Ecole Polytechnique Fédérale de Lausanne

Model Predictive Control Project

Micro-engineering - Master of Robotics

Control of a Quadricopter

Use of linear and non-linear sub-system control

Authors:
De Lajarte Albéric
Martin Yves
Brosset Maxime

Professor:
Jones Colin

July 13, 2021

Contents

1 System Dynamics

2 Linearization and Diagonalization

2.1 Why does the $A \cdot x + B \cdot T^{-1} \cdot v$ model breaks the system into four independent/non-interacting systems ? For which other steady-state conditions would this occur ? . .

3 Design MPC Controllers for Each Sub-System

3.1 Design MPC regulators . . . .

3.2 Design MPC tracking Controllers . . . .

4 Simulation with Nonlinear Quadcopter

4.1 Simulation with Nonlinear Quadcopter . . . .

5 Offset-Free Tracking

5.1 Design an offset-free tracking controller . . . .

6 Nonlinear MPC (Bonus)

1 System Dynamics

No deliverable for this part

2 Linearization and Diagonalization

2.1 Why does the $A \cdot x + B \cdot T^{-1} \cdot v$ model breaks the system into four independent/non-interacting systems ? For which other steady-state conditions would this occur ?

Let's choose steady state conditions.

(1) : As $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) = \mathbf{0}$, we can deduce that $\dot{\theta} = \mathbf{0}$ and $\dot{\mathbf{p}} = \mathbf{0}$.

(2) : As $\dot{\mathbf{p}} = \mathbf{0}$, then $\ddot{\mathbf{p}} = \mathbf{0}$. Considering the equation $\ddot{\mathbf{p}} = -m \cdot g \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + u_{tot} \cdot R \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

We deduce that the form of the rotation matrix R has to be $\begin{pmatrix} a & b & 0 \\ d & e & 0 \\ g & h & i \end{pmatrix}$. Then, roll and pitch, respectively θ_x and θ_y, are equal to zero. Then, $\mathbf{x} = \begin{pmatrix} \dot{\theta}_x \\ \dot{\theta}_y \\ \dot{\theta}_z \\ \theta_x \\ \theta_y \\ \theta_z \\ \dot{p_x} \\ \dot{p_y} \\ \dot{p_z} \\ p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ ? \\ 0 \\ 0 \\ 0 \\ ? \\ ? \\ ? \end{pmatrix}$

Those mathematical results seems connected to the reality : the θ_z angle has no influence on the steady state. This is also true for the position of the quadricopter.

Let's prove separately that $A \cdot x$ (3) and $B \cdot T^{-1} \cdot v$ (4) divide the system into 4 independent sub-systems.

(3) : A is the Jacobian Matrix of **F(x,u)** by the **x** state vector. It is the Newtonian mechanics part of the problem. A is a 12x12 matrix. $A = \begin{pmatrix} \partial\dot{x_1}(\mathbf{x,u})\overline{\partial x_1} & \dots & \partial\dot{x_1}(\mathbf{x,u})\overline{\partial x_i} & \dots & \partial\dot{x_1}(\mathbf{x,u})\overline{\partial x_{12}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \partial\dot{x_j}(\mathbf{x,u})\overline{\partial x_1} & \dots & \partial\dot{x_j}(\mathbf{x,u})\overline{\partial x_i} & \dots & \partial\dot{x_j}(\mathbf{x,u})\overline{\partial x_{12}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \partial\dot{x_{12}}(\mathbf{x,u})\overline{\partial x_1} & \dots & \partial\dot{x_{12}}(\mathbf{x,u})\overline{\partial x_i} & \dots & \partial\dot{x_{12}}(\mathbf{x,u})\overline{\partial x_{12}} \end{pmatrix}$

With the previous steady-state conditions, A has one non-zero component per line, which means the system is divided into 4 independent sub-systems. Indeed, $A \cdot \mathbf{x} = \begin{pmatrix} \sum_{i=1}^{12} A_{(1,i)} \cdot x_i \\ \sum_{i=1}^{12} A_{(2,i)} \cdot x_i \\ \vdots \\ \sum_{i=1}^{12} A_{(12,i)} \cdot x_i \end{pmatrix}$, then for each component of $A \cdot \mathbf{x}$, only one element of $\mathbf{x}$ is a factor.

(4) : B is the Jacobian Matrix of **F(x,u)** by the **u** input vector.

1. B is a 12x4 matrix. For the right steady-state conditions, B is written is this form :

$$B = \begin{pmatrix} 0 & a & 0 & a \\ b & 0 & b & 0 \\ c & -c & c & -c \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ d & d & d & d \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

   By Gauss method, one can see that the columns are linearly independent, so $dim(img(B)) = 4$.

2. $T \cdot u = v$ and T is reversible, then $u = T^{-1} \cdot v$ and $dim(img(B \cdot T^{-1})) = 4$. T^{-1} is a algebraic base to write the inputs in a sub-systems perspective.

3. With the previous steady-state conditions, B has maximum one non-zero component per line.

4. **v** divides the **u** vector into 4 subsystems, as $\mathbf{v} = \begin{pmatrix} F \\ M_\alpha \\ M_\beta \\ M_\gamma \end{pmatrix}$. If a vector **z** is written such as **z** =

$B \cdot T^{-1} \cdot \mathbf{v}$, where $B \cdot T^{-1}$ has one or zero element per line, then $\mathbf{z} = \begin{pmatrix} \sum_{i=1}^{4} (B \cdot T^{-1})_{(1,i)} \cdot v_i \\ \sum_{i=1}^{4} (B \cdot T^{-1})_{(2,i)} \cdot v_i \\ \vdots \\ \sum_{i=1}^{4} (B \cdot T^{-1})_{(12,i)} \cdot v_i \end{pmatrix}$, then for each component of $\mathbf{z}$, only one element of $\mathbf{v}$ is a factor. Then the four subsystems are non-interacting and independent.

3 Design MPC Controllers for Each Sub-System

3.1 Design MPC regulators

To design our controllers, we have first computed the maximum controlled invariant set. The constraints are represented as polytopes:

```
M = [1; -1]; m = [0.3; 0.3];
F = [0 1 0 0; 0 -1 0 0]; f = [0.035; 0.035];
% Compute maximal invariant set
Xf = polytope([F;M*K],[f;m]);
```

and to find the invariant set, we compute the pre-set until we cannot see any change:

```
Acl =  mpc.A + mpc.B*K;
while 1
   prevXf = Xf;
   [T,t] = double(Xf);
   preXf = polytope(T*Acl,t);
   Xf = intersect(Xf, preXf);
   if isequal(prevXf, Xf)
       break
   end
end
[Ff,ff] = double(Xf);
```

As we compute the **controlled** invariant set, we need to define a LQR controller:

```
[K, Qf, ~] = dlqr(mpc.A, mpc.B, Q, R);
K = -K;
```

The choice of the Q and R parameters are thus really important as they will define the performance of our controller. Our strategy will be to start with Q as the identity matrix and R = 1. We will then change either the diagonal values of Q or the value of R, to find rough values that gives us a settling time of 8 seconds for a deviation of 2 meters from the origin. Then we will play with the individual values of Q to further optimize the settling time and size of the invariant set. As we see in figure 1, the terminal set is reduced when we increase the diagonal values of Q.

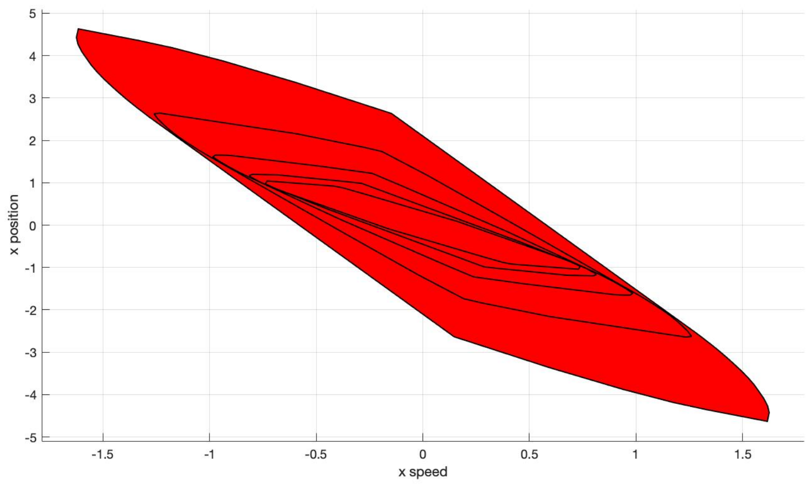

Figure 1: Projection of the mpcX controller's maximum controlled invariant set. The diagonal values of Q ranges from 10^{-3}(biggest set) to 10^{2}(smallest set)

However we also observed that the settling time decreases, so we cannot simply use the minimum values of Q. For R, we observe the opposite, that is when we increase R, the invariant set and the settling time increase.

In the end, the values chosen for Q and R are: $Q = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 6.3 & 0 \\ 0 & 0 & 0 & 8.5 \end{pmatrix}$ and R = 3.1 for the X and Y controllers.
For the Z and Yaw controllers, $Q = \begin{pmatrix} 6.3 & 0 \\ 0 & 8.5 \end{pmatrix}$ and R = 3.1.

Now that we have our maximum controlled invariant set, we can define the constraints and objectives for the optimizer. We do this in two steps: first by calculating over a finite horizon of N steps the states of the system:

```
con = (x(:,2) == mpc.A*x(:,1) + mpc.B*u(1)) + (M*u(1) <= m);
obj = u(1)'*R*u(1);

for i = 2:N-1
    con = [con, x(:,i+1) == mpc.A*x(:,i) + mpc.B*u(i)];   % System dynamics
    con = [con, F*x(:,i) <= f];                           % State constraint
    con = [con, M*u(i) <= m];                             % Input constraints
    obj = obj + x(:,i)'*Q*x(:,i) + u(i)'*R*u(i);          % Cost function
end
```

Then we simulate the cost and objectives over an infinite horizon using the invariant set calculated earlier:

```
con = [con, Ff*x(:,N) <= ff]; % Terminal constraint
```

```
obj = obj + x(:,N)'*Qf*x(:,N); % Terminal weight
```

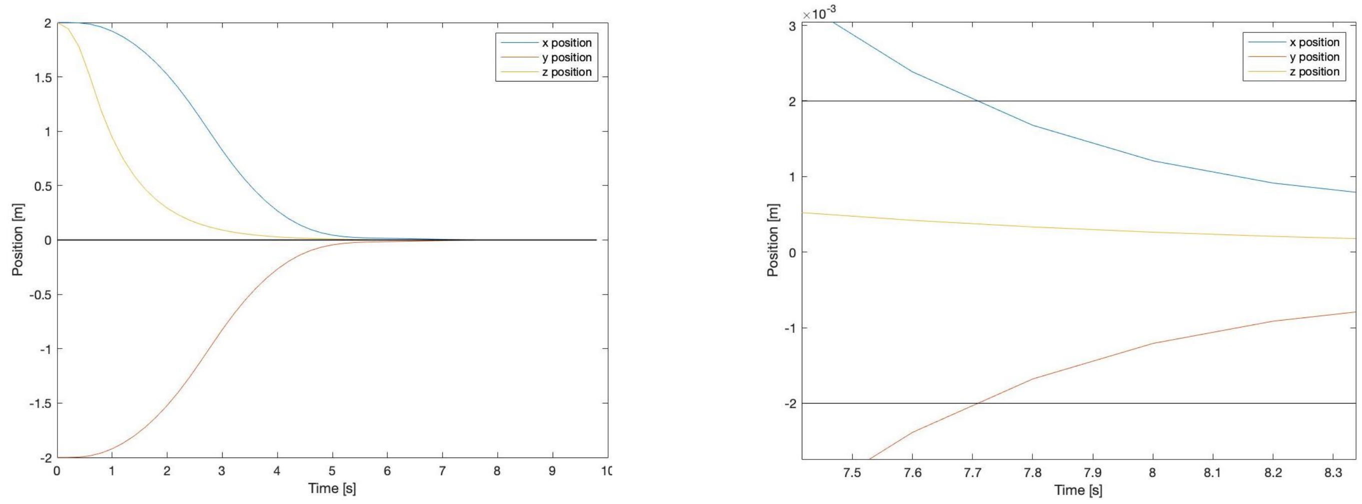

Figure 2: X, Y and Z positions of the drone starting at ±2 meters from the origin. The right image shows a zoomed version of the left image where we can see the drone converging below ±0.1% of the error in 7.7 seconds

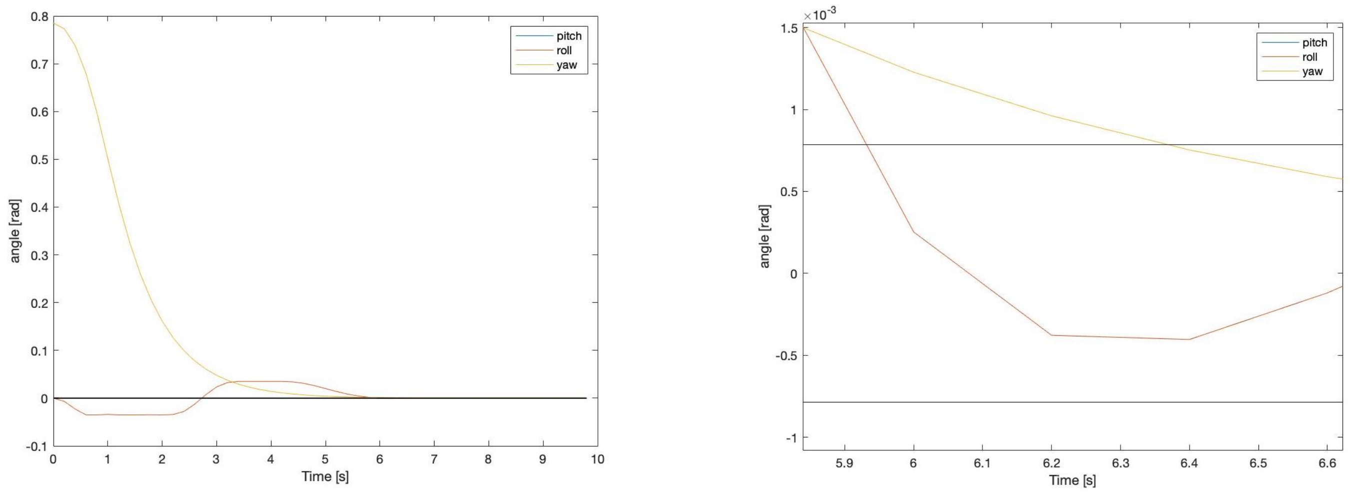

Figure 3: Pitch, yaw and roll of the drone starting at the origin for pitch and roll and 0.785 radian for yaw. The right image shows a zoomed version of the left image where we can see the drone converging below ±0.1% of the error in around 6 seconds

Obviously the choice of the parameter N is really important here. As we see in figure 4, the invariant set doesn't allow the drone to converge when starting at 2 meters with zero velocity. This is why we need the MPC controller to bring the initial point inside the invariant set. By doing some test, we saw that the minimal value of 14 steps was just enough for the controller to work. Thus we used N = 14.

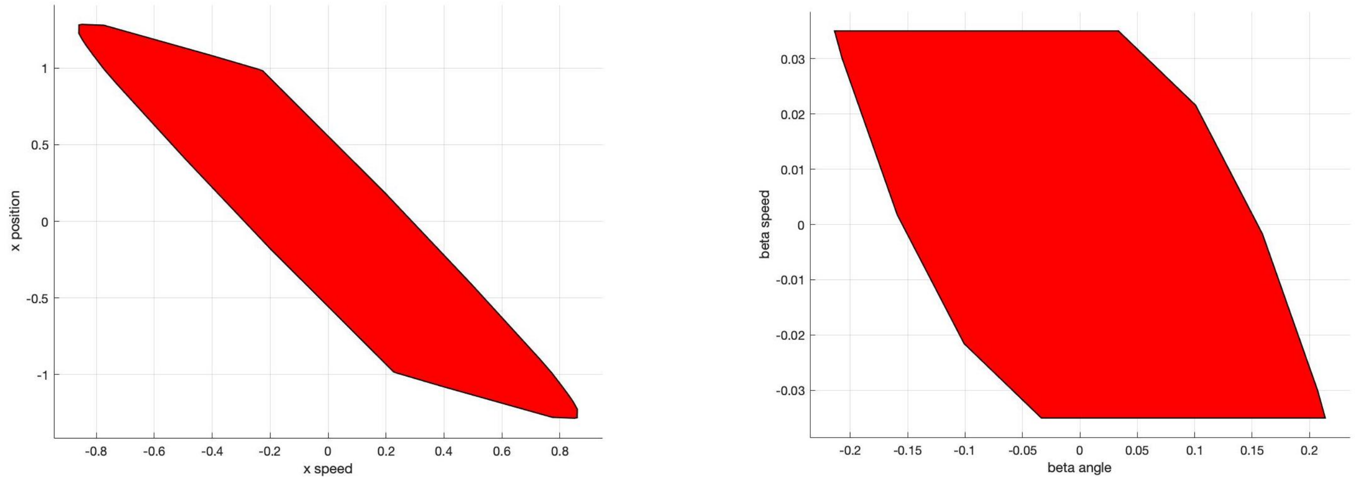

Figure 4: Projections of the invariant set. The left image shows the speed and position limits of the x axis, and the right image the angle and angular velocity limit of the β angle

3.2 Design MPC tracking Controllers

To allow our controller to track a reference, we need two steps.
First we convert the X, Y, Z and Yaw reference into a steady state target (x_s, u_s). Note that as it was demonstrated in part 2.1, there is no other variables that can be tracked, as they have to be equal to zero in order to respect the steady state conditions.

```
% WRITE THE CONSTRAINTS AND OBJECTIVE HERE
Q = 10;
M = [1; -1]; m = [0.3; 0.3];
F = [0 1 0 0; 0 -1 0 0]; f = [0.035; 0.035];

con = [M*us <= m            ,...
       F*xs <= f            ,...
       xs == mpc.A*xs + mpc.B*us ];

obj  = (mpc.C*xs - ref)'*Q*(mpc.C*xs - ref);
% Compute the steady-state target
target_opt = optimizer(con, obj, sdpsettings('solver', 'gurobi'), ref, {xs, us});
```

To compute the steady state target, the new constraint is $x_s = A * x_s + B * u_s$ which is the definition of the steady state. For the objective function, we try to minimize the distance to the reference $(C * x_s - ref)' * Q * (C * x_s - ref)$, with Q a scalar used as parameter. After some test, it seemed that Q had very little effect on the overall performance.

The second part is to modify our controller to integrate the steady state target. For this, we will use the Delta formulation, defined as:

$$\begin{Bmatrix} \Delta x = x - x_s \\ \Delta u = u - u_s \end{Bmatrix}$$

The rest of the controller is however very similar to non-tracking regulator:

```
% WRITE THE CONSTRAINTS AND OBJECTIVE HERE
Q = 2*eye(4); Q(3,3) = 6.3; Q(4,4) = 8.5;
R = 3.1;

M = [1; -1]; m = [0.3; 0.3];
F = [0 1 0 0; 0 -1 0 0]; f = [0.035; 0.035];

con = (x(:,2)-xs == mpc.A*(x(:,1)-xs) + mpc.B*(u(1)-us)) + (M*(u(1)-us) <= m-M*us);
obj = ((x(:,1)-xs)'*Q*(x(:,1)-xs))+(u(:,1)-us)'*R*(u(:,1)-us);

for i = 2:N-1
    con = [con, (x(:,i+1)-xs) == mpc.A*(x(:,i)-xs) + mpc.B*(u(i)-us)]; % System dynamics
    con = [con, M*(u(i)-us) <= m-M*us];                             % Input constraints
    con = [con, F*(x(:,i)-xs) <= f-F*xs];                            % State constraint
    obj = obj + (x(:,i)-xs)'*Q*(x(:,i)-xs) + (u(i)-us)'*R*(u(i)-us);  % Cost function
end
```

As we see, there is no terminal set in this controller, which means that we don't have the guarantee that the controller will actually converge to our reference, but the advantage is that the controller runs now much faster, because there is no need to compute the invariant set. However it seems that not using the terminal set yields less good result, as the X and Y controller took 8.5 seconds to converge below 0.1 % of the error, the Yaw controller took 7.6 seconds and the Z controller 6.6 seconds. To guarantee a settling time of less than 8 seconds, the horizon size was thus increase from 14 steps to 20 steps for the X and Y controllers. We also kept the same values of Q and R, as we saw after some test that it still gave us the best performance.

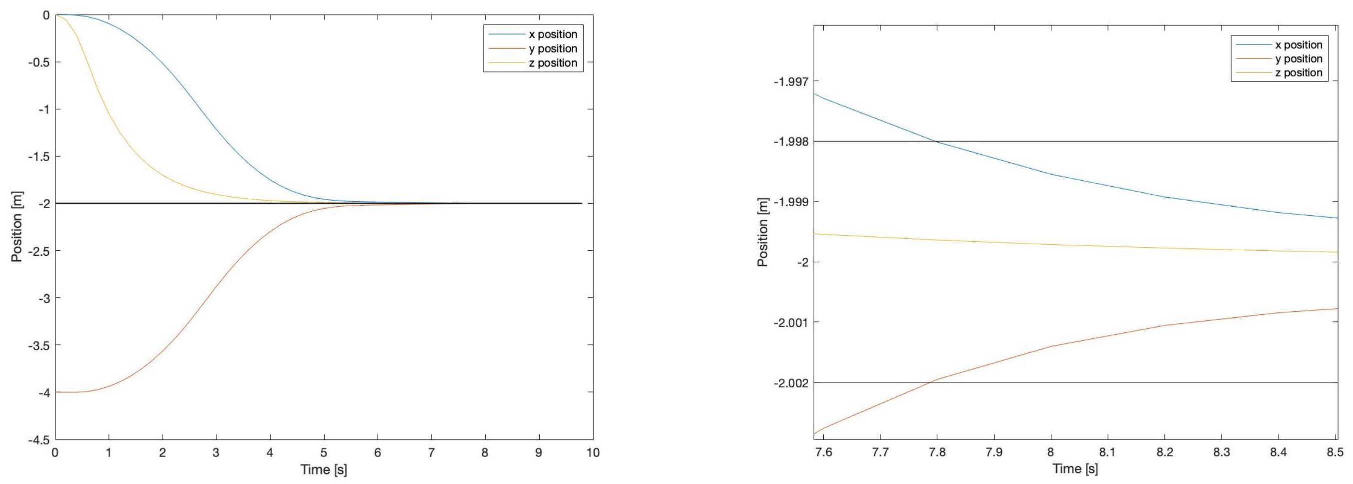

Figure 5: X, Y and Z positions of the drone starting at 0, -4 and 0 meters from the origin respectively. The right image shows a zoomed version of the left image where we can see the X, Y and Z position converging to the reference in less than 8 seconds

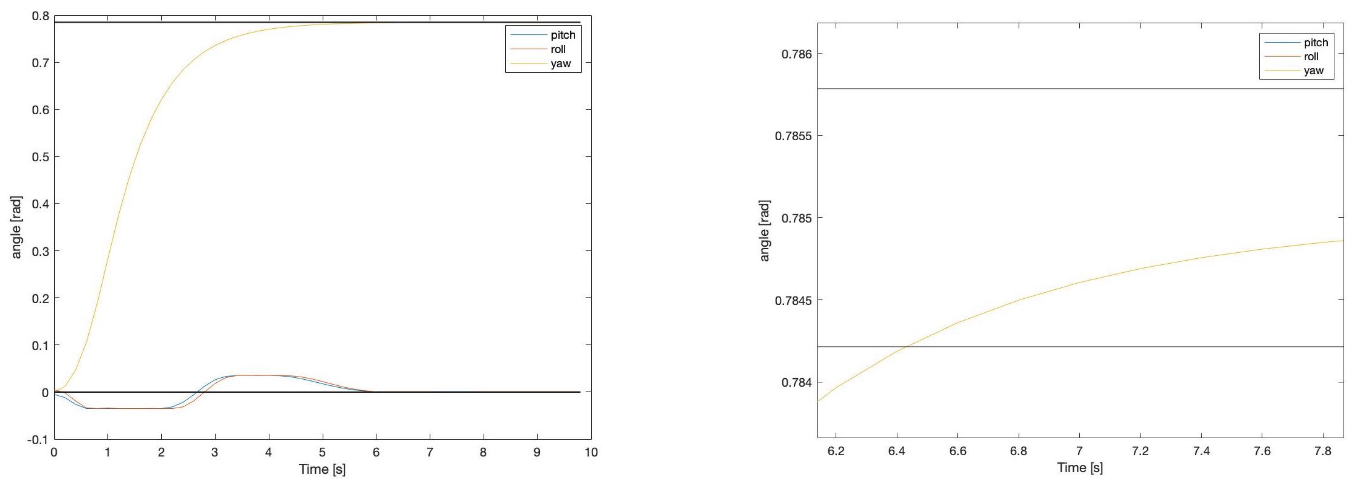

Figure 6: Pitch, yaw and roll of the drone starting at $\pm 5 * 10^{-3}$ radian for pitch and roll and 0 radian for yaw. The right image shows a zoomed version of the left image where we can see the yaw angle converging to the reference in around 6.4 seconds

4 Simulation with Nonlinear Quadcopter

4.1 Simulation with Nonlinear Quadcopter

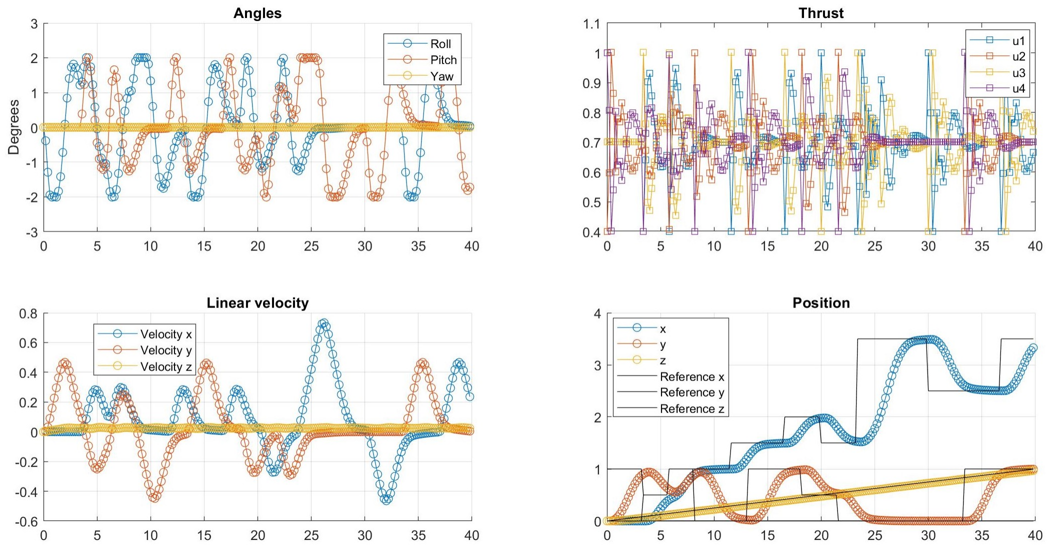

Figure 7: Detailled view of the the reference tracking in X, Y and Z

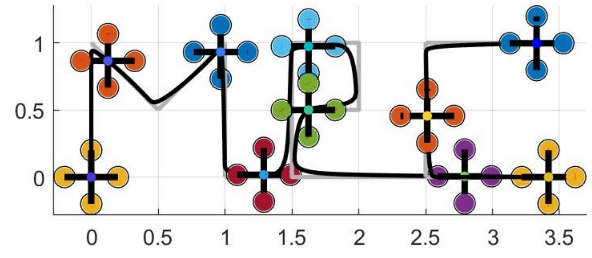

Figure 8: Reference tracking in X-Y plane

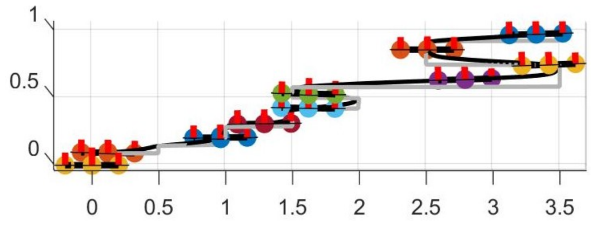

Figure 9: Reference tracking in Y-Z plane

5 Offset-Free Tracking

5.1 Design an offset-free tracking controller

To design our offset-free controller, we first had to define our new augmented model incorporating our disturbance model. This new augmented model is defined by:

```
nx    = size(mpc.A,1);
nu    = size(mpc.B,2);
ny    = size(mpc.C,1);
A_bar = [mpc.A           , mpc.B;
         zeros(1,nx),1          ];
B_bar = [mpc.B;zeros(1,nu)];
C_bar = [mpc.C,ones(ny,1)];
```

To design our state estimator, we needed to choose an estimator gain L such that the error dynamic is stable and converges to zero. To do so, we used the matlab **place** command allowing us to design L such that the eigenvalues of $\bar{A} - L * \bar{C}$ are $[0.5, 0.6, 0.7]$ making our estimator stable.

```
L = -place(A_bar',C_bar',[0.5,0.6,0.7])';
```

We choose $[0.5, 0.6, 0.7]$ for our eigenvalues as these values seemed to be a good trade-off between stability and performance. Also, $max(abs([0.5, 0.6, 0.7])) < 1$, which means the system is stable.

We also had to modify the **target-opt** function to take the disturbance in account when defining the constraints and objectives to compute the steady-states targets.

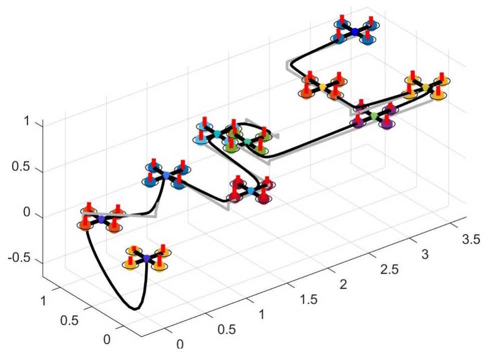

Figure 10: Offset free Reference tracking

6 Nonlinear MPC (Bonus)

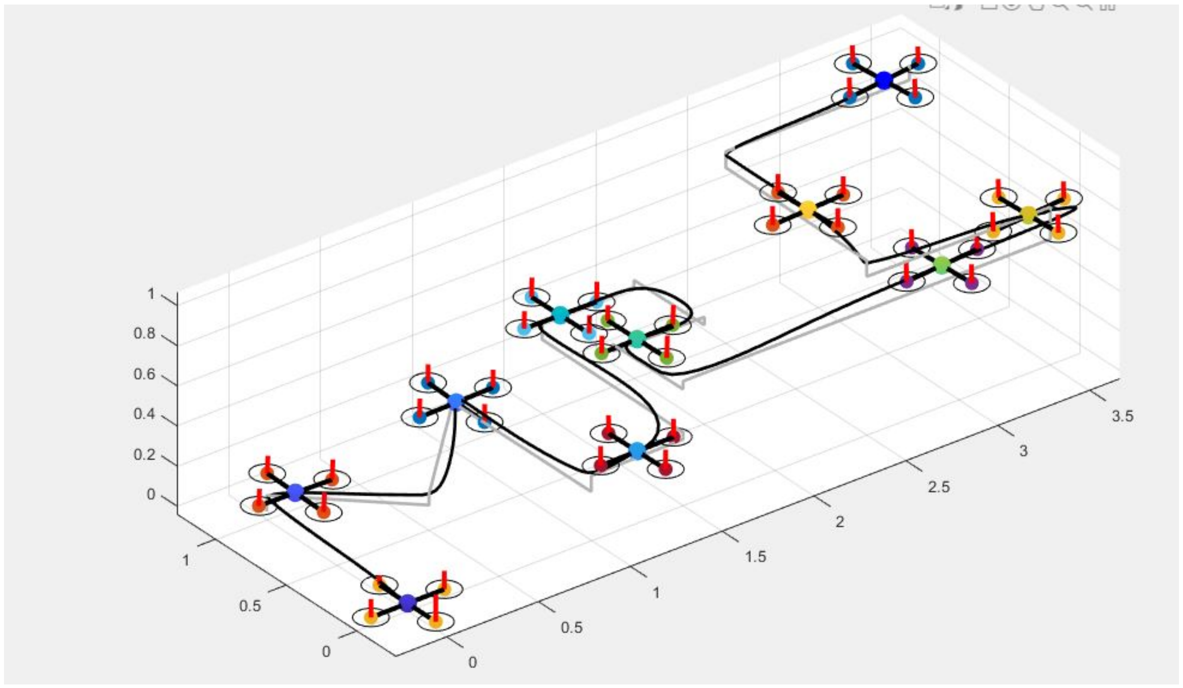

Figure 11: Nonlinear controller global view

The nonlinear MPC is implemented with the casadi library : the code optimizes a Lyapunov cost function while respecting the constraints. It is quite similar to the previous part, except that we use only one controller instead of four, as the controller optimises the whole $\mathbf{x}$ state-vector. Also, the $x^+ = f(x, u)$ function is nonlinear, as we use directly quad.f(). The cost function has the following value :

$obj = \sum_{n=1}^{N} [\sum_{i=1}^{12} weight_x(i) \cdot (x(i,n) - ref(i))^2 + \sum_{j=1}^{4} weight_u(j) \cdot (u(j,n) - us(j))^2]$

The $[x, y, z, yaw]$ reference vector is given by quad.MPC_ref(). The other components of the reference vector are fixed to zero. Then, the weight vector allow us to tune every state-vector component, especially the speed and position components. The tuning vector is the following :

$$weight_x = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 4 \\ 4 \\ 8 \\ 10 \\ 10 \\ 20 \end{Bmatrix} weight_u = \begin{Bmatrix} 10 \\ 10 \\ 10 \\ 10 \end{Bmatrix}$$

One can see that the speed has a cost, in order to lower the overshoot. Also, the weight is 40% of the cost of the corresponding position error, which is the best ratio we found by testing and testing again.

```
%Choose tuning parameters
w_vit_xy = 4;  w_vit_z = 8;
w_pos_xy = 10; w_pos_z = 20;
w_pos_yaw = 0; w_u = 10;

weight_x = [0 0 0 0 0 w_pos_yaw w_vit_xy w_vit_xy w_vit_z w_pos_xy w_pos_xy w_pos_z];

%compute the objective to minimize
obj = 0;
for i = 1:12
    obj = obj + weight_x(i) * (X(i,:)-vec_ref(i)) * (X(i,:)-vec_ref(i))';
end
obj = obj + w_u*sum(diag((U-repmat(quad.us,1,N))*(U-repmat(quad.us,1,N))'));

opti.minimize(obj);
```

The optimizer also follows the constraints : as for the previous part, the **X(i)** state vector are computed from 1 to the horizon N, fixed to 10. We use the 4^{th} order Runge-Kutta method applied to the non linear function quad.f() to discretise it. This predictive trajectory allows us to formulate the cost function. Eventually, the usual constraints are followed (0<u<1.5 and -2°< pitch & roll < 2°).

```
%compute the constraints
opti.subject_to(0 <= U <= 1.5);
opti.subject_to(X(:,1)==X0);   % use initial position
opti.subject_to(-0.035*ones(2,N+1) <= X(4:5,:) <= 0.035*ones(2,N+1));
```

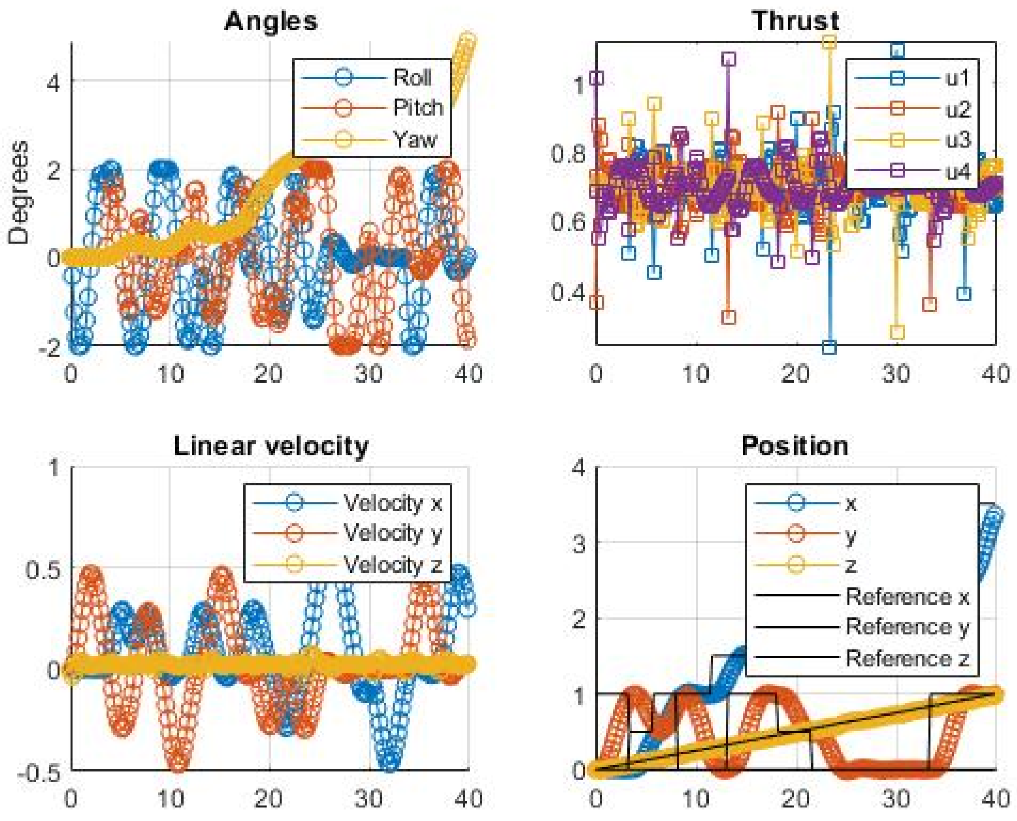

Figure 12: Detailled view of the reference tacking in X, Y and Z

```
h = quad.Ts;
quad_f = @(x,u) quad.f(x,u);
f_discrete = @(x,u) RK4(x,u,h,quad_f);

for k = 1:N
    opti.subject_to( X(:,k+1) == f_discrete(X(:,k),U(:,k)));
end
```

The nonlinear MPC works better than the previous one because the state vector is globally optimized instead of 4 times locally optimized. There is no "interface issue" between different controllers as there is only one controller now. Also, the predictive function is closer to the reality as it is not linearized. Eventually, it is easier to tune a weight vector than the Q, R and N parameters in a LQR controller.

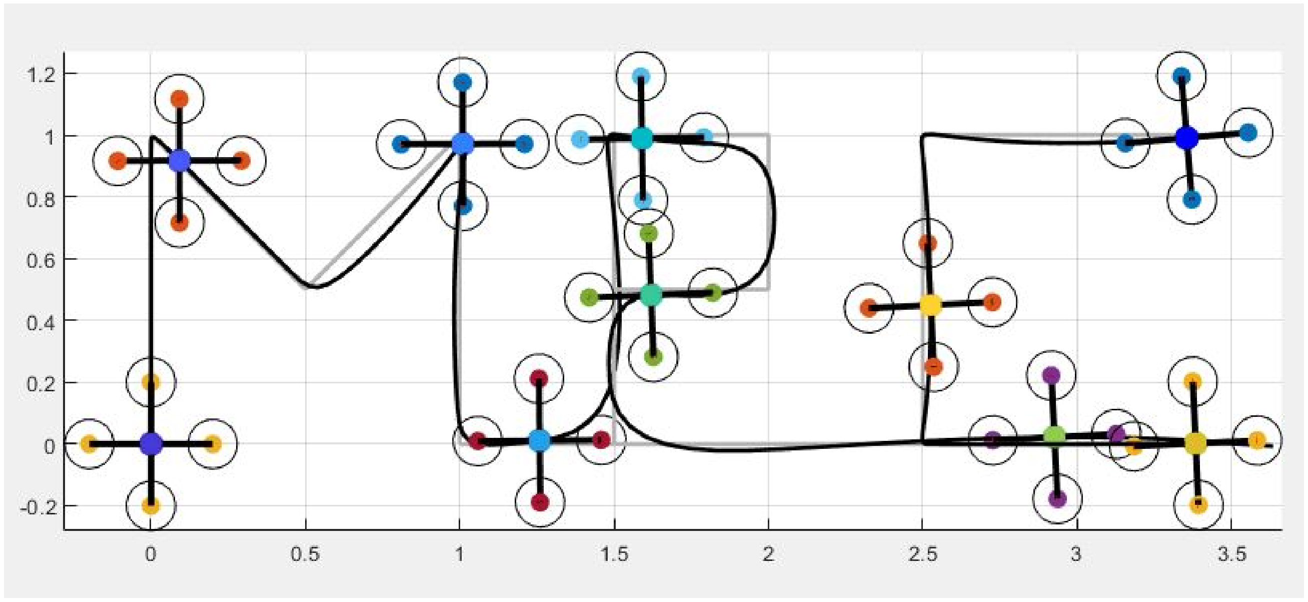

Figure 13: Reference tracking in X-Y plane

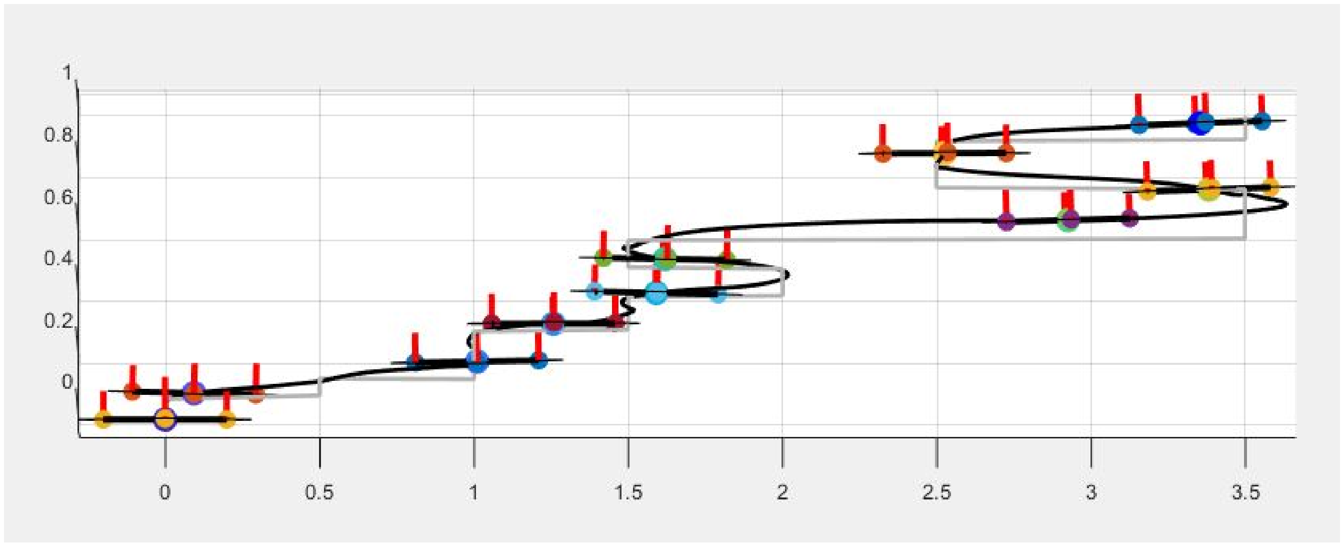

Figure 14: Reference tracking in Y-Z plane