– umi se vyhybat pohybujicim se objektum – 2 positions in convex space -¿ all trajectory – zmit polomr helikoptry – algoritmus setpointu

1 Model predictive control unconstrained

1.1 UAV dynamics

For understanding the UAV's controller, it is important to understand the physical properties of UAV. It has 4 rotors,

1.2 MPC Introduction

MPC is an advanced regulator. It using prediction of future states of the system to determine system input actions. This prediction runs constantly in a loop. Because of it's computational demands, it is used mainly in processes with long time constants. A motivation for it's development has been control of chemical processes, where the computational time is not limiting. Using MPC for controlling UAV is a big challenge, because it is hard to implement on embedded hardware. Controlling real time system, such as UAV, requires regulation in tens of Hz, giving the hardware very little time to compute such a complex problem. The MPC needs to know the system's state space model, initial condition and, unlike other controllers, a sequence of desired future states. A great advantage of MPC is applying large variety of constraints, which can be useful for example in the regulation of chemical processes. On the other hand, MPC is sensitive to the model inaccuracy and to sensory noise. Output of the 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. This can be also very useful. Unlike standard controllers like PID, MPC can adjust the input action based on future demands.

– designed to be robust

1.3 Formulation of quadratic programming

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

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

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

1.4 Coordinate system

This whole task will be solved in 2D only. In these 2 dimensions, there is an aileron axis x with the direction to the right and elevator axis y with the

direction forward. These both axes are perpendicular. There are two coordinate systems, which will be used. The first one is a standard world coordinate system W. The second one is a coordinate system U, which is a system with the origin in the center of the mass of UAV. 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 axis 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 would rotate over time, the model would no more be linear and control of this system would be much more complex, therefore slower. These coordinate systems transformations work as

$$\begin{aligned} x^{(W)} &= x^{(U)} + \Delta x \\ y^{(W)} &= y^{(U)} + \Delta y \\ \dot{x}^{(W)} &= \dot{x}^{(U)} + \Delta \dot{x} \\ \dot{y}^{(W)} &= \dot{y}^{(U)} + \Delta \dot{y} \\ \ddot{x}^{(W)} &= \ddot{x}^{(U)} + \Delta \ddot{x} \\ \ddot{y}^{(W)} &= \ddot{y}^{(U)} + \Delta \ddot{y}. \end{aligned} \tag{2}$$

In the following text, the world coordination system W will be used unless stated otherwise. The UAV has it's own size. However, it is complicated to compute with the whole UAV model. It is much easier to proximate the UAV's body with a single mass point in it's center with a constant orientation. This way only just one position can be easily observed and no information will be lost. The height of the UAV is controlled separately. There are several reasons to do that. The first one is, that most applications use constant height, for example building interiors. The desired trajectory is usually also given in 2D. The second reason is, as mentioned above, MPC is very demanding on computing time. To save computational capacity, standard PID controller can be used instead. The height PID controller has already been implemented [**?**] and it is will not be part of this thesis.

1.5 One axis model

The UAV model has been already analyzed –cite Tomas–. Thanks to symmetrical body of the UAV, both axis have the same model. For one axis, the states take form of $\vec{x}_x = (x_x, \dot{x}_x, \ddot{x}_x)^T$ and $\vec{x}_y = (x_y, \dot{x}_y, \ddot{x}_y)^T$, 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

$$\vec{x}_{x,y,[t+1]} = \mathbf{A}_s \vec{x}_{x,y,[t]} + \mathbf{B}_s u_{x,y,[t]}, \tag{3}$$

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

$$\mathbf{A}_s = \begin{bmatrix} 1 & \Delta t & 0 \\ 0 & 1 & \Delta t \\ 0 & 0 & p_1 \end{bmatrix}, \mathbf{B}_s = \begin{bmatrix} 0 \\ 0 \\ p_2 \end{bmatrix}, \tag{4}$$

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.

1.6 Linked/Extended model

For more complex constraints, such as position constraints, one axis position is a function of the other axis position. For these kinds of constraints more complicated system is needed. The state space system must be preserved, connecting both identical systems for each axes into one system extending equation 5 into

$$\mathbf{x}_{[t+1]} = \mathbf{A}\mathbf{x}_{[t]} + \mathbf{B}\mathbf{u}_{[t]} \tag{5}$$

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 $\mathbf{x}_{[t]} = (x_{[t]}, \dot{x}_{[t]}, \ddot{x}_{[t]}, y_{[t]}, \dot{y}_{[t]}, \ddot{y}_{[t]})^T$ contains positions x, y and theirs derivatives at the time t.

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

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_s & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_s \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_s & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_s \end{bmatrix}. \tag{6}$$

1.7 Kalman

As mentioned in introduction, MPC is very sensitive to noise and errors. Wrong initial condition can result in a bad prediction because of the double integration of acceleration into position. To work properly, the MPC has to have access to very accurate initial condition. These measurements are position, speed and acceleration in both axis. An Kalman estimator has been already implemented -cite Tomas- to estimate all states of the UAV. Above all of that, it estimates disturbances of the acceleration.

The condition

1.7.1

1.8 System prediction

The MPC algorithm is based on predicting future states based on initial condition and system input. Such a general equation must be found. Let's repeat the equation 5.

$$\mathbf{x}_{[t+1]} = \mathbf{A}\mathbf{x}_{[t]} + \mathbf{B}\mathbf{u}_{[t]}, \tag{7}$$

With a simple substitution we can get prediction of the states at the time $t = 2$.

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

The equation 8 can be rewritten in a more general way:

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

Let's combine the sequence of predicted states into one vector $\underline{\mathbf{x}} = (\mathbf{x}_{[1]}^T, \mathbf{x}_{[2]}^T, ..., \mathbf{x}_{[T]}^T)^T$ and the sequence of inputs into $\underline{\mathbf{u}} = (\mathbf{u}_{x,[0]}, \mathbf{u}_{y,[0]}, \mathbf{u}_{x,[1]}, \mathbf{u}_{y,[1]}, ..., \mathbf{u}_{x,[T-1]}, \mathbf{u}_{y,[T-1]})^T$ With this notation, equation 9 can be represented as a simple matrix multiplication.

$$
\underbrace{\begin{bmatrix} \mathbf{x}_{[1]} \\ \mathbf{x}_{[2]} \\ \vdots \\ \mathbf{x}_{[T]} \end{bmatrix}}_{\underline{\mathbf{x}}} = \underbrace{\begin{bmatrix} \mathbf{A} \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^{(T-1)} \end{bmatrix}}_{\hat{\mathbf{A}}} \mathbf{x}_{[0]} + \underbrace{\begin{bmatrix} \mathbf{B} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}\mathbf{B} & \mathbf{B} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{(T-1)}\mathbf{B} & \mathbf{A}^{(T-2)}\mathbf{B} & \dots & \mathbf{B} \end{bmatrix}}_{\hat{\mathbf{B}}} \cdot \underbrace{\begin{bmatrix} \mathbf{u}_{[0]} \\ \mathbf{u}_{[1]} \\ \vdots \\ \mathbf{u}_{[T-1]} \end{bmatrix}}_{\underline{\mathbf{u}}} \tag{10}
$$

Using this new notation, we can rewrite it in a simpler form

$$
\underline{\mathbf{x}} = \hat{\mathbf{A}}\mathbf{x}_{[0]} + \hat{\mathbf{B}}\underline{\mathbf{u}}. \tag{11}
$$

2 Problem formulation

As mentioned in a section ???, MPC uses a quadratic optimization problem. Let's first solve the problem, where obstacles are not involved.

2.1 Trajectory

For every task, there is given a desired trajectory $\underline{\mathbf{x}}_d = (\mathbf{x}_{d,[1]}^T, \mathbf{x}_{d,[2]}^T, ..., \mathbf{x}_{d,[T]}^T)^T$, 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 outside position. However, The UAV desired velocity is already given by the desired positions at certain time as the distance $d = \sqrt{(x_{d,[t]} - x_{d,[t+1]})^2 + (y_{d,[t]} - y_{d,[t+1]})^2}$. The same can be applied for acceleration. Therefore, the velocity and acceleration is demanded in the sequence of desired positions, rather than the the states. The desired velocity and acceleration is than ignored, which will be discussed in a section ??? and it can hold any value, for example 0. The $\mathbf{x}_{d,[t]}$

than 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.

2.2 Objective function

As in many other areas of engineering, the problem can be solved in 2 independent steps. The first one is creating an objective function, sometimes called cost function. This function describes, how good is the particular solution, in this case vector $\underline{\mathbf{u}}$. It is usually a vector or scalar. The lower is the value of the objective function, the better is the particular solution. If found a solution with the minimal objective function, it can be said, that this is the optimal solution. In some cases, a fitness function can be created. In this case, the higher the value, the better the solution and the goal is the maximize this function. These functions are very similar and can be created by multiplying -1 the other function.

The goal of unconstrained MPC is to follow the given trajectory as well as possible. This means, that we want to minimize the error between all the predicted positions and the desired positions, gaining error:

$$\begin{aligned} e_{x,t} &= x_{[t]} - x_{d,[t]} \\ e_{y,t} &= y_{[t]} - y_{d,[t]} \end{aligned} \tag{12}$$

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 [**?**]. This is a very useful property of being able to get gradient of the final objective function. 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 the 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 following 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 penalized also. This combined together, we get the following objective function

$$\mathrm{V}\left(\underline{\mathbf{x}}, \underline{\mathbf{u}}\right) = \frac{1}{2} \sum_{i=1}^{T} \left(k_q \cdot (e_{x,[i]}^2 + e_{y,[i]}^2) + k_s \cdot (u_{x,[i-1]}^2 + u_{y,[i-1]}^2) \right) \tag{13}$$

where k_q and k_s are constants, whose ratio 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 on the input

action $\underline{\mathbf{u}}$. Because the function is to be optimized, this error can be left out. The equation 13 can be rewritten in a matrix form using penalizing matrices $\mathbf{Q}$ and $\mathbf{S}$

$$\mathrm{V}\left(\underline{\mathbf{x}}, \underline{\mathbf{u}}\right) = \frac{1}{2} \sum_{i=1}^{T} \left(\mathbf{e}_{[i]}^T \mathbf{Q} \mathbf{e}_{[i]} + \mathbf{u}_{[i-1]}^T \mathbf{P} \mathbf{u}_{[i-1]} \right) \tag{14}$$

where $\mathbf{e}_{[t]} = \mathbf{x}_{[t]} - \mathbf{x}_{d,[t]}$ is the error of all states at time t and matrices $\mathbf{Q}$ and $\mathbf{S}$ are

$$\mathbf{Q} = \begin{bmatrix} k_q & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & k_q & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{P} = \begin{bmatrix} k_p & 0 \\ 0 & k_p \end{bmatrix}. \tag{15}$$

This form of $\mathbf{Q}$ allows to penalize only position errors and ignore the velocity and acceleration errors. To ensure, that the function $\mathrm{V}\left(\underline{\mathbf{x}}, \underline{\mathbf{u}}\right)$ is strictly convex, the matrix $\mathbf{Q}$ must be positive semi-definite ($\mathbf{Q} \succeq 0$) and $\mathbf{P}$ must be positive definite ($\mathbf{P} \succ 0$) –cite Tom–. The matrix $\mathbf{Q}$ has it's 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 – cite ??? –, 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 in section ???.

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

$$\hat{\mathbf{Q}} = \begin{bmatrix} \mathbf{Q} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{Q} & \dots & \vdots \\ \mathbf{0} & \dots & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & \mathbf{Q} \end{bmatrix}, \hat{\mathbf{P}} = \begin{bmatrix} \mathbf{P} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{P} & \dots & \vdots \\ \mathbf{0} & \dots & \ddots & \vdots \\ \mathbf{0} & \dots & \dots & \mathbf{P} \end{bmatrix}. \tag{16}$$

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 equation 14 using the equation 11.

$$\begin{aligned} \mathrm{J}(\underline{\mathbf{u}}) &= \frac{1}{2} \left(\underline{\mathbf{e}}^T \hat{\mathbf{Q}} \underline{\mathbf{e}} + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} \right) \\ &= \frac{1}{2} \left((\hat{\mathbf{A}} \mathbf{x}_{[0]} + \hat{\mathbf{B}} \underline{\mathbf{u}} - \underline{\tilde{\mathbf{x}}})^T \hat{\mathbf{Q}} (\hat{\mathbf{A}} \mathbf{x}_{[0]} + \hat{\mathbf{B}} \underline{\mathbf{u}} - \underline{\tilde{\mathbf{x}}}) + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} \right) \\ &= \frac{1}{2} \left(2(\hat{\mathbf{Q}} \hat{\mathbf{B}})^T \hat{\mathbf{A}} \mathbf{x}_{[0]} \underline{\mathbf{u}} + \underline{\mathbf{u}}^T \hat{\mathbf{B}}^T \hat{\mathbf{Q}} \hat{\mathbf{B}} \underline{\mathbf{u}} - 2(\hat{\mathbf{Q}} \hat{\mathbf{B}})^T \underline{\tilde{\mathbf{x}}}\, \underline{\mathbf{u}} + \underline{\mathbf{u}}^T \hat{\mathbf{P}} \underline{\mathbf{u}} + const. \right) \end{aligned} \tag{17}$$

In the equation 17, all constant elements have been put into the *const* part. Because the goal is to minimize the objective function $\mathrm{J}(\underline{\mathbf{u}})$, parts of the equation, that do not depend on the $\underline{\mathbf{u}}$ can be left out. The equation 17 can be rewritten as

$$\mathrm{J}(\underline{\mathbf{u}}) = \frac{1}{2}\underline{\mathbf{u}}^T \underbrace{\left(\hat{\mathbf{B}}^T \hat{\mathbf{Q}} \hat{\mathbf{B}} + \hat{\mathbf{P}}\right)}_{\hat{\mathbf{H}}} \underline{\mathbf{u}} + \underbrace{\left(\hat{\mathbf{Q}}\hat{\mathbf{B}}\right)^T \left(\hat{\mathbf{A}}\mathbf{x}_{[0]} - \underline{\tilde{\mathbf{x}}}\right)}_{\hat{\mathbf{c}}} \underline{\mathbf{u}}. \tag{18}$$

The task can be formulated as

$$\begin{aligned} \min_{\underline{\mathbf{u}} \in \mathrm{R}^{kM}} \quad & \mathrm{J}(\underline{\mathbf{u}}) = \frac{1}{2}\underline{\mathbf{u}}^T \hat{\mathbf{H}} \underline{\mathbf{u}} + \hat{\mathbf{c}}\underline{\mathbf{u}} \\ \text{s.t.} \quad & \mathbf{A}_c \underline{\mathbf{u}} \leq \mathbf{B}_c \end{aligned} \tag{19}$$

The constraints will be further discussed in the section **??**, for unconstrained system we can ignore them. Solving unconstrained MPC task is very easy.

2.3 solving QP unconstrained

We want to solve the problem 19 while ignoring the constraints. For this task to have a single solution $\underline{\mathbf{u}}^\star$, 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 [**?**].

$$\nabla \mathrm{J}(\underline{\mathbf{u}}) = \hat{\mathbf{H}}\underline{\mathbf{u}} + \hat{\mathbf{c}} \tag{20}$$

The gradient $\nabla \mathrm{J}(\underline{\mathbf{u}}) \in R^{2 \cdot T}$ is a vector, with the opposite direction to the global minimum, so the $-\nabla \mathrm{J}(\underline{\mathbf{u}})$ is a vector pointing the direction towards the global minimum $\underline{\mathbf{u}}^\star$. For solving this task, gradient descent algorithm can be used, but also analytic solution can be found. Because the $\hat{\mathbf{H}}$ is positive semi-definite, the quadratic form $\frac{1}{2}\underline{\mathbf{u}}^T \hat{\mathbf{H}} \underline{\mathbf{u}}$ is convex. $\hat{\mathbf{c}}\underline{\mathbf{u}}$ 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

$$\begin{aligned} \nabla \mathrm{J}(\underline{\mathbf{u}}^\star) = \hat{\mathbf{H}}\underline{\mathbf{u}}^\star + \hat{\mathbf{c}} &= 0 \\ \hat{\mathbf{H}}\underline{\mathbf{u}}^\star &= -\hat{\mathbf{c}} \\ \underline{\mathbf{u}}^\star &= -\hat{\mathbf{H}}^{-1}\hat{\mathbf{c}} \end{aligned} \tag{21}$$

The inverse of $\hat{\mathbf{H}}$ can be done, because $\hat{\mathbf{H}}$ is positive-definite, therefore regular. This is guaranteed by the definition of positive-definite property, that it's leading principal minors are all positive [**?**]. On-board computation is very 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 on computer and stored as a constant in the UAV's memory. Finding the $\underline{\mathbf{u}}^\star$ is only matter of creating matrix $\hat{\mathbf{c}}$ and matrix multiplication.

2.4 system constraints

One of the greatest advantages of MPC is being able to apply large variety of constraints. As mentioned in 1, the MPC uses linearly constraint quadratic programming for finding the input action prediction. There can be many different constraints required and the solution has to obey all of them. These constraints take form of linear inequalities

$$\vec{\omega}_i \cdot \underline{\mathbf{u}} \leq b_i, \quad i \in \{1, 2, ..., m\} \tag{22}$$

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 in a space $\mathbf{R}^{2T}$ constraining it by hyperplane with a normal vector $\vec{\omega}_i$ shifted by b_i. To apply multiple constraints at the same time, the equation 22 can be rewritten in matrix form as

$$\mathbf{A}_c \underline{\mathbf{u}} \leq \mathbf{B}_c \tag{23}$$

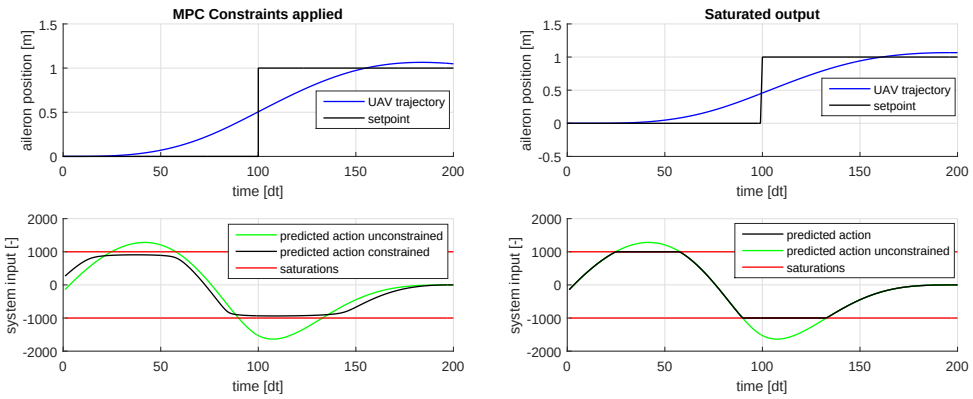
where $\mathbf{A}_c$ is a matrix of the width 2T and height m - the number of constraints applied. $\mathbf{B}_c$ is a column vector of the same height created as

$$\mathbf{A}_c = \begin{bmatrix} \vec{\omega}_1 \\ \vec{\omega}_2 \\ ... \\ \vec{\omega}_m \end{bmatrix}, \mathbf{B}_c = \begin{bmatrix} b_1 \\ b_2 \\ ... \\ b_m \end{bmatrix}, \tag{24}$$

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

2.4.1 Input constraints

In control, one of the biggest problems 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 capapibilities [**?**]. For example motor can spin in a certain maximum speed despite input voltage. Also, if the system receives too high input actions from the controller, it can be destroyed. 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 kind of problem is simply saturating the output of the controller. This is a very simple solution and for many tasks it is enough. However, because the controller doesn't know about this saturation, the system can behave incorrectly. The great 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



(a) constrained MPC. (b) saturated MPC.

Figure 1: Step response of the system using two approaches.

and at the same time will achieve the desired output. This is achieved by the constraints. The system saturation thresholds are u_{min} and u_{max}

We can see in figure 1b simple saturated output.

3 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 1.4 has been introduced, that the UAV is approximated with a mass point. 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.

3.1 Convexity

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

$$\forall\{x_1, x_2\} \in S, \forall t \in <0, 1> \implies t \cdot x_1 + (1 - t) \cdot x_2 \in S \tag{25}$$

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 proven in section ???. Lets introduce a space S of dimension $2T+1$, where the axes will be $(\underline{\mathbf{u}}^T, z)$. Than introduce a subset J_s of the space S as $\{J_s \in S : \mathrm{J}(\underline{\mathbf{u}}) \leq z\}$. The function J($\underline{\mathbf{u}}$) is convex, if and only if the subset J_s is convex.

3.1.1 Convexity of u

Lets have a look at the vector $\underline{\mathbf{u}}$. For MPC purposes this vector is 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 23 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 subsection **??**, 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.

3.1.2 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 2 in only 2 dimensions constrained by 3 half spaces.

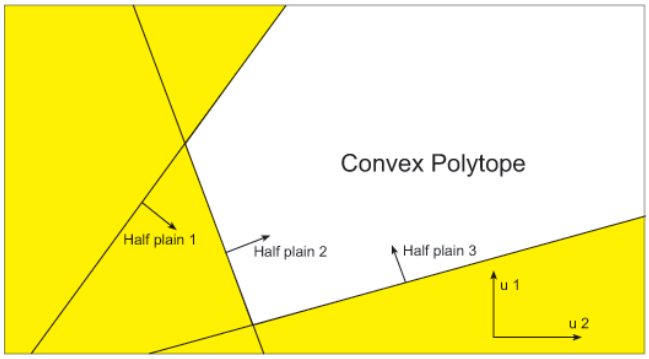


Figure 2: convex polytope.

At this time is left to prove, that a single half space is a convex set. Half space is described in equation 22 as $\{\underline{\mathbf{u}} : \omega_i \cdot \underline{\mathbf{u}} \leq b_i \text{ for } i \in \{1, 2, ..., m\}\}$. If we take any 2 points $\underline{\mathbf{u}}_1, \underline{\mathbf{u}}_2$ belonging to the half space given by the normal vector ω_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

$$t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2; \quad t \in <0,1> \tag{26}$$

The problem of proving convexity has been transformed using the equation 26 into equation

$$\omega_i \cdot (t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) \leq b_i \tag{27}$$

An assumption has been made, that $\underline{\mathbf{u}}_1$ and $\underline{\mathbf{u}}_2$ belong to the half space, therefore

$$\begin{aligned} \omega_i \cdot \underline{\mathbf{u}}_1 &\leq b_i \\ \omega_i \cdot \underline{\mathbf{u}}_2 &\leq b_i. \end{aligned} \tag{28}$$

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 $\omega_i \cdot \underline{\mathbf{u}}_1 = \omega_i \cdot \underline{\mathbf{u}}_2$:

$$\begin{aligned} \omega_i \cdot (t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq b_i \\ \omega_i \cdot (\underline{\mathbf{u}}_1 (t+1-t) &\leq b_i \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq b_i \end{aligned} \tag{29}$$

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

$$\begin{aligned} \omega_i (t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq \omega_i \cdot \underline{\mathbf{u}}_2 \\ \omega_i (t \cdot \underline{\mathbf{u}}_1 - t \cdot \underline{\mathbf{u}}_2) &\leq 0 \\ t \cdot \omega_i (\underline{\mathbf{u}}_1 - \underline{\mathbf{u}}_2) &\leq 0 \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq \omega_i \cdot \underline{\mathbf{u}}_2 \end{aligned} \tag{30}$$

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.

$$\begin{aligned} \omega_i (t \cdot \underline{\mathbf{u}}_1 + (1-t) \cdot \underline{\mathbf{u}}_2) &\leq \omega_i \cdot \underline{\mathbf{u}}_1 \\ \omega_i ((t-1) \cdot \underline{\mathbf{u}}_1 - (t-1) \cdot \underline{\mathbf{u}}_2) &\leq 0 \\ (t-1) \cdot \omega_i \cdot (\underline{\mathbf{u}}_1 - \underline{\mathbf{u}}_2) &\leq 0 \\ \omega_i \cdot \underline{\mathbf{u}}_1 &\leq \omega_i \cdot \underline{\mathbf{u}}_2 \end{aligned} \tag{31}$$

Similarly to the equation 30, the third line of the equation 31 has been divided by $t-1$. It has been proven, that all $\underline{\mathbf{u}}$, that obey the equation 23 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.

4 Obstacles

Through all this thesis, obstacles will be represented in 2D only by circles. These representations have 3 parameters: position x_{obs}, position y_{obs} and diameter r_{obs}. Of course, obstacles can have various shapes, but the final allowed space will be very simplified and this is a good representation of for example people. Because the MPC runs constantly in a loop and reacts to the changing environment, it also works great with moving objects. Until now, the UAV body has been approximated with one mass point without considering the real size of the UAV. It would be difficult to calculate, whether the UAV's body has collided in the particular position or not. There is an algorithm using Minkovski Sum. Instead computing with the real size of the UAV, the representation of all obstacles will change according to the UAV's size. Let's create a circle with the center in the UAV's center, that will completely wrap the UAV. This circle will have the smallest possible diameter R_{UAV} as shown in the figure 3.

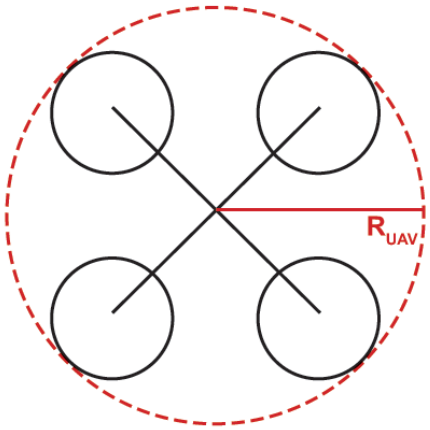


Figure 3: UAV dimensions

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

$$(x_{obs} - x)^2 + (y_{obs} - y)^2 \geq (R_{UAV} + r_{obs})^2. \tag{32}$$

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

Obstacle avoidance task is actually a task for searching a path between obstacles trying to follow a desired trajectory. From the model equation an initial condition x_0 and vector of input prediction $\underline{\mathbf{u}}$ can be transformed as a vector of positions, velocities and accelerations at a certain predicted times. It is just positions we are interested in.

Let's make an assumption, that the UAV's trajectory is a line created by connecting all the predicted positions. Because the UAV can not change vector of speed immediately, the real trajectory has a continuous first time derivative. Also because the second derivative is a result of the UAV's pitch and roll, which also can't 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. It is very similar to approximating a function with the Taylor polynomial. To be able to approximate the real trajectory by the simplified trajectory, we need to make sure, that the predicted positions are far smaller, than the obstacles sizes. This is easy to achieve, because the time between the predicted positions is $\Delta t = 1/70s$.

Easy approach would be to say, that every trajectory is feasible, if any predicted position does not collide with any obstacle. However if the obstacle is just a simple circle in the way, the allowed space is highly non-convex. As mentioned in the section 3.1, for keeping the quadratic programming convex, the allowed space for UAV has to be convex. This is the greatest disadvantage of using MPC for obstacle avoidance.

4.1 Creating allowed space

There are many ways of creating a convex space from the knowledge of the obstacles. The approach for creating a convex allowed space is restricting any position 'behind' any obstacle. This space would be polytope defined by a set of half planes. These plains are constantly adjusting to the relative positions of the obstacles, which positions are continuously changing due to the change of the UAV position and the noise of the cameras. Let's first look at a situation with only one obstacle.

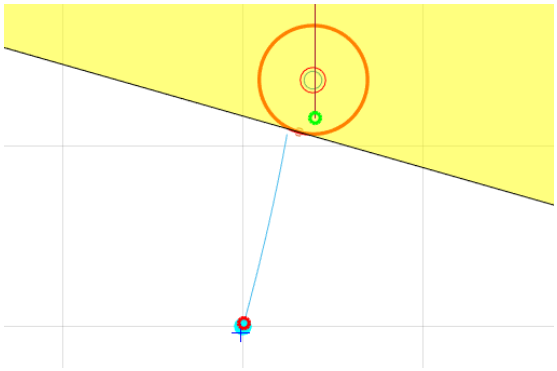


Figure 4: half plane, –template of the final picture

Half plane can be 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

$$s(y - kx - q) \leq 0 \tag{33}$$

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

$$\begin{aligned} k &= -\frac{y_{obs} - y_{UAV}}{x_{obs} - x_{UAV}} \\ b_x &= (x_{obs} - x_{UAV}) \cdot \big(1 - \frac{r_{obs} + R_{UAV}}{\sqrt{(x_{obs} - x)^2 + (y_{obs} - y)^2}}\big) \\ b_y &= (y_{obs} - y_{UAV}) \cdot \big(1 - \frac{r_{obs} + R_{UAV}}{\sqrt{(x_{obs} - x)^2 + (y_{obs} - y)^2}}\big) \\ q &= b_y - kb_x \\ s &= sign(y). \end{aligned} \tag{34}$$

The b_x and b_y is a position of the intersection of the constraining line and the obstacle circle with diameter $R_{UAV} + r_{obs}$. Until now, we have been computing in the space of positions. These constraints have to be transformed into an input action constraints. This is an inverse transformation to what we have been doing until now. For the purposes of MPC matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ have to be found to fit the condition 23. We already know the transformation 11 of input actions and initial condition to the predicted states. However, the vector $\underline{\mathbf{x}}$ contains all the predicted states for both axis. We are interested in separating just positions x and y into the column vectors $\vec{x}$ and $\vec{y}$ of size T as

$$\vec{x} = \begin{bmatrix} x_{[1]} \\ x_{[2]} \\ \dots \\ x_{[T]} \end{bmatrix}, \vec{y} = \begin{bmatrix} y_{[1]} \\ y_{[2]} \\ \dots \\ y_{[T]} \end{bmatrix}, \tag{35}$$

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 found with the submatrices $\hat{\mathbf{A}}_x$, $\hat{\mathbf{B}}_x$, $\hat{\mathbf{A}}_y$, $\hat{\mathbf{B}}_y$ as

$$\begin{aligned} \vec{x} &= \hat{\mathbf{A}}_x \mathbf{x}_{[0]} + \hat{\mathbf{B}}_x \underline{\mathbf{u}} \\ \vec{y} &= \hat{\mathbf{A}}_y \mathbf{x}_{[0]} + \hat{\mathbf{B}}_y \underline{\mathbf{u}}, \end{aligned} \tag{36}$$

where the matrices $\hat{\mathbf{A}}_x, \hat{\mathbf{A}}_y \in R^{T \times 6}$ and $\hat{\mathbf{B}}_x, \hat{\mathbf{B}}_y \in R^{T \times 2T}$. These matrices are constant and for the computation purposes can be computed in advance and stored in memory. They take form of

$$\mathbf{A}_x = \begin{bmatrix} \mathbf{A}_{1,1:6} \\ \mathbf{A}_{1,1:6}^2 \\ \vdots \\ \mathbf{A}_{1,1:6}^{(T-1)} \end{bmatrix}, \mathbf{A}_y = \begin{bmatrix} \mathbf{A}_{4,1:6} \\ \mathbf{A}_{4,1:6}^2 \\ \vdots \\ \mathbf{A}_{4,1:6}^{(T-1)} \end{bmatrix}$$

aa

$$\mathbf{B}_x = \begin{bmatrix} \mathbf{B}_{1,1:2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{1,1:6}\mathbf{B} & \mathbf{B}_{1,1:2} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{1,1:6}^{(T-1)}\mathbf{B} & \mathbf{A}_{1,1:6}^{(T-2)}\mathbf{B} & \dots & \mathbf{B}_{1,1:2} \end{bmatrix}, \mathbf{B}_y = \begin{bmatrix} \mathbf{B}_{4,1:2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{4,1:6}\mathbf{B} & \mathbf{B}_{4,1:2} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{4,1:6}^{(T-1)}\mathbf{B} & \mathbf{A}_{4,1:6}^{(T-2)}\mathbf{B} & \dots & \mathbf{B}_{4,1:2} \end{bmatrix} \tag{37}$$

where $\mathbf{A}_{i_1,i_2:i_3}$ 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 equation 33 is defined only for one position. It is important, that the UAV does not collide in any predicted position. Than the equation can be rewritten in the form

$$s(\vec{y} - k\vec{x} - \vec{q}) \leq 0. \tag{38}$$

$\vec{q}$ is the column vector of the size T and every member is q. After the substitution of equation 36 into equation 38 we get

$$\begin{aligned} s\big(\hat{\mathbf{A}}_y\mathbf{x}_{[0]} + \hat{\mathbf{B}}_y\underline{\mathbf{u}} - k(\hat{\mathbf{A}}_x\mathbf{x}_{[0]} + \hat{\mathbf{B}}_x\underline{\mathbf{u}}) - \vec{q}\big) &\leq 0 \\ \underbrace{s\big(\hat{\mathbf{B}}_y - k\hat{\mathbf{B}}_x\big)}_{\hat{\mathbf{A}}_c}\underline{\mathbf{u}} + \underbrace{s\big(\hat{\mathbf{A}}_y\mathbf{x}_{[0]} - k\hat{\mathbf{A}}_x\mathbf{x}_{[0]} - k\vec{q}\big)}_{\mathbf{B}_c} &\leq 0. \end{aligned} \tag{39}$$

One more improvement can be done to this algorithm. In this version, all the half planes of the obstacles are computed from the relative position of the initial condition. 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 that one. The MPC loop runs in tens of Hz and the UAV doesn't change it's position very much in the one time step. This improvement is done by computing the previous trajectory using equation 11. Than the constants k_i, q_i, s_i need to be found for each predicted position $i, i \in \{1, ..., T\}$ using the equation 34. Than the equation 38 can be rewritten in a matrix form as

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

where $\mathbf{s}$ and $\mathbf{k}$ are square diagonal matrices of the size T and $\vec{q}$ is a column vector of the size T

$$\mathbf{s} = \begin{bmatrix} s_1 & 0 & \dots & 0 \\ 0 & s_2 & \dots & \vdots \\ 0 & \dots & \ddots & \vdots \\ 0 & \dots & \dots & s_T \end{bmatrix}, \mathbf{k} = \begin{bmatrix} k_1 & 0 & \dots & 0 \\ 0 & k_2 & \dots & \vdots \\ 0 & \dots & \ddots & \vdots \\ 0 & \dots & \dots & k_T \end{bmatrix}, \vec{q} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_t \end{bmatrix}, \tag{41}$$

This algorithm takes much more time to compute, so it has not been implemented in experiments or simulations. Because of the UAV's hardware computational limitations, the loop must run as fast as possible. The standard solution uses only the initial condition for knowing position and computes once the constants from equation 34. This is very fast operation. If we want to get different constants for each future positions, first we need to compute the predicted trajectory from the previous prediction using equation 11. Because MPC is only required to compute action inputs, the predicted trajectory is not usually computed. Above that, the equation 34 would had to been computed for each time prediction, this means T times. T can be a large number, for example in the simulation sometimes as high as 500.

Until now, we have been describing the case with only one obstacle. As described in the section 2.4, the lines of constraining matrices $\mathbf{A}_c$ and $\mathbf{B}_c$ enforce independent conditions. If the computation would have been done independently for N obstacles using the same procedure marking computed matrices $\mathbf{A}_{c,i}$ and $\mathbf{B}_{c,i}$, where $i \in 1, 2, ..., N$ the final constraining matrices would take form of

$$\mathbf{A}_c = \begin{bmatrix} \mathbf{A}_{c,1} \\ \mathbf{A}_{c,2} \\ \dots \\ \mathbf{A}_{c,N} \end{bmatrix}, \mathbf{B}_c = \begin{bmatrix} \mathbf{B}_{c,1} \\ \mathbf{B}_{c,2} \\ \dots \\ \mathbf{B}_{c,N} \end{bmatrix}, \tag{42}$$

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

5 solving constrained LCQP

The MPC has 2 different parts. The first one is creating the objective function J($\underline{\mathbf{u}}$) by matrices $\hat{\mathbf{H}}$ and $\hat{\mathbf{c}}$ 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 equation 19. We have already talked about solving this problem unconstrained using inverse matrix. This problem can't be solved analytically and there are only iterative methods. This means, that the solution is usually optimal, but close to optimal. One more property of the solver is desirable. Instead of finding 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 means 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 wanted to have some kind of 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 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. First of all, the thrust would have to be enormous, but mainly, the position is a second integration of the input and each integration takes one time step. This means, that the input action inflates the position first in two time steps. Even a small amount of noise can result in a task formulation, that has no solution. This is of course not applicable. Making the final solution *vec* 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 it's 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 $\mathrm{J}(\underline{\mathbf{u}})$, or lies on the constraining border.

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 it's 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 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 the minimum lies on the constraining border. This is forced with the constraints defined as $g_i(\underline{\mathbf{u}}) = 0, i \in \{1, 2, ..., N\}$ for N constraint.This method requires, that the objective function and the constraints are differentiable. It uses a function called Lagrangian $\mathcal{L}$

$$\mathcal{L}(\underline{\mathbf{u}}, \lambda_1, \lambda_2, ..., \lambda_N) = \mathrm{J}(\underline{\mathbf{u}}) - \sum_{i=1}^{N} \lambda_i g_i(\underline{\mathbf{u}}) \tag{43}$$

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

$$\frac{\partial \mathrm{J}(\underline{\mathbf{u}})}{\partial u_j} = \sum_{i=1}^{N} \frac{\partial \lambda_i g_i(\underline{\mathbf{u}})}{\partial u_j} \quad and \quad \frac{\partial \mathrm{J}(\underline{\mathbf{u}})}{\partial \lambda_i} = \sum_{i=1}^{N} \frac{\partial \lambda_i g_i(\underline{\mathbf{u}})}{\partial \lambda_i} \tag{44}$$
$$i \in \{1, 2, ..., N\}, \quad j \in \{1, 2, ..., 2T\}$$

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, as shows the figure —????—. 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 exists 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 2.4.1. 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

$$min \;\; \mathrm{J}(\underline{\mathbf{u}}) + \sigma \sum_{i=0}^{N} g(c_i(\underline{\mathbf{u}})) \tag{45}$$

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't be used either.

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. Than a gradient descent algorithm is used to find the local minimum of the function $f(\underline{\mathbf{u}}) = \mathrm{J}(\underline{\mathbf{u}}) + g(\underline{\mathbf{u}})$. 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. Than 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

$$labeleq : sum_conv_1 f_3(tx_1+(1-t)x_2) \leq tf_3(x_1)+(1-t)f_3(x_2), \quad t \in < 0, 1 > \quad (46)$$

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

$$\begin{aligned} f_1(tx_1 + (1-t)x_2) &\leq tf_1(x_1) + (1-t)f_1(x_2) \\ f_2(tx_1 + (1-t)x_2) &\leq tf_2(x_1) + (1-t)f_2(x_2) \end{aligned} \quad (47)$$

The inequality 46 is actually a sum of the two lines of the inequalities 47. Adding these 2 lines preserves the inequality. We can say, that the inequality 46 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

$$d = abs\left(\frac{\omega_i \cdot \underline{\mathbf{u}} + b_i}{\sqrt{\vec{\omega_i}^T \cdot \vec{\omega_i}}}\right) \quad (48)$$

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

$$g(\underline{\mathbf{u}}) = k_g \sum_{i=1}^{M} \frac{1}{d_i} \quad (49)$$

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.

The figure 5 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 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 – ??? kontrola—.

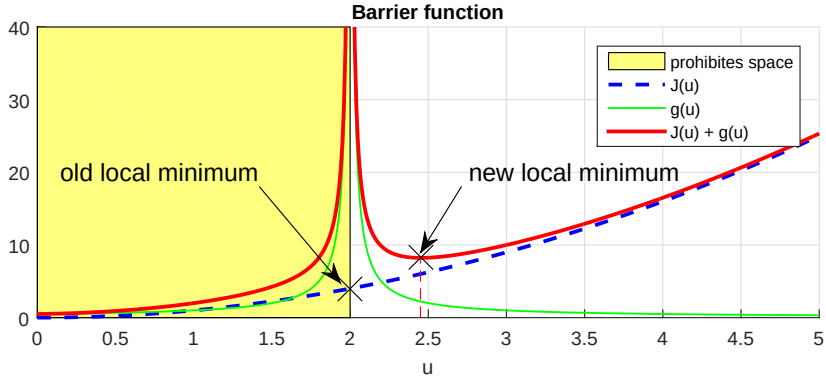


Figure 5: Barrier function

5.1 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 $\mathrm{J}(\underline{\mathbf{u}})$, the gradient descent optimizes the function $f(\underline{\mathbf{u}}) = \mathrm{J}(\underline{\mathbf{u}}) + g(\underline{\mathbf{u}})$ where $g(\underline{\mathbf{u}})$ is the barrier function from the equation 49. 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

$$\underline{\mathbf{u}}_{[t+1]} = \underline{\mathbf{u}}_t - k_d \nabla f(\underline{\mathbf{u}}) \tag{50}$$

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.

$$\begin{aligned} \nabla f(\underline{\mathbf{u}}) &= \nabla \mathrm{J}(\underline{\mathbf{u}}) + \nabla g(\underline{\mathbf{u}}) \\ \nabla \mathrm{J}(\underline{\mathbf{u}}) &= \hat{\mathbf{H}}\underline{\mathbf{u}} + \hat{\mathbf{c}} \\ \nabla g(\underline{\mathbf{u}}) &= k_g \sum_{i=1}^{N} \frac{\vec{\omega}_i.^2 sig(\vec{\omega}_i \cdot \underline{\mathbf{u}} + b_i)}{|\vec{\omega}_i| \cdot (\vec{\omega}_i \cdot \underline{\mathbf{u}} + b_i)} \end{aligned} \tag{51}$$

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 $\vec{\omega}_i.^2$ represents a line vector as the diagonal of the $\vec{\omega}_i^T \cdot \vec{\omega}_i$.

5.1.1 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 than 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 $\underline{\mathbf{u}}_0$. It works as follows:

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 $u_{[t+1]}$ 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 very well. This algorithm works simply, that if an

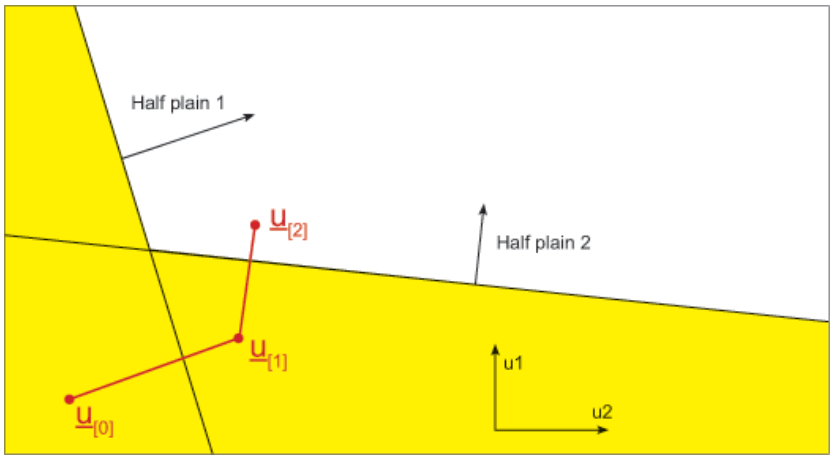


Figure 6: feasibility algorithm.

The figure 6 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.

When the algorithm for following trajectory starts, the UAV has no velocity, so the gradient descent algorithm can use the initial search point $\underline{\mathbf{u}}_{[0]}$ 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 $\underline{\mathbf{u}}_{[0]}$, 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 $f(\underline{\mathbf{u}})$ 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 $\underline{\mathbf{u}}$ 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 $\underline{\mathbf{u}}$ to a new location, that is still very close to the previous $\underline{\mathbf{u}}$, so very close to the local minimum. One more simple improvement needs to be done. There can be a case, 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, than the $R_{UAV} + r_obs$. With this initial condition there is usually no solution. To prevent the UAV to freeze, a counter needs to be added to terminate the feasibility algorithm after running too long.

5.1.2 Implementation of the gradient descend

The gradient descend is implemented by the equation 50. The only difference is, that every $\underline{\mathbf{u}}_{[t]}$ is checked for feasibility by applying the feasibility algorithm. This is for the case, that the gradient step is too big and after shifting the search point, it could end up outside the feasible domain. A simulation has been run for no barrier function using only the $\nabla \mathrm{J}(\underline{\mathbf{u}})$ and the feasibility algorithm and still getting 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 small number of total iterations can be helpful by providing a kind of low pass filter in situations with high sensory noise. Experiments have shown, that 100 is a good compromise between the accuracy and speed.

6 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. We have to keep this in mind all the time. 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 1MB 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.

6.1 Move blocking implementation

The computation complexity can be greatly reduced by the discretization of the prediction is very soft, meaning, that the Δt is very small. There is not a reason to predict position 70 times a second. The input actions tend to take form of a nice, continuous function. One solution would be to change the system's Δt, but there is a better solution. Move blocking algorithm allows us to choose freely exactly which time steps will be used. The algorithm will generate constant input action between those predicted as shown in the figure 9 or like zero-order hold model. This uses a matrix $\mathbf{U}$ to set, which time steps will be used. This allows only small number of variables to cover long prediction horizon. The algorithm uses a matrix $2T$ by $2T_n$.

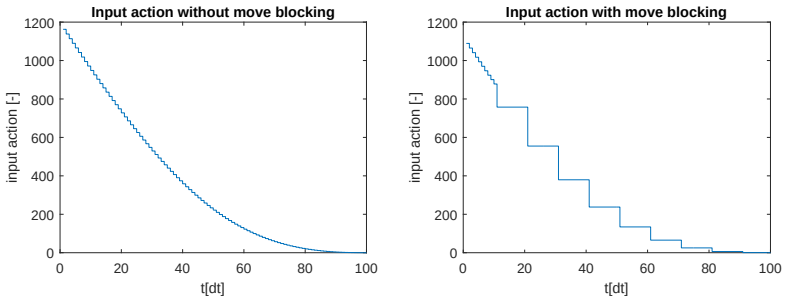


Figure 7: Move blocking algorithm in one axis

$$\underline{\mathbf{u}} = \underbrace{\begin{bmatrix} \begin{bmatrix}1\\\vdots\\1\end{bmatrix} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \begin{bmatrix}1\\\vdots\\1\end{bmatrix} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \begin{bmatrix}1\\\vdots\\1\end{bmatrix} \end{bmatrix}}_{\mathbf{U}} \underline{\mathbf{u}}_r, \quad \underline{\mathbf{u}}_r = \begin{bmatrix} \mathbf{u}_{[t_1]} \\ \mathbf{u}_{[t_2]} \\ \vdots \\ \mathbf{u}_{[T_r]} \end{bmatrix}, \tag{52}$$

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

In the figure 9 have been used prediction times $1, 2, ..., 9, 10, 20, 30, ..., 100$. The number of the prediction times is T_r. Using this particular vector, the T_r is 19 instead of T as 100. The operation of system matrix multiplication is now $(T/T_r)^2 = 27.7$ times faster and creates the same memory savings with the matrices. This is a huge 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 created accordingly. That is, why the main diagonal has to have it's members individually multiplied by the corresponding constant, that is the number of the representing time steps.

7 Simulations and Hardware

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. Also to minimize the probability of an expensive crash happening, the algorithm should be tested as much as possible.

During developing the MPC controller, all steps were being tested by Matlab simulations. This program is very good for these purposes, because it is optimized for matrix multiplication, which is the core of computing the right input actions. There have been written two matlab separate modules: the Environment module and the UAV module.

7.1 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.

7.2 The UAV module

The second module simulates the 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.

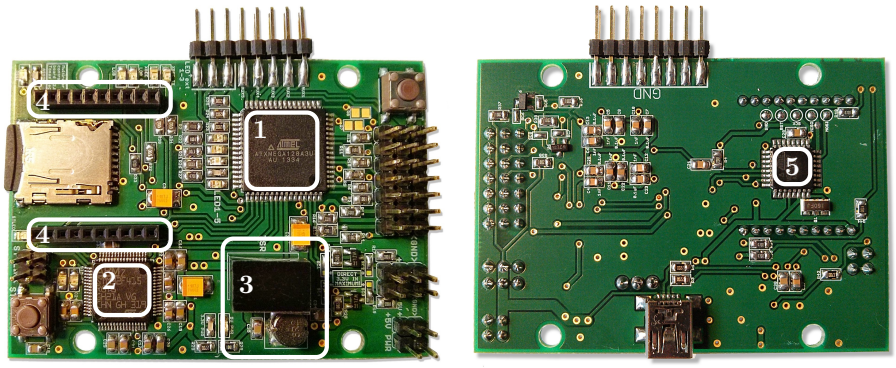
and the LCQP solver. This solver has been written by the method described in this thesis. It's inputs are the gradient step size, the total number of iterations and the problem defining matrices, which are identical to the function quadprog from the optimization toolbox. It returns slightly different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

— obrazek —

7.3 UAV custom board

The UAV custom board [**?**] as shown in the figure **??**, 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, a simple messages can be exchanged between the computer and



(a) board's top (b) board's bottom

Figure 8: 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 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. 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 2MB 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 is 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 less than a third of RAM —-xTaskCreate(mpcTask, (char*) "mpcTask", 4092, NULL, 2, NULL);???—-. Because of very limited computational power, the MPC task has to be programmed very efficiently.

7.4 MPC 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, that 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 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.

7.5 Hardware in the loop

This is a method of testing embedded hardware. The board is connected to a computer. The inputs are provided by the computer and outputs are also sent to the computer. 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.

The second experiment was actually connecting the board to the UAV, without the ability of controlling the motors. It received inputs from real sensors, such as velocity and obstacle positions. The board was connected to matlab using wireless Xbee. The 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.

After testing in various situations and behaving correctly, it was time to do an experiment with real flying UAV.

7.6 Experiments

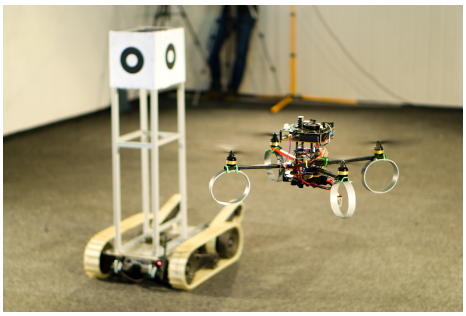


Figure 9: Photo of the experiment