MPC for path-following

TTK4551 Specialization Project

Report 2020

Thomas Leirfall

Trondheim, 22/12/2020

NTNU In Trondheim
PROJECT ASSIGNMENT

NTNU
Knowledge for a better world

Tittel:

MPC for path-following

Candidate number (name):
Thomas Leirfall

Date:	Code:	Subject:	Document access:
22/12/2020	TTK4551	Specialization Project	Public

Studium:	Nr. pages/attachments:	Bibl. nr:
Cybernetics and Robotics	73 / 6	

Supervisor:
Dirk Reinhardt
Tor Arne Johansen

Table of contents

SUMMARY

In my thesis, I will use theory and methods on optimization, airplane dynamics and simulations to develop a controller for fixed-wing aerial vehicles. Through the thesis, I will focus on subjects I have acquired during the cybernetics and robotics courses at NTNU Trondheim.

model predictiv controller (MPC) can be used to follow geometrically challenging curves, and at the same time performing optimal with respect to user-defined cost function and constraints is that of (nonlinear) model predictive controllers (MPCs). Given a set of waypoints for the MPC to follow, there are different ways to frame a path that the vehicle can follow in order to pass them. However, their design has an impact on how well the MPC converges to a solution of the underlying optimization problem.

The thesis is made up of 8 chapters where the project plan is formulated in chapter 1 which is then answered in chapter 4. Theory, methods, results, and discussions are described in the following sections:

- In section 1 is an introduction to the thesis.
- In section 2 is the notation and abbreviations.
- In section 3 presents relevant theory divided into 18 different subsections.
- In section 4 the control algorithm are described. Here the path-following algorithms for straight lines are presented along with the successive loops controlling the airplane. Also, the optimal control problem is chosen for the Nonlinear model predictiv controller (NMPC).
- In section 5 the results of the controller are presented.
- In section 6 is a discussion on how the controllers did.
- Section 7 concludes in relation to the methods and solutions chosen on the optimal control problem presented in section 4 to follow the line and further work needed.
- In the Appendix the programs are included.

1 INTRODUCTION

1.1 Background

This report is a preparation for my master's thesis in the field of cybernetics and robotics at NTNU Trondheim. The thesis is about model predictive control for path-following control of autonomous fixed-wing aerial vehicles.

1.2 Project plan

In this project the following plan is defined

1. Learning Ubuntu.
2. Familarize yourself with casadi and acados, which we use to implement the MPCs.
3. Implement the path-following algorithms for straight lines.
4. Implement a switching logic and follow a rectangular pattern.
5. Implement a model predictive path-following control (MPFC) (straight lines in the lateral/longitudinal plane).
6. Implement a MPFC (for following straight lines in $\mathcal{R}^3$).

1.3 Limitations

This thesis is limited to only computer simulation with approximated models. Furhter on, program libraries for PID controllers as well as libraries for aircraft dynamics are used for the simulation, and acados are used for solving the NMPC.

1.4 Structure of the thesis

The thesis is divided into 8 chapters. Starting with focus on theory and methods that are important for the concept and the solutions used are presented in Attachments.

2 NOTATION

Acronyms

DAE Differential algebraic equation. 41

KKT Karush–Kuhn–Tucker. 26

MPC model predictiv controller. 5, 26, 32

MPFC model predictive path-following control. 6, 41, 50, 52

NED north east down. 9, 10, 14, 16, 17

NMPC Nonlinear model predictiv controller. 5, 6, 27, 41

OCP optimal control problem. 27, 28, 31, 37, 38, 41, 46, 48

ODE Ordinary differential equation. 41

SSA Smallest sign angle. 25

UAV Unmanned aerial vehicle. 23, 50, 52

$\mathcal{F}^i$	NED frame
$\mathcal{F}^v$	Vehicle frame
$\mathcal{F}^{v1}$	Vehicle-1 frame
$\mathcal{F}^{v2}$	Vehicle-2 frame
$\mathcal{F}^b$	Body frame
$\mathcal{F}^s$	Stability frame
$\mathcal{F}^w$	Wind frame
ϕ	Roll
θ	Pitch
ψ	Yaw
Θ	Euler angles $[\phi \quad \theta \quad \psi]^T$
α	Angle of attack
β	Side-slip
$S(\cdot)$	Skew symmetric matrix
δ_a	Aileron
δ_e	Elevation
δ_r	Rudder
δ_t	Throttle
$Q \succeq 0$	Positive definite matrix Q
$\|x\|^2$	2-norm of vector
$\|x\|^2_Q$	$x^T Q x$
n_x	Number of state
n_u	Number of input
n_y	Number of output
w_i	Waypoint index i
θ	Path parameter
v	Virtual input

3 THEORY

In this section I am going trough some basic concepts need to understand the airplane theory. First, we are going to look some important angles on the plane, and then some reference planes and coordinate systems.

3.1 Frames

There is some important coordinate system for the plane we need to look at. The following flow chart can be defined for the plane. The orange is the frames, the white is either translation or angles between the frames. Also see figure 2 to 7.

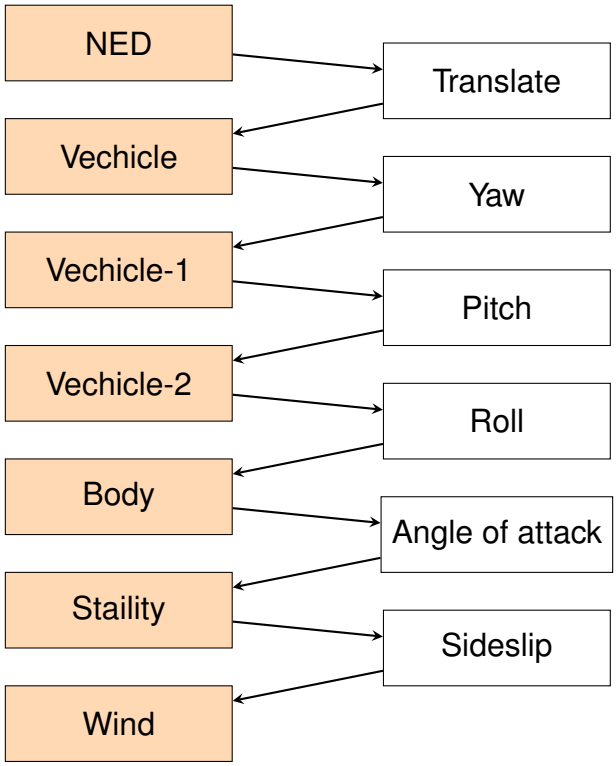


Figure 1: An overview of frames, angles and translation.

3.1.1 NED

north east down (NED) means North-East-Down, where this is the direction the plane is pointing. So, we have x pointing to the north, y to the east, and z down. Note that we say that downwards is positive. This frame is on the curve of the earth and is a tangential frame. We assume that this frame is an inertial frame, which is where the forces occur

We denote the NED frame as $\mathcal{F}^i$. Also denote i, j and k to be x-,y- and z-axis, respectively.

3.1.2 Vehicle

This frame is located at the plane, and does not rotate. Meaning that this has the same orientation as the NED-frame. This is seen in figure 2.

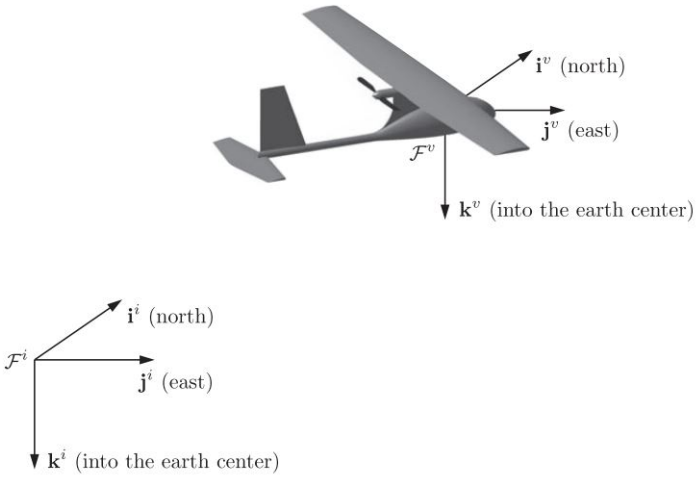


Figure 2: NED and Vehicle frame [1]

The rest of the frames has the same origo as the vehicle frame, there is only rotation difference.

3.1.3 Vehicle-1

Going from vehicle to vehicle-1 we rotate around the $\mathcal{F}^i$ z-axis. We denote this as around j^i. This is the plane's yaw angle, and we denote the yaw angle as ψ. This is seen in figure 3, where a positive direction is shown. We denote this frame as $\mathcal{F}^v$.

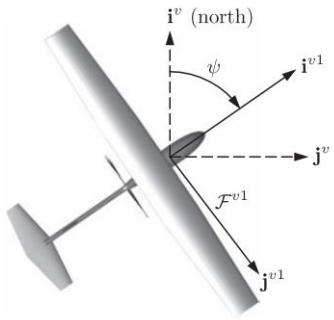


Figure 3: Vehicle-1 frame [1]

3.1.4 Vehicle-2

Next rotation is around j^v, y-axis of $\mathcal{F}^{v1}$. This is the plans pitch angle, and we denote this as θ. See figure 4, where positive direction is shown.

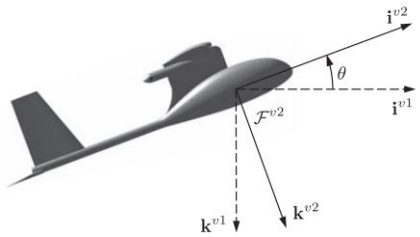


Figure 4: Vehicle-2 frame [1]

3.1.5 Body

The last plane configuration for rotation is the roll or bank angle. This is around the x-axis of $\mathcal{F}^{v2}$. We denote the roll angle as ϕ. The body frame is denoted $\mathcal{F}^b$. This is seen in figure 3, where a positive direction is shown.

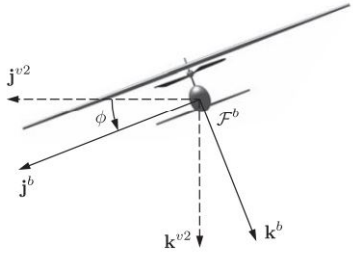


Figure 5: Body frame [1]

We denote the euler angles as

$$\Theta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \tag{1}$$

The total rotation matrix from Vehicle to Body is

$$R_v^b(\Theta) = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}}_{\text{Around } i^{v2}} \underbrace{\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}}_{\text{Around } j^{v1}} \underbrace{\begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Around } k^{v}} \tag{2}$$

Note that this has a singularity if the pitch angle is $\pm$ 90°. We could therefore use quaternion instead, which do not give us singularity.

3.1.6 Stability and wind

The stability- and wind frame is used for calculation purposes. We use the stability frame to calculate the decoupled lateral forces. While we define the speed over ground in the wind frame.

To understand these two frames, we need to look at some features first. Starting with the angle of attack, denoted α. This is the angle we rotate around j^b. If we have a positive angle of attack, the wings will create a lift and the plane will rise. In other words, it is the airspeed, denoted as $\mathbf{V_a}$, that hits the wings to create lift. Note that V_a is the magnitude of the relative velocity vector, see equation 13. If we look at figure 6 we can see how the stability frame is denoted.

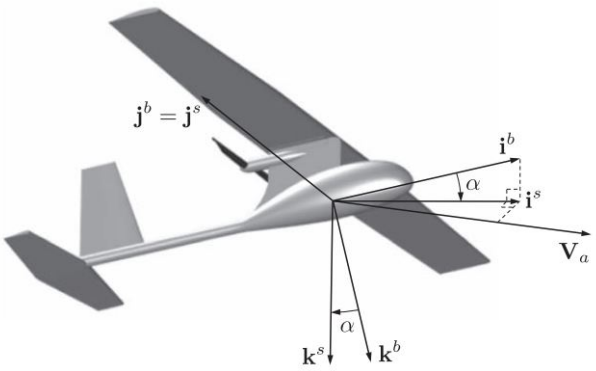


Figure 6: Stability frame [1]

Going from stability to wind frame, we rotate around k^s with an angle called β. This is the sideslip angle. This can be seen as an angle the plane is *sliding* out of straight forward along the i^b direction.

After this rotation, we see that now the i^w is aligned with $\mathbf{V_a}$. This means that we can denote the matrix for $\mathbf{V_a}$ as

$$\mathbf{V_a^w} = \begin{bmatrix} V_a \\ 0 \\ 0 \end{bmatrix} \tag{3}$$

With just one element in the x-direction. This is trivial. The frame can be seen in figure 7.

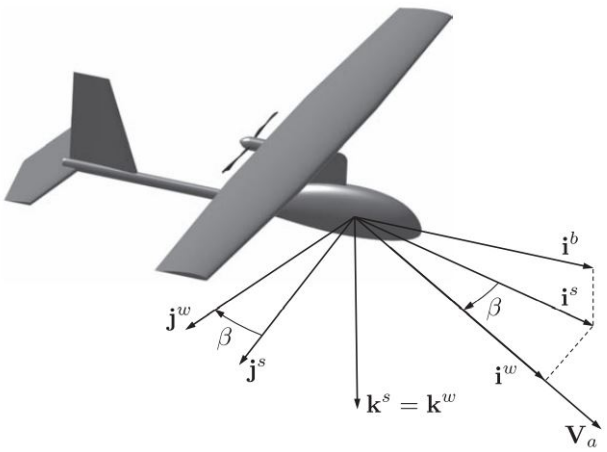


Figure 7: Wind frame [1]

The two rotation matrices can be defined as

$$R_b^s(\alpha) = \underbrace{\begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}}_{\text{Around } j^b} \tag{4}$$

$$R_s^w(\beta) = \underbrace{\begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Around } k^s} \tag{5}$$

3.2 Wind

There are two types of wind: the steady ambient wind in NED frame and stochastic (gust) wind which is defined body frame. The wind vector is defined as, $\mathbf{V_w^b}$ denoted in the body frame.

$$\mathbf{V_w^b} = \begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} \tag{6}$$

$$\mathbf{V_w^b} = R_v^b(\Theta) \begin{bmatrix} w_{n_s} \\ w_{e_s} \\ w_{d_s} \end{bmatrix} + \begin{bmatrix} w_{n_g} \\ w_{e_g} \\ w_{d_g} \end{bmatrix} \tag{7}$$

The way to get values for gust wind is to filter white noise through a Dryden transfer function. Here the user can adjust for altitude (height of the airplane) and strength of turbulence. The steady ambient wind is constant.

3.3 Air- and groundspeed

Airspeed is denoted as $\mathbf{V_a}$, with wind, $\mathbf{V_w}$, and the groundspeed, $\mathbf{V_g}$. The relationship between this speed is

$$\mathbf{V_a} = \mathbf{V_g} - \mathbf{V_w} \tag{8}$$

The groundspeed vector is defined in the body frame

$$\mathbf{V_g^b} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{9}$$

which are states in the system.

In equation 3, the airspeed is denoted in the wind frame, rotating this to body gives

$$\mathbf{V_a^b} = \begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = R_w^b(\alpha, \beta)\mathbf{V_a^w} \tag{10}$$

where u_r, v_r and w_r as the relative speed in the body frame. The roation from body to wind is

$$R_b^w(\alpha, \beta) = R_s^w(\beta)R_b^s(\alpha) \tag{11}$$

$$V_g = V_a + V_w \tag{12}$$

The relationship gives

$$\begin{aligned} V_a &= \sqrt{u_r^2 + v_r^2 + w_r^2} \\ \alpha &= \arctan \frac{w_r}{u_r} \\ \beta &= \arcsin \frac{v_r}{V_a} \end{aligned} \tag{13}$$

In figure 8 we can see airspeed, ground speed, and wind speed.

3.4 Course and heading angle

Both course and heading angle can describe the direction of the aircraft. The yaw angle ψ is the heading angle.

The course angle, χ, which is much used in the navigation of airplanes, can be seen in figure 8, and is the angle going from the north (y-axis of NED) and to ground speed.

The reason why the course is typical used instead of yaw is that we usually use speed sensors on a plane or GPS, and we can then measure the ground speed of the aircraft. Velocity is usually estimated. While for boats, we could use a compass (which shows the north) and then get the yaw angle.

The last angle is the crab angle χ_c. This is defined as:

$$\chi_c = \chi - \psi \tag{14}$$

And is the aircraft crab into the wind.

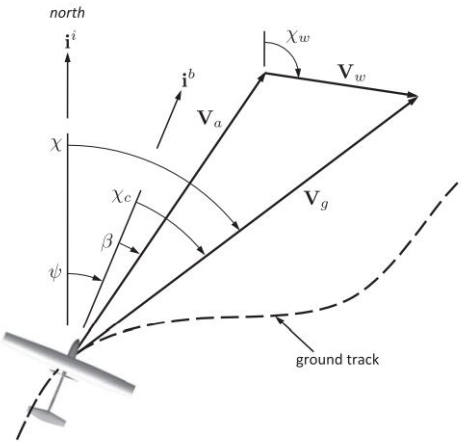


Figure 8: Wind triangle [1]

3.5 Flight path angle

This is in the vertical plane. Earlier seen the pitch angle θ and angle of attack α. The flight path angle γ is the angle between the horizontal plane, which is the i^{v1} (same as i^v) and the ground speed. The last angle is the air-mass-reference flight path angle γ_a, and is the angle from horizontal to airspeed. This can be seen in figure 9.

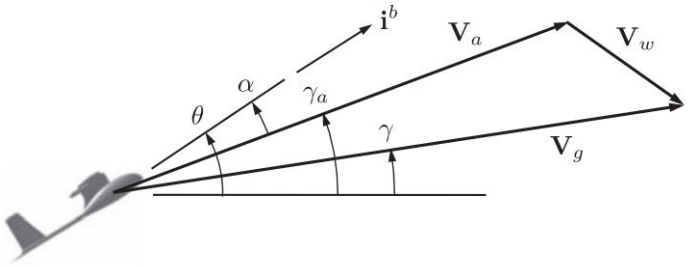


Figure 9: Wind triangle vertical plane [1]

3.6 Dynamics

Here the states of the aircraft is defined the states of the aircraft, see table 1.

Name	Description
p_n	Postion along i^i in NED
p_e	Postion along i^i in NED
p_d	Postion along k^i in NED
u	Velocity along i^b in body
v	Velocity along j^b in body
w	Velocity along k^b in body
ϕ	Roll angle defined with respect to vehicle 2
θ	Pitch angle defined with respect to vehicle 1
ψ	Heading (yaw) angle defined with respect to vehicle
p	Roll rate measured along i^b in body
q	Pitch rate measured along j^b in body
r	Yaw rate measured along k^b in body

Table 1: States

3.6.1 Translation

The first 6 states are the translation states. This is $[p_n \quad p_e \quad p_d \quad u \quad v \quad w]^T$.

All the states need to be defined in body, however, position is defined in NED.

$[u \quad v \quad w]^T$ is in vehicle frame. Time differential of the position gives:

$$\frac{d}{dt}\begin{bmatrix} p_n \\ p_e \\ p_d \end{bmatrix} = R_b^v \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{15}$$

The properties of the rotation matrix give that the inverse is the same as the transpose matrix. The expression for R_v^b, see equation 2. Then

$$\begin{bmatrix} \dot{p_n} \\ \dot{p_d} \\ \dot{p_d} \end{bmatrix} = (R_v^b)^T \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{16}$$

3.6.2 Rotation

The total rotation from vehicle to body is a total of three rotations: yaw, pitch and roll. This is the equation 2, again here:

$$R_v^b(\Theta) = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}}_{R_{v2}^b(\phi)} \underbrace{\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}}_{R_{v1}^{v2}(\theta)} \underbrace{\begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{R_v^{v1}(\psi)} \tag{17}$$

Example 3.1. *Differentiate a rotation matrix.*

To find the time differention of a rotation matrix, we have that

$$\frac{d}{dt}R(t) = S(w(t))R(t) \tag{18}$$

Where, R is a roation matrix and $S(\cdot)$ is a skew symetric matrix.
Starting with rotation matrix R_v^b

$$\begin{aligned}\frac{d}{dt}R_v^b =& \frac{d}{dt}[R_{v2}^b(\phi)]R_{v1}^{v2}(\theta)R_v^{v1}(\psi) + R_{v2}^b(\phi)\frac{d}{dt}[R_{v1}^{v2}(\theta)]R_v^{v1}(\psi) + R_{v2}^b(\phi)R_{v1}^{v2}(\theta)\frac{d}{dt}[R_v^{v1}(\psi)] \\ =& S(\dot{\phi}\mathbf{i})R_v^b(\Theta) + S(R_{v2}^b(\phi)\dot{\theta}\mathbf{j})R_v^b(\Theta) + S(R_{v2}^b(\phi)R_{v1}^{v2}(\theta)\dot{\psi}\mathbf{k})R_v^b(\Theta) \\ =& [S(\dot{\phi}\mathbf{i}) + S(R_{v2}^b(\phi)\dot{\theta}\mathbf{j}) + S(R_{v2}^b(\phi)R_{v1}^{v2}(\theta)\dot{\psi}\mathbf{k})]R_v^b(\Theta)\end{aligned}$$

Here we have utilize that fact that $RS(a)R^T = S(Ra)$. Using equation 18 we have that the rotation velocity is given by:

$$\omega = \dot{\phi}\mathbf{i} + R_{v2}^b(\phi)\dot{\theta}\mathbf{j} + R_{v2}^b(\phi)R_{v1}^{v2}(\theta)\dot{\psi}\mathbf{k}$$

Where

$$\begin{aligned}\mathbf{i} =& [1 \quad 0 \quad 0]^T \\ \mathbf{j} =& [0 \quad 1 \quad 0]^T \\ \mathbf{k} =& [0 \quad 0 \quad 1]^T\end{aligned} \tag{19}$$

This represent the rotation we rotate, so when we say that i is in the first part, this is because this rotation matrix is around x. So j is around y and k is around z.
This gives:

$$\omega = \begin{bmatrix}\dot{\phi}\\0\\0\end{bmatrix} + R_{v2}^b(\phi)\begin{bmatrix}0\\\dot{\theta}\\0\end{bmatrix} + R_{v2}^b(\phi)R_{v1}^{v2}(\theta)\begin{bmatrix}0\\0\\\dot{\psi}\end{bmatrix}$$

Continue from the example, by some rearranging, and defined that $\omega = [p \quad q \quad r]^T$ then:

$$\begin{bmatrix}p\\q\\r\end{bmatrix} = \begin{bmatrix}1 & 0 & -\sin\theta\\0 & \cos\phi & \sin\phi\cos\theta\\0 & -\sin\phi & \cos\phi\cos\theta\end{bmatrix}\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix} \tag{20}$$

Gives the state explicit

$$\begin{bmatrix}\dot{\phi}\\\dot{\theta}\\\dot{\psi}\end{bmatrix} = \begin{bmatrix}1 & \sin\phi\tan\theta & \cos\phi\tan\theta\\0 & \cos\phi & -\sin\phi\\0 & -\sin\phi\sec\theta & \cos\phi\sec\theta\end{bmatrix}\begin{bmatrix}p\\q\\r\end{bmatrix} \tag{21}$$

3.7 Rigid-body Dynamics

Starting with the typical newtons second law,

$$\begin{aligned} \sum F =& ma \\ \sum F =& m\frac{d\mathbf{V}_g}{dt_i} \end{aligned} \tag{22}$$

The time differentiation of the ground speed vector is the acceleration.

Example 3.2. *Time differentiation a vector:*
Given a vector expressed in body, where i,j,k is defined in equation 19

$$\mathbf{p} = p_x\mathbf{i}^b + p_y\mathbf{j}^b + p_z\mathbf{k}^b \tag{23}$$

Then the time differentiation in the inertia frame, i, is

$$\frac{d}{dt_i}\mathbf{p} = \frac{d}{dt_b}\mathbf{p} + \omega_{b/i} \times \mathbf{p} \tag{24}$$

This gives for equation 22, that:

$$\begin{aligned} \sum F =& m(\frac{d}{dt_b}\mathbf{V}_g + \omega_{b/i} \times \mathbf{V}_g) \\ \sum F^b =& m(\frac{d}{dt_b}\mathbf{V}_g^b + \omega_{b/i}^b \times \mathbf{V}_g) \end{aligned} \tag{25}$$

Write that $\mathbf{V}_g^b = [u \quad v \quad w]^T$ and $\omega_{b/i}^b = [p \quad q \quad r]^T$, and get that:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m}\sum F^b \tag{26}$$

Next is to find the angular acceleration. Finding the sum of moments

$$\begin{aligned} \sum M =& I\alpha \\ \sum M =& \frac{d\mathbf{h}}{dt_i} \end{aligned} \tag{27}$$

Where M is total moments, I is interia of moments and h is the angular momentum. As for forces can write out the $\mathbf{h}$ to be:

$$\frac{h}{dt_i}\mathbf{h} = \frac{h}{dt_b}\mathbf{h} + \omega_{b/i} \times \mathbf{h} \tag{28}$$

Insert this into equation 27 gives

$$\begin{aligned}\sum M &= (\frac{d}{dt_b}\mathbf{h} + \omega_{b/i} \times \mathbf{h}) \\ \sum M^b &= (\frac{d}{dt_b}\mathbf{h}^b + \omega^b_{b/i} \times \mathbf{h}^b)\end{aligned} \tag{29}$$

Where $\mathbf{h}^b = \mathbf{J}\omega^b_{b/i}$ is defined, where J is the interia of moments. Insert this gives

$$\sum M^b = \frac{d}{dt_b}\mathbf{J}\omega^b_{b/i} + \omega^b_{b/i} \times \mathbf{J}\omega^b_{b/i} \tag{30}$$

With the interia matrix constant, can write

$$\sum M^b = \mathbf{J}\frac{d}{dt_b}\omega^b_{b/i} + \omega^b_{b/i} \times \mathbf{J}\omega^b_{b/i} \tag{31}$$

Can get the states from $\frac{d}{dt_b}\omega^b_{b/i} = [\dot{p} \quad \dot{q} \quad \dot{r}]^T$, this gives

$$\sum M^b = \mathbf{J}\begin{bmatrix}\dot{p} \\ \dot{q} \\ \dot{r}\end{bmatrix} + \omega^b_{b/i} \times \mathbf{J}\omega^b_{b/i} \tag{32}$$

Rearranging, then the state is explicit

$$\begin{bmatrix}\dot{p} \\ \dot{q} \\ \dot{r}\end{bmatrix} = \mathbf{J}^{-1}\left(\sum M^b - \omega^b_{b/i} \times \mathbf{J}\omega^b_{b/i}\right) \tag{33}$$

Since the aircraft is assumed symmetric, the inertia matrix $\mathbf{J}$ can be written as, with $J_{XY} = J_{YZ} = 0$

$$\mathbf{J} = \begin{bmatrix} J_X & 0 & -J_{XZ} \\ 0 & J_Y & 0 \\ -J_{XZ} & 0 & J_Z \end{bmatrix} \tag{34}$$

3.8 Forces and Moments

From the states equation 26 and 33 we defined the sum of forces and mometns to be:

$$\begin{aligned}\sum F^b &= \mathbf{f}_g + \mathbf{f}_a + \mathbf{f}_p \\ \sum M^b &= \mathbf{m}_a + \mathbf{m}_p\end{aligned} \tag{35}$$

Here the subscript g is gravity, a is aerodynamics and p is due to propulsion. Superscript b is body. The aerodynamics forces are divided into lift and drag, which again give a moment. Propulsion forces are given from example propeller thrust with a propeller torque which give a moment. This thesis will not go deeper into these forces and moments.

3.9 Control surfaces

To control the aircraft we have what we call control surfaces. Which we use to control the air around the aircraft to pitch, roll or yaw the aircraft. There is

- Aileron
- Elevator
- Tail rudder
- Throttle

Aileron
This is used to control the roll of the flight and it is on the wings of the plain. Meaning, to turn the plain. Usually, the ailerons on the two wings move in the opposite direction, which makes sense so the plain roll. If the aileron moves in the same direction, we could use this as an elevation.

Elevation
Controls the pitch of aircraft. Moves the tail wing with an angle to make the noise go up or down.

Rudder
On the tail of the aircraft. Makes the plane yaw.

Throttle
We also have the throttle which controls the aircraft's speed

There is a lot of different ways these control surfaces can be designed, but in figure 10 we see an example. We denote

Symbol	Name
δ_a	Aileron
δ_e	Elevation
δ_r	Rudder
δ_t	Throttle

Table 2: Control surfaces

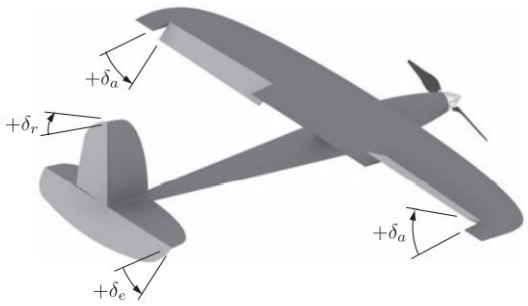


Figure 10: Control surfaces [1]

3.10 Successive loop

Here the goal is to close a loop inside another loop. When the inner loop is closed, this inner loop must be faster than the outer loop. With a ratio of 5-10 times faster. The concept is that we want the inner loop to be stabilized, meaning finishing so that we can approximate this inner loop to a gain constant of 1 [1].

In, for example, a bank to turn, the successive loop is used. Here the inner loop is the control of the roll, PD controller, see equation 36. The outer loop is the course control, PI controller see equation 37.

$$\delta_a = k_{p_\psi}(\psi^c - \psi) - k_{d_\psi}\dot{\psi} \tag{36}$$

$$\phi^c = k_{p_\chi}(\chi^c - \chi) + \frac{k_{i_\chi}}{s}(\chi^c - \chi) \tag{37}$$

The inner loop is the transfer function from roll to the aileron. This loop is closed, with a fast enough bandwidth, so that $\phi^c = \phi$. Then the outer loop from course to roll, threat the inner loop as a gain constant of 1.

It is not wanted to have an integral effect in the inner loop, because having an integral effect inside a loop can cause a problem with bandwidth. This is because an integral is time-consuming, and this can slow down the inner loop. This means that the inner loop no longer has the gain of 1. This could mean that $\phi^c \neq \phi$, and we chasing our tail. The integral effect on the outer loop can correct the steady-state error in both loops.

3.11 Straight-line Path Following

In Beard and McLain [1] a straight-line path following is described as

$$\mathcal{P}(\mathbf{r}, \mathbf{q}) = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{x} = \mathbf{r} + \lambda \mathbf{q}, \lambda \in \mathbb{R}\} \tag{38}$$

Since this is a path following, the position of the Unmanned aerial vehicle (UAV) along the path, and only consider the error in y, the cross-track error, to be driven to zero. The cross-track error is defined as

$$e_{py} = -\sin(\chi_q)(p_n - r_n) + \cos(\chi_q)(p_e - r_e) \tag{39}$$

where this is trigonometry from figure 11

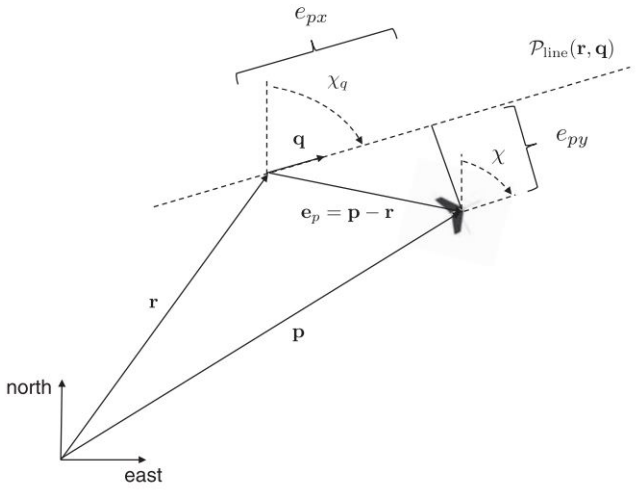


Figure 11: North East plane [1]

Beard and McLain[1] prove stability, when driving the cross track error to zero, with the following controller

$$\chi_d = x_q - \chi^{\infty} \frac{2}{\pi} \arctan(k_{path} e_{py}) \tag{40}$$

Here the χ^{∞} is the commanded coruse angle when the aircraft is far from the line, see figure 12a . The gain k_{path} is a parameter to tune how fast the sigmoid function, $\arctan(\cdot)$, curve around zero, see figure 12b.

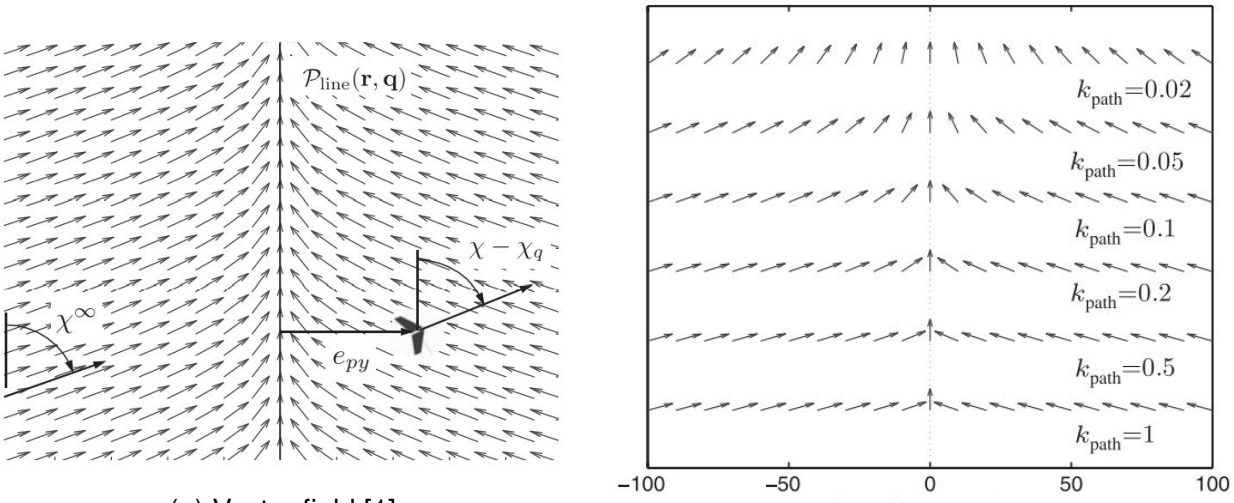
The correction of course, χ_q is defined as

$$\chi_q = atan2(q_e, q_n) \tag{41}$$

Where the $\mathbf{q}$ vector is defined as

$$\mathbf{q} = \mathbf{w}_{i+1} - \mathbf{w}_i \tag{42}$$

where $\mathbf{w}$ is waypoints.



(a) Vector field [1]

(b) k_{path} [1]

Figure 12: Paramters

To get the commanded height, the trigonometry from figure 13b is used. Figure 13b is a representation from where the $\mathbf{q}$ and $\mathbf{k}$ axis is merge. Then

$$\frac{-s_d}{\sqrt{s_n^2 + s_e^2}} = \frac{-q_d}{\sqrt{q_n^2 + q_e^2}} \tag{43}$$

Rearrange this equation to

$$s_d = -\sqrt{s_n^2 + s_e^2}\frac{q_d}{\sqrt{q_n^2 + q_e^2}} \tag{44}$$

Where the s vector is

$$\mathbf{s}^i = \mathbf{e}_i^p - (\mathbf{e}_i^p \mathbf{n})\mathbf{n} \tag{45}$$

Were the normal vector $\mathbf{n}$ is

$$\mathbf{n} = \frac{\mathbf{n} \times \mathbf{k}^i}{\|\mathbf{n} \times \mathbf{k}^i\|} \tag{46}$$

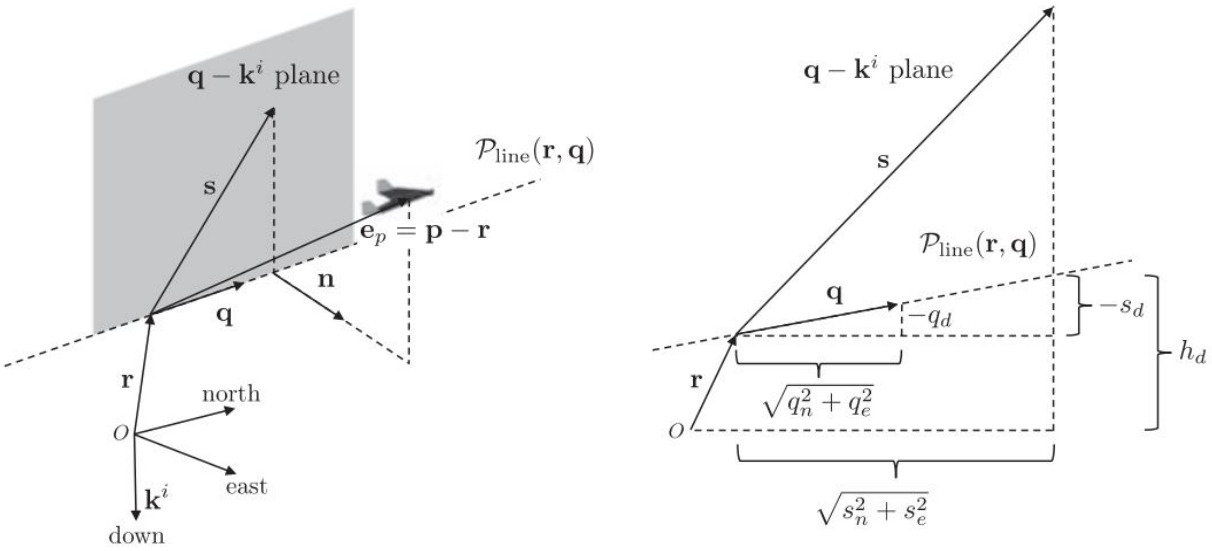
and the error e_i^p is

$$\mathbf{e_i^p} = \begin{bmatrix} e_{pn} \\ e_{pe} \\ e_{pn} \end{bmatrix} = \mathbf{p}^i - \mathbf{r}^i = \begin{bmatrix} p_n - r_n \\ p_e - r_e \\ p_d - r_d \end{bmatrix} \tag{47}$$

where the $\mathbf{r}^i$ is the first waypoint, and $\mathbf{p}^i$ is the position for the aircraft.

The commanded height is then

$$h^c = -r_d - s_d \tag{48}$$



(a) Full plane [1]

(b) Norm of plane[1]

Figure 13: Planes

Note then the Smallest sign angle (SSA) function is used in the algorithm, which maps the angle from $[-\pi\pi)$ to prevent discontinuous jumps in angles [2]. The function is called $SSA(\cdot)$ where the input is an angle in radians.

The whole algorithm can be found in attachments 8.2.

3.12 Optimizing

Optimizing is to minimize or maximise a cost function. In this context, we will minimize. With this objective function, there is some constrains, which needs to be respected. There are equality constrains, which means that something has to be equal. And inequality, which means greater or smaller. Typically we denote it:

$$\begin{matrix} min \\ z \in \mathbb{R}^n \end{matrix} \quad f(z) \tag{49}$$

subject to:

$$\begin{aligned} c_i(z) &= 0, \quad i \in \mathcal{E} \\ c_i(z) &\geq 0, \quad i \in \mathcal{I} \end{aligned} \tag{50}$$

Here we see that $\mathcal{E}$ are the equality constraints, and $\mathcal{I}$ are the inequality constraints. $f(z)$ is the cost function and z is the decision variable.

Some important terms

1. Feasible area: an area where all the constraints hold, and we can find a solution
2. Convex: if the function is convex, then we can connect any two points on the function with a line that does not cross the function itself. With this, we can say if our solution is a global minimum or only local. If convex, then we can say it is global, if not, we could have a locally. Intuitive explanation, if water flows down the mountainside. If there are no obstacles (think of a U-shape), then the water flows to the bottom, and it is convex and we have a global minimum. If there are obstacles, then the water can be collected there, and we only have a local minimum. With this, it is not convex, because the obstacles will clearly block the line.
3. Karush–Kuhn–Tucker (KKT) conditions: define necessary conditions for optimality. For more information, the read should look into [3] or [4].

3.13 Positive definite

A matrix, Q, is said to be positive invariant if

$$x^T Q x \geq 0 \quad \forall x \tag{51}$$

From here, the notation of this is $Q \succeq 0$.

3.14 Nonlinear Model Predictive Control

As seen in section 3.12, optimization is to minimize. In a model predictive control, the goal is to minimization an objective function, which for example can be the error between the wanted position and actual position.

The MPC solve the optimizing problem over a time horizontal. When all is solved, the controller only applies the first input, and disregard the rest. Meaning that for every time step, the MPC calculates the optimal solution. This is illustrated in figure 14. Here the upper part of the figure shows how the MPC calculate the trajectory and only apply the first step to the plant, the lower part.

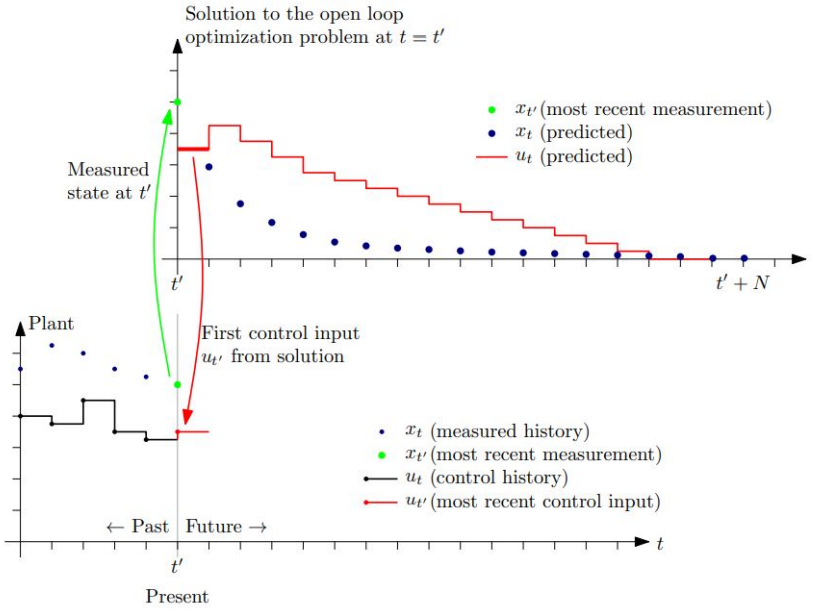


Figure 14: MPC [3]

A NMPC can be formulated based on the optimal control problem (OCP).

$$\begin{aligned} &min \\ &z \in \mathbb{R}^n \end{aligned} \quad f(z) \tag{52}$$

where

$$f(z) = \sum_{t=0}^{N-1} \frac{1}{2} x_{t+1}^T Q_{t+1} x_{t+1} + d_{xt+1} x_{t+1} + \frac{1}{2} u_t^T R_t u_t + d_{ut} u_t + \frac{1}{2} \Delta u_t^T R_{\Delta t} u_t \tag{53}$$

subject to

$$x_{t+1} = g(x_t, u_t) \tag{54}$$

$$x_o = given \tag{55}$$

$$x^{low} \leq x_t \leq x^{high} \tag{56}$$

$$u^{low} \leq u_t \leq u^{high} \tag{57}$$

$$-\Delta u^{high} \leq u_t \leq \Delta u^{high} \tag{58}$$

where

$$\begin{aligned} Q_t &\succeq 0 \\ R_t &\succeq 0 \\ R_{\Delta t} &\succeq 0 \end{aligned} \tag{59}$$

The OCP in equation 53 is a quadratic function, with linear terms. If the OCP is ex the error, this can be seen as computing the sum of all errors from time 0 to time N-1, and then finding the optimal input to the system which gives the least error.

The last term in the OCP, is to control input change, in this thisi this is not relevant and hence removed. This also includes the constraints on the change of input in 58.

The 54 is the nonlinear equation for the system. 55 is the start condition for state and input. Notice that index 0 on the state, and -1 on the input. 56 and 57 is the constraint on the state and input.

The 59 is the gain matrix. Note that $R_{\Delta t}$ is removed (see above). All this matrix needs to be positive invariant, see section 3.13. This matrix is here time-variant, and in this thesis this matrix is considered time-invariant, meaning constant.

3.15 Acados form

From this point, the notation of the OCP while be on the Acados form [5]. Here the OCP is defined as

$$l\Big(x(t), z(t), u(t)\Big) = \|\mathbf{V}_x x + \mathbf{V}_u u + \mathbf{V}_z z - \mathbf{y}_{ref}\|^2_W \tag{60}$$

With the terminal cost

$$m(x) = \|\mathbf{V}^e_x x - \mathbf{y}^e_{ref}\|^2_{W^e} \tag{61}$$

Here the gain matrix W is

$$W = diag\{\begin{bmatrix} Q & R \end{bmatrix}\} \tag{62}$$

Where the matrix Q and R can be recognized as weight parameters controlling the stats and inputs, respectively. Where both Q and R is positive definite and diagonal.

Note also the algebraic state z, which should not be confused with the virtual state introduce in section 3.18.

The matrix V_x and V_u maps from n_x and n_u to n_y, where n_x, n_u and n_y is the number states, input and output, respectively.

3.16 Path parameterizing

In this thesis the problem is a path following. Meaning, the goal is for the plane to follow a line. Time is not important here, meaning that the as long as on the line is good, and where on the line at given time not. The reference is denoted as path, $\mathcal{P}$ [6], and it is given parametrized regular curve in the output space, in general

$$\mathcal{P} = \{y \in \mathcal{R}^{n_y} | \theta \in \begin{bmatrix} \theta_0, \theta_1 \end{bmatrix}\} \rightarrow p(\theta) \tag{63}$$

Here the variable θ is the scalar path parameter. The range of θ is bounend by θ_0 and θ_1.θ_0 can be choosen as a negativ number, but θ_1 must be zero. This is because the goal is to end up at the end point. In this thises, this values are given as $\theta_0 = -1$ and $\theta_1 = 0$. The path equation is in this thisis defined as

$$p(\theta) = w_2 - \theta(w_1 - w_2) \tag{64}$$

Where the w_1 and w_2 is waypoints in cartesian coordinates defined in $\mathcal{F}^i$. Where w_1 is the start waypoint and w_2 is the end, define as:

$$\begin{aligned} w_1^i &= \begin{bmatrix} w_{1_x}^i & w_{1_y}^i & w_{1_z}^i \end{bmatrix}^T \\ w_2^i &= \begin{bmatrix} w_{2_x}^i & w_{2_y}^i & w_{2_z}^i \end{bmatrix}^T \end{aligned} \tag{65}$$

From equation 64, it can be seen that when the path parameter is θ_0, the $p(\theta) = w_1$. And at θ_1, $p(\theta) = w_2$. This means that the goal is for $\theta \to \theta_1$ when $t \to \infty$.

In figure 15, it can be seen as a visual overview of the goal. In the beginning, the airplane aims at w1, meaning $\theta = \theta_0$. As time goes, $\theta \to \theta_1$ and the airplane aims more and more at w_2. These are the green lines in the figure. So for each step, the aiming is more and more at w_2.

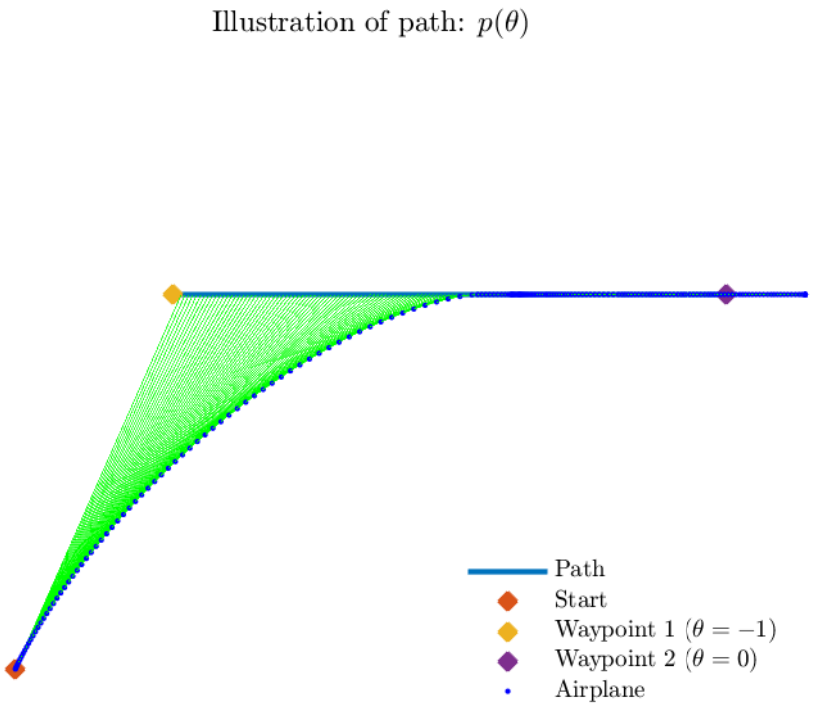


Figure 15: Illustration of path

The overall goal is then that:

$$\lim_{t \to \infty} \left\| \mathbf{y}(t) - p(\theta(t)) \right\| = 0 \tag{66}$$

where $\mathbf{y}$ is the position state of the airplane. Note that this is the error between the position and the path, so the error can be defined as:

$$e(t) = \mathbf{y}(t) - p(\theta(t)) \tag{67}$$

3.17 Timing law

The path parameter θ is introduce as a virtual state. This needs to be controlled, and here the timing law is introduce.

$$\begin{aligned} &g\Big(\theta^{(k)}, \theta^{(k-1)}, ..., \dot{\theta}, \theta, v\Big) = 0 \\ &\forall i \in \{1, ..., k\} : \theta^{(i)}(t_0) = \theta_0^{(i)}, \theta(t_0) = \theta_0 \end{aligned} \tag{68}$$

The first line defined the timing law $g(\cdot)$, the second line defined the initialization condition to the virtual states. The state v is the virtual input. This can be seen as an input to control behavior along the path.

Timing law is chosen as [6]:

$$\begin{aligned} &\theta^{(\hat{r}+1)} = v \\ &\theta(t_0) = \theta_0 \quad \forall j \in \{1, ..., \hat{r}\} : \theta^{(j)}(t_0) = 0 \end{aligned} \tag{69}$$

Where

$$\hat{r} = max\{r_1, ..., r_{n_y}\} \tag{70}$$

Where $\{r_1, ...r_{n_y}\}$ is the relevant degrees.

3.18 Augmented system

Given that the nonlinear system is defined as:

$$\dot{x}(t) = f(x(t), u(t)), \quad x(t_0) = x_0 \tag{71}$$

$$y(t) = h(x(t)) \tag{72}$$

The size of the input u and state x is n_u and n_x. The size of output is n_y. The function $h(x(t))$ maps from n_x to n_y.

The virtual system is defined as

$$\begin{aligned}\dot{z} =& \mathbf{A}z + \mathbf{B}v \\ =& \underbrace{\begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \\ 0 & \cdots & 0 & \cdots & 1 \\ 0 & \cdots & 0 & \cdots & 0 \end{bmatrix}}_{\text{A}} z + \underbrace{\begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 1 \end{bmatrix}}_{\text{B}} v\end{aligned} \tag{73}$$

$$\begin{aligned}\theta =& Cz \\ =& \underbrace{\begin{bmatrix} 1 & \cdots & 0 \end{bmatrix}}_{\text{C}} \\ =& z1\end{aligned} \tag{74}$$

In 73 the virtual input v is

$$z^{\hat{r}+1} = v \tag{75}$$

and the states are

$$z^k = z_{k+1} \quad k = \{1, ..., \hat{r}\} \tag{76}$$

Adding the timing law with the virtual state and input, the augmented system is then defined as:

$$\begin{bmatrix} \dot{x} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} f(x,u) \\ l(z,u) \end{bmatrix} \tag{77}$$

$$\begin{bmatrix} e \\ \theta \end{bmatrix} = \begin{bmatrix} h(x) - p(z1) \\ z1 \end{bmatrix} \tag{78}$$

The size of virtual state z, is n_z. Note the the state z is not an algebraic state. Where n_z is given by

$$n_z = \hat{r} + 1 \tag{79}$$

The total size of equation 77 is $n_x + n_z$. Equation 78 is the OCP. Where error dynamics can be extended according to section 3.17.

4 Control Algorithm Design

For comparison, there will be designed to types of system. One with MPC and one with successive loops.

4.1 Implement a switching logic and follow a rectangular pattern

If the airplane is within a radius of a threshold, then switch to the next waypoints. The code for this function is in attachemnts 8.1.

4.2 Path-following algorithms for straight lines

The dynamics, introduce in section 3.6, for the airplane is choosen from the UAV lab folder. Next, the system will be exposed for gust wind (3.2). The control inputs (3.9 is aileron and rudder.

The approach is to use two successive loops, 3.10, where subsystem 1 uses aileron to control the roll and in subsystem 2 elevator the pitch. An overview of the system can be seen in figure 16. Here the subsystem 1 is called course, and subsystem 2 is height. A path algorithm design by Beard and McLain [1] will be used. The output is the desired course angle and height.

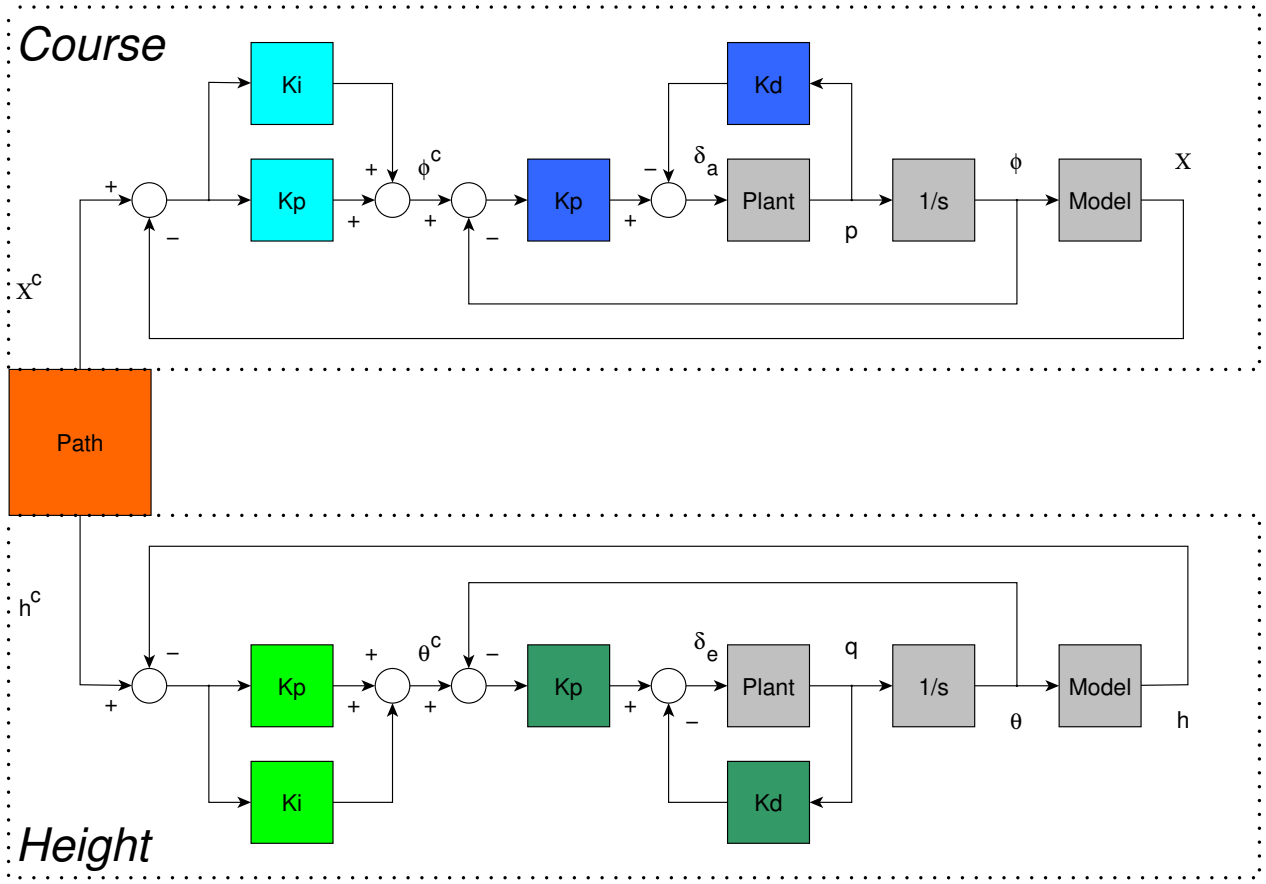


Figure 16: Flow chart system[1]

Subsystem 1

The upper loop is to control the course angle of the airplane. Path gives the commanded course angle. The light blue boxes is the PI controller for the roll. This is the outer loop. The controller is defined as

$$\phi^c = k_{p_\chi}(\chi^c - \chi) + \frac{k_{i_\chi}}{s}(\chi^c - \chi) \tag{80}$$

This desired roll angle can be used to find the aileron input, defined as the PD controller, which is the blue boxes and the inner loop. Defined as

$$\delta_a = k_{p_\phi}(\phi^c - \phi) - k_{d_\phi} p \tag{81}$$

The gains are found through tuning.

Subsystem 2

The lower loop is to controller the height of the airplane. Path gives the commanded height of the airplane. The outer loop, the green boxes, is the controller for the pitch angle θ. The controller is defined as the PI controller:

$$\theta^c = k_{p_h}(h^c - h) + \frac{k_{i_h}}{s}(h^c - h) \tag{82}$$

The inner loop is to controll the elevator input δ_e, which is the dark green boxes. The PD controller is defined as: beginequation

$$\delta_e = k_{p_\theta}(\theta^c - \theta) - k_{d_\theta} q \tag{83}$$

The output of the algorithm is commanded course and height, as seen above.

Since the underlying model dynamics is collected from a library, the gains are found through trial and error. The following gains are defined for subsystem 1

Gains	Value
k_{p_ϕ}	5
k_{d_ϕ}	0.1
k_{p_χ}	2
k_{i_χ}	1.4

Table 3: Subsystem 2 gains

And for subsystem 2:

Gains	Value
k_{p_θ}	-1.2
k_{d_θ}	-0.1
k_{p_h}	-0.2
k_{i_h}	-0.1

Table 4: Subsystem 2 gains

Path

To defined the path to follow, the straight-line path algorithm of descried in section 3.11 is used. The algorithm can be seen in attachments 8.2.

4.3 MPFC

The model is chosen from Beard and McLain [1] where it is assumed that the autopilot controls airspeed, altitude and course angle, denoted V_a^c, h^c and χ^c respectively. From equation 9.19 in Beard and McLain[1] the guidance model is

$$\begin{aligned}
\dot{p}_n &= V_a \cos\psi + w_n \\
\dot{p}_e &= V_a \sin\psi + w_e \\
\ddot{\chi} &= b_{\dot{\chi}}(\dot{\chi}^c - \dot{\chi}) + b_{\chi}(\chi^c - \chi) \\
\ddot{h} &= b_{\dot{h}}(\dot{h}^c - \dot{h}) + b_h(h^c - h) \\
\dot{V}_a &= b_{V_a}(V_a^c - V_a)
\end{aligned} \tag{84}$$

Where the heading angle ψ is defined as:

$$\psi = \chi - \arcsin\left(\frac{1}{V_a \cos\gamma_a}\begin{bmatrix} w_n \\ w_e \end{bmatrix}^T \begin{bmatrix} -\sin\chi \\ \cos\chi \end{bmatrix}\right) \tag{85}$$

and the air-mass-referenced flight path angle $\gamma_a = 0$. Inserting this into equation 85

$$\psi = \chi - \arcsin\left(\frac{1}{V_a}\begin{bmatrix} w_n \\ w_e \end{bmatrix}^T \begin{bmatrix} -\sin\chi \\ \cos\chi \end{bmatrix}\right) \tag{86}$$

Also introduce the time differenceti $\ddot{V}a$

$$\ddot{V}_a = b_{V_a}(\dot{V}_a^c - V_a) + b_{V_a}(V_a^c - V_a) \tag{87}$$

Define the states to be:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \end{bmatrix} = \begin{bmatrix} p_n \\ p_e \\ h \\ \dot{h} \\ \chi \\ \dot{\chi} \\ V_a \\ \dot{V}_a \\ V_a^c \\ h^c \\ \chi^c \end{bmatrix} \tag{88}$$

We have that $\dot{x}(t) = f(x(t), u(t)$, and define

$$f(x,u) = \begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{h} \\ \ddot{h} \\ \dot{\chi} \\ \ddot{\chi} \\ \dot{V_a} \\ \ddot{V_a} \\ \dot{V_a^c} \\ \dot{h^c} \\ \dot{\chi^c} \end{bmatrix} = \begin{bmatrix} V_a \cos\psi + w_n \\ V_a \sin\psi + w_e \\ \dot{h} \\ b_{\dot{h}}(\dot{h}^c - \dot{h}) + b_h(h^c - h) \\ \dot{\chi} \\ b_{\dot{\chi}}(\dot{\chi}^c - \dot{\chi}) + b_\chi(\chi^c - \chi) \\ b_{V_a}(V_a^c - V_a) \\ b_{V_a}(\dot{V_a^c} - V_a) + b_{V_a}(V_a^c - V_a) \\ \dot{V_a}^c \\ \dot{h^c} \\ \dot{\chi^c} \end{bmatrix} \tag{89}$$

Choose the following input to the system:

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} \dot{h}^c \\ \dot{\chi}^c \\ \dot{V_a}^c \end{bmatrix} \tag{90}$$

And the output to be

$$y = \begin{bmatrix} p_n \\ p_e \\ h \\ V_a \end{bmatrix} \tag{91}$$

Where $b_{\dot{h}}$, b_h, $b_{\dot{\chi}}$ and b_χ is constants, and w_n and w_e is wind defined in the NED frame, and the direction is north and east respectively. Note also that heading angle ψ is defined as equation 86, but for simplicity it is not inserted in equation 89.

In section 3.2 was described, here the wind vector is set to a constant,

$$\mathbf{V}_w = \begin{bmatrix} w_n \\ w_e \\ w_d \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix} \tag{92}$$

The other constants are defined as

$$\begin{aligned} b_{\dot{h}} &= \frac{(\omega_h)^2}{K_\theta V_a} \\ b_h &= \frac{2\zeta\omega_h}{K_\theta V_a} \end{aligned} \tag{93}$$

[1] And

$$\begin{aligned} \dot{b_\chi} &= \omega_\chi \frac{V_g}{g} \\ b_\chi &= 2\zeta\omega_\chi \frac{V_g}{g} \end{aligned} \tag{94}$$

[1] And

$$b_{V_a} = 0.69 \tag{95}$$

Where variables are choosen to be.

ζ	1
ω_h	0.5
K_θ	1.2
V_a	20
ω_χ	0.5
V_g	24.2
g	9.81

Table 5: Constants

Note that the ground speed is defined as equation 12, so $V_g = V_a + V_w = 20 + \sqrt{3^2 + 3^2} = 24.2$.

Next is to find the OCP, and here to error between position and path is chosen. The path parameter and timing law is described in section 3.16 and 3.17. The error is

The augmented state dynamics 3.18is defined as

$$f(x,u) = \begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{h} \\ \ddot{h} \\ \dot{\chi} \\ \ddot{\chi} \\ \dot{V_a} \\ \ddot{V_a} \\ \dot{V_a}^c \\ \dot{h}^c \\ \dot{\chi}^c \\ \dot{z_1} \\ \dot{z_2} \end{bmatrix} = \begin{bmatrix} V_a \cos\psi + w_n \\ V_a \sin\psi + w_e \\ \dot{h} \\ b_{\dot{h}}(\dot{h}^c - \dot{h}) + b_h(h^c - h) \\ \dot{\chi} \\ b_{\dot{\chi}}(\dot{\chi}^c - \dot{\chi}) + b_\chi(\chi^c - \chi) \\ b_{V_a}(V_a^c - V_a) \\ b_{V_a}(\dot{V}_a^c - V_a) + b_{V_a}(V_a^c - V_a) \\ \dot{V_a}^c \\ \dot{h}^c \\ \dot{\chi}^c \\ z_2 \\ v \end{bmatrix} \tag{96}$$

From equation 69, the $\hat{r} = 2$, this gives that the timing law is

$$\begin{aligned} z_1 &= \theta \\ \dot{z}_1 &= z_2 \\ \dot{z}_2 &= v \end{aligned} \tag{97}$$

Which is what is in equation 96. Reasoning for why $\hat{r} = 2$ is that if two times time difference the output in equation 91 then the inputs are delivered, from equation 90.

The state virtual state z_1 can be seen as the position on our path line. And z_2 is the velocity on the line. Note that this is not the actual speed of the airplane, but a virtual state. The airplane needs enough speed to create lift to maintain height and to basically fly. So the movement on the line can not be zero or small due to lift. The following constraint is defined:

$$-1 < z_1 \leq 0 \tag{98}$$

$$z_{2_{min}} < z_2 \leq z_{2_{max}} \tag{99}$$

Including the virtual input, and get

$$u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} \dot{h}^c \\ \dot{\chi}^c \\ \dot{V}_a^c \\ v \end{bmatrix} \tag{100}$$

To the defined the OCP, one option is to choose to minimize the error between the actual position agings the path. This could be seen as

$$\lim_{t \to \infty} \| \begin{bmatrix} p_n & p_e & h \end{bmatrix}^T - p(\theta(t)) \|_2 = 0 \tag{101}$$

Where the the error is defined:

$$\begin{aligned} e &= \begin{bmatrix} p_n \\ p_e \\ h \end{bmatrix} - p(\theta) \\ &= \begin{bmatrix} p_n \\ p_e \\ h \end{bmatrix} - \left(\begin{bmatrix} w_{2_y}^i \\ w_{2_x}^i \\ w_{2_z}^i \end{bmatrix} - \theta \begin{bmatrix} w_{1_y}^i - w_{2_y}^i \\ w_{1_x}^i - w_{2_x}^i \\ w_{1_z}^i - w_{2_z}^i \end{bmatrix} \right) \\ &= \begin{bmatrix} p_n \\ p_e \\ h \end{bmatrix} - \left(\begin{bmatrix} w_{2_y}^i \\ w_{2_x}^i \\ w_{2_z}^i \end{bmatrix} - \theta \begin{bmatrix} dw_y \\ dw_x \\ dw_z \end{bmatrix} \right) \end{aligned} \tag{102}$$

Where

$$dw = w_1 - w_2 \tag{103}$$

and the error dynamics

$$\begin{aligned}
\dot{e} &= \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} - \begin{bmatrix} \frac{\partial p_1}{\partial \theta} \\ \frac{\partial p_2}{\partial \theta} \\ \frac{\partial p_3}{\partial \theta} \end{bmatrix} \dot{\theta} \\
&= \begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{h} \end{bmatrix} - \begin{bmatrix} \frac{\partial \left(w_{2_y}^i - z_1 dw_y \right)}{\partial z_1} \\ \frac{\partial \left(w_{2_x}^i - z_1 dw_x \right)}{\partial z_1} \\ \frac{\partial \left(w_{2_z}^i - z_1 dw_z \right)}{\partial z_1} \end{bmatrix} \dot{\theta} \\
&= \begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{h} \end{bmatrix} - \begin{bmatrix} -dw_y \\ -dw_x \\ -dw_z \end{bmatrix} \dot{\theta} \\
&= \begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{h} \end{bmatrix} + \begin{bmatrix} dw_y \\ dw_x \\ dw_z \end{bmatrix} \dot{\theta}
\end{aligned} \tag{104}$$

and time differnce one more time gives:

$$\ddot{e} = \begin{bmatrix} \ddot{p_n} \\ \ddot{p_e} \\ \ddot{h} \end{bmatrix} + \begin{bmatrix} dw_y \\ dw_x \\ dw_z \end{bmatrix} \ddot{\theta} \tag{105}$$

Define the optimal problem states to be

$$\mathbf{x}_{ocp} = \begin{bmatrix} e_n \\ e_e \\ e_h \\ \dot{e_n} \\ \dot{e_e} \\ \dot{e_h} \\ V_a \\ \dot{V_a} \\ V_a^c \\ h \\ \dot{h} \\ h^c \\ \chi \\ \dot{\chi} \\ \chi^c \\ z_1 \\ z_2 \end{bmatrix} \tag{106}$$

$$e_n \; e_e \; e_h \; \dot{e_n} \; \dot{e_e} \; \dot{e_h} \; V_a \; \theta$$

$$\mathbf{u}_{ocp} = \begin{bmatrix} \dot{h}^c \\ \dot{\chi}^c \\ \dot{V}_a^c \\ v \end{bmatrix} \tag{107}$$

The objectiv function is defined as

$$F(e, \dot{e}, V_a, \theta) = \|(e, \dot{e}, V_a, \theta)\|_Q^2 \tag{108}$$

This gives that $n_y = 8 + 4 = 12$, where there is 8 states and 4 inputs.

Want this on Acados form 3.15, meaning

$$l\Big(x(t), z(t), u(t)\Big) = F(e, \dot{e}) \tag{109}$$

Define the system inside the norm, to be:

$$l\Big(x(t), z(t), u(t)\Big) = \mathbf{V}_x \mathbf{x}_{ocp} + \mathbf{V}_u \mathbf{u}_{ocp} - \mathbf{y_{ref}} \tag{110}$$

The matrix $\mathbf{V}_x$ is to map from $\mathbf{x}_{ocp}$ to $\mathbf{y}_{ocp}$ and $\mathbf{V}_u$ to $\mathbf{u}_{ocp}$ to $\mathbf{y}_{ocp}$.

$\mathbf{V}_x$ is designed so that is chosen form equation 106, and $\mathbf{V}_u$ is designed so that al inputs in 107 is choosen. The following stacking is wanted

$$l\Big(x(t), z(t), u(t)\Big) = \| \underbrace{\begin{bmatrix} e_n \\ e_e \\ e_h \\ \dot{e_n} \\ \dot{e_e} \\ \dot{e_h} \\ V_a \\ \theta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\mathbf{V}_x \mathbf{x}_{ocp}} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \dot{h}^c \\ \dot{\chi}^c \\ \dot{V}_a^c \\ v \end{bmatrix}}_{\mathbf{V}_u \mathbf{u}_{ocp}} - \mathbf{y_{ref}} \|_W^2 \tag{111}$$

Need the following python scripts:

1. model

2. acados settings
3. main

Where in *model* the $\mathbf{x}_{ocp}$ and $\mathbf{u}_{ocp}$ is defined with the state equation defined in 96. Along with the start conditions and constraint on states, which is need in NMPC, see section 3.14.

In *acados settings* the gain matrix $\mathbf{W}$, $\mathbf{V}_x$ and $\mathbf{V}_u$, $\mathbf{y_{ref}}$ is deifined along with simulation settings and creating the solver. *Main* is just for simulations.

The overall system is then defined as in figure 17. Note that the state estimation is neglected away in this thesis.

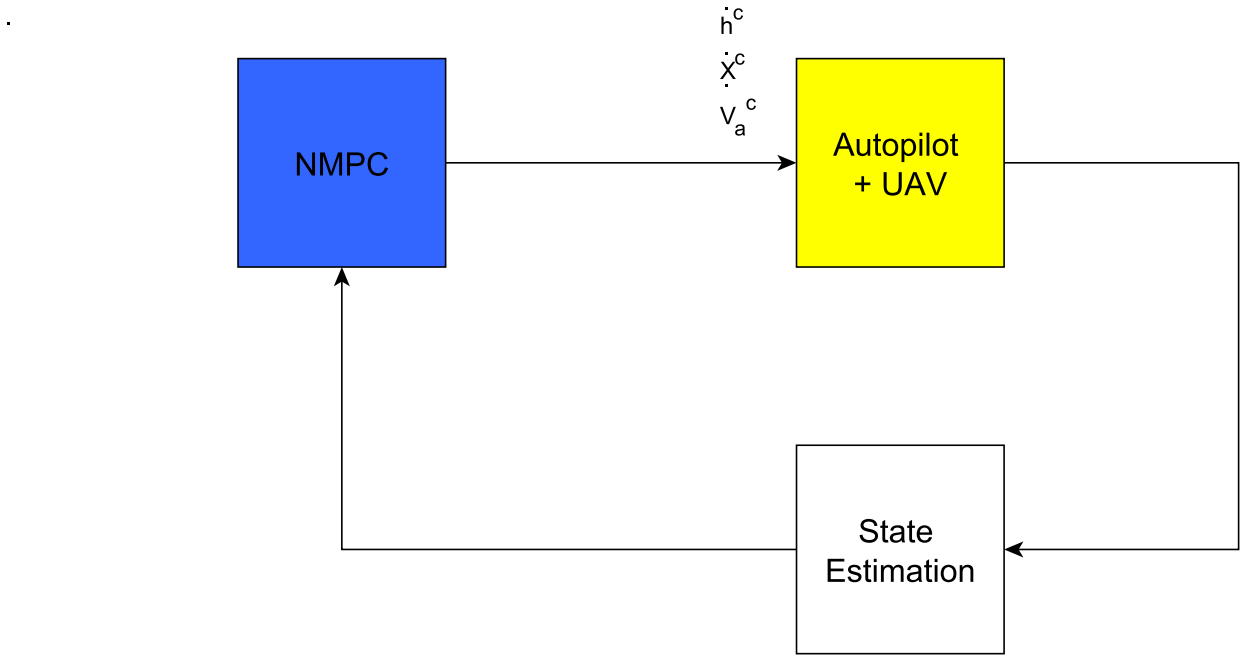


Figure 17: Flow chart system of the MPFC

4.4 MPFC alternative

On alternative way of solving the OCP is to treat the system as a Differential algebraic equation (DAE) instead of Ordinary differential equation (ODE) which is presented above. With an DAE system introduce the algebraic states as the error between state and path. For example

$$\mathbf{z} = \begin{bmatrix} p_n - w_2 + p(\theta)(w_1 - w_2) \\ p_e - w_2 + p(\theta)(w_1 - w_2) \\ h - w_2 + p(\theta)(w_1 - w_2) \end{bmatrix} \tag{112}$$

where $\mathbf{z}$ is the algebraic state.

5 RESULTS

5.1 Path-following algorithms for straight lines

This system is described in section 4.2. The results can be seen in figure 18. Here the goal is to follow a straight line. In the figure, the airplane is able to follow the line. Moreover, it can be seen in figure 19 that the states are able to follow the commanded course and height.

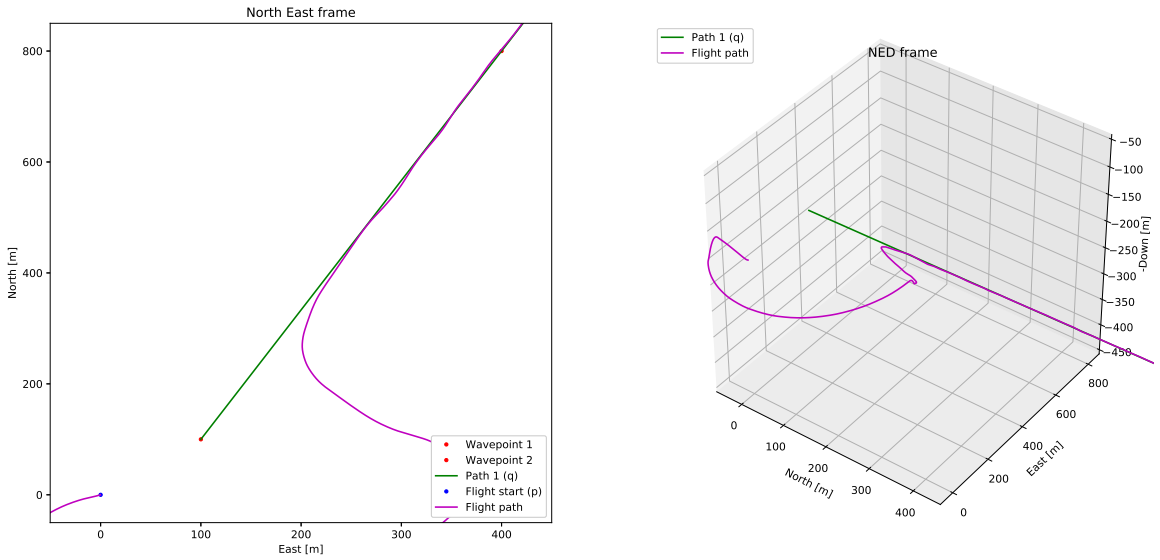


Figure 18: Result path-following algorithms for straight lines

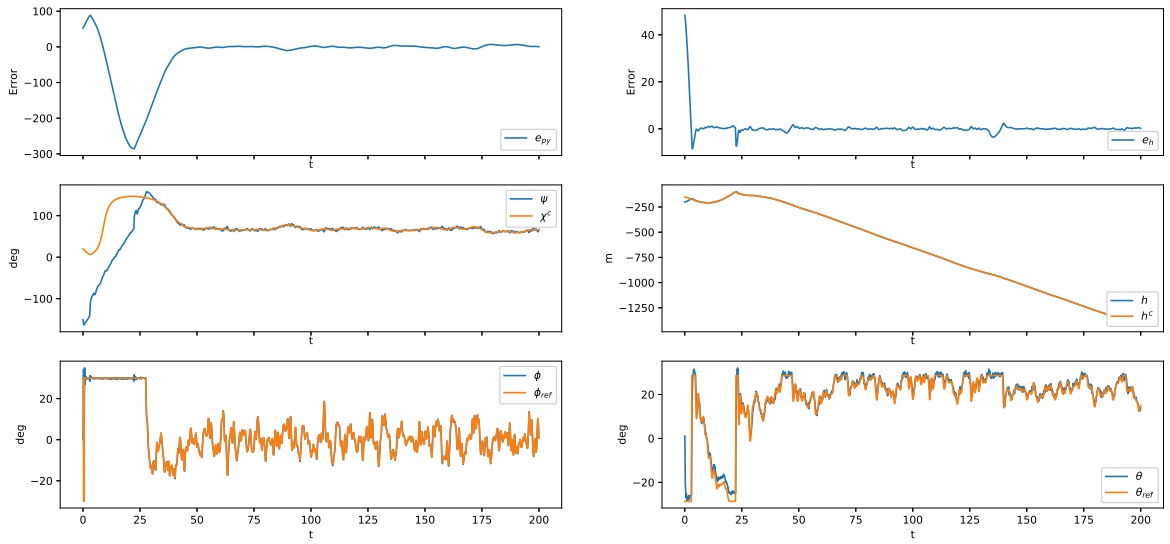


Figure 19: Overview of states

5.2 Path-following algorithms for straight lines with a switching logic and follow a rectangular pattern

Here the switch logic is included, from section 4.1, and the results can be seen in figure 20, with the controller in figure 21. Here, the goal is to follow the given path. When a waypoint is intersected, a new path is desired to follow. Also here the airplane is following the path quite good.

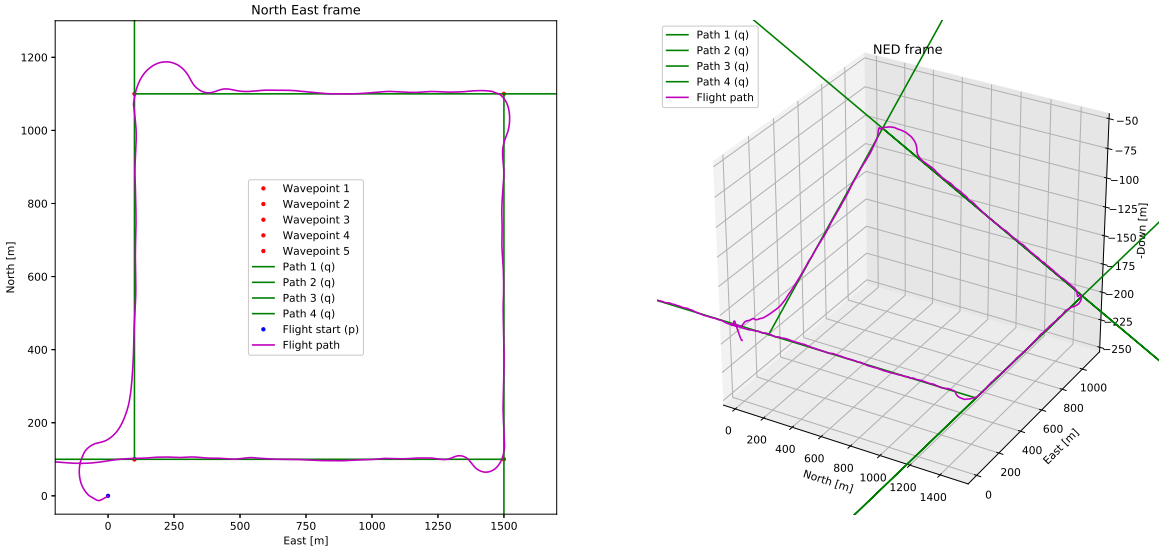


Figure 20: Result PID

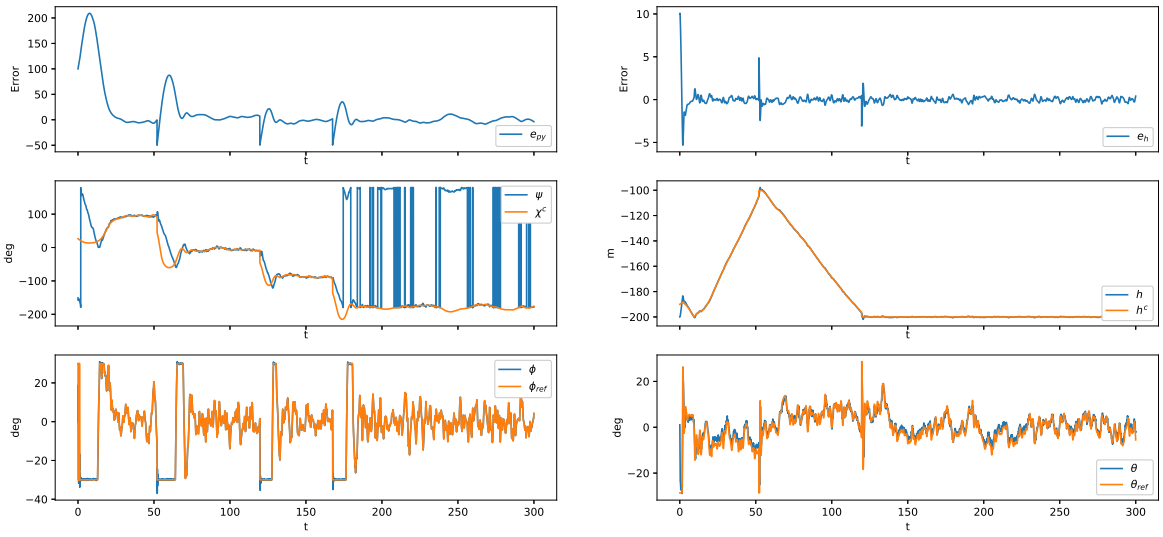


Figure 21: Result PID

5.3 MPFC straight lines in the longitudinal plane).

This system is described in section 4.2. The results can be seen in figure 22. Here the goal is to follow a line in $\mathcal{R}^2$. The airplane is here following the line quite good, to the point where

there is doodles. Also see that in the 3-D plots, that it looks like the airplane is moving very much in the vertical plane, but the movement is very small (see the axis).

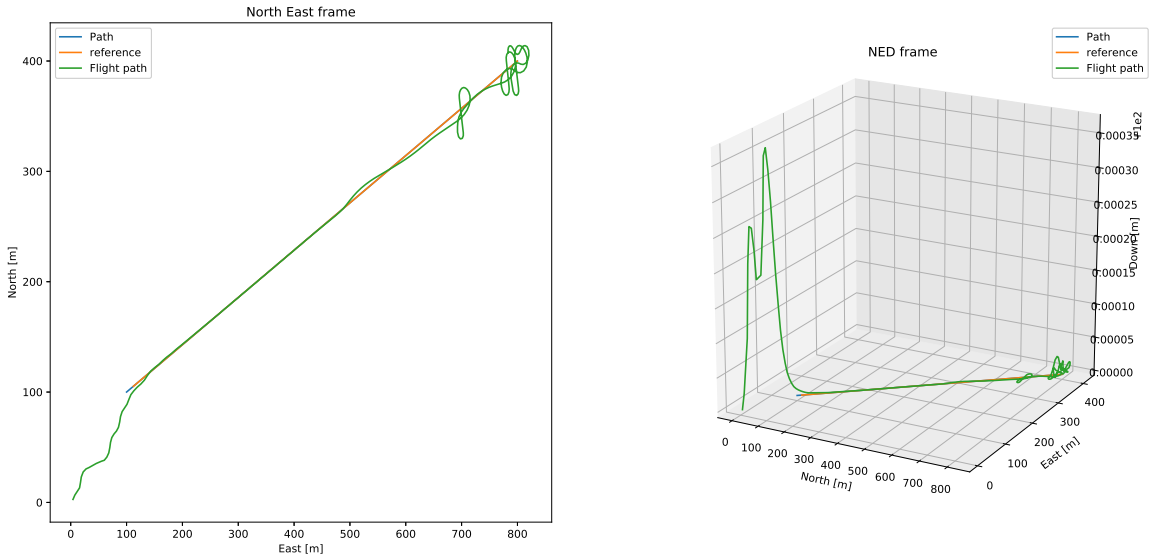
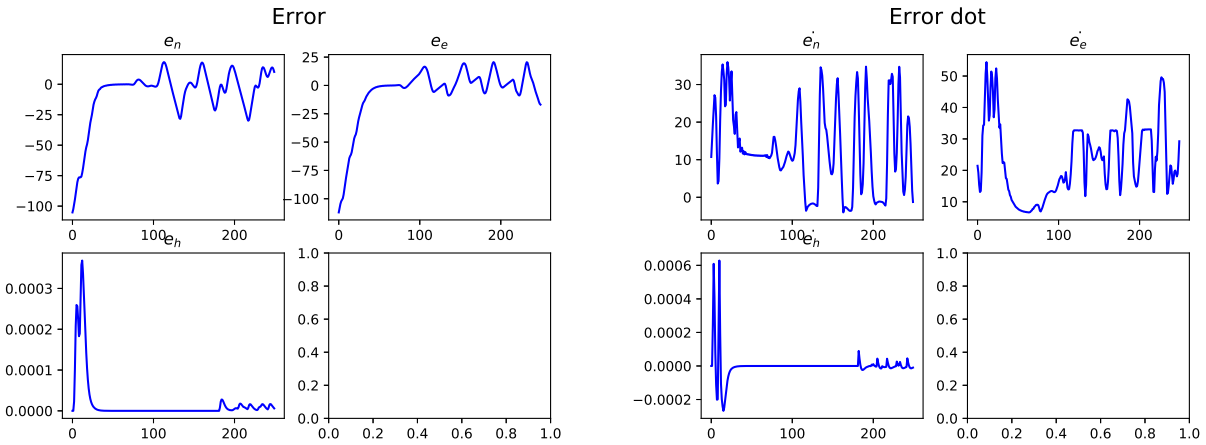


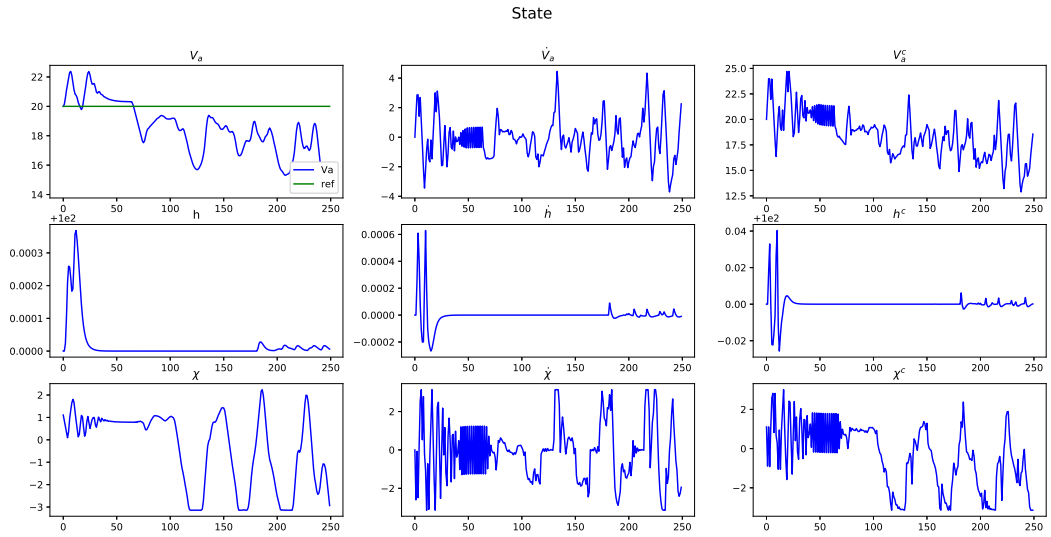
Figure 22: Result MPC path-following, straight lines in the longitudinal plane

In figure 23, figure 23a, 23b, V_a from figure 23c and z_1 from figure 23d is the output of the OCP. The inputs is in figure 23e. The rest in figure 23c and 23d is states and virtual state. In figure 23c it can be observed that V_a is not following its reference which is $20m/s$.

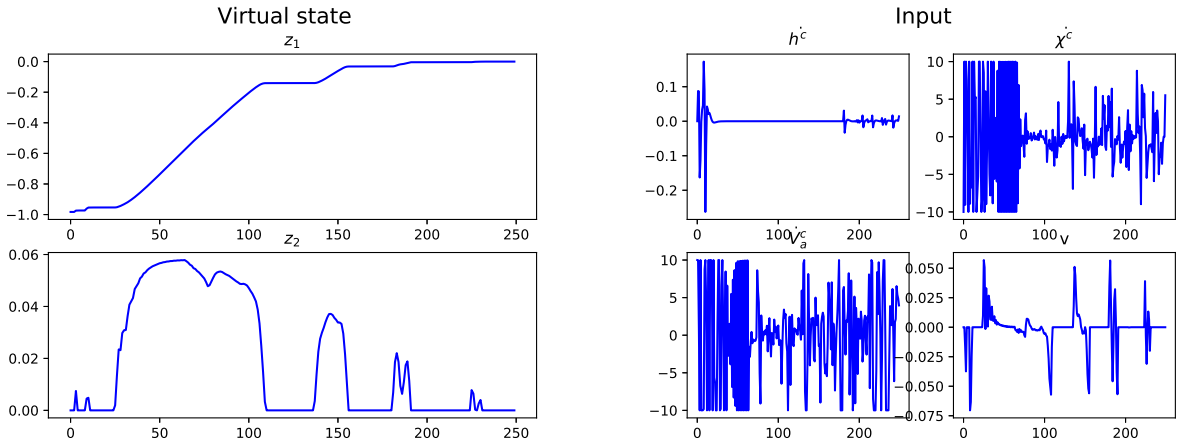


(a) Error

(b) Error dot



(c) States



(d) Virtual states

(e) Inputs

Figure 23: Overview of states and inputs

5.4 MPFC for following straight lines in both planes

Here the path to follow is in R^3. This means including a change in height. A result of this can be seen in figure 24. Here the same doodles from figure 22 happens. Other than that, the aircraft is following the path good. In the 3-D plot the aircraft manage converge to the path right before the doodle happens.

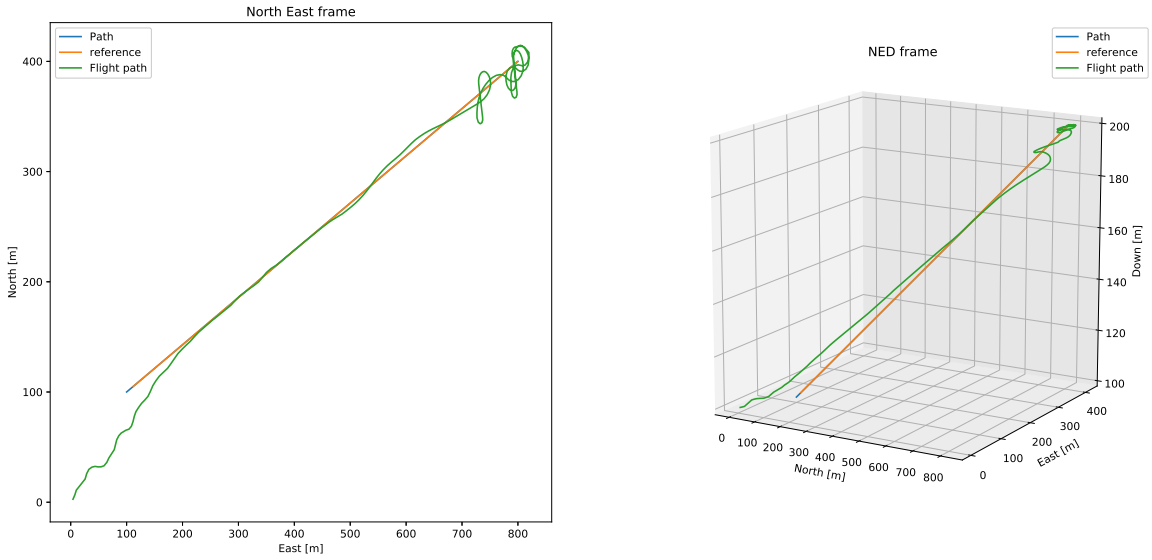
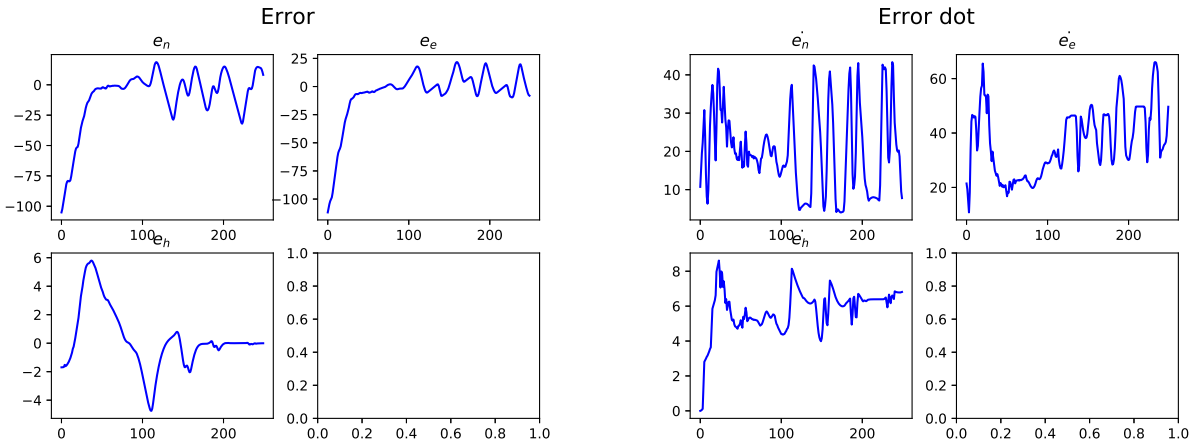


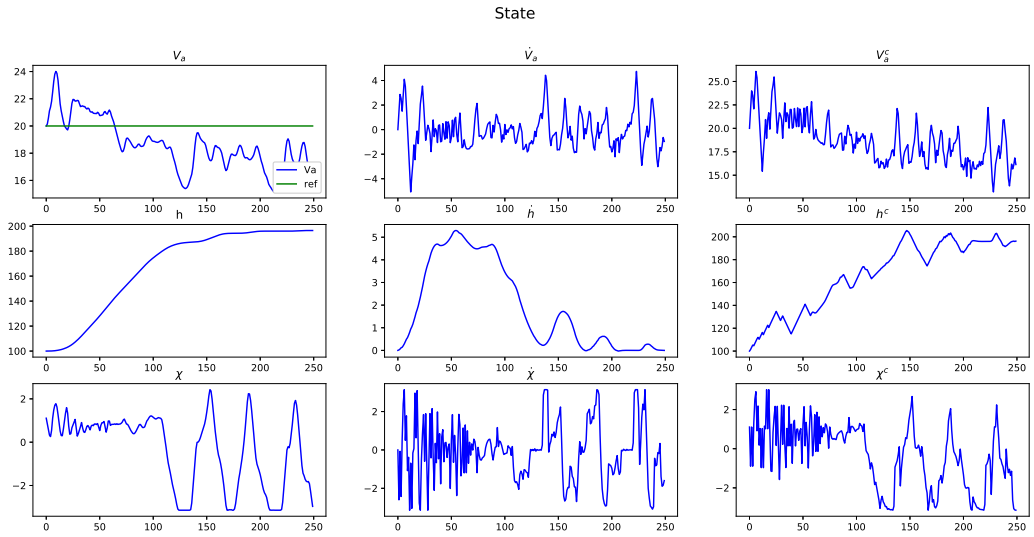
Figure 24: Result NMPC

In figure 25, figure 25a, 25b, V_a from figure 25c and z_1 from figure 25d is the output of the OCP. The inputs is in figure 25e. The rest in figure 25c and 25d is states and virtual state. In figure 25c the V_a is not following its reference which is $20m/s$.

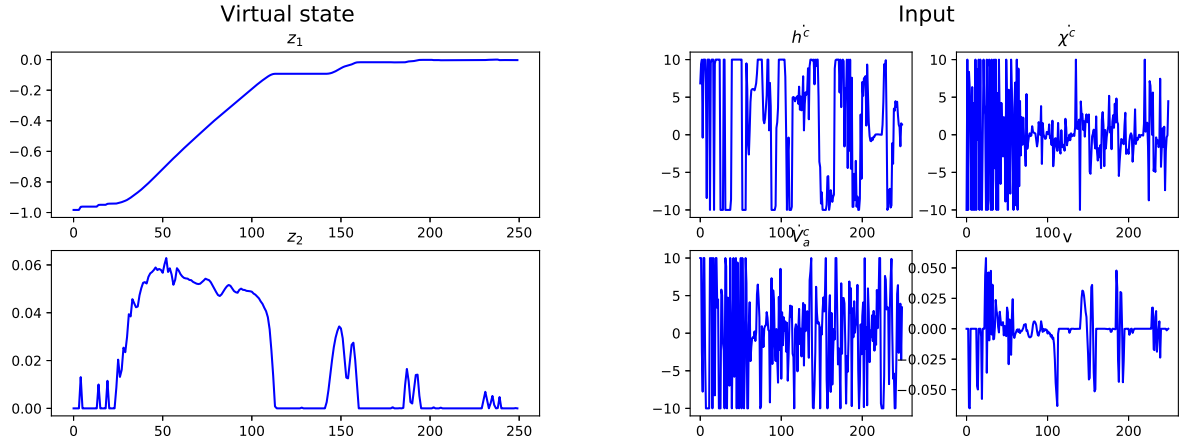


(a) Error

(b) Error dot



(c) States



(d) Virtuel states

(e) Inputs

Figure 25: NMPC 2

6 DISCUSSION

6.1 Path-following algorithms for straight lines

The aircraft is taking a long detour before converging to the path, see figure 18. This is because the aircraft starts at a trim condition, set up by the UAV library. The aircraft converges faster if a change in the trim conditions is made. Other than that, the performance was within reasonable accuracy. The controller was capable of following the commanded course and height. And also for the inner loop to follow the reference, roll and pitch, respectively. The cross-track error and height error both converged to zero, which is wanted. See figure 19.

6.2 Path-following algorithms for straight lines with a switching logic and follow a rectangular pattern

The results in section 5.2 with path-following algorithms for straight lines with a switching logic and following a rectangular pattern work very well. There was a bit of overshoot at the change of waypoints, but this as expected.

One could tune the gains to avoid such overshoot. Another way to handle this could be to increase the circle radius of which the operator predetermined accepted that the aircraft is close enough to waypoints to go to the next one. In the simulation, this radius was 10 meters.

From figure 21, where the control is seen, there is noise at the yaw angle. This is because the angle is right between 0 and 180 degrees, and there is then a discontinuous jump. In the controller, this is solved using the function $SSA(\cdot)$, which maps angles between $-\pi$ and π [2]. So this is solved.

From figure 21 the cross-track error and height error was oscillating with a very small value around zero, which is wanted. Comparison for commanded height and height is very good. Same for roll and pitch, where they followed the reference good.

6.3 MPFC straight lines

For the MPFC, in section 5.3 and 5.4 the aircraft manage to follow the reference quite good in the start.

The virtual state z_1 in figure 23d, is converging towards zero, which is wanted. From section 3.16, when $z_1 = \theta = 0$, the aircraft is at waypoint 2 (target waypoint). In figure 23e there is a loot of saturation of the inputs for $\dot{\chi}^c$ and $\dot{V}_a^c$. This can be seen when the spikes

are removed from the plo. It is not wanted to increase the constraints on input mentioned, because of physically reasons.

In figure 23a, the error is converging to zero, then, around 100 iterations, the error starts to oscillate. This can also be seen in figure 22 where at the end, it is bad and do not follow anything. This is because the z_1 values is reaching its constraints, which is 0. This can be seen in figure 23d. While it is wanted that $z_1 = 0$, it is also destroys the simulation because of constraints. In figure 23d from 0 to 100 iterations (in x-direction), the value is increasing steady, while beyond 100 iterations, z_1 increases slowly. So to fix this doodling, there could be introduced a new waypoint when, for example z_1 is greater than a threshold. For example -0.2. The same incident is happening in figure 24.

Overall, the aircraft manage to follow the path adequately

7 CONCLUSION

In this project, the main focus has been control algorithm design for following a path for fixed-wing aerial vehicles. Two different solution has been deigned, where they have been divided into two subproblems, where the task is in four parts:

Path-following algorithms for straight lines : Using the path algorithm of Beard and McLain [1] a successful path algorithm with two separate successive loops for controll of the commanded course and height was designed. This worked very well and the results where good.

Path-following algorithms for straight lines with a switching : A switch was designed which made the aircraft change to a new path when the aircraft was close enough to a (desired) waypoint. A successful simulation of a route with 4 waypoints in rectangular pattern was completed.

MPFC for following straight lines in the longitudinal plane : An MPFC was designed with the path parameterizing from Faulwasser[6]. The MPFC worked well until the end, where z_1 became to small, a possible solution is pointed out under future work.

MPFC for following straight lines in R^3 : Same MPFC as above, did relative good to the point where the results was bad, obviously.

Future work

Path-following algorithms for straight lines : One could tune the gains to avoid overshoot at the corners. Another way to handle this could be to increase the radius of the circle of which the operator predetermined accepted that the aircraft is close enough to waypoints to go to the next one. In the simulation, this radius was 10 meters.

MPFC for following straight lines in R^3 : Implement a switch so the MPFC is following a new waypoint before z_1 is too small. Using the value of z_1 for the switch can be used. One can include it so that the rectangular movement can be achieved, as in figure 20. The wind is now constant, so the wind could be included, the same way as introduced in section 3.2. The system uses simplified dynamics. So it is natural to try to expand to more advanced dynamics. For the path-following algorithms, the dynamics were from the UAV library. Also, in figure 23c, the controller did not manage to follow the reference on the airspeed. So this is also something that could be done better.

8 ATTACHMENTS

8.1 Switching logic

```
def HitWavePoint(p,w):
    tresh = 10
    for i in range(len(p)):
        if not abs(p[i]-w[i]) < tresh:
            return False
    return True
```

8.2 Straight-line Algorithm

```
import numpy as np
from numpy import linalg as LA
import math


def SSA(angle):
    return (angle + np.pi) % (2 * np.pi) - np.pi

def StraightLine(p,w1,w2,chi_inf=math.pi/2,k_path=0.02):
    '''
    Input:
        w1      : Wavepoint 1     (3x1 array) (NED)
        w2      : Wavepoint 2     (3x1 array) (NED)
        p       : Start pos plane (3x1 array) (NED)

        chi_inf : Desired course angle far from path-line
                      Defult pi/2 (90 deg)
        k_path  : Gain to the sigmond funksion atan.
                      Desied when to follow chi_inf or chi_d
                      Defult 0.02


    Output:
        h_c     : Commanded hight (desired)
        chi_c   : Commanded coruse (desired)

    '''

    p = np.array(p)
    r = np.array(w1)
    q_vec = np.array(w2)-np.array(w1)

    ## Height:
    r = r.flatten()
    q_vec = q_vec.flatten()
    p = p.flatten()
    e_i_p = p-r
    q = q_vec/LA.norm(q_vec)
    k = np.array([0,0,1])
    cross_q_k = np.cross(q, k)
    n = cross_q_k / LA.norm(cross_q_k)

    s_i = e_i_p- np.outer(n, n)@ e_i_p

    s_d = math.sqrt(s_i[0]**2+s_i[1]**2)*( q[2]/ (    math.sqrt(  q[0]**2+q
    [1]**2)   ) )
    h_c =r[2]+s_d

    ## Course:
    chi_q = math.atan2(q[1],q[0])
    chi_q = SSA(chi_q)

```

```
52      e_py = -math.sin(chi_q)*(p[0]-r[0])+math.cos(chi_q)*(p[1]-r[1])
53      chi_c = chi_q - chi_inf*2/np.pi * math.atan(k_path*e_py)
54
55      return [h_c, chi_c,e_py]
```

8.3 Main Straight-line Algorithm

```
from lib.casadi_lib import state
from lib.casadi_lib import trim
from lib.filter.fir_lowpass import Fir_lowpass
from lib.wind.dryden import generateWindGust
import uav.models.X8 as model
import matplotlib.pyplot as plt
import numpy as np
import casadi as cs
from uav.controllers import PID
from uav import uav_v2
import importlib
importlib.reload(uav_v2)


from AddOnsPlott import *
from StraightLineAlg import *

from numpy import linalg as LA
from mpl_toolkits.mplot3d import Axes3D


def Main(wavepoints):
    # Set up simulation
    dt = 0.02
    t0 = 0
    tf = 800.0
    t = np.arange(t0, tf, dt)
    N = t.shape[0]
    space, frame, rotation = 'full', 'b', 'quat'
    idx = state.convert_idx_to_numpy(state.idx['full']['b']['quat'])
    wind_static_n = np.array([1, 1, 0])
    wind_gust_n = generateWindGust(dt, tf, turbulenceLevel=1)[0]
    w = wind_static_n + wind_gust_n

    # Initialize trimmed vehicle
    Va = 20
    gamma = 0*np.pi/180

    R = np.inf
    nx, nu, nw = state.get_dim_x_u_wind(space, frame, rotation)
    x = trim.trim(Va, R, gamma, model, print_level=5, wind=wind_static_n)
    x = state.convert_rotation[space][frame]['euler'](cs.SX(x), rotation)
    x = cs.DM(x).full().flatten()
    u = np.zeros(nu)

    uav = uav_v2.Vehicle(x, u, wind_static_n, model, {})
    delta_trim = x[-4:]

    # Design PID for speed/attitude
    P = model.P
    pidV = PID(0.69, 1.0, 0.0, -0.9, 0.9, 1.0, P['throttle_min'], P['
   throttle_max'])
```

```
52      piYaw = PID(2, 1.4, 0., -0.9, 0.9, 0.1, -30*np.pi/180, 30*np.pi/180)
53      pdRoll = PID(5, 0, 0.1, -0.9, 0.9, 0.1, P['aileron_min'], P['aileron_max
        '])
54      piHeight = PID(-0.2, -0.1, 0., -0.9, 0.9, 0.1,-0.5,0.5)
55      pdPitch = PID(-1.1, 0.0, -0.1, -.1, 0.1, 0.1, P['elevator_min'], P['
        elevator_max'])
56
57      # Plane start:
58      plane_start = uav.x[idx['pos']]
59
60      ref = dict()
61
62      ref['Va'] = uav.get('Va') * np.ones(N)
63
64      X = np.zeros((N, nx))
65      euler = np.zeros((N, 3))
66      euler_ref = np.zeros((N, 3))
67      delta = np.zeros((N, 4))
68
69      errors = np.zeros((N, 2))
70      hight = np.zeros((N,2))
71
72      waypoint = 0
73
74      print("\n\nSTART ::::::")
75
76      for k in range(t.shape[0]):
77          rpy = uav.get('euler')
78          roll, pitch, yaw = rpy[0], rpy[1], rpy[2]
79
80          wavepoint1 = wavepoints[waypoint]
81          wavepoint2 = wavepoints[waypoint+1]
82
83          [h_c,chi_c,ep_y] = StraightLine(uav.get('pos'),wavepoint1,wavepoint2
        )
84
85          ## Coruse(yaw) Control:
86          yawError = SSA(chi_c - yaw)
87          ref_roll = piYaw.update(t[k], yawError)
88          ddelta_a = pdRoll.update(t[k], ref_roll - roll)
89
90          ## Hight control:
91          hightError = h_c - uav.get('pos')[2]
92          ref_pitch = piHeight.update(t[k], hightError)
93          ddelta_e = pdPitch.update(t[k], ref_pitch - pitch)
94
95          ## Throttle:
96          ddelta_t = pidV.update(t[k], ref['Va'][k] - uav.get('Va'))
97
98          ## Output:
99          u = delta_trim + np.array((ddelta_a, ddelta_e, 0, ddelta_t))
100         u = np.clip(u,
101                     np.array((P['aileron_min'], P['elevator_min'], P['
        rudder_min'], P['throttle_min'])),
```

```
102                     np.array((P['aileron_max'], P['elevator_max'], P['
    rudder_max'], P['throttle_max'])))
103
104         ## Update UAV:
105         uav.update(uav.solver.t + dt, u, w[k, :])
106
107         euler[k, :] = rpy
108         euler_ref[k,:] = [ref_roll,ref_pitch,chi_c]
109
110         errors[k,:] = [ep_y,hightError]
111         hight[k,:] = [h_c,uav.get('pos')[2]]
112
113         X[k,:] = uav.x#uav.get('pos')#uav.x
114         delta[k, :] = u
115
116         if HitWavePoint(uav.get('pos'),wavepoint2):
117
118             print("Hit wavepoint {0}".format(waypoint+1))
119
120             if waypoint < len(wavepoints)-2:
121
122                 waypoint +=1
123
124
125
126     # NE and NED frame:
127
128     limits = SetLimits(X[:,0],X[:,1],X[:,2],wavepoints)
129     q_lines = GetQPlot(wavepoints)
130
131     fig = plt.figure()
132     ax=plt.subplot(1, 2, 1)
133     ax.set_title('North East frame')
134     for i in range(len(wavepoints)):
135         ax.plot(wavepoints[i][0],wavepoints[i][1], '.r', label='Wavepoint
    {0}'.format(i+1))
136     for i in range(len(q_lines)):
137         ax.plot(q_lines[i][0],q_lines[i][1], 'g', label='Path {0} (q)'.
    format(i+1))
138     ax.plot(plane_start[0],plane_start[1], '.b', label='Flight start (p)')
139     ax.plot(X[:,0],X[:,1],'m',label='Flight path')
140     ax.legend(loc='lower right')
141     ax.set_xlabel('East [m]')
142     ax.set_ylabel('North [m]')
143     ax.set_xlim([limits['xmin'],limits['xmax']])
144     ax.set_ylim([limits['ymin'],limits['ymax']])
145
146     ax = fig.add_subplot(1, 2, 2, projection='3d')
147     ax.set_title('NED frame')
148     for i in range(len(q_lines)):
149         ax.plot3D(q_lines[i][0],q_lines[i][1],q_lines[i][2], 'g', label='
    Path {0} (q)'.format(i+1))
150     ax.plot3D(X[:,0],X[:,1],X[:,2], 'm',label='Flight path')
151     ax.set_xlim([limits['xmin'],limits['xmax']])
152     ax.set_ylim([limits['ymin'],limits['ymax']])
```

```
153     ax.set_zlim([limits['zmin'],limits['zmax']])
154     ax.set_xlabel('North [m]')
155     ax.set_ylabel('East [m]')
156     ax.set_zlabel('-Down [m]')
157     ax.legend(loc='upper left')
158 
159     fig, ax = plt.subplots(3, 2, sharex=True)
160     ax[1, 0].plot(t,euler[:,2]* 180/np.pi, label=r'$\psi$')
161     ax[1, 0].plot(t,euler_ref[:,2]* 180/np.pi, label=r'$\chi^c$')
162     ax[1, 0].set_xlabel('t')
163     ax[1, 0].set_ylabel('deg')
164     ax[1, 0].legend(loc='upper right')
165 
166     ax[2, 1].plot(t,euler[:,1]* 180/np.pi, label=r'$\theta$')
167     ax[2, 1].plot(t,euler_ref[:,1]* 180/np.pi, label=r'$\theta_{ref}$')
168     ax[2, 1].set_xlabel('t')
169     ax[2, 1].set_ylabel('deg')
170     ax[2, 1].legend(loc='lower right')
171 
172     ax[2, 0].plot(t,euler[:,0]* 180/np.pi, label=r'$\phi$')
173     ax[2, 0].plot(t,euler_ref[:,0]* 180/np.pi, label=r'$\phi_{ref}$')
174     ax[2, 0].set_xlabel('t')
175     ax[2, 0].set_ylabel('deg')
176     ax[2, 0].legend(loc='upper right')
177 
178     ax[1, 1].plot(t,hight[:,1], label=r'$h$')
179     ax[1, 1].plot(t,hight[:,0], label=r'$h^c$')
180     ax[1, 1].set_xlabel('t')
181     ax[1, 1].set_ylabel('m')
182     ax[1, 1].legend(loc='lower right')
183 
184     ax[0, 1].plot(t,errors[:,1], label=r'$e_{h}$')
185     ax[0, 1].set_xlabel('t')
186     ax[0, 1].set_ylabel('Error')
187     ax[0, 1].legend(loc='lower right')
188 
189 
190 
191     ax[0, 0].plot(t,errors[:,0], label=r'$e_{py}$')
192     ax[0, 0].set_xlabel('t')
193     ax[0, 0].set_ylabel('Error')
194     ax[0, 0].legend(loc='lower right')
195 
196 
197     plt.show()
198 
199 
200 
201 
202 
203 if __name__ == '__main__':
204 
205     # Wavepoint 1:
206     w1 = [100,100,-200]
207     # Wavepoint 2:
```

```
208     w2 = [100,1100,-100]
209     # Wavepoint 3:
210     w3 = [1500,1100,-200]
211     # Wavepoint 4:
212     w4 = [1500,100,-200]
213 
214     w_all = [w1,w2,w3,w4,w1]
215 
216     Main(w_all)
```

8.4 Model MPC

```
import casadi as cs
import numpy as np
import math

def uav_model_t():
    # define structs
    constraint = cs.types.SimpleNamespace()
    model = cs.types.SimpleNamespace()
    model_name = "Thomas_NMPC"

    # To ddot{h}:
    zeta = 1
    omega_h = 0.5
    K_theta = 1.2
    Vaa = 20

    b_h_d = omega_h**2 / (K_theta*Vaa)
    b_h = 2*zeta*omega_h/(K_theta*Vaa)


    # To ddot{\chi}:
    zeta = 1
    omega_chi = 0.5
    Vgg = 24.2
    g = 9.81

    b_chi_d = omega_chi * Vgg/g
    b_chi = 2*zeta*omega_chi * Vgg/g

    # To dot{V_a}:
    b_va = 0.69


    def compute_path_reference(theta, wp):
        return wp[1] - theta*(wp[0] - wp[1])

    def compute_psi_chi(chi, Va, wind):
        return chi - cs.arcsin((-cs.sin(chi)*wind[0] + cs.cos(chi)*wind[1])/
    Va)

    nx = 17
    x = cs.SX.sym('x', nx)
    # e_pos = x[0:3]
    # de_pos = x[3:6]
    Va = x[6]
    dVa = x[7]
    Va_c = x[8]
    h = x[9]
    dh = x[10]
    h_c = x[11]
    chi = x[12]
    dchi = x[13]
```

```
    chi_c = x[14]
    # z1 = x[15]
    z2 = x[16]

    nu = 4
    u = cs.SX.sym('u', nu)
    dh_c   = u[0]
    dchi_c = u[1]
    dVa_c  = u[2]
    dz2    = u[3]

    nparam = 9
    p = cs.SX.sym('p', nparam)
    wp = [p[0:3], p[3:6]]
    wind = p[6:]

    dx = cs.SX.sym('dx', nx)

    # algebraic variables
    z = cs.vertcat([])

    psi = chi

    ddVa = b_va*(dVa_c - dVa) + b_va*(Va_c-Va)
    dwp = wp[0] - wp[1]
    dwx = dwp[0]
    dwy = dwp[1]
    dwz = dwp[2]

    f_expl = cs.vertcat(
        Va*cs.cos(psi) + wind[0] +z2*(wp[0][0] - wp[1][0]),
        Va*cs.sin(psi) + wind[1] +z2*(wp[0][0] - wp[1][0]),
        dh + z2*(wp[0][2] - wp[1][2]),
        dVa * cs.cos(psi) - Va*dchi * cs.sin(psi) + dz2 * dwx,
        dVa * cs.sin(psi) + Va*dchi * cs.cos(psi) + dz2 * dwy,
        b_h_d*(dh_c - dh) + b_h * (h_c - h) + dz2 * dwz,
        dVa,
        ddVa,
        dVa_c,
        dh,
        b_h_d*(dh_c-dh) + b_h * (h_c - h),
        dh_c,
        dchi,
        b_chi_d * (dchi_c - dchi) + b_chi * (chi_c - chi),
        dchi_c,
        z2,
        dz2
    )

    model.x0 = np.zeros(nx)

    # Define model struct
    model.f_impl_expr = dx - f_expl
    model.f_expl_expr = f_expl
    model.x = x
```

```
107     model.xdot = dx
108     model.u = u
109     model.z = z
110     model.p = p
111     model.name = model_name
112 
113     return model, constraint
```

8.5 Acados settings

```
from acados_template import AcadosModel, AcadosOcp, AcadosOcpSolver
import scipy.linalg
import numpy as np
from model_new import *

def acados_settings(Tf, N, x0, wp, wind):

    # create render arguments
    ocp = AcadosOcp()

    # export model
    model, constraint = uav_model_t()

    # define acados ODE
    model_ac = AcadosModel()
    model_ac.f_impl_expr = model.f_impl_expr
    model_ac.f_expl_expr = model.f_expl_expr
    model_ac.x = model.x
    model_ac.xdot = model.xdot
    model_ac.u = model.u
    model_ac.z = model.z
    model_ac.p = model.p
    model_ac.name = model.name
    ocp.model = model_ac


    nx = model.x.size()[0]
    nu = model.u.size()[0]
    ny = 12#
    ny_e = 8#
    ns = 0
    nsbx = 0
    nsh = 0

    ocp.dims.nx = nx
    ocp.dims.np = model.p.size()[0]
    ocp.dims.ny = ny
    ocp.dims.ny_e = ny_e
    ocp.dims.nsbx = 0
    ocp.dims.nu = nu
    ocp.dims.N = N
    ocp.dims.nsh = 0
    ocp.dims.nh = 0
    ocp.dims.ns = 0

    ocp.parameter_values = np.concatenate([wp[0], wp[1], wind])

    ## Gains Q and R
    # Q
    q1 = 100
    q2 = 100
    q3 = 100
```

```
53
54      q4 = 0.5*q1
55      q5 = 0.5*q2
56      q6 = 0.5*q3
57
58      q7 = 1000     # Va
59
60      q11 = 1 # Theta
61
62      # R
63      r1 = 1e-3
64      r2 = 1e-2
65      r3 =1e-3
66      r4 = 1e-3
67
68      # discretization
69      ocp.dims.N = N
70      Q = np.diag([q1,q2,q3,q4,q5,q6,q7,q11])
71      Qe = Q
72      R = np.diag([r1,r2,r3,r4])
73      ocp.cost.cost_type = "LINEAR_LS"
74      ocp.cost.cost_type_e = "LINEAR_LS"
75      unscale = N / Tf
76      ocp.cost.W = unscale * scipy.linalg.block_diag(Q, R)
77      ocp.cost.W_e = Qe / unscale
78
79      Vx = np.zeros([ny,nx])
80      Vx[0:6,0:6] = np.identity(6)
81      Vx[6,6] = 1
82      Vx[7, 15] = 1
83      ocp.cost.Vx = Vx
84
85      Vx_e = Vx[:ny_e,:]
86      ocp.cost.Vx_e = Vx_e
87
88      Vu = np.zeros([ny,nu])
89      Vu[8:ny,:]= np.identity(nu)
90      ocp.cost.Vu = Vu
91
92      ocp.cost.zl = 100 * np.ones((ns,))
93      ocp.cost.zu = 100 * np.ones((ns,))
94      ocp.cost.Zl = 0 * np.ones((ns,))
95      ocp.cost.Zu = 0 * np.ones((ns,))
96
97      # set intial references
98      ocp.cost.yref = np.array([0,0,0,0,0,0,20,0,0,0,0,0])
99      ocp.cost.yref_e = ocp.cost.yref[:8]
100
101     # setting constraints
102     ocp.constraints.lbx = np.array([])
103     ocp.constraints.ubx = np.array([])
104     ocp.constraints.idxbx = np.array([])
105
106     lbx = np.array([-1e3,
107                     -1e3,
```

```
                    -1e3,
                    -1e3,
                    -1e3,
                    -1e3,
                    10,
                    -10,
                    10,
                    0,
                    -10,
                    0,
                    -np.pi,
                    -np.pi,
                    -np.pi,
                    -1,
                    0])

    ubx = np.array([+1e3,
                    +1e3,
                    +1e3,
                    +1e3,
                    +1e3,
                    +1e3,
                    30,
                    +10,
                    30,
                    1e3,
                    10,
                    1e3,
                    +np.pi,
                    +np.pi,
                    +np.pi,
                    0,
                    1e3])

    ocp.constraints.lbx = lbx
    ocp.constraints.ubx = ubx
    ocp.constraints.idxbx = np.arange(nx)

    lbu = np.array([-10, -10, -10, -10])
    ubu = -lbu
    ocp.constraints.lbu = lbu
    ocp.constraints.ubu = ubu
    ocp.constraints.idxbu = np.arange(nu)

    ocp.dims.nbx = ocp.constraints.idxbx.shape[0]
    ocp.dims.nbu = ocp.constraints.idxbu.shape[0]

    # # set intial condition
    ocp.constraints.x0 = x0

    # set QP solver and integration
    ocp.solver_options.tf = Tf
    # ocp.solver_options.qp_solver = 'FULL_CONDENSING_QPOASES'
    ocp.solver_options.qp_solver = "PARTIAL_CONDENSING_HPIPM"
    ocp.solver_options.nlp_solver_type = "SQP_RTI"
```

```
163     ocp.solver_options.hessian_approx = "GAUSS_NEWTON"
164     ocp.solver_options.integrator_type = "IRK"
165     #ocp.solver_options.integrator_type = "ERK"
166     ocp.solver_options.sim_method_num_stages = 4
167     ocp.solver_options.sim_method_num_steps = 3
168     # ocp.solver_options.nlp_solver_step_length = 0.05
169     # ocp.solver_options.nlp_solver_max_iter = 1000
170     ocp.solver_options.tol = 1e-4
171     # ocp.solver_options.nlp_solver_tol_comp = 1e-1
172 
173     # create solver
174     acados_solver = AcadosOcpSolver(ocp, json_file="acados_ocp.json")
175 
176     return constraint, model, acados_solver
```

8.6 Main NMPC

```
from acados_template import AcadosModel, AcadosOcp, AcadosOcpSolver
import scipy.linalg
from casadi import *
import numpy as np
import math
import matplotlib.pyplot as plt
from acados_settings import *

from mpl_toolkits.mplot3d import Axes3D
# Set up mpc
Tf = 10.0  # prediction horizon
N = 50  # number of discretization steps
T = 50.00  # maximum simulation time[s]


# Waypoints
wp1 = np.array((100, 100, 100))
wp2 = np.array((400, 800, 100))

wp = [wp1, wp2]

wind = np.array((3, 3, 0))

Nsim = int(T * N / Tf)

# Initial position
pos_init = np.array((0, 600, 100))

# Airspeed reference
Va_ref = 20

Va_ref_dat = Va_ref * np.ones((Nsim, 1))

NameStates = ['en','ee','eh','en_dot','ee_dot','eh_dot','Va','Va_dot','
    Va_com','h','h_dot','h_com','chi','chi_dot','chi_com','z1','z2']

sim_res_state = {'t': 10,
        'en'      : np.zeros((Nsim, 1)),
        'ee'      : np.zeros((Nsim, 1)),
        'eh'      : np.zeros((Nsim, 1)),
        'en_dot'  : np.zeros((Nsim, 1)),
        'ee_dot'  : np.zeros((Nsim, 1)),
        'eh_dot'  : np.zeros((Nsim, 1)),
        'Va'      : np.zeros((Nsim, 1)),
        'Va_dot'  : np.zeros((Nsim, 1)),
        'Va_com'  : np.zeros((Nsim, 1)),
        'h'       : np.zeros((Nsim, 1)),
        'h_dot'   : np.zeros((Nsim, 1)),
        'h_com'   : np.zeros((Nsim, 1)),
        'chi'     : np.zeros((Nsim, 1)),
        'chi_dot' : np.zeros((Nsim, 1)),
        'chi_com' : np.zeros((Nsim, 1)),
```

```
52          'z1'       : np.zeros((Nsim, 1)),
53          'z2'       : np.zeros((Nsim, 1))}
54
55
56  NameInput = ['h_c_d','chi_c_d','Va_c','v']
57
58  sim_res_input = {'h_c': np.zeros((Nsim, 1)),
59          'h_c_d'    : np.zeros((Nsim, 1)),
60          'chi_c_d'  : np.zeros((Nsim, 1)),
61          'Va_c'     : np.zeros((Nsim, 1)),
62          'v'        : np.zeros((Nsim, 1))}
63
64  sim_res_pos = {'pe'   : np.zeros((Nsim, 1)),
65                 'pn'   : np.zeros((Nsim, 1)),
66                 'h'    : np.zeros((Nsim, 1)),
67                  'Va'  : np.zeros((Nsim, 1))}
68
69  def find_initial_path_variable(pos, wp):
70      return -1*(1 - max([0, np.inner(pos, wp[0])/np.inner(wp[1] - wp[0], wp
        [1] - wp[0])]))
71
72  def compute_path_reference(theta, wp):
73      return wp[1] - theta*(wp[0] - wp[1])
74
75  def compute_psi_chi(chi, Va, wind):
76      return chi - np.arcsin((-np.sin(chi)*wind[0] + np.cos(chi)*wind[1])/Va)
77
78  theta_init = find_initial_path_variable(pos_init, wp)
79  pos_ref_init = compute_path_reference(theta_init, wp)
80  e_init = pos_init - pos_ref_init
81
82  chi_init = math.atan2(wp[1][1]-pos_init[1], wp[1][0]-pos_init[0])
83  Va_init = Va_ref
84  psi_init = compute_psi_chi(chi_init, Va_init, wind)
85
86  chidot_init = 0
87  hdot_init = 0
88  Vadot_init = 0
89
90  pndot_init = Va_init*np.cos(psi_init) + wind[0]
91  pedot_init = Va_init*np.sin(psi_init) + wind[1]
92
93  edot_init = np.array((pndot_init, pedot_init, hdot_init))
94
95  h_init = pos_init[2]
96
97  h_c_init = h_init
98  Va_c_init = Va_init
99  chi_c_init = chi_init
100
101 z1_init = theta_init
102 z2_init = 0
103
104 x_init = np.array([e_init[0],
105                    e_init[1],
```

```
106                     e_init[2],
107                     edot_init[0],
108                     edot_init[1],
109                     edot_init[2],
110                     Va_init,
111                     Vadot_init,
112                     Va_c_init,
113                     h_init,
114                     hdot_init,
115                     h_c_init,
116                     chi_init,
117                     chidot_init,
118                     chi_c_init,
119                     z1_init,
120                     z2_init])
121
122 ref_init = compute_path_reference(theta_init, wp)
123
124 constraints, model, acados_solver = acados_settings(Tf, N, x_init, wp, wind)
125 acados_solver.set(0, "x", x_init)
126
127 # simulate
128 Vx = acados_solver.acados_ocp.cost.Vx
129 Vx_e = acados_solver.acados_ocp.cost.Vx_e
130 Vu = acados_solver.acados_ocp.cost.Vu
131 W = acados_solver.acados_ocp.cost.W
132 W_e = acados_solver.acados_ocp.cost.W_e
133 y_init = Vx @ x_init
134
135 epos = np.zeros((Nsim, 3))
136 pos = np.zeros((Nsim, 3))
137 ref_pos = np.zeros((Nsim, 3))
138 for i in range(Nsim):
139     print("Finished  {}/{}.".format(i, Nsim))
140     for j in range(N):
141         yref = np.array([0,0,0,0,0,0,Va_ref,0,0,0,0,0])
142         acados_solver.set(j, "yref", yref)
143
144     yref_N = np.array([0,0,0,0,0,0,Va_ref,0])
145     acados_solver.set(N, "yref", yref_N)
146
147     # solve ocp
148     status = acados_solver.solve()
149
150     if status != 0:
151         print("acados returned status {} in closed loop iteration {}.".
    format(status, i))
152         break
153
154
155     # get solution
156     x0 = acados_solver.get(0, "x")
157     u0 = acados_solver.get(0, "u")
158
159
```

```
160     for ver,name in enumerate(NameStates):
161         sim_res_state[name][i] = x0[ver]
162 
163     for ver,name in enumerate(NameInput):
164         sim_res_input[name][i] = u0[ver]
165 
166     # update initial condition
167     x0 = acados_solver.get(1, "x")
168 
169 
170     epos[i, :] = x0[:3]
171     theta = x0[15]
172     ref_pos[i, :] = compute_path_reference(theta, wp)
173     pos[i, :] = epos[i, :] + ref_pos[i, :]
174 
175     acados_solver.set(0, "lbx", x0)
176     acados_solver.set(0, "ubx", x0)
177 
178 
179 fig = plt.figure()
180 ax=plt.subplot(1, 2, 1)
181 
182 #fig, ax = plt.subplots()
183 ax.plot([wp[0][1], wp[1][1]], [wp[0][0], wp[1][0]])
184 ax.plot(ref_pos[:, 1], ref_pos[:, 0])
185 ax.plot(pos[:, 1], pos[:, 0])
186 ax.set_title('North East frame')
187 ax.legend(['Path', 'reference', 'Flight path'])
188 ax.set_xlabel('East [m]')
189 ax.set_ylabel('North [m]')
190 
191 ax = fig.add_subplot(1, 2, 2, projection='3d')
192 ax.set_title('NED frame')
193 ax.plot([wp[0][1], wp[1][1]], [wp[0][0], wp[1][0]], [wp[0][2], wp[1][2]])
194 ax.plot(ref_pos[:, 1], ref_pos[:, 0], ref_pos[:, 2])
195 ax.plot(pos[:, 1], pos[:, 0], pos[:, 2])
196 ax.legend(['Path', 'reference', 'Flight path'])
197 ax.set_xlabel('North [m]')
198 ax.set_ylabel('East [m]')
199 ax.set_zlabel('Down [m]')
200 
201 
202 fig, ax = plt.subplots(2, 2)
203 ax[0, 0].set_title(r'$e_n$')
204 ax[0, 0].plot(sim_res_state['en'],'b')#,label='en')
205 ax[0, 1].set_title(r'$e_e$')
206 ax[0, 1].plot(sim_res_state['ee'],'b')#,label='ee')
207 ax[1, 0].set_title(r'$e_h$')
208 ax[1, 0].plot(sim_res_state['eh'],'b')#,label='eh')
209 
210 fig.suptitle('Error', fontsize=16)
211 fig, ax = plt.subplots(2, 2)
212 ax[0, 0].set_title(r'$\dot{e_n}$')
213 ax[0, 0].plot(sim_res_state['en_dot'],'b')#,label='en_dot')
214 ax[0, 1].set_title(r'$\dot{e_e}$')
```

```
215 ax[0, 1].plot(sim_res_state['ee_dot'],'b')#,label='ee_dot')
216 ax[1, 0].set_title(r'$\dot{e_h}$')
217 ax[1, 0].plot(sim_res_state['eh_dot'],'b')#,label='eh_dot')
218 fig.suptitle('Error dot', fontsize=16)
219
220 fig, ax = plt.subplots(3, 3)
221 ax[0, 0].set_title(r'$V_a$')
222 ax[0, 0].plot(sim_res_state['Va'],'b',label='Va')
223 ax[0, 0].plot(Va_ref_dat,'g', label='ref')
224 ax[0, 0].legend(loc='lower right')
225 ax[0, 1].set_title(r'$\dot{V}_a$')
226 ax[0, 1].plot(sim_res_state['Va_dot'],'b',label='Va')
227 ax[0, 2].set_title(r'$V_a^c$')
228 ax[0, 2].plot(sim_res_state['Va_com'],'b',label='Va')
229 ax[1, 0].set_title('h')
230 ax[1, 0].plot(sim_res_state['h'],'b')#,label='h')
231 ax[1, 1].set_title(r'$\dot{h}$')
232 ax[1, 1].plot(sim_res_state['h_dot'],'b')#,label='h_dot')
233 ax[1, 2].set_title(r'$h^c$')
234 ax[1, 2].plot(sim_res_state['h_com'],'b')#,label='h_com')
235 ax[2, 0].set_title(r'$\chi$')
236 ax[2, 0].plot(sim_res_state['chi'],'b')#,label='chi')
237 ax[2, 1].set_title(r'$\dot{\chi}$')
238 ax[2, 1].plot(sim_res_state['chi_dot'],'b')#,label='chi_dot')
239 ax[2, 2].set_title(r'$\chi^c$')
240 ax[2, 2].plot(sim_res_state['chi_com'],'b')#,label='chi_com')
241
242 fig.suptitle('State', fontsize=16)
243 fig, ax = plt.subplots(nrows=2, ncols=1)
244 ax[0].set_title(r'$z_1$')
245 ax[0].plot(sim_res_state['z1'],'b')#,label='z1')
246
247 ax[1].set_title(r'$z_2$')
248 ax[1].plot(sim_res_state['z2'],'b')#,label='z2')
249
250 fig.suptitle('Virtual state', fontsize=16)
251
252 fig, ax = plt.subplots(2, 2)
253 ax[0, 0].set_title(r'$\dot{h^c}$')
254 ax[0, 0].plot(sim_res_input['h_c_d'],'b')#,label='h_c_d')
255 ax[0, 1].set_title(r'$\dot{\chi^c}$')
256 ax[0, 1].plot(sim_res_input['chi_c_d'],'b')#,label='chi_c_d')
257 ax[1, 0].set_title(r'$\dot{V}_a^c$')
258 ax[1, 0].plot(sim_res_input['Va_c'],'b')#,label='Va_c')
259 ax[1, 1].set_title('v')
260 ax[1, 1].plot(sim_res_input['v'],'b')#,label='v')
261
262 fig.suptitle('Input', fontsize=16)
263
264 plt.show()
```

References

[1] R.W Beard and McLain T.W. *Small Unmanned Aircraft : Theory and Practice*. Princeton University Press, 2012.

[2] Thor I Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley, Hoboken, 1. aufl. edition, 2011.

[3] Bjarne Foss and Tor Aksel N. Heirung. *Merging Optimization and Control*. 03 2016.

[4] Jorge Nocedal. Numerical optimization, 2006.

[5] Robin Verschueren, Gianluca Frison, Dimitris Kouzoupis, Niels van Duijkeren, Andrea Zanelli, Branimir Novoselnik, Jonathan Frey, Thivaharan Albin, Rien Quirynen, and Moritz Diehl. acados: a modular open-source framework for fast embedded optimal control. *arXiv preprint*, 2019.

[6] Timm Faulwasser. *Optimization-based solutions to constrained trajectory-tracking and path-following problems*. 01 2013.