
279   f << dot(p) == Lv*v + Lp*p + Lr*r
280               + Lda*aileron + Ldr*rudder;
281
282   f << dot(r) == Nv*v + Np*p + Nr*r
283               + Nda*aileron + Ndr*rudder;
284
285   f << dot(phi) == p
286                  + cos(phi_trim)*tan(theta_trim)*r
287                  + (q_trim*cos(phi_trim)
288                           *tan(theta_trim)
289                  - r_trim*sin(phi_trim)
290                           *tan(theta_trim))*phi;
291
292   f << dot(psi) == cos(phi_trim)*(1/cos(theta_trim))*r
293                  + (p_trim*cos(phi_trim)
294                          *(1/cos(theta_trim))
295                  - r_trim*sin(phi_trim)
296                          *(1/cos(theta_trim)))*phi;
297
298
299
300   //________________________________________
301   /* Longitudinal Differential Equations */
302
303   f << dot(u) == Xu*u + Xw*w + Xq*q
304               - g*cos(theta_trim)*theta
```

```
305             + Xde*elevator + Xdt*throttle;
306
307   f << dot(w) == Zu*u + Zw*w + Zq*q
308             - g*sin(theta_trim)*theta
309             + Zde*elevator;
310
311   f << dot(q) == Mu*u + Mw*w + Mq*q + Mde*elevator;
312
313   f << dot(theta) == q;
314
315   f << dot(h) == sin(theta_trim)*u
316             - cos(theta_trim)*w
317             + (u_trim*cos(theta_trim)
318             + w_trim*sin(theta_trim))*theta;
319
320
321
322   //----------------------------------------
323   /* Control Differential Equation */
324
325   f << dot(elevator) == d_elevator;
326   f << dot(aileron)  == d_aileron;
327   f << dot(rudder)   == d_rudder;
328   f << dot(throttle) == d_throttle;
329
330
331
332   //----------------------------------------
333   /* Calculate Gamma and Chi */
334
335   GAMMA = -atan(p_D_dot/sqrt(p_N_dot*p_N_dot
336                            + p_E_dot*p_E_dot
337                            + p_D_dot*p_D_dot));
338   CHI = asin(p_E_dot/sqrt(p_N_dot*p_N_dot
339                         + p_E_dot*p_E_dot
340                         + p_D_dot*p_D_dot));
341
342
343
344   //----------------------------------------
345   /* Calculate Camera Position */
346
347   cx = p_N + (h+h_trim)*tan(theta+theta_trim)
348                         *cos(psi+psi_trim)
349            - (h+h_trim)*tan(-phi+phi_trim)
350                         *sin(psi+psi_trim);
```

```
351  cy = p_E + (h+h_trim)*tan(theta+theta_trim)
352                       *sin(psi+psi_trim)
353           + (h+h_trim)*tan(-phi+phi_trim)
354                       *cos(psi+psi_trim);
355
356
357  //----------------------------------------
358  /* Least Squares Problem */
359
360  Function trajectory;
361
362  trajectory << cx;
363  trajectory << cy;
364  trajectory << u;
365  trajectory << h;
366
367  trajectory << d_elevator;
368  trajectory << d_aileron;
369  trajectory << d_rudder;
370  trajectory << d_throttle;
371
372
373  DMatrix Q(8,8); Q.setIdentity();
374
375  Q(0,0) = 1e-1;   // p_N
376  Q(1,1) = 1e-1;   // p_E
377  Q(2,2) = 1e1;   // u
378  Q(3,3) = 1e1;   // h
379
380  Q(4,4) = 1e0;   // d_elevator
381  Q(5,5) = 1e-2;   // d_aileron
382  Q(6,6) = 1e6;   // d_rudder
383  Q(7,7) = 1e0;   // d_throttle
384
385
386  //----------------------------------------
387  /* Initialize Optimal Control Problem */
388  OCP ocp( path.getTimePoints() );
389
390  ocp.subjectTo( f );
391
392  ocp.minimizeLSQ( Q, trajectory, path );
393
394
395
396  //----------------------------------------
```

```
397  /* Start Configuration */
398
399  ocp.subjectTo( AT_START, p_N == X0(0) );
400  ocp.subjectTo( AT_START, p_E == X0(1) );
401  ocp.subjectTo( AT_START,   h == X0(2) );
402
403  ocp.subjectTo( AT_START, u == X0(3) );
404  ocp.subjectTo( AT_START, v == X0(4) );
405  ocp.subjectTo( AT_START, w == X0(5) );
406
407  ocp.subjectTo( AT_START, phi   == X0(6) );
408  ocp.subjectTo( AT_START, theta == X0(7) );
409  ocp.subjectTo( AT_START, psi   == X0(8) );
410
411  ocp.subjectTo( AT_START, p == X0(9)  );
412  ocp.subjectTo( AT_START, q == X0(10) );
413  ocp.subjectTo( AT_START, r == X0(11) );
414
415  ocp.subjectTo( AT_START, elevator == U0(0) );
416  ocp.subjectTo( AT_START,  aileron == U0(1) );
417  ocp.subjectTo( AT_START,   rudder == U0(2) );
418  ocp.subjectTo( AT_START, throttle == U0(3)  );
419
420  ocp.subjectTo( AT_START, d_elevator == DU0(0) );
421  ocp.subjectTo( AT_START, d_aileron  == DU0(1) );
422  ocp.subjectTo( AT_START, d_rudder   == DU0(2) );
423  ocp.subjectTo( AT_START, d_throttle == DU0(3) );
424
425
426
427  //_________________________________________
428  /* Constraints */
429
430  //ocp.subjectTo( 0 <= u );
431
432  ocp.subjectTo( -PI/2 <= theta <= PI/2 );
433  ocp.subjectTo( -PI/2 <= phi <= PI/2 );
434  ocp.subjectTo( -PI/6 <= elevator <= PI/6 );
435  ocp.subjectTo( -PI/6 <= aileron  <= PI/6 );
436  ocp.subjectTo( -PI/6 <= rudder   <= PI/6 );
437  ocp.subjectTo(     0 <= throttle <= 1 );
438
439  ocp.subjectTo( -0.2 <= d_elevator <= 0.2 );
440  ocp.subjectTo( -0.2 <= d_aileron  <= 0.2 );
441  ocp.subjectTo( -0.2 <= d_rudder   <= 0.2 );
442  ocp.subjectTo( -0.2 <= d_throttle <= 0.2 );
```

```
443
444
445
446     //-----------------------------------------
447     /* Configure Solver Algorithm */
448
449     OptimizationAlgorithm algorithm( ocp );
450
451     algorithm.set( KKT_TOLERANCE, 1e-4 );
452     algorithm.set( INTEGRATOR_TYPE, INT_RK78 );
453
454     algorithm.set(PRINT_COPYRIGHT, BT_FALSE);
455
456     algorithm.solve();
457
458
459
460     //-----------------------------------------
461     /* Prepare Solution */
462
463     VariablesGrid states, controls;
464     algorithm.getDifferentialStates(states);
465     algorithm.getControls(controls);
466
467     DMatrix ret_values(path.getLastTime(), 20);
468
469     for(int i = 0 ; i < path.getLastTime() ; i++){
470         for(int j = 0 ; j < 16 ; j++){
471             ret_values(i, j) = states(i+1, j);
472         }
473         ret_values(i, 16) = controls(i+1, 0);
474         ret_values(i, 17) = controls(i+1, 1);
475         ret_values(i, 18) = controls(i+1, 2);
476         ret_values(i, 19) = controls(i+1, 3);
477     }
478
479     return ret_values;
480
481 }
```

Appendix C

MPC Code

C.1 Algorithms

Algorithm 1 Offline Intervalwise MPC Algorithm

procedure MPC
path ← path from file
timestep ← duration of timestep [s]
horizonlen ← number of *timestep* in horizon
intervallen ← number of *timestep* in interval
intervals ← number of intervals needed to cover *path*
x_0 ← initial values of states
u_0 ← initial values of control states
Δu_0 ← inital values of control rates
results[] ← empty list to store result from optimization
for each *interval* **do**
$\mathbf{c}^n$ ← calculate camera centre position using equation 2.14
trajectory ← GENERATEHORIZON(*path, timestep, horizonlen,* $\mathbf{c}^n$)
Solve optimization with initial states x_0, u_0, Δu_0 for current *horizon*
x_0 ← last x value in the *interval*
u_0 ← last u value in the *interval*
Δu_0 ← last Δu value in the *interval*
result[] ← the first *intervallen* number of *timesteps* from *horizon*

Algorithm 2 Generate horizon

procedure GENERATEHORIZON(*path, timestep, horizonlen,* $\mathbf{c}^n$)
  distance ← distance travelled during one timestep
  pos ← find the point in *path* that is closes to current camera position $\mathbf{c}^n$
  trajectory[] ← empty list to store the generated trajectory
  for each *timestep* in *horizonlen* **do**
    Find point pos_{temp} on *path* with the given *distance* away from current *pos*
    trajectory[] ← pos_{temp}
    pos ← pos_{temp}
  return *trajectory*

C.2 Code

C.2.1 Offline Intervalwise MPC

```
1  #include <acado_optimal_control.hpp>
2
3  #include <iostream>
4  #include <fstream>
5  #include <sstream>
6  #include <cmath>
7  #include <ctime>
8
9  #include "mpc_script.hpp"
10 #include "path.hpp"
11
12 using namespace std;
13 USING_NAMESPACE_ACADO
14
15
16 int main(){
17
18   /* Read path from file */
19   int path_length = 15001;
20   ifstream file("./../path_curved2.txt");
21   double **path_data;
22
23   path_data = readPathFile(file, path_length);
24
25
26   /* Initialize MPC variables */
27   int horizon_length  = 14;
28   int interval_length = 5;
29   double timestep     = 0.1; // [s]
```

```
30  int no_intervals    = 10;
31
32
33  //_____________________________________________
34  /* Start Configuration */
35
36  DVector X0(12);
37  DVector U0(4);
38  DVector DU0(4);
39
40  X0(0)  =  0.0; // p_N
41  X0(1)  =  0.0; // p_E
42  X0(2)  =  0.0; // h
43  X0(3)  =  0.0; // u
44  X0(4)  =  0.0; // v
45  X0(5)  =  0.0; // w
46  X0(6)  =  0.0; // phi
47  X0(7)  =  0.0; // theta
48  X0(8)  =  0.0; // psi
49  X0(9)  =  0.0; // p
50  X0(10) =  0.0; // q
51  X0(11) =  0.0; // r
52
53  U0(0) = 0.0; // Elevator
54  U0(1) = 0.0; // Aileron
55  U0(2) = 0.0; // Rudder
56  U0(3) = 0.0; // Throttle
57
58  DU0(0) = 0.0; // D_Elevator
59  DU0(1) = 0.0; // D_Aileron
60  DU0(2) = 0.0; // D_Rudder
61  DU0(3) = 0.0; // D_Throttle
62
63
64  //_____________________________________________
65  /* Trim Conditions */
66
67  DVector X0_trim(12);
68  DVector U0_trim(4);
69  DVector DU0_trim(4);
70
71  X0_trim(0)  =  0.0;   // p_N
72  X0_trim(1)  =  0.0;   // p_E
73  X0_trim(2)  =150.0;   // h
74  X0_trim(3)  = 25.0;   // u
75  X0_trim(4)  =  0.0;   // v
```

```
76   X0_trim(5)  =  0.0;   // w
77   X0_trim(6)  =  0.0;   // phi
78   X0_trim(7)  =  0.066; // theta
79   X0_trim(8)  =  0.0;   // psi
80   X0_trim(9)  =  0.0;   // p
81   X0_trim(10) =  0.0;   // q
82   X0_trim(11) =  0.0;   // r
83
84   U0_trim(0) = -0.15; // Elevator
85   U0_trim(1) = 0.0;   // Aileron
86   U0_trim(2) = 0.0;   // Rudder
87   U0_trim(3) = 0.1;   // Throttle
88
89   DU0_trim(0) = 0.0; // D_Elevator
90   DU0_trim(1) = 0.0; // D_Aileron
91   DU0_trim(2) = 0.0; // D_Rudder
92   DU0_trim(3) = 0.0; // D_Throttle
93
94   //-----------------------------------------
95   /* Initialize result storage */
96
97   double ** result;
98   result = new double*[no_intervals*interval_length];
99   for(int i = 0; i<no_intervals*interval_length; i++){
100      result[i] = new double[21];
101  }
102  result[0][0] = 0.0;
103  for(int k = 1 ; k < 13 ; k++){
104      result[0][k] = X0(k-1) + X0_trim(k-1);
105  }
106  result[0][13] = U0(0) + U0_trim(0);
107  result[0][14] = U0(1) + U0_trim(1);
108  result[0][15] = U0(2) + U0_trim(2);
109  result[0][16] = U0(3) + U0_trim(3);
110
111  //-----------------------------------------
112  /* MPC Loop */
113  cout << "Starting␣MPC.\n";
114  for(int i = 0 ; i < no_intervals ; i++){
115      int index = findClosestPoint(X0(0),
116                                   X0(1),
117                                   X0(2)+X0_trim(2),
118                                   X0(6)+X0_trim(6),
119                                   X0(7)+X0_trim(7),
120                                   X0(8)+X0_trim(8),
121                                   path_data,
```

```
122                                     path_length);
123
124     double closest_x = path_data[index][0];
125     double closest_y = path_data[index][1];
126
127     VariablesGrid path = generateHorizon(
128                                 path_data,
129                                 timestep,
130                                 interval_length,
131                                 path_length,
132                                 closest_x,
133                                 closest_y);
134
135     DMatrix states = optimize_path(path, X0, U0, DU0);
136
137     for(int l = 0 ; l < 12 ; l++){
138       X0(l) = states(section_length-1, l);
139     }
140     U0(0)  = states(section_length-1, 12);
141     U0(1)  = states(section_length-1, 13);
142     U0(2)  = states(section_length-1, 14);
143     U0(3)  = states(section_length-1, 15);
144
145     DU0(0) = states(section_length-1, 16);
146     DU0(1) = states(section_length-1, 17);
147     DU0(2) = states(section_length-1, 18);
148     DU0(3) = states(section_length-1, 19);
149
150     for(int j = 0 ; j < section_length ; j++){
151       int idx = i*section_length + j;
152       result[idx][0] = idx*timestep;
153
154       for(int k = 1 ; k < 13 ; k++){
155         result[idx][k]  = states(j, k-1)
156                           + X0_trim(k-1);
157       }
158
159       result[idx][13]  = states(j, 12) + U0_trim(0);
160       result[idx][14]  = states(j, 13) + U0_trim(1);
161       result[idx][15]  = states(j, 14) + U0_trim(2);
162       result[idx][16]  = states(j, 15) + U0_trim(3);
163
164       result[idx][17]  = states(j, 16);
165       result[idx][18]  = states(j, 17);
166       result[idx][19]  = states(j, 18);
167       result[idx][20]  = states(j, 19);
```

```
168
169     }
170
171     clearAllStaticCounters();
172   }
173   saveResults(result, no_sections*section_length,
174                             path_data, path_length);
175
176
177   //-----------------------------------------
178   /* Delete path array */
179   for( int i = 0 ; i < path_length ; i++){
180       delete [] path_data[i];
181   }
182   delete [] path_data;
183
184   return 0;
185 }
```

C.2.2 Generate Horizon

```
1  VariablesGrid generateHorizon(double** path_data,
2                                double timestep,
3                                int horizon_length,
4                                int path_length,
5                                double x_start,
6                                double y_start){
7
8    /* Initalize storage */
9    int no_timesteps = horizon_length/timestep;
10   VariablesGrid path(8, 0, horizon_length,
11                                    no_timesteps+1);
12
13   DVector points(8);
14
15   /* Initialize variables */
16   double  x =  x_start;
17   double  y =  y_start;
18
19   double speed = 25.0;
20   double distance = speed*timestep; // [m]
21
22   points(0) = x;    // p_N
23   points(1) = y;    // p_E
```

```
24    points(2) = 0.0; // Speed (u)
25    points(3) = 0.0; // h
26    points(4) = 0.0; // d_elevator
27    points(5) = 0.0; // d_aileron
28    points(6) = 0.0; // d_rudder
29    points(7) = 0.0; // d_throttle
30    path.setVector(0, points);
31
32    /* Generate path */
33    int point_found = 0;
34    for( int step = 0 ; step < no_timesteps ; step++ ){
35        for( int i = path_length-1 ; i >= 0 ; --i ){
36            double x_dist = path_data[i][0] - x;
37            double y_dist = path_data[i][1] - y;
38            double radius = sqrt(x_dist*x_dist
39                                 + y_dist*y_dist);
40
41            if( (radius > distance-0.08)
42                        && (radius < distance+0.08)){
43
44              x = path_data[i][0];
45              y = path_data[i][1];
46
47              points(0) = path_data[i][0];
48              points(1) = path_data[i][1];
49
50              path.setVector(step+1, points);
51              point_found = 1;
52              break;
53            }
54        }
55        if(!point_found){
56          cout << "Could not find a point!\n";
57          throw 1;
58        }
59    }
60    return path;
61  }
```

Bibliography

[1] W. H Kwon and S. H. Han. *Receding Horizon Control: Model Predictive Control for State Models*. Advanced Textbooks in Control and Signal Processing. Springer London, 2005.

[2] E. Skjong, S. A. Nundal, F. S. Leira, and T. A. Johansen. 2015 international conference on unmanned aircraft systems (ICUAS). In *Autonomous Search and Tracking of Objects Using Model Predictive Control of Unmanned Aerial Vehicle and Gimbal: Hardware-in-the-loop Simulation of Payload and Avionics*, Denver, Colorado, USA, June 2015. IEEE.

[3] J. Egbert and R. W. Beard. Proceedings of the 2007 American control conference. In *Low Altitude Road Following Constraints Using Strap-Down EO Cameras on Miniature Air Vehicles*, New York City, USA, July 2007. IEEE.

[4] Thomas M. Fisher. Rudder augmented trajectory correction for unmanned aerial vehicles to decrease lateral image errors of fixed camera payloads. *All Graduate Theses and Dissertations*, 2016.

[5] S. Mills, J. J. Ford, and L. Mejias. Vision based control for fixed wing UAVs inspecting locally linear infrastructure using skid-to-turn maneuvers. *Journal of Intelligent and Robotic Systems*, 61(1):29–42, 2011.

[6] M. Ahsan, H. Rafique, and Z. Abbas. Multitopic conference (INMIC). In *Heading Control of a Fixed Wing UAV Using Alternate Control Surfaces*. IEEE, December 2012.

[7] Stephen P. Jackson. Controlling small fixed wing UAVs to optimize image quality from on-board cameras. *ProQuest Dissertations and Theses*, 2011.

[8] Randall B. Smith. *Introduction to Hyperspectral Imaging*. MicroImages, Inc., 2012.

[9] R. Nsi, E. Honkavaara, P. Lyytikinen-Saarenmaa, M. Blomqvist, P. Litkey, T. Hakala, N. Viljanen, T. Kantola, T. Tanhuanp, and M. Holopainen. Using

UAV-based photogrammetry and hyperspectral imaging for mapping bark beetle damage at tree-level. *Remote Sensing*, 7(15467-15493), 2015.

[10] P. J. Zarco-Tejada, V. Gonzlez-Dugo, and J. A. J. Berni. Fluorescence, temperature and narrow-band indices acquired from a UAV platform for water stress detection using a micro-hyperspectral imager and a thermal camera. *Remote Sensing of Environment*, 117(322-337), 2012.

[11] J. Suomalainen, N. Anders, S. Iqbal, G. Roerink, J. Franke, P. Wenting, D. Hnniger, H. Bartholomeus, R. Becker, and L. Kooistra. A lightweight hyperspectral mapping system and photogrammetric processing chain for unmanned aerial vehicles. *Remote Sensing*, 6(11013-11030), 2014.

[12] Thor I. Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley & Sons, Ltd, 2011.

[13] Guowei Cai, Ben M. Chen, and Tong Heng Lee. *Unmanned Rotorcraft Systems*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[14] Randal W. Beard and Timothy W. McLain. *Small Unmanned Aircraft: Theory and Practice*. Princeton University Press, Princeton, NJ, USA, 2012.

[15] O. Egeland and J. T. Gravdahl. *Modeling and Simulation for Automatic Control*. Marine Cybernetics, Trondheim, Norway, 2002.

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

[17] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006.

[18] M. Diehl, H.G. Bock, H. Diedam, and P.-B. Wieber. *Fast Direct Multiple Shooting Algorithms for Optimal Robot Control*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[19] S. H. Mathisen, T. I. Fossen, and T. A. Johansen. Non-linear model predictive control for guidance of a fixed-wing uav in precision deep stall landing. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 356–365, June 2015.

[20] J. B. Rawlings and D. Q. Mayne. *Model Predictive Control: Theory and Design*. Nob Hill Publishing, Madison, Wisconsin, 2015.

[21] B. Houska, H.J. Ferreau, and M. Diehl. ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[22] TD Bui and TR Bui. Numerical methods for extremely stiff systems of ordinary differential equations. *Applied Mathematical Modelling*, 3(5):355–358, 1979.

[23] Facculty of Engineering University of Porto. LSTS: DUNE Unified Navigation Environment. http://www.lsts.pt/toolchain/dune. Accessed: 23.05.2017.

[24] ArduPilot. http://ardupilot.org/. Accessed: 23.05.2017.

[25] JSBSim. http://jsbsim.sourceforge.net/index.html. Accessed: 23.05.2017.

[26] Facculty of Engineering University of Porto. Neptus. http://www.lsts.pt/toolchain/neptus. Accessed: 23.05.2017.

[27] K. Gryte and T. I. Fossen. *High Angle of Attack Landing of an Unmanned Aerial Vehicle*. Norges Teknisk-Naturvitenskapelige Universitet, Trondheim, Norway, 2015.

[28] L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497–516, 1957.