

dopsat: – carkovaney graf saturation – subsection Simulation obrazek simulace

## 1 Introduction

Unmanned aerial vehicles (UAV) have become very popular in the last few years. It is mainly thanks to the multicopters, especially quadcopters and their dropping prizes. They are used in many areas. They allow many amateur film makers to capture aerial shots and often substitute expensive helicopters in professional movie shooting. Many companies, such as DJI, have developed quadcopters for professional and amateur film making as shown in Fig. 1. The use of quadcopters is also in military industry mainly for search and rescue missions. Some quadcopters, carrying defibrillators, are used for medical assistance in cases of heart failures, for their much faster reaction time than ambulance vehicles. The future use of quadcopters is almost limitless, from monitoring cities, delivering packages, to collecting the air condition data.

Quadcopters body is simple compared with classical helicopter. It has rigid body and four propellers with a fixed pitch angles. In this thesis, the term UAV will refer to a quadcopter.

Many quadcopters possess global position system (GPS). This allows better outdoor control of the aircraft, such as position hold. Quadcopters are usually directly controlled by people, but in many applications an ability of autonomous flight is welcomed. For example if the UAV is too far from the operator, it can return to the takeoff place using GPS and even land. This mechanism has been developed by the company DJI.

A system Vicon, using outside cameras, has been developed for more precise localization. This localization uses wireless communication between the on the ground computer and the UAV. The UAV's controller is usually also implemented in the on the ground computer, because of the computational demands. Because of this, UAV can not fly outside the laboratory.

There has been a demand for on board controller, working outside the lab that depends only on the on board computer and sensors. Some high level unit plan trajectories for one or multiple UAV's. For example for movie shooting, just trajectory is given by this unit. It is not designed to work with a specific UAV model. The MPC tracks this trajectory and if there is and obstacle in the way, it prevents collision.

### 1.1 Problem statement

The task of this thesis is to design and implement a model predictive controller capable of precise trajectory tracking and obstacle avoidance. The trajectory will be given by higher level planning unit and the UAV will track the trajectory, unless obstacles appear. In this case, the MPC automatically



**Figure 1:** DJI Phantom 3

avoids the obstacles. This is useful for swarms of UAVs and other trajectory planning algorithms, which do not have to take into consideration either the actual control of the particular UAV or avoiding unexpected obstacles. The appropriate input actions will be found as a mathematical optimization of linearly constrained quadratic function.

The UAV will be fully autonomous with the localization, obstacle detection and computing happening solely on board. This allows the UAV to operate outside a laboratory conditions.

## 1.2 Previous work

Since 2011, the laboratory of Intelligent and Mobile Robotics (IMR) group at FEE CTU has been developing UAV controllers. At the early stages the group concentrated on simulations of swarms and formations. In the following years hardware platforms were developed.

The hardware platform, which is further described in Sec. 9.2, was developed in [?]. This platform was used for real flight experiments. An MPC, capable of tracking a trajectory was developed for this platform, but it was not designed to avoid obstacles. For obstacle avoidance different approaches have to be used mainly in the mathematical optimization part.

A system for sensor px4flow [?] allowes to measure speed and, combined with a previously implemented state observer[?], allows position measurements.

A system WhyCon[?][?] is used for obstacle detection. It allows to detect multiple circular markers, shown in Fig. ??, used in the experiments to represent obstacles.

### 1.3 Related work

Many laboratories have been developing different obstacle avoidance systems. Most of them develop precise trajectory tracking. If an obstacle appears, a higher level planning unit updates the desired trajectory.

Such a system can be used for on board 3D mapping and obstacle avoidance[?] at the same time. This system is used mainly for mapping unknown areas.

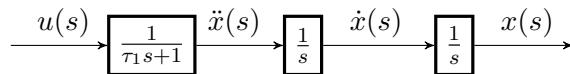
Obstacles do not have to be detected by the cameras, ultrasonic sensors can be used instead. A system for indoor obstacle avoidance[?] has been developed to allow computer assisted maneuvering in static indoor environment.

Another obstacle avoidance system is based on the potential field principle, which is a popular model for swarm control[?]. In this model, the acceleration of the UAV is the sum of fictional forces, which repulse the UAV from obstacles and push it towards the desired position.

Not all obstacle avoidance systems are designed for multicopters. A flapping wing micro areal vehicle(MAV) system has been also equipped by an on board obstacle avoidance system [?], using trajectory updating.

## 2 UAV dynamics

For understanding the UAV's controller, it is important to understand the physical properties of the UAV. The UAV has 6 degrees of freedom which are the position  $x, y, z$  and the rotation pitch( $\theta$ ), roll ( $\psi$ ) and yaw( $\phi$ ). The UAV is equipped with the KK2 board providing basic stabilization. The inputs, that are used in this thesis are roll and pitch rotation references. After linearisation at the equilibrium at the horizontal position, with rotations close to the equilibrium, it can be said, that  $\psi \propto \ddot{x}$  and  $\theta \propto -\ddot{y}$ , which are the accelerations in each axis. The KK2's inputs are desired  $\psi$  and  $\theta$  and output is the voltage on the individual motors. This controller provides a time-invariant linear system. The continuous transfer diagram is shown in the Fig. 2



**Figure 2:** Diagram of the continuous system.

The input action  $u(s)$ , as a from the MPC, is given to the KK2 and transformed to the the UAV's tilt, which corresponds to the real acceleration  $\ddot{x}(s)$ , and then twice integrated to the position  $x(s)$ .

## 2.1 Coordinate systems

This whole thesis will consider 2D coordinate system only. In these 2 dimensions, there is an aileron axis  $x$  with the direction to the right and the elevator axis  $y$  with the direction forwards. These both axes are perpendicular. There are two coordinate systems which will be used. The first system is a standard world coordinate system W with the origin usually at the starting point of the UAV. The second system is a coordinate system U, which is a system with the origin in the center of the mass of the UAV. This is for example the coordinate system of detecting obstacles, which are observed relatively to the UAV by on-board cameras. Coordinate system U is created only by the translation of the system W by the vector  $\vec{r} = (\Delta x, \Delta y)$ . There is no rotation between the two coordinate systems, so the axes of the both coordinate systems are parallel. Because the UAV can move easily along each axis, there is no need to introduce the UAV's rotation. If the UAV rotated over time, the model would no longer be linear and the control of this system would be much more complex, therefore slower. These coordinate systems transformations follow equations

$$\begin{aligned} x^{(W)} &= x^{(U)} + \Delta x \\ y^{(W)} &= y^{(U)} + \Delta y \\ \dot{x}^{(W)} &= \dot{x}^{(U)} + \Delta \dot{x} \\ \dot{y}^{(W)} &= \dot{y}^{(U)} + \Delta \dot{y} \\ \ddot{x}^{(W)} &= \ddot{x}^{(U)} + \Delta \ddot{x} \\ \ddot{y}^{(W)} &= \ddot{y}^{(U)} + \Delta \ddot{y}. \end{aligned} \tag{1}$$

In the following text the world coordination system W will be used unless stated otherwise. The UAV has its own proportions. However, it is complicated to take into consideration the whole UAV's body. It is easier to proximate the UAV's body with a single mass point in its center. The orientation of the UAV is constant and will not be taken into consideration, thus the position of the UAV can be described by two coordinates.

The altitude of the UAV is controlled separately. There are several reasons to do that. The first reason is, that many applications treat the altitude differently and enforce constant altitude for long periods of time. For example, in building interiors, the altitude is constant most of the time. The desired trajectory is also usually given in 2D and the altitude is described separately. The second reason is, as mentioned above, an MPC is very demanding on computing time. To save computational capacity, a standard PID controller can be used instead of an MPC for controlling the altitude. The altitude PID controller has already been implemented [?] and it is not a part of this thesis.

## 2.2 State observer

An MPC is very sensitive to a data noise and errors. Inaccurate initial condition can result in a bad prediction because of the double integration of acceleration into position. To work properly, the MPC requires very accurate initial conditions. These are position, speed and acceleration in both axes. A Kalman estimator has been implemented[?] to estimate states of the UAV including disturbances of the acceleration. The inputs of the state observer are the system's input action and measured velocity by a camera sensor.

## 2.3 One axis model

The UAV model has been analyzed[?]. Thanks to the symmetrical body of the UAV, both axes share the same model. For aileron axis, the states take form of  $\vec{x}_x = (x_x, \dot{x}_x, \ddot{x}_x)^T$  and  $\vec{x}_y = (x_y, \dot{x}_y, \ddot{x}_y)^T$  for elevator axis, where  $x_x$  is the aileron and  $x_y$  is the elevator position. The aileron and elevator models are mathematically identical. The discrete state space model takes form of system matrices  $\mathbf{A}_s, \mathbf{B}_s$  as

$$\vec{x}_{x,y,[t+1]} = \mathbf{A}_s \vec{x}_{x,y,[t]} + \mathbf{B}_s u_{x,y,[t]}, \quad (2)$$

where  $u_{x,y,[t]}$  is an input at time  $t$ , sampling with the frequency of  $1/\Delta t = 70$  Hz.

$$\mathbf{A}_s = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & p_1 \end{bmatrix}, \mathbf{B}_s = \begin{bmatrix} 0 \\ 0 \\ p_2 \end{bmatrix}, \quad (3)$$

where  $p_1 = 0.9799$  and  $p_2 = 5.0719 \cdot 10^{-5}$ . The unconstrained MPC can be solved separately for elevator and aileron axis. Simple constraints, such as input saturation can be applied.

## 3 Model predictive control formulation

An MPC is an advanced regulator. It uses prediction of future states of the system to determine system input actions. This prediction runs constantly in a loop. Because of its computational demands, it is used mainly in processes with long time constants. A motivation for its development was to control various chemical processes, where the computational time was not limiting. Using an MPC for controlling a UAV is a challenge, because it is hard to implement on embedded hardware. Controlling a real time system, such as UAV, requires regulation in tens of Hz, giving the hardware very little time to compute such a complex problem.

An MPC requires the system's state space model, initial condition and, unlike other controllers, a sequence of desired future states. An advantage

of an MPC is applying large variety of constraints, which can be useful for example in the regulation of chemical processes. On the other hand, an MPC is sensitive to the model's inaccuracy and to sensory noise.

Output of an MPC is not only the desired input action for the next time step, but also predicted input actions and predicted behavior of the system in the whole prediction horizon for the  $T$  following time steps. Unlike standard controllers like a PID, an MPC can adjust the input action based on future demands.

The MPC controller for tracking desired trajectory was developed [?]. The goal of the thesis is to extend the controller to be capable of obstacle avoidance by using linearly constrained quadratic optimization. Obstacles are not previously known and can change position.

### 3.1 Formulation of quadratic programming

MPC formulates optimization problem, that then has to be solved. The problem takes usually form of Linear Programming (LP) or, in this case, Quadratic Programming (QP). A vector objective function has to be defined

$$V(\underline{\mathbf{x}}, \underline{\mathbf{u}}) = \frac{1}{2} \sum_{i=0}^T \mathbf{e}_{[i]}^T \mathbf{Q} \mathbf{e}_{[i]} + \mathbf{u}_{[i]}^T \mathbf{P} \mathbf{u}_{[i]} \quad (4)$$

There are several ways to solve this problem, which will be discussed later.

### 3.2 Extended model

For complex, such as position constraints for obstacle avoidance, position of UAV in one axis is a function of the position in the other axis. For these kinds of constraints axes can not be treated separately and more complicated system description is needed. The state space system must be preserved, connecting both identical systems for each axis into one system extending Eq. (5) into

$$\mathbf{x}_{[t+1]} = \mathbf{A}\mathbf{x}_{[t]} + \mathbf{B}\mathbf{u}_{[t]}, \quad (5)$$

where  $\mathbf{u}_{[t]} = (u_{x,[t]}, u_{y,[t]})^T$  is input vector containing elevator and aileron system inputs at the time  $t$ . Extended state vector

$$\mathbf{x}_{[t]} = (x_{[t]}, \dot{x}_{[t]}, \ddot{x}_{[t]}, y_{[t]}, \dot{y}_{[t]}, \ddot{y}_{[t]})^T \quad (6)$$

contains positions  $x, y$  and their derivatives at the time  $t$ .

By connecting these 2 systems, we get the following state space matrices of the whole system

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_s & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_s \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_s & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_s \end{bmatrix}. \quad (7)$$

### 3.3 System prediction

The MPC algorithm is based on predicting future states based on initial condition and system input. Such a general equation can be derived from the Eq. (5). With a simple substitution we can get prediction of the states at the time  $t = 2$ :

$$\begin{aligned}\mathbf{x}_{[1]} &= \mathbf{A}\mathbf{x}_{[0]} + \mathbf{B}\mathbf{u}_{[0]}, \\ \mathbf{x}_{[2]} &= \mathbf{A}\mathbf{x}_{[1]} + \mathbf{B}\mathbf{u}_{[1]} \\ &= \mathbf{A} \cdot (\mathbf{A}\mathbf{x}_{[0]} + \mathbf{B}\mathbf{u}_{[0]}) + \mathbf{B}\mathbf{u}_{[1]} \\ &= \mathbf{A}^2\mathbf{x}_{[0]} + \mathbf{AB}\mathbf{u}_{[0]} + \mathbf{B}\mathbf{u}_{[1]}.\end{aligned}\tag{8}$$

The Eq. (8) can be rewritten in a more general way:

$$\mathbf{x}_{[t]} = \mathbf{A}^t\mathbf{x}_{[0]} + \sum_{i=1}^{t-1} \mathbf{A}^i \mathbf{B} \mathbf{u}_{[i-1]} + \mathbf{B} \mathbf{u}_{[0]}.\tag{9}$$

Let's combine the sequence of predicted states into one vector  $\underline{\mathbf{x}} = (\mathbf{x}_{[1]}^T, \mathbf{x}_{[2]}^T, \dots, \mathbf{x}_{[T]}^T)^T$  and the sequence of inputs into

$$\underline{\mathbf{u}} = (\mathbf{u}_{x,[0]}, \mathbf{u}_{y,[0]}, \mathbf{u}_{x,[1]}, \mathbf{u}_{y,[1]}, \dots, \mathbf{u}_{x,[T-1]}, \mathbf{u}_{y,[T-1]})^T.\tag{10}$$

With this notation, Eq. (9) can be represented as a simple matrix multiplication

$$\underbrace{\begin{bmatrix} \mathbf{x}_{[1]} \\ \mathbf{x}_{[2]} \\ \vdots \\ \mathbf{x}_{[T]} \end{bmatrix}}_{\underline{\mathbf{x}}} = \underbrace{\begin{bmatrix} \mathbf{A} \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^{(T-1)} \end{bmatrix}}_{\hat{\mathbf{A}}} \mathbf{x}_{[0]} + \underbrace{\begin{bmatrix} \mathbf{B} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{AB} & \mathbf{B} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{(T-1)}\mathbf{B} & \mathbf{A}^{(T-2)}\mathbf{B} & \dots & \mathbf{B} \end{bmatrix}}_{\hat{\mathbf{B}}} \cdot \underbrace{\begin{bmatrix} \mathbf{u}_{[0]} \\ \mathbf{u}_{[1]} \\ \vdots \\ \mathbf{u}_{[T-1]} \end{bmatrix}}_{\underline{\mathbf{u}}}.\tag{11}$$

Using new notation, this equation can be rewritten in a simpler form as

$$\underline{\mathbf{x}} = \hat{\mathbf{A}} \mathbf{x}_{[0]} + \hat{\mathbf{B}} \underline{\mathbf{u}}.\tag{12}$$

## 4 MPC implementation

As mentioned in the Sec. 3, MPC formulates a quadratic optimization problem. Let's first solve the problem, where the obstacles are not involved.

## 4.1 Trajectory

For MPC to compute input actions, there must be given a desired trajectory  $\underline{\mathbf{x}}_d = (\mathbf{x}_{d,[1]}^T, \mathbf{x}_{d,[2]}^T, \dots, \mathbf{x}_{d,[T]}^T)^T$ , where the desired state at the time  $t$  is  $\mathbf{x}_{d,[t]} = (x_{d,[t]}, \dot{x}_{d,[t]}, \ddot{x}_{d,[t]}, y_{d,[t]}, \dot{y}_{d,[t]}, \ddot{y}_{d,[t]})^T$ . These desired states contain, besides aileron and elevator position, also velocity and acceleration. This gives the MPC a chance to enforce other properties besides position. However, The UAV desired velocity is already given by the desired positions at certain time as the distance

$$d = \sqrt{(x_{d,[t]} - x_{d,[t+1]})^2 + (y_{d,[t]} - y_{d,[t+1]})^2}. \quad (13)$$

The same can be applied for acceleration. Therefore, the velocity and acceleration are demanded in the sequence of desired positions, rather than the desired states. The desired velocity and acceleration are then ignored and it can hold any value, for example 0. The  $\mathbf{x}_{d,[t]}$  then takes form of  $\mathbf{x}_{d,[t]} = (x_{d,[t]}, 0, 0, y_{d,[t]}, 0, 0)^T$ . When creating the desired trajectory, one should always keep in mind, that the desired positions hold also information about velocity and acceleration.

## 4.2 Objective function

As in many other areas of engineering, the problem can be solved in two independent steps. Creating an objective function and then minimize it. The objective function's (sometimes called cost function) value describes, how good is the particular solution from its domain. In this case the solution takes form of a vector  $\underline{\mathbf{u}}$ , which can be directly transformed into predicted trajectory. The lower is the value of the objective function, the better the particular solution is. The solution with the minimal objective function's value is called an optimal solution.

The goal of unconstrained MPC is to track a given trajectory as well as possible. This means, that we want to minimize the distance between all the predicted and the desired positions. This error in both axes can be computed as

$$\begin{aligned} e_{x,t} &= x_{[t]} - x_{d,[t]} \\ e_{y,t} &= y_{[t]} - y_{d,[t]}. \end{aligned} \quad (14)$$

Using this kind of error has many downsides. The obvious one is, that it penalizes error in only one direction and favors the other. If we want to use the distance, we would have to take the absolute value. However this function would not be differentiable [?] and solving this task would be more complicated. Differentiability is a very useful property. Being able to compute gradient of the final objective function results in fast task solving. Above that we don't get very good results if penalizing the error linearly.

From the experience, it has come beneficial in many ways to use the square of the error. This preserves the condition of not prioritizing one direction error. The function is also easily differentiable. The next great advantage is penalizing big distances disproportionately more and ignoring very small errors. This also describes our requirements, where the exact tracking of the trajectory is not as important as eliminating big deviations from the trajectory. If a simple sum of  $e_{x,t}^2$  and  $e_{y,t}^2$  was applied, the system would behave very wildly, generating very high input actions to correct the error. However, this is not in the capabilities of the real system and could result in an unstable control. Therefore input actions must be also penalized. When combined together, we get the following objective function

$$V(\mathbf{e}_x, \mathbf{e}_y, \underline{\mathbf{u}}) = \frac{1}{2} \sum_{i=1}^T \left( k_q \cdot (e_{x,[i]}^2 + e_{y,[i]}^2) + k_s \cdot (\mathbf{u}_{x,[i-1]}^2 + \mathbf{u}_{y,[i-1]}^2) \right), \quad (15)$$

where  $\mathbf{e}_x = (e_{x,[1]}, e_{x,[2]}, \dots, e_{x,[T]})^T$  and  $\mathbf{e}_y = (e_{y,[1]}, e_{y,[2]}, \dots, e_{y,[T]})^T$ . The ratio of the constants  $k_q$  and  $k_s$  is the only parameter of the MPC and determines how wildly the system behaves. The error at the time  $t = 0$  is determined only by the initial condition and doesn't depend on the input action  $\underline{\mathbf{u}}$ . Because the function is to be optimized, this error can be left out. The Eq. (15) can be rewritten in a matrix form using penalizing matrices  $\mathbf{Q}$  and  $\mathbf{P}$

$$V(\underline{\mathbf{e}}, \underline{\mathbf{u}}) = \frac{1}{2} \sum_{i=1}^T \left( \mathbf{e}_{[i]}^T \mathbf{Q} \mathbf{e}_{[i]} + \mathbf{u}_{[i-1]}^T \mathbf{P} \mathbf{u}_{[i-1]} \right), \quad (16)$$

where  $\underline{\mathbf{e}} = \underline{\mathbf{x}} - \underline{\mathbf{x}}_d$  is the error of all states at the time  $t$  and matrices  $\mathbf{Q}$  and  $\mathbf{P}$  are

$$\mathbf{Q} = \begin{bmatrix} k_q & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & k_q & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{P} = \begin{bmatrix} k_p & 0 \\ 0 & k_p \end{bmatrix}. \quad (17)$$

This form of  $\mathbf{Q}$  allows to penalize only position errors and ignore the velocity and acceleration errors. To ensure, that the function  $V(\underline{\mathbf{x}}, \underline{\mathbf{u}})$  is strictly convex, the matrix  $\mathbf{Q}$  must be positive semi-definite ( $\mathbf{Q} \succeq 0$ ) and  $\mathbf{P}$  must be positive definite ( $\mathbf{P} \succ 0$ ). The matrix  $\mathbf{Q}$  has its eigenvalues 0 and  $k_q$ , therefore  $k_q \geq 0$ . The eigenvalues of  $\mathbf{P}$  are  $k_p$ , so  $k_p > 0$ . In some MPC algorithms the last error is penalized with higher weight, forcing the system to end up in the last state more. It turned out, this can be counterproductive in obstacle avoidance system for reasons that will be discussed later.

We want to get rid of the sum in Eq. (16) and substitute it with matrix multiplication. Let's first introduce the matrices  $\hat{\mathbf{Q}}$  and  $\hat{\mathbf{P}}$  as

$$\hat{\mathbf{Q}} = \begin{bmatrix} \mathbf{Q} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{Q} & \dots & \vdots \\ \mathbf{0} & \dots & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & \mathbf{Q} \end{bmatrix}, \hat{\mathbf{P}} = \begin{bmatrix} \mathbf{P} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{P} & \dots & \vdots \\ \mathbf{0} & \dots & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & \mathbf{P} \end{bmatrix}. \quad (18)$$

If the last error should be penalized more, the last matrix  $\mathbf{Q}$  on the diagonal of the matrix  $\hat{\mathbf{Q}}$  would consist of different constants  $k_q$ . Lets rewrite the Eq. (16) using the Eq. (12):

$$\begin{aligned} J(\underline{\mathbf{u}}) &= \frac{1}{2} \left( \underline{\mathbf{e}}^T \hat{\mathbf{Q}} \underline{\mathbf{e}} + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} \right) \\ &= \frac{1}{2} \left( (\hat{\mathbf{A}} \mathbf{x}_{[0]} + \hat{\mathbf{B}} \underline{\mathbf{u}} - \tilde{\mathbf{x}})^T \hat{\mathbf{Q}} (\hat{\mathbf{A}} \mathbf{x}_{[0]} + \hat{\mathbf{B}} \underline{\mathbf{u}} - \tilde{\mathbf{x}}) + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} \right) \\ &= \frac{1}{2} \left( 2(\hat{\mathbf{Q}} \hat{\mathbf{B}})^T \hat{\mathbf{A}} \mathbf{x}_{[0]} \underline{\mathbf{u}} + \underline{\mathbf{u}}^T \hat{\mathbf{B}}^T \hat{\mathbf{Q}} \hat{\mathbf{B}} \underline{\mathbf{u}} - 2(\hat{\mathbf{Q}} \hat{\mathbf{B}})^T \tilde{\mathbf{x}} \underline{\mathbf{u}} + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} + const. \right) \end{aligned} \quad (19)$$

In the Eq. (19), all constant elements have been put into the *const* part. Because the goal is to minimize the objective function  $J(\underline{\mathbf{u}})$ , parts of the equation, that do not depend on the  $\underline{\mathbf{u}}$  can be left out. The Eq. (19) can be rewritten as

$$J(\underline{\mathbf{u}}) = \frac{1}{2} \underline{\mathbf{u}}^T \underbrace{(\hat{\mathbf{B}}^T \hat{\mathbf{Q}} \hat{\mathbf{B}} + \hat{\mathbf{P}})}_{\hat{\mathbf{H}}} \underline{\mathbf{u}} + \underbrace{(\hat{\mathbf{Q}} \hat{\mathbf{B}})^T (\hat{\mathbf{A}} \mathbf{x}_{[0]} - \tilde{\mathbf{x}})}_{\hat{\mathbf{c}}} \underline{\mathbf{u}}. \quad (20)$$

The task can be formulated as

$$\begin{aligned} \min_{\underline{\mathbf{u}} \in \mathbb{R}^{k_M}} \quad J(\underline{\mathbf{u}}) &= \frac{1}{2} \underline{\mathbf{u}}^T \hat{\mathbf{H}} \underline{\mathbf{u}} + \hat{\mathbf{c}} \underline{\mathbf{u}} \\ \text{s.t.} \quad \mathbf{A}_c \underline{\mathbf{u}} &\leq \mathbf{B}_c. \end{aligned} \quad (21)$$

The constraints will be further discussed in section 4.4. For unconstrained system we can ignore them. Solving unconstrained MPC task is described in the following section.

### 4.3 Solving QP unconstrained

We want to solve the problem 21 while ignoring the constraints. For this task to have a single optimal solution  $\underline{\mathbf{u}}^*$ , the objective function has to be convex. Therefore the matrix  $\hat{\mathbf{H}}$  has to be positive semi definite. This is guaranteed, because the matrices  $\mathbf{Q}$  and  $\mathbf{S}$  are positive semi-definite ( $\mathbf{Q}, \mathbf{S} \succeq 0$ ) and  $\mathbf{P}$  is

positive definite ( $\mathbf{P} \succ 0$ ). For solving this task, we need to find the gradient of the objective function [?] as

$$\nabla J(\underline{\mathbf{u}}) = \hat{\mathbf{H}}\underline{\mathbf{u}} + \hat{\mathbf{c}}. \quad (22)$$

The gradient  $\nabla J(\underline{\mathbf{u}}) \in R^{2T}$  is a vector with the opposite direction to the global minimum, so the  $-\nabla J(\underline{\mathbf{u}})$  is a vector pointing the direction towards the global minimum  $\underline{\mathbf{u}}^*$ . For solving this task, a gradient descent algorithm can be used, but also analytic solution can be found.

Because the  $\hat{\mathbf{H}}$  is positive semidefinite, the quadratic form  $\frac{1}{2}\underline{\mathbf{u}}^T \hat{\mathbf{H}}\underline{\mathbf{u}}$  is convex. The term  $\hat{\mathbf{c}}\underline{\mathbf{u}}$  is a linear function and adding it to a convex function does not change the convexity of the task. Because of this, we know, that a local minimum is also a global minimum. The local minimum can be found as

$$\begin{aligned} \nabla J(\underline{\mathbf{u}}^*) &= \hat{\mathbf{H}}\underline{\mathbf{u}}^* + \hat{\mathbf{c}} = 0 \\ \hat{\mathbf{H}}\underline{\mathbf{u}}^* &= -\hat{\mathbf{c}} \\ \underline{\mathbf{u}}^* &= -\hat{\mathbf{H}}^{-1}\hat{\mathbf{c}}. \end{aligned} \quad (23)$$

The inverse of  $\hat{\mathbf{H}}$  can be computed, because  $\hat{\mathbf{H}}$  is positive-definite, therefore regular. This is guaranteed by the definition of positive-definite property, that its leading principal minors are all positive [?]. On-board computation is fast, because matrix  $\hat{\mathbf{H}}^{-1}$  does not depend on the system state and is only the function of the constants  $k_q$ ,  $k_p$  and the system model. It can be generated in advance and stored as a constant in the UAV's read-only memory. Finding the  $\underline{\mathbf{u}}^*$  is only matter of creating vector  $\hat{\mathbf{c}}$  and computing  $-\hat{\mathbf{H}}^{-1}\hat{\mathbf{c}}$ .

#### 4.4 System constraints

One of the greatest advantages of MPC is being able to apply large variety of constraints. As mentioned in section 4, the MPC uses linearly constraint quadratic programming for finding the input action prediction. There can be many different kinds of constraints and the MPC has to obey all of them. These constraints take form of linear inequalities

$$\vec{\omega}_i \cdot \underline{\mathbf{u}} \leq b_i, \quad i \in \{1, 2, \dots, m\}, \quad (24)$$

where  $\vec{\omega}_i$  is a vector of the length  $2T$  and  $b_i$  is a scalar, both describing the constraint number  $i$ .  $m$  is the total number of constraints applied. Each constraint takes form of a half space in a space  $R^{2T}$  constraining it by hyperplane with a normal vector  $\vec{\omega}_i$  shifted by  $b_i$ . To apply multiple constraints at the same time, the Eq. (24) can be rewritten in matrix form as

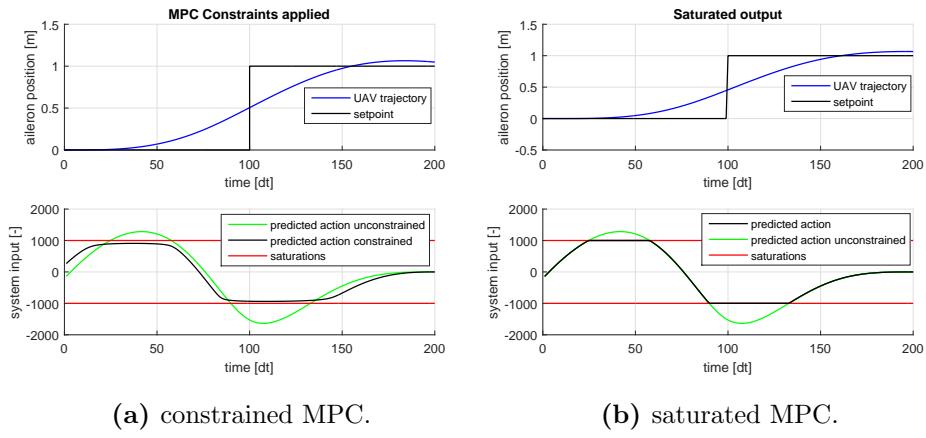
$$\mathbf{A}_c \underline{\mathbf{u}} \leq \mathbf{B}_c, \quad (25)$$

where  $\mathbf{A}_c$  is a matrix of the width  $2T$  and height  $m$  - the number of constraints applied.  $\mathbf{B}_c$  is a column vector of the same height created as

$$\mathbf{A}_c = \begin{bmatrix} \vec{\omega}_1 \\ \vec{\omega}_2 \\ \dots \\ \vec{\omega}_m \end{bmatrix}, \mathbf{B}_c = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{bmatrix}. \quad (26)$$

Each line of the matrix  $\mathbf{A}_c$  together with a particular member of  $\mathbf{B}_c$  represents one constraint.

#### 4.5 Input constraints



**Figure 3:** Step response of the system using two approaches of saturation.

In control engineering, a common problem one must overcome is a system saturation. This means, that the real system is linear only on a certain range of inputs and has limited capabilities [?]. For example motor can spin in a certain maximum speed despite input voltage. If the system receives too high input actions from the controller, it can get damaged. This can happen for example in PD control, when the difference between real and desired output changes rapidly in time, for example because of a sensory noise. A standard solution for this problem is a simple saturation of the output of the controller. This is a simple solution and for many tasks it is enough. However, because the controller doesn't take into consideration this saturation, the system can behave incorrectly. The advantage of MPC is, that the controller can consider these aspects of real system and find an input prediction, that will not violate the constraints and at the same time will achieve the desired output. This is achieved by the MPC constraints.

The system saturation thresholds are  $u_{min}$  and  $u_{max}$ . We can see in fig. (3b) simply saturated input action with the computed trajectory and in fig. (3a) the input action computed by the MPC using system constraints.

## 5 Collision Avoidance

Collision avoidance in MPC is realized through constraining the position. This means, that we allow every predicted position to be in a certain allowed space and every violation of this space is considered to be a collision. For the UAV to get outside this allowed space will be prohibited by the algorithm used in this section. In the subsection 2.1 has been introduced, that the UAV is approximated with a single point of mass. Using this approximation would result only for the center point of the UAV not to collide. In the real world the size matters. it is not necessary to bring the whole model back. Instead the whole body can be approximated with a ring with the same center  $(x, y)$  and radius  $R$ , such as every part of the body  $(x_b, y_b)$  follows  $|x - x_b, y - y_b| \leq R$ . If the distance between the position of UAV and the edge of the allowed space is less or equal  $R$ , it can be said, that the UAV will not collide.

The biggest problem in using MPC as a collision avoidance is creating this allowed space and converting it to the MPC constraints.

### 5.1 Convexity

Let's start with the definition of convexity. A set  $S$  is convex if

$$\forall\{x_1, x_2\} \in S, \forall t \in \langle 0, 1 \rangle \implies t \cdot x_1 + (1 - t) \cdot x_2 \in S. \quad (27)$$

In simple worlds, if every 2 points of a set are connected with a line, the whole line has to belong to the set.

For easily finding the global minimum of the cost function  $J(\underline{\mathbf{u}})$ , it is essential to keep the problem convex. This has many levels. First we have to ensure, that the function  $J$  is convex. This has been discussed in Sec. 4.2. Lets introduce a space  $S$  of dimension  $2T + 1$ , where the axes will be  $(\underline{\mathbf{u}}^T, z)$ . Then introduce a subset  $J_s$  of the space  $S$  as  $\{J_s \in S : J(\underline{\mathbf{u}}) \leq z\}$ . The function  $J(\underline{\mathbf{u}})$  is convex, if and only if the subset  $J_s$  is convex.

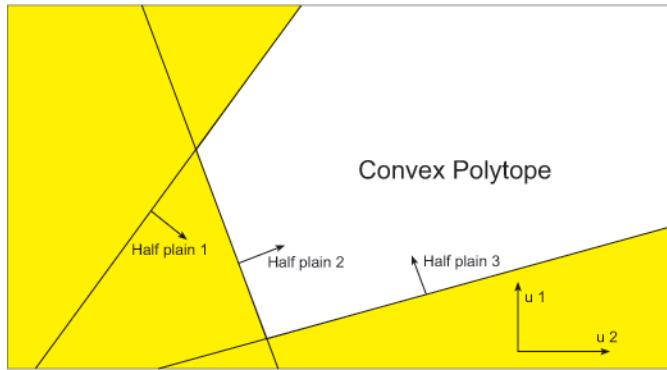
#### 5.1.1 Convexity of $\mathbf{u}$

Lets have a look at the vector  $\underline{\mathbf{u}}$ . For MPC purposes this vector is needed to be constrained. Because the space  $S$  has one more dimension than  $\underline{\mathbf{u}}$ , an extended version will be needed such as  $(\underline{\mathbf{u}}^T, z)^T$ . All points, that satisfy the condition 25 will create a subset  $U_s$  such as  $\{U_s \in S : \mathbf{A}_c \cdot \underline{\mathbf{u}} \leq \mathbf{B}_c\}$ . The condition is independent of  $z$ , so it is defined for all  $z$ .

We can look at this as the feasible  $\underline{\mathbf{u}}$  is the domain of the function  $J(\underline{\mathbf{u}})$ . Let's examine, how the  $U_s$  actually looks. As mentioned in the Sec. 4.4, the feasible  $\underline{\mathbf{u}}$  is an  $\underline{\mathbf{u}}$ , that satisfies  $m$  linear inequalities represented as half spaces. To satisfy all of them, set  $U_s$  has to be an extended intersection of half spaces. This is important, because the convexity of set  $U_s$  can be now proven.

### 5.1.2 Proof of the convexity of polytope

A theorem states, that an intersection of convex sets creates a convex set. An intersection of finite number of half spaces is a convex polytope. This polytope doesn't have to be bounded. Such a polytope is shown on a figure 4 in only 2 dimensions constrained by 3 half spaces.



**Figure 4:** convex polytope.

At this time is left to prove, that a single half space is a convex set. Half space is described in Eq. (24) as  $\{\underline{\mathbf{u}} : \omega_i \cdot \underline{\mathbf{u}} \leq b_i \text{ for } i \in \{1, 2, \dots, m\}\}$ . If we take any 2 points  $\underline{\mathbf{u}}_1, \underline{\mathbf{u}}_2$  belonging to the half space given by the normal vector  $\omega_i$  and bias  $b_i$ , the line segment between them has to lie in the half space. This can be proved for any dimension. The line segment is described as

$$t \cdot \underline{\mathbf{u}}_1 + (1 - t) \cdot \underline{\mathbf{u}}_2; \quad t \in \langle 0, 1 \rangle \quad (28)$$

The problem of proving convexity has been transformed using the Eq. (28) into equation

$$\omega_i \cdot (t \cdot \underline{\mathbf{u}}_1 + (1 - t) \cdot \underline{\mathbf{u}}_2) \leq b_i. \quad (29)$$

We know, that  $\underline{\mathbf{u}}_1$  and  $\underline{\mathbf{u}}_2$  belong to the half space, therefore

$$\begin{aligned} \omega_i \cdot \underline{\mathbf{u}}_1 &\leq b_i \\ \omega_i \cdot \underline{\mathbf{u}}_2 &\leq b_i. \end{aligned} \quad (30)$$

Because  $\omega_i \cdot \underline{\mathbf{u}}$  is a scalar, there are just 3 possible outcomes.  $\omega_i \cdot \underline{\mathbf{u}}_1 = \omega_i \cdot \underline{\mathbf{u}}_2$  or  $\omega_i \cdot \underline{\mathbf{u}}_1 < \omega_i \cdot \underline{\mathbf{u}}_2$  or  $\omega_i \cdot \underline{\mathbf{u}}_1 > \omega_i \cdot \underline{\mathbf{u}}_2$ . Lets have a look at the first case  $\omega_i \cdot \underline{\mathbf{u}}_1 = \omega_i \cdot \underline{\mathbf{u}}_2$ :

$$\begin{aligned} \omega_i \cdot (t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq b_i \\ \omega_i \cdot (\underline{\mathbf{u}}_1(t+1-t)) &\leq b_i \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq b_i. \end{aligned} \tag{31}$$

This case has been proven. The second case  $\omega_i \cdot \underline{\mathbf{u}}_1 < \omega_i \cdot \underline{\mathbf{u}}_2$  is a little bit more complicated. Because  $\omega_i \cdot \underline{\mathbf{u}}_2 \leq b_i$ . It would be sufficient to prove

$$\begin{aligned} \omega_i(t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq \omega_i \cdot \underline{\mathbf{u}}_2 \\ \omega_i(t \cdot \underline{\mathbf{u}}_1 - t \cdot \underline{\mathbf{u}}_2) &\leq 0 \\ t \cdot \omega_i(\underline{\mathbf{u}}_1 - \underline{\mathbf{u}}_2) &\leq 0 \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq \omega_i \cdot \underline{\mathbf{u}}_2. \end{aligned} \tag{32}$$

The third line has been divided by  $t$ . In case  $t = 0$ , the proof ends by  $0 \leq 0$ , otherwise continuing to the fourth line. This case has been also proven. Proving the last case  $\omega_i \cdot \underline{\mathbf{u}}_2 < \omega_i \cdot \underline{\mathbf{u}}_1$  is very similar.

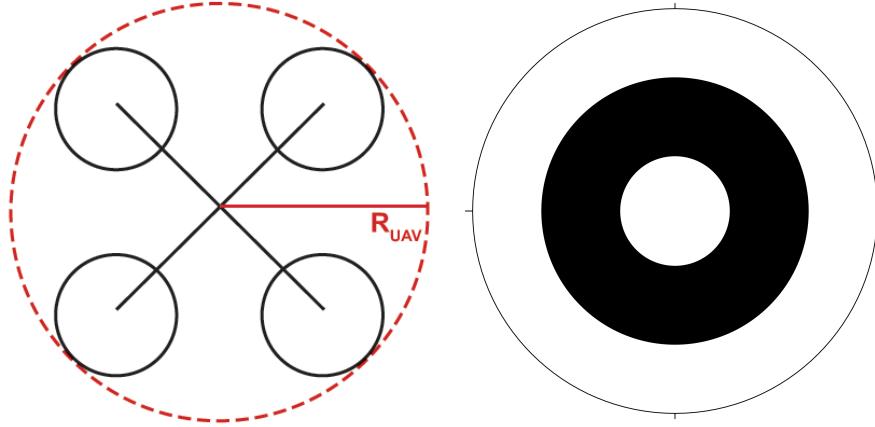
$$\begin{aligned} \omega_i(t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq \omega_i \cdot \underline{\mathbf{u}}_1 \\ \omega_i((t-1) \cdot \underline{\mathbf{u}}_1 - (t-1) \cdot \underline{\mathbf{u}}_2) &\leq 0 \\ (t-1) \cdot \omega_i \cdot (\underline{\mathbf{u}}_1 - \underline{\mathbf{u}}_2) &\leq 0 \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq \omega_i \cdot \underline{\mathbf{u}}_2 \end{aligned} \tag{33}$$

Similarly to the Eq. (32), the third line of the Eq. (33) has been divided by  $t-1$ . It has been proven, that all  $\underline{\mathbf{u}}$ , that obey the Eq. (25 c) create a convex set. This means, that also the set  $U_s$  is convex. Intersection of the convex set  $J_s$  and  $U_s$  is a convex set making the function  $J(\underline{\mathbf{u}})$  with the domain as polytope defined  $\mathbf{A}_c \underline{\mathbf{u}} \leq \mathbf{B}_c$  is a convex function.

## 6 Obstacles

Through this thesis, obstacles are represented by circles in 2D only. They have 3 parameters: position  $x_{obs}$ , position  $y_{obs}$  and diameter  $r_{obs}$ . Despite obstacles can have various shapes, we simplify them to allow simpler computations. Because the MPC runs constantly in the loop and reacts to the changing environment, it also works with moving objects. Until now, the UAV body has been approximated with one mass point without considering its real size. Since it would be difficult to calculate, whether the UAV's body has collided, we use Minkovski sum. Instead of computing with the real size

of the UAV, the representation of all obstacles will change according to the UAV's size, creating a circle with the UAV's dimensions. This circle will have the smallest possible diameter  $R_{UAV}$  while still wrapping the UAV as shown in the Fig. 5.



**Figure 5:** UAV dimensions

**Figure 6:** Recognizable marker

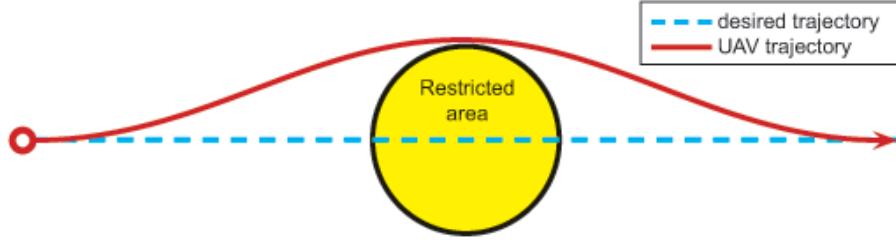
Let's define a simplified representation of the UAV's body. It can be now said, that the UAV will not collide, if

$$(x_{obs} - x)^2 + (y_{obs} - y)^2 \geq (R_{UAV} + r_{obs})^2. \quad (34)$$

From now on we will compute with the UAV as a single mass point and every obstacle's diameter  $r_{obs}$  substitute with the  $r_{obs} + R_{UAV}$ .

Let's make an assumption, that the UAV's trajectory is a line created by connecting all the predicted positions. This line has a zigzag shape. Because the UAV can not change vector of speed immediately, the real trajectory has to have a continuous first time derivative. Also because the second derivative(acceleration) of the UAV's position is a result of the UAV's pitch and roll, which also can not be changed immediately, the second derivative is also continuous. In short, the real trajectory intersects all the predicted positions, but there is a small deviation caused by the first and second derivatives of the real system. The real trajectory has to be smooth up to second derivative. To be able to approximate the real trajectory by the simplified zigzag trajectory, we need to make sure, that the predicted positions are far closer to each other, than the obstacles sizes. This is easy to achieve, because the time between the predicted positions is  $\Delta t = 1/70s$ , which with the maximum speed of  $0.35 \text{ ms}^{-1}$  is distance of 5 mm.

The forward approach would be to say, that every trajectory is feasible, if any predicted position does not collide with any obstacle. This approach would search for any trajectory in the allowed space. However if the obstacle



**Figure 7:** UAV avoidance in non-convex space.

is just a simple circle, the allowed space is non-convex as shown in Fig. 7. As mentioned in Sec. 5.1, for keeping the quadratic programming convex, the allowed space for UAV has to be convex. This turned out to be the greatest disadvantage of using MPC for obstacle avoidance.

This is also the reason, why greater penalization of the last state deviation is not desirable. The allowed space is not ideal and the prediction of the last position

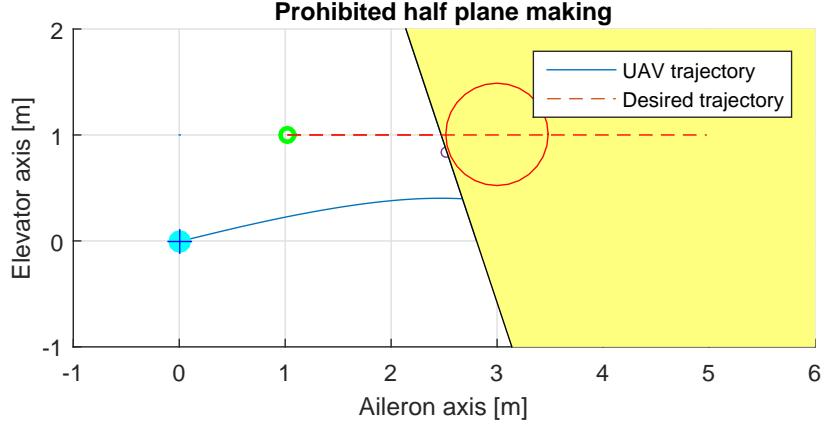
## 6.1 Creating allowed space

A convex allowed space for trajectory planning has to be created to approximate the real world situation. However, there can not be enough good convex representation, that is robust at the same time. The approach for creating a convex allowed space in this thesis is restricting any position in a half plain 'behind' any obstacle as shown in Fig. 8. Combining multiple obstacle constraints, this space becomes polytope defined by a set of half planes. These plains are in each MPC iteration updated by the relative positions of the obstacles. These positions continuously change due to the UAV's movement and the sensory noise. Let's first look at a situation with a single obstacle.

Half plane can be represented by the equation  $y \leq kx + q$  where  $x$  and  $y$  are the allowed positions of UAV and  $k$  is the slope of the line and  $q$  is the bias of the line. For easier work, these equations can be rewritten as

$$s(y - kx - q) \leq 0, \quad (35)$$

where  $s = \pm 1$ . This is an inequality defining one obstacle by a single half plain. These constants  $k, q$  and  $s$  can be found as



**Figure 8:** Creating a prohibited half plain

$$\begin{aligned}
 k &= -\frac{y_{obs} - y_{UAV}}{x_{obs} - x_{UAV}} \\
 b_x &= (x_{obs} - x_{UAV}) \cdot \left(1 - \frac{r_{obs} + R_{UAV}}{\sqrt{(x_{obs} - x)^2 + (y_{obs} - y)^2}}\right) \\
 b_y &= (y_{obs} - y_{UAV}) \cdot \left(1 - \frac{r_{obs} + R_{UAV}}{\sqrt{(x_{obs} - x)^2 + (y_{obs} - y)^2}}\right) \\
 q &= b_y - kb_x \\
 s &= \text{sign}(y).
 \end{aligned} \tag{36}$$

The  $b_x$  and  $b_y$  is a position of the intersection of the constraining line and the obstacle circle with diameter  $R_{UAV} + r_{obs}$ .

We have found the constrains for position. These constraints have to be further transformed into an input action constraints. This is an inverse transformation to the Eq. 12, where input actions have been transformed to position and its derivatives. For the purposes of MPC, matrices  $\mathbf{A}_c$  and  $\mathbf{B}_c$  have to be created to fit the condition 25. The vector  $\underline{\mathbf{x}}$  from the Eq. 12 contains all predicted states for both axes. We are interested in separating just the predicted positions  $x$  and  $y$  into the position column vectors  $\mathbf{x}_p$  and  $\mathbf{y}_p$  of size  $T$  as

$$\mathbf{x}_p = \begin{bmatrix} x[1] \\ x[2] \\ \dots \\ x[T] \end{bmatrix}, \mathbf{y}_p = \begin{bmatrix} y[1] \\ y[2] \\ \dots \\ y[T] \end{bmatrix}, \tag{37}$$

where  $x_{[t]}$  is the predicted position  $x$  at the time  $t$  and  $y_{[t]}$  is the predicted position  $y$  at the time  $t$ . These vectors are related to  $\underline{\mathbf{u}}$  by the matrices  $\hat{\mathbf{A}}_x$ ,  $\hat{\mathbf{B}}_x$ ,  $\hat{\mathbf{A}}_y$ ,  $\hat{\mathbf{B}}_y$  as

$$\begin{aligned}\mathbf{x}_p &= \hat{\mathbf{A}}_x \mathbf{x}_{[0]} + \hat{\mathbf{B}}_x \underline{\mathbf{u}} \\ \mathbf{y}_p &= \hat{\mathbf{A}}_y \mathbf{x}_{[0]} + \hat{\mathbf{B}}_y \underline{\mathbf{u}},\end{aligned}\tag{38}$$

where the matrices  $\hat{\mathbf{A}}_x, \hat{\mathbf{A}}_y \in R^{T \times 6}$  and  $\hat{\mathbf{B}}_x, \hat{\mathbf{B}}_y \in R^{T \times 2T}$  are submatrices of their corresponding matrices  $\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{A}}, \hat{\mathbf{B}}$  as

$$\begin{aligned}\mathbf{B}_x &= \begin{bmatrix} \mathbf{B}_{1,1:2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{1,1:6}\mathbf{B} & \mathbf{B}_{1,1:2} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{1,1:6}^{(T-1)}\mathbf{B} & \mathbf{A}_{1,1:6}^{(T-2)}\mathbf{B} & \dots & \mathbf{B}_{1,1:2} \end{bmatrix}, \quad \mathbf{A}_x = \begin{bmatrix} \mathbf{A}_{1,1:6} \\ \mathbf{A}_{1,1:6}^2 \\ \vdots \\ \mathbf{A}_{1,1:6}^{(T-1)} \end{bmatrix}, \\ \mathbf{B}_y &= \begin{bmatrix} \mathbf{B}_{4,1:2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{4,1:6}\mathbf{B} & \mathbf{B}_{4,1:2} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{4,1:6}^{(T-1)}\mathbf{B} & \mathbf{A}_{4,1:6}^{(T-2)}\mathbf{B} & \dots & \mathbf{B}_{4,1:2} \end{bmatrix}, \quad \mathbf{A}_y = \begin{bmatrix} \mathbf{A}_{4,1:6} \\ \mathbf{A}_{4,1:6}^2 \\ \vdots \\ \mathbf{A}_{4,1:6}^{(T-1)} \end{bmatrix}.\end{aligned}\tag{39}$$

$\mathbf{A}_{i_1,i_2:i_3}$  is a matlab-like notation describing a submatrix of the matrix  $\mathbf{A}$  with the line  $i_1$  and columns from  $i_2$  to  $i_3$ . The Eq. (35) is defined only for one position. These matrices are constant, so they can be computed in advance and stored in the read-only memory. It is important, that the UAV does not collide in any predicted position in the prediction horizon. Then this equation is rewritten in a vector form

$$s(\mathbf{y}_p - k\mathbf{x}_p - \mathbf{q}) \leq 0,\tag{40}$$

where  $\mathbf{q}$  is the column vector of the size  $T$  and every member is  $q$ . After the substitution of Eq. (38) into Eq. (40) we get

$$\begin{aligned}s(\hat{\mathbf{A}}_y \mathbf{x}_{[0]} + \hat{\mathbf{B}}_y \underline{\mathbf{u}} - k(\hat{\mathbf{A}}_x \mathbf{x}_{[0]} + \hat{\mathbf{B}}_x \underline{\mathbf{u}}) - \mathbf{q}) &\leq 0 \\ \underbrace{s(\hat{\mathbf{B}}_y - k\hat{\mathbf{B}}_x)}_{\hat{\mathbf{A}}_c} \underline{\mathbf{u}} + \underbrace{s(\hat{\mathbf{A}}_y \mathbf{x}_{[0]} - k\hat{\mathbf{A}}_x \mathbf{x}_{[0]} - k\mathbf{q})}_{\mathbf{B}_c} &\leq 0.\end{aligned}\tag{41}$$

Until now, all the half planes of the obstacles have been computed from the relative position of the initial position of the UAV. This means, that for example the last predicted position will be still constrained by the position some time ago. However we don't know the last position, because it can be computed from the vector  $\underline{\mathbf{u}}$  that we are searching for. If an assumption is made, that the predicted trajectory is very similar to the one in the previous

step, we can use the previous one. The MPC loop runs in tens of Hz and the UAV doesn't change its position very much in the one time step. This improvement is done by computing the previous trajectory using Eq. (12). Then the constants  $k_i, q_i, s_i$  need to be found for each predicted position  $i, i \in \{1, \dots, T\}$  using the Eq. (36). Then the Eq. (40) can be rewritten in a matrix form as

$$\hat{\mathbf{s}}(\mathbf{y}_p - \hat{\mathbf{k}}\mathbf{x}_p - \mathbf{q}) \leq 0, \quad (42)$$

where  $\hat{\mathbf{s}}$  and  $\hat{\mathbf{k}}$  are square diagonal matrices of the size  $T$ .

$$\hat{\mathbf{s}} = \begin{bmatrix} s_1 & 0 & \dots & 0 \\ 0 & s_2 & \dots & \vdots \\ 0 & \dots & \ddots & \vdots \\ 0 & \dots & \dots & s_T \end{bmatrix}, \hat{\mathbf{k}} = \begin{bmatrix} k_1 & 0 & \dots & 0 \\ 0 & k_2 & \dots & \vdots \\ 0 & \dots & \ddots & \vdots \\ 0 & \dots & \dots & k_T \end{bmatrix}, \mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_T \end{bmatrix}. \quad (43)$$

Because of high computational demands, this algorithm has not been implemented. The MPC loop must run as fast as possible. Because of the UAV's hardware computational limitations. The standard solution uses only the initial condition for knowing position and computes once the constants from Eq. (36). This is very fast operation. If we want to compute different constants for each future position, we need to compute the predicted trajectory from the previous prediction using Eq. (12) and compute the constraining conditions  $T$  times.

Let's now consider a case with multiple obstacles. As described in the section 4.4, the lines of constraining matrices  $\mathbf{A}_c$  and  $\mathbf{B}_c$  enforce independent conditions. The computation can be done independently for  $N$  obstacles using the same procedure, marking the matrices  $\mathbf{A}_{c,i}$  and  $\mathbf{B}_{c,i}$ , where  $i \in 1, 2, \dots, N$ . The final constraining matrices would take form of

$$\mathbf{A}_c = \begin{bmatrix} \mathbf{A}_{c,1} \\ \mathbf{A}_{c,2} \\ \dots \\ \mathbf{A}_{c,N} \end{bmatrix}, \mathbf{B}_c = \begin{bmatrix} \mathbf{B}_{c,1} \\ \mathbf{B}_{c,2} \\ \dots \\ \mathbf{B}_{c,N} \end{bmatrix}. \quad (44)$$

Now the task is fully defined and can be given to a LCQP solver to find the optimal input action  $\mathbf{u}$ .

## 7 Solving Linearly Constrained QP

Formulating the MPC has 2 parts. The first one is creating the objective function  $J(\underline{\mathbf{u}})$  with the matrices  $\hat{\mathbf{H}}$ ,  $\hat{\mathbf{c}}$  and creating the constraining matrices  $\mathbf{A}_c$  and  $\mathbf{B}_c$ . This part has been described in the previous sections. The second part is solving this problem defined in the Eq. (21). We have already

discussed solving the unconstrained problem using the inverse  $\hat{\mathbf{H}}^{-1}$  matrix. The constrained problem can not be solved analytically and iterative methods are commonly used. This means, that the solution is usually not optimal, but close to optimal.

## 7.1 Safe margin

One more property of the solver is desirable in this particular case of obstacle avoidance. Instead of finding a strictly feasible solution (an  $\underline{\mathbf{u}}$ , that lies on at least one constraining hyper plain), a solution with some margin is better. Strictly feasible solution is actually touching the obstacle. The MPC finds the closest avoidance trajectory, where at least one state prediction is equal to the position constraint. If we want to have a safe distance from the obstacle, there are 2 solutions.

One is simply making  $R_{UAV}$  bigger than the real UAV body and taking the strictly feasible solution. This solution has a one big disadvantage. The mathematical model will still touch the obstacle, even that the UAV body is actually smaller and will fly in a bigger distance. Because of the sensory noise, the measured relative position of the obstacle is continuously changing. At the time, when UAV model touches the obstacle, the measurement of the obstacle can change a little bit closer to the UAV and suddenly the UAV is mathematically inside the obstacle. The initial condition does not satisfy the constraints any more. The constraints are not defined for the initial condition, but from the first time step, because the initial condition does not depend on  $\underline{\mathbf{u}}$ . The optimization algorithm would try to find an input action to escape the constraint in the first time step. The position is a second integration of the input and each integration takes one time step. This means, that the input action first influences the position in two time steps ahead.

Even a small amount of noise can result in a task formulation, that has no solution. Choosing the strictly feasible solution is not applicable. Making the final solution  $\underline{\mathbf{u}}$  further from the constraining hyper plains solves this problem well. The UAV will fly with some safe distance from the obstacle and if the obstacle changes its position as a result of the sensory noise, the initial condition will still satisfy the position constraints. This is the solution, that will be used. For finding this kind of solution, a special optimization method has to be used. Because of the convexity of the function and the convexity of the constraints, the solution is either a global minimum of the function  $J(\underline{\mathbf{u}})$ , or lies on the constraining border.

## 7.2 LCQP solvers

There are several methods of solving the constraint optimization problem. The first one is called Active set method [?]. The constraints are solved

almost exactly. This method supposes, that the minimum lies on the constraint border. It searches only strictly feasible solutions. Some constraints play no role for the final result and some parts of constraints are prohibited by other constraints. This brings a high complexity. Because of its analytic approach, it works well on noise-free problems and when the exact minimum is needed. However, UAV's optical registration of obstacles and optical localization brings a high noise. Active set method is not a good fit for this problem.

The second method is a method of Lagrange multipliers. This is a very widely used strategy in many different tasks. It also assumes, that the minimum lies on the constraining border. This is forced with the constraints defined as  $g_i(\underline{\mathbf{u}}) = 0, i \in \{1, 2, \dots, N\}$  for  $N$  constraint. This method requires, that the objective function and the constraints are differentiable. It uses a function called Lagrangian  $\mathcal{L}$

$$\mathcal{L}(\underline{\mathbf{u}}, \lambda_1, \lambda_2, \dots, \lambda_N) = J(\underline{\mathbf{u}}) - \sum_{i=1}^N \lambda_i g_i(\underline{\mathbf{u}}) \quad (45)$$

Then a solution must be found for  $\nabla \mathcal{L} = 0$ . This would mean to solve a set of  $T + N$  equations

$$\frac{\partial J(\underline{\mathbf{u}})}{\partial u_j} = \sum_{i=1}^N \frac{\partial \lambda_i g_i(\underline{\mathbf{u}})}{\partial u_j} \quad \text{and} \quad \frac{\partial J(\underline{\mathbf{u}})}{\partial \lambda_i} = \sum_{j=1}^N \frac{\partial \lambda_i g_i(\underline{\mathbf{u}})}{\partial \lambda_i} \quad (46)$$

$$i \in \{1, 2, \dots, N\}, \quad j \in \{1, 2, \dots, 2T\}$$

This is a simple idea, that the gradient of the objective function  $J(\underline{\mathbf{u}})$  is perpendicular to all constraints. If not, it means, that there is a better solution. However, in our case the perpendicularity is not always true. The solution can lie for example in an intersection of two half plains. The problem is also in the defining the constraints, which are not in the form of inequality but equality and it is required to satisfy all of them. This means, that there has to exist an intersection of all the constraints. This is not very likely. Our constraints are defined with hyper plains. If even 2 hyper plains are parallel, this algorithm does not find any solution. These hyper plains have been described in the section 4.5. If there is constrained a maximum thrust forwards and backwards. If defined as equality, this would mean, that the thrust must be maximum forwards and maximum backwards at the same time, which is logically impossible. This approach has to be also rejected.

The third method is a Penalty method. It is used for example in optimizing SVM algorithm. It penalizes crossings of the constraints. It modifies the objective function by adding a penalty function, if a constraint is violated. The task is modified as

$$\min \mathbf{J}(\underline{\mathbf{u}}) + \sigma \sum_{i=0}^N g(c_i(\underline{\mathbf{u}})) \quad (47)$$

where  $g(c_i(\underline{\mathbf{u}})) = \min(0, c_i(\underline{\mathbf{u}})^2)$  is a positive number if the constraint  $c_i$  has been violated and zero otherwise. This function is not easily differentiable, but after some adjustments, gradient descent can be used. The biggest disadvantage is, that the final solution very likely violates the constraints. Violating these constraints in our case means either giving higher input actions or a collision. It is obvious, that this algorithm can not be used either.

### 7.3 Barrier method

The final method is called a Barrier method. This method has been implemented. It modifies the objective function by adding a barrier function  $g(\underline{\mathbf{u}})$  penalizing points too close to the border. Then a gradient descent algorithm is used to find the local minimum of the function  $f(\underline{\mathbf{u}}) = \mathbf{J}(\underline{\mathbf{u}}) + g(\underline{\mathbf{u}})$ . The function  $g(\underline{\mathbf{u}})$  must possess several properties.

- the function has to be defined for all feasible  $\underline{\mathbf{u}}$ . It doesn't have to be defined elsewhere.
- The function  $g(\underline{\mathbf{u}})$  should be convex to preserve convexity of the sum of the function  $J(\underline{\mathbf{u}})$  and  $g(\underline{\mathbf{u}})$ .
- To make sure, the constraints will be satisfied, it should equal to infinity when approaching the border.
- It must be differentiable over all the domain to be able to use the gradient descent method.
- From these assumptions we can say without the loss of generality, that the barrier function should be a function of the distance from the constraining hyper plain.

The second requirement supposes, that an addition of two convex functions is a convex function. Let's choose some convex function  $f_1(x), f_2(x)$  and some variable  $x$ . Then we want to prove, that  $f_3(x) = f_1(x) + f_2(x)$  is also a convex function. This is proven by proving the definition of convexity

$$f_3(tx_1 + (1-t)x_2) \leq tf_3(x_1) + (1-t)f_3(x_2), \quad t \in \langle 0, 1 \rangle \quad (48)$$

where  $x_1$  and  $x_2$  are any points form the domain. Then because of the convexity of  $f_1$  and  $f_2$ ,

$$\begin{aligned} f_1(tx_1 + (1-t)x_2) &\leq tf_1(x_1) + (1-t)f_1(x_2) \\ f_2(tx_1 + (1-t)x_2) &\leq tf_2(x_1) + (1-t)f_2(x_2) \end{aligned} \quad (49)$$

The inequality 48 is actually a sum of the two lines of the inequalities 49. Adding these 2 lines preserves the inequality. We can say, that the inequality 48 is valid and that the result of summing convex functions is a convex function.

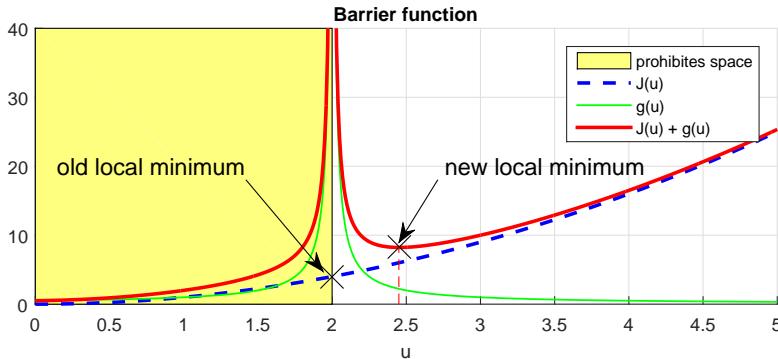
The distance in a space of any dimension of a point from a hyper plain is

$$d = \text{abs} \left( \frac{\omega_i \cdot \underline{\mathbf{u}} + b_i}{\sqrt{\vec{\omega}_i^T \cdot \vec{\omega}_i}} \right) \quad (50)$$

The notation  $|\vec{\omega}_i|$  represents the Euclidean norm as  $\sqrt{\vec{\omega}_i \cdot \vec{\omega}_i^T}$ . The function  $f$  can be then defined for  $\underline{\mathbf{u}}$  or  $d$  depending on the requirements. After trying various forms of functions  $g(\underline{\mathbf{u}})$ , the function

$$g(\underline{\mathbf{u}}) = k_g \sum_{i=1}^M \frac{1}{d_i} \quad (51)$$

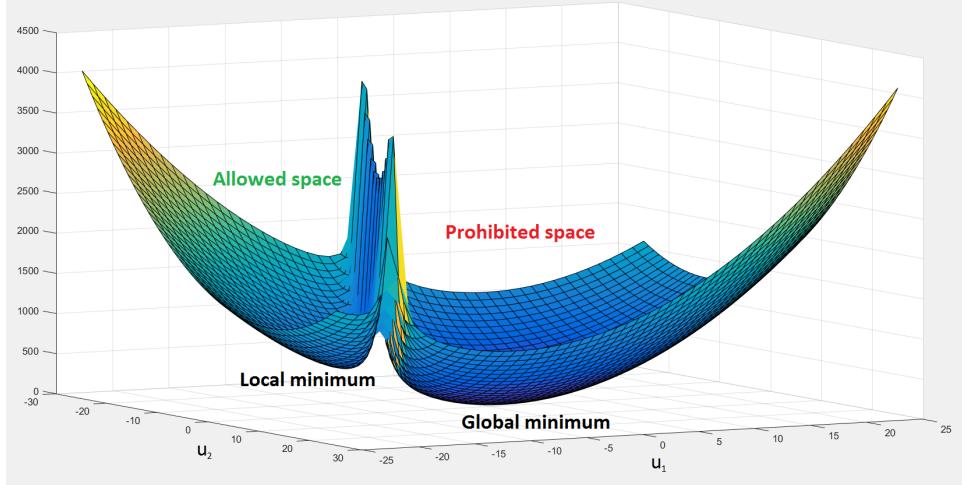
has been chosen, where  $k_g$  is a constant. It has been experimentally chosen to be  $10^4$ . This function has all the properties required. If the point is close to any on the constraining hyper plains, the barrier function rapidly grows. It could look strange, that this function is defined for all points, not only for the feasible set. And it doesn't make sense in the not feasible area. Again, it penalizes less the points far from the hyper plain so far behind the constraint. This is actually no problem, because the function will be only used in the feasible set.



**Figure 9:** Barrier function with one variable

The Fig. 9 shows, how the barrier method works in one dimension. It moves the location of the strictly feasible local minimum away from the prohibited space to a new local minimum. This is the desired property, because the strictly feasible solution can result in a bad behavior as mentioned in the beginning of this section.

The Fig. 10 shows the function  $J(\underline{\mathbf{u}}) + g(\underline{\mathbf{u}})$  with the 2 dimensional domain. The real optimizing function is in very high dimension and can not



**Figure 10:** The final optimization function with two variables

be graphed. Principal component analysis (PCA), which is an algorithm for reducing space dimensions by projecting objects to a created basis can not be used, because it will highly deform the space.

#### 7.4 Gradient descent

The optimization problem will be solved with slightly modified standard gradient descent. This is a very used algorithm for optimizing functions. It has two big requirements. One is, that the function must be differentiable. This has been satisfied by choosing quadratic function as the objective function and hyperbola as the barrier function. The second requirement is, that the function must be convex for Using the barrier method, instead of the function  $J(\underline{\mathbf{u}})$ , the gradient descent optimizes the function  $f(\underline{\mathbf{u}}) = J(\underline{\mathbf{u}}) + g(\underline{\mathbf{u}})$  where  $g(\underline{\mathbf{u}})$  is the barrier function from the Eq. (51). It is an iterative method, where in each iteration a point  $\underline{\mathbf{u}}$  is moved in the negative direction of the gradient of the function  $f(\underline{\mathbf{u}})$ . This means, that

$$\underline{\mathbf{u}}_{[t+1]} = \underline{\mathbf{u}}_t - k_d \nabla f(\underline{\mathbf{u}}), \quad (52)$$

where  $\underline{\mathbf{u}}_{[t]}$  is the  $\underline{\mathbf{u}}$  at the  $t$ -th iteration. The size of the gradient step at  $i$ -th iteration is  $k_d \nabla f(\underline{\mathbf{u}})$ . This algorithm runs for a certain number of steps until the final  $\underline{\mathbf{u}}$  is used as the result. For using this algorithm, the gradient  $\nabla f(\underline{\mathbf{u}})$  is needed to be computed as

$$\begin{aligned}
\nabla f(\underline{\mathbf{u}}) &= \nabla J(\underline{\mathbf{u}}) + \nabla g(\underline{\mathbf{u}}) \\
\nabla J(\underline{\mathbf{u}}) &= \hat{\mathbf{H}}\underline{\mathbf{u}} + \hat{\mathbf{c}} \\
\nabla g(\underline{\mathbf{u}}) &= k_g \sum_{i=1}^N \frac{\vec{\omega}_i \cdot \text{sig}(\vec{\omega}_i \cdot \underline{\mathbf{u}} + b_i)}{|\vec{\omega}_i| \cdot (\vec{\omega}_i \cdot \underline{\mathbf{u}} + b_i)},
\end{aligned} \tag{53}$$

where  $\vec{\omega}_i$  is the  $i$ -th line of the matrix  $\mathbf{A}_c$  and  $b_i$  is the  $i$ -th member of the column matrix  $\mathbf{B}_c$ . The notation  $\vec{\omega}_i \cdot \vec{\omega}_i$  represents a line vector as the diagonal of the  $\vec{\omega}_i^T \cdot \vec{\omega}_i$ .

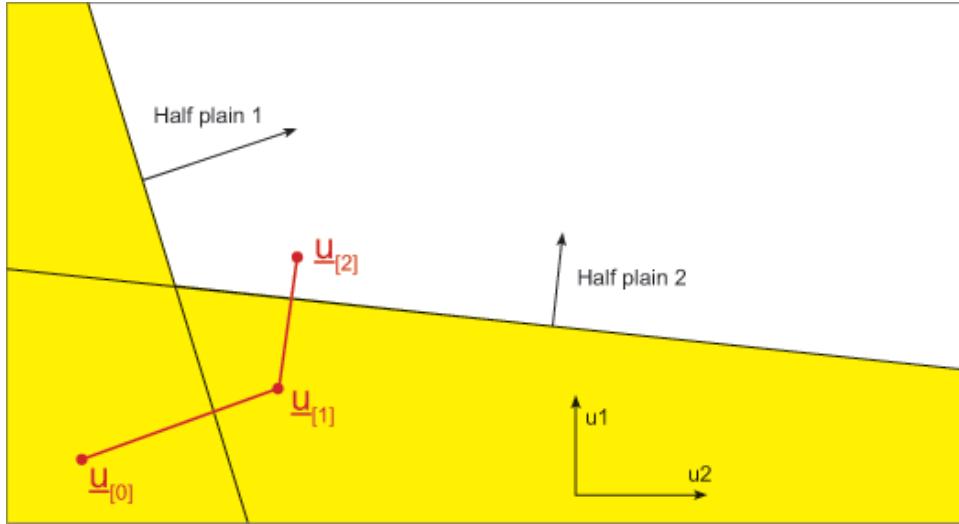
## 7.5 Feasibility algorithm

Now, that we have the  $\nabla f(\underline{\mathbf{u}})$ , we have to find the initial search point  $\underline{\mathbf{u}}_{[0]}$ . Because the function  $f(\underline{\mathbf{u}})$  is convex only on the feasible domain, the  $\underline{\mathbf{u}}_{[0]}$  has to lie there. To find a feasible  $\underline{\mathbf{u}}$  is not as simple task, as it looks. Especially if the UAV flies among many obstacle with higher velocity. A zero vector means flying in the initial direction and back thrust is not robust and could not work. An algorithm for finding a feasible  $\underline{\mathbf{u}}$  has been developed — chci napsat, ye jsem ho vztvoril ja—, that has been named a feasibility algorithm. While developing this algorithm, a useful feature had been required. This algorithm takes a  $\underline{\mathbf{u}}$ , that may or may not be feasible. It then finds a feasible alternative of the point, that is very close to the given one. It will be very useful for the speed and accuracy of the whole MPC, that will be discussed later. The iterative algorithm inputs are matrices  $\mathbf{A}_c$ ,  $\mathbf{A}_c$  and a  $\underline{\mathbf{u}}_0$ . The algorithm can be formulated in the following steps:

1. Find the first constraint  $\vec{\omega}_i \cdot \underline{\mathbf{u}}_{[t]} \leq b_i$ , that is violated. If such a constraint does not exist, successfully terminate the algorithm and return the  $\underline{\mathbf{u}}_{[t]}$ .
2. Count the distance  $d$  from the constraint  $i$  and compute the normalized gradient  $dg = \frac{\nabla g(\underline{\mathbf{u}})}{|\nabla g(\underline{\mathbf{u}})|}$ .
3. Move the search point along the gradient such as  $\underline{\mathbf{u}}_{[t+1]} = \underline{\mathbf{u}}_{[t]} + k_f \cdot d \cdot dg$  and continue to step 1.

The constant  $k_f$  should be between 1 and 2. If the  $k_f$  is 1, the  $\underline{\mathbf{u}}_{[t+1]}$  ends exactly on the constraint. This is not very good, because our function  $g(\underline{\mathbf{u}})$  is not defined there. If the constant  $k_f$  is higher than 2, the algorithm could oscillate, especially between 2 parallel constraints. From experiments, a constant  $k_f = 1.5$  has been chosen and works well.

The Fig. 16 shows, how the algorithm can correct a  $\underline{\mathbf{u}}$ , that is not feasible into a feasible one with a position very close to the initial one.



**Figure 11:** Feasibility algorithm.

When the algorithm for tracking trajectory starts, the UAV has no velocity, so the gradient descent algorithm can use the initial search point  $\underline{u}_{[0]}$  as a zero vector. It means, that it will stay at one place, so there is no predicted collision. The MPC loop runs very fast and if every time a new gradient descend is initialized with a random  $\underline{u}_{[0]}$ , it would take a long time to come anywhere to the local minimum. The knowledge, that the UAV moves relatively slow compared with the MPC loop, can be very useful, because the function  $f(\underline{u})$  is very similar in the next step. A final prediction from previous MPC step can be used as an initial search point. This incredibly increases the speed and accuracy of the gradient descend algorithm. Of course, this could not have been done without the feasibility algorithm. A final  $\underline{u}$  from previous MPC step can be feasible no more with a different UAV position and velocity. This is, where the required feature of the feasibility algorithm comes handy. It moves the previous  $\underline{u}$  to a new location, that is still very close to the previous  $\underline{u}$ , so very close to the local minimum.

There can be a situation, where a feasible solution does not exist. In this case, the loop would run infinitely. This can be achieved especially, if the sensory noise is too high that a distance to an obstacle is evaluated closer, then the  $R_{UAV} + r_{obs}$ . With this initial condition there is usually no solution as described in the section 7.1. To prevent the UAV to freeze, a counter needs to be added to terminate the feasibility algorithm after running too long.

## 7.6 Implementation of the gradient descend

The gradient descend is implemented by the equation 52. The only difference is, that in every iteration, every  $\underline{\mathbf{u}}_{[t]}$  is checked for feasibility by applying the feasibility algorithm. The reason to do this is, that the gradient step can be too big and after shifting the search point, it can end up outside the feasible domain. If the gradient step constant  $k_d$  was set too small, it would take longer time to converge to the local minimum. The  $k_d = 150$  based on simulations and experiments.

A simulation has been run for no barrier function using only the  $\nabla J(\underline{\mathbf{u}})$  and the feasibility algorithm and still providing fairly good results. This approach has not been used in later simulations or experiments.

Choosing the total number of iterations of the gradient algorithm generally depends on the requirements of accuracy vs speed. Experimentally has been observed, that a small number of total iterations can be helpful by providing a low pass filter in situations with high sensory noise. Experiments have shown, that 100 iterations is a good compromise between the accuracy and the speed.

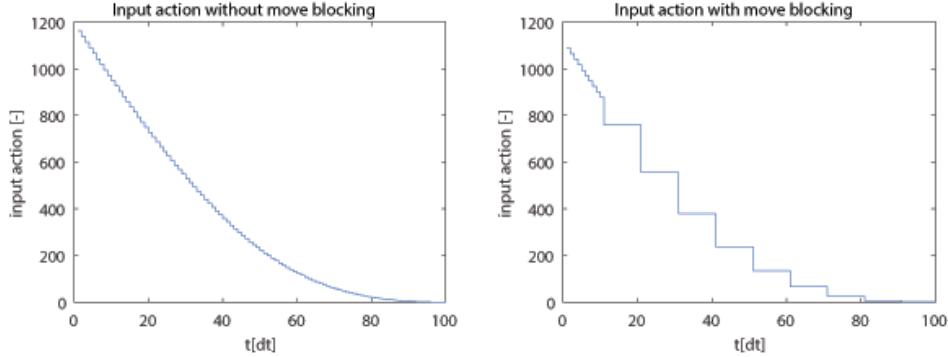
## 8 Move Blocking

The whole MPC algorithm is very demanding on computational time and memory. This can result in a very slow frequency of the MPC controller. A matrix multiplication of a  $n$  by  $m$  matrix and a column vector of the length  $m$  has the computational complexity  $O(nm)$ . For example the matrix  $\hat{\mathbf{B}}$  is the size of  $6T$  by  $2T$  and the vector  $\underline{\mathbf{u}}$  is the length of  $2T$ . This makes the computational complexity  $O(12T^2)$ . When considering, that the system model is created for  $dt = 1/70s$ , for prediction of 2 seconds for a simple matrix multiplication, processor would have to do at least  $12 \cdot 140^2 = 235200$  operations. With the size of 4 bytes for one float, the matrix  $\hat{\mathbf{B}}$  would take almost 1 MB of memory. This is not that much for PC simulation, but it is a lot for the custom board flying on the UAV with only 192 kB RAM, 2 MB of ROM and 168 MHz processor with one instruction for float multiplication.

### 8.1 Move blocking implementation

The computation complexity can be greatly reduced by reducing the  $T$ , which is the total number of future predictions. The discretization of the prediction is very soft, meaning, that the  $\Delta t$  is very small. With the maximum speed this is a distance 0.5 mm between predicted positions. There is no reason to predict position 70 times a second. The input actions tend to take form of a continuous function. One solution would be to change the system's  $\Delta t$ , but there is a better solution. Move blocking algorithm allows us to choose exactly which time steps will be used. This allows only small

number of variables to cover long prediction horizon. The algorithm will generate constant input action between those predicted as shown in Fig. 17. Let's introduce a matrix  $\mathbf{U}$  of the height  $2T$  and width  $2T_n$ , where  $T_n$  is the new number of predicted states. This matrix determines, which time steps will be used.



**Figure 12:** Move blocking algorithm in one axis

$$\underline{\mathbf{u}} = \underbrace{\begin{bmatrix} \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} & 0 & \dots & 0 \\ 0 & \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \end{bmatrix}}_{\mathbf{U}}, \quad \underline{\mathbf{u}}_r, \quad \underline{\mathbf{u}}_r = \begin{bmatrix} \mathbf{u}_{[t_1]} \\ \mathbf{u}_{[t_2]} \\ \vdots \\ \mathbf{u}_{[T_r]} \end{bmatrix}, \quad (54)$$

$\underline{\mathbf{u}}_r$  is the reduced column vector of the length  $2T_r$ . Its members are time predictions at certain times  $t_1, t_2, \dots, T_r$ . All matrices introduced in MPC must be reduced. The process is quite complicated and will not be described step by step.

In the figure 17 have been used prediction times  $1, 2, \dots, 9, 10, 20, 30, \dots, 100$ . The number of the prediction times is  $T_r$ . Using this particular vector, the  $T_r$  is 19 instead of  $T$  as 100. The acceleration of matrix multiplication can be computed as  $(T/T_r)^2 = 27.7$ . This means 27.7 times faster. The move blocking algorithm also creates the same memory savings with some matrices. This is a crucial improvement of the MPC algorithm, allowing to cover a longer prediction horizon without having too big matrices.

The penalization matrices  $\mathbf{Q}$  and  $\mathbf{P}$  need to be edited a little bit differently. Without move blocking, all predicted positions and inputs are penalized evenly. If now the predicted inputs represents inputs of different length, the penalization must be set accordingly. The main diagonal has to have its members individually multiplied by the corresponding constant of the number of the representing time steps.

## 9 Simulations and Hardware



**Figure 14:** UAV body

Developing the collision avoidance system just on the paper and then programming it directly to the UAV’s on-board computer is a very naive approach that very likely would not work. To minimize the probability of an expensive crash happening, the algorithm should be tested as much as possible.

### 9.1 Simulation

During the development of the MPC, all steps were being tested by Matlab simulations. This program is good for these purposes, because it is optimized for matrix multiplication, which is the core of he MPC computing. The matlab simulation consists of two separate modules: the Environment module and the UAV module.

#### 9.1.1 The Environment module

The Environment module simulates the flight. This simulates the physical world and the sensors of the UAV. It is initialized by the obstacles positions, the UAV’s initial condition and desired trajectory. It takes care of the UAV’s dynamics and simulates the flight. The main program runs in a loop. It receives the computed input actions from the UAV module and computes

the UAV's movement. Then it updates the relative obstacles positions, the absolute UAV position and the velocity and the desired trajectory. It then gives this information as an input to the UAV module. This module also sets the parameters for the UAV module, such as penalization of the position errors  $k_q$  and the input  $k_p$ , the indexes of the move blocking time predictions and gradient descend iterations and step size. This makes it easier to tune the overall constants.

It has also the ability to visualize the UAV, obstacles, the desired trajectory, the predicted trajectory and the convex feasible space. This graph updates with the loop, so it simulates the whole flight.

### 9.1.2 The UAV module

The second module simulates the MPC controller on UAV's on-board computer. It has the ability to compute the input actions based on the information received from the Environment module. It consists of two parts.

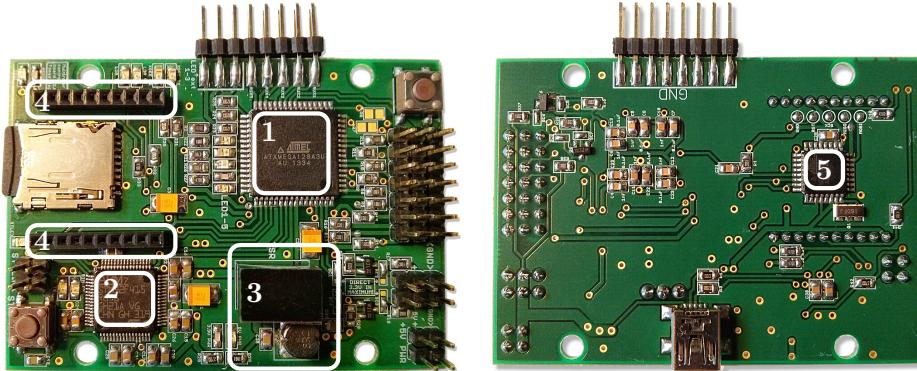
First, the MPC problem definition, which creates the constraining matrices  $\mathbf{A}_c$ ,  $\mathbf{B}_c$  from the information about the initial condition and obstacles positions. Only the objective function matrix  $\hat{\mathbf{c}}$  is created, because the  $\hat{\mathbf{H}}$  is a constant and can be stored in memory.

Second, the LCQP solver. This solver has been written by the method described in the Sec. 7.6. Its inputs are the gradient step size, the total number of iterations and the problem defining matrices, which are identical to the function quadprog from the optimization toolbox. It returns slightly different result, because it returns on purpose not strictly feasible solution.

## 9.2 UAV custom board

The UAV custom board [?] as shown in the Fig. ??, is a computer with very limited computational power compared to matlab. This board has been designed [?] especially for MPC control and communication. This board is whole programmed in the language C. It has two computational units: xMega and STM. For debugging and data logging is a socket for XBee and logging data to micro SD card. This board sticks to the philosophy of distributed computing.

The xMega is a slow computational unit. It has 8-bit architecture with 32 MHz clock and 8 kB SRAM. It is mainly handling the communication between sensors and STM. The communication between peripherals goes through 7 UART ports. Sensors or a computer can be connected. With the help of the program Putty, simple messages can be exchanged between the computer and the board. Most of the protocols have been already designed, but minor adjustments had to be done to fit the exact requirements of collision avoidance. For example the communication between the xMega and STM had to be extended to transfer information about obstacle positions.



(a) board's top

(b) board's bottom

**Figure 15:** Custom control board v.2, key components are placed at follows: 1 – xMega, 2 – STM, 3 – switching power supply, 4 – socket for XBee, 5 – data logging MCU.

The xMega also stores the desired trajectory or the location of the desired position(setpoint) and it sends this data to the STM.

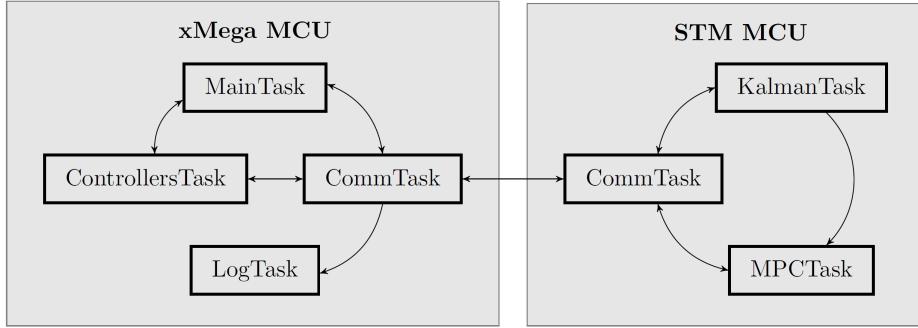
The STM is a powerful 32-bit unit with 168 MHz clock and 192 kB of RAM and 2 MB ROM. This runs three tasks:

- CommTask is responsible for communicating between the xMega and STM.
- KalmanTask is the state estimator task responsible for estimating positions, velocity, acceleration and input error based on information from the px4flow sensor and the inputs. When it computes the state, it sends it to the MPCTask.
- MPCTask is the task, that computes the MPC controller. The MPC is triggered by the KalmanTask, if the previous iteration has finished. This means, that if the MPC runs slower, than 70 Hz, it does not cause big problems, because the input actions are held, until they are replaced by a new ones. This mechanism allows tuning between the speed and accuracy of the MPC, because there is no need to regulate the UAV with 70 Hz.

Each task uses a third of the CPU time and 16 kB RAM. Because of very limited computational power, the MPCTask has to be programmed very efficiently.

### 9.3 Obstacles detection

A system WhyCon[?][?] is used for obstacle detection. It uses 3 Mobius Actioncam cameras, two pointing to the sides and one forwards. This allows



**Figure 16:** Block diagram of information flow between tasks of xMega and STM MCUs.

obstacle detection at  $270^\circ$ . Each camera has the capability of recording in the resolution 1920x1080, but for faster performance the video is shoot in 1280x720 and processed 640x480. The system is capable of precise detecting of multiple circular markers shown in Fig. ???. The 3-axis positions of these markers are then sent by UART to the custom board, where the xMega receives the information. The image processing runs in a loop on a computational unit NVIDIA Jetson TK1.

#### 9.4 MPC hardware implementation

The exact algorithm, as written in matlab, has been rewritten to the C code. This has been the most time demanding part of this thesis. The programmed code uses a CMATRIXLIB library, that allows simple matrix and vector operations. This library has been extended by more functions for the purposes of this thesis. Because the RAM is very limited, the matrices have to be divided into constant matrices, that are stored in ROM, and the changing matrices, which have allocated memory in RAM.

The constant matrices depend only on the system model and move blocking algorithm. These matrices are:  $\hat{\mathbf{A}}_x$ ,  $\hat{\mathbf{A}}_y$ ,  $\hat{\mathbf{B}}_x$ ,  $\hat{\mathbf{B}}_y$ ,  $(\hat{\mathbf{Q}}\hat{\mathbf{B}})^T$ ,  $\hat{\mathbf{A}}$  and  $\hat{\mathbf{H}}$ . They had been adapted by the move blocking algorithm, generated in matlab and stored in the ROM memory of the STM unit.

The UAV's max speed has been set to  $0.35 \text{ ms}^{-1}$ . This is enforced by editing the desired trajectory. If the trajectory doesn't violate the maximum speed, it is used unchanged. In the other case, the desired positions are moved closer to the UAV. If given only setpoint, the desired trajectory is created as a straight line with the maximum speed.

The input action can not be infinite and some method of constraint has to be applied. Applying the constraints in the MPC control slows the algorithm down. A saturation of the output has been used instead.

## 9.5 Hardware in the loop

This is a method of testing embedded hardware. The custom board is connected to a computer instead of UAV's body. The board's inputs are provided by the computer and outputs are sent back. The board is in the same situation, as if it would be flying. This has been simulated in two experiments.

The first one was connecting the board to the computer using the putty terminal. The inputs have been sent through the terminal and outputs were received. These outputs were compared with the matlab simulations, to check if they match. The UAV has been tested in various situations, such as different initial conditions, desired trajectories and obstacle positions. This allowed to test and debug the various segments of the code, as well as the MPC algorithm as whole. This method also allowed to test xMega as well as STM. After successfully finishing this testing, it was sure, that the implemented algorithm behaves exactly in the same way as the simulation.

The second experiment was connecting the board to the UAV, without the ability of controlling the motors. It received inputs from the UAV's sensors, such as velocity and obstacle positions. The board was connected to matlab using wireless Xbee. The Xbee communication protocol has been extended for the important information to be transferred. A matlab visualization allowed to see the predicted trajectory in real time without the need to be connected to the UAV with a cable. This allowed testing the algorithm in real situations without worrying of damaging the UAV. It also allowed testing communication with the on-board sensors. The frequency of the MPC loop has been measured close to 20 Hz. This is an sufficient regulator frequency.

After testing in various situations and behaving correctly, it was time to do a real flight experiment.

## 10 Experiments

The implemented MPC controller has been verified by a real world experiments. Simulations, that have been done, differ from the real world experiments mainly by the absence of sensory noise. Because of the construction of the UAV with 4 propellers on sides, most collisions can be dangerous not only for the UAV, but for the operator as well. That is the reason, why the operator has to have the access to take over the control any time during these experiments. During the described experiment this possibility has not been used and the presented data are solely while the MPC was in control.



**Figure 17:** Photo of the obstacle avoidance experiment.

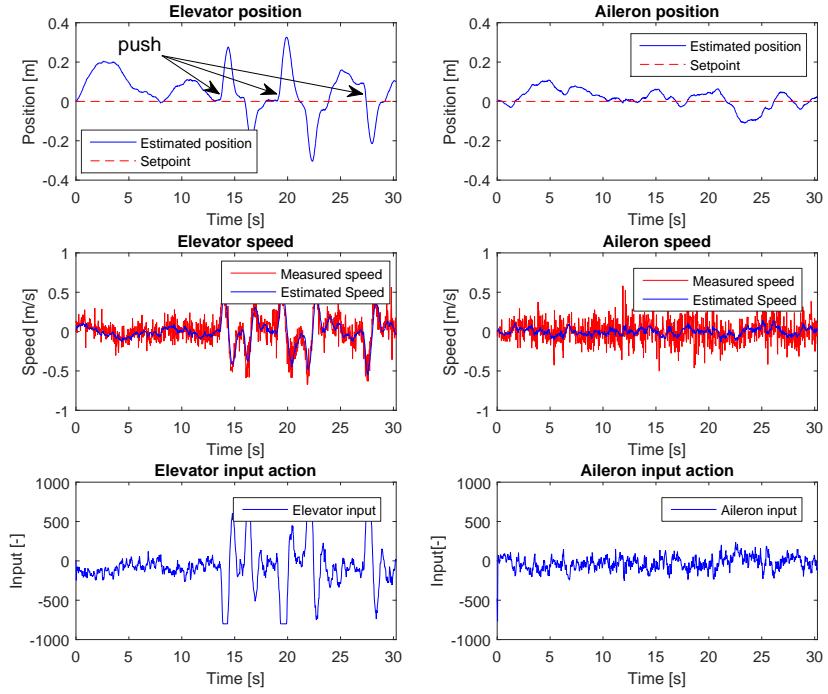
### 10.1 Stabilization

The first experiment was to test the stability and settle time of the MPC. The position setpoint was set to the origin of the world coordinate system. The task for the UAV was to track this setpoint while the UAV was physically pushed by the operator. This tested reactions of the controller to an outside disturbances.

The Fig. 18 shows the position, speed and input action during the stabilization experiment. During the experiment, the UAV has been three times pushed in the positive direction of the elevator axis. The UAV returned then to the original position. The aileron position stayed away from the setpoint within 11 cm, which is a good performance considering the size of the UAV's body. At the elevator axis it is 20 cm if not considering the disturbances.

### 10.2 Obstacle avoidance

The obstacle avoidance system has been tested. The obstacle was represented by a blob - recognizable pattern by the camera. This was especially risky experiment, because the UAV had to get close to the obstacle. The maximum speed was set to  $0.2 \text{ ms}^{-1}$  for the operator to have enough time to take control of the UAV in case incorrect behavior. The UAV was given a setpoint 4 m in the front. The obstacle was in the middle between the UAV's initial position and the setpoint and did not move. The obstacle



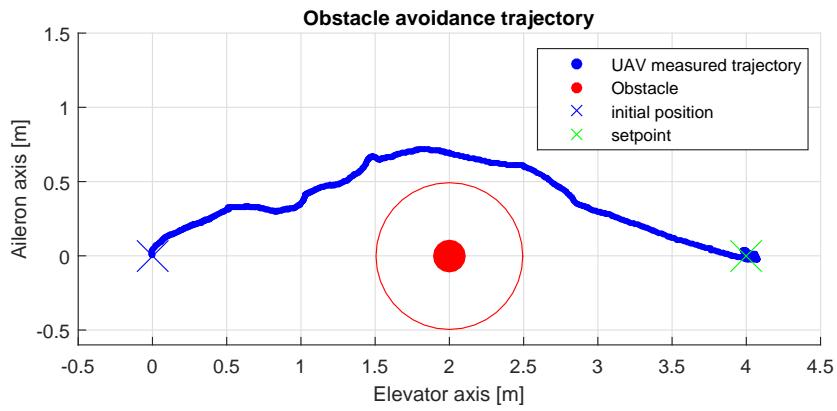
**Figure 18:** Data from the stabilization experiment

was given a predefined diameter of 0.5 m. The trajectory was being created on-board. This way the algorithm of creating trajectory was tested as well. The task was to avoid the obstacle and arrive to the setpoint position.

The fig. 19 shows the UAV's obstacle avoidance trajectory. Around the obstacle is a red circle of 0.5 m, which is the minimal distance allowed from the obstacle. The figure also shows, that there is a safe distance from the obstacle, which was enforced by the barrier function. The UAV also finishes on the setpoint and stabilizes there.

## 11 Conclusion

In this thesis a Model predictive controller for UAV has been developed. It is capable of trajectory or setpoint tracking and obstacle avoidance. The MPC algorithm and linearly constrained quadratic programming solver has been designed and tested by matlab simulations. The solution was then implemented on embedded hardware and tested again. Real world experiments have been successfully conducted. The UAV was capable of trajectory tracking while detecting and avoiding an obstacle. The entire assignment of this thesis has been successfully fulfilled.



**Figure 19:** Obstacle avoidance experiment

### 11.1 Future work

Although simulations and experiments have been successfully executed, there is still a room for future improvements. The greatest limitation of the introduced algorithm is keeping the convexity of the mathematical optimization. If this condition was lost, the optimization problem would have local minimums, but the designed prediction space could be much more complex, allowing the UAV to work with more complicated obstacles. This would allow UAV to fly faster and safer.