Model Predictive Control : Project

In this project you will develop an MPC controller to fly a quadcopter.

The project is worth 40% **of your final grade and is due on January** 8^{th}**.**

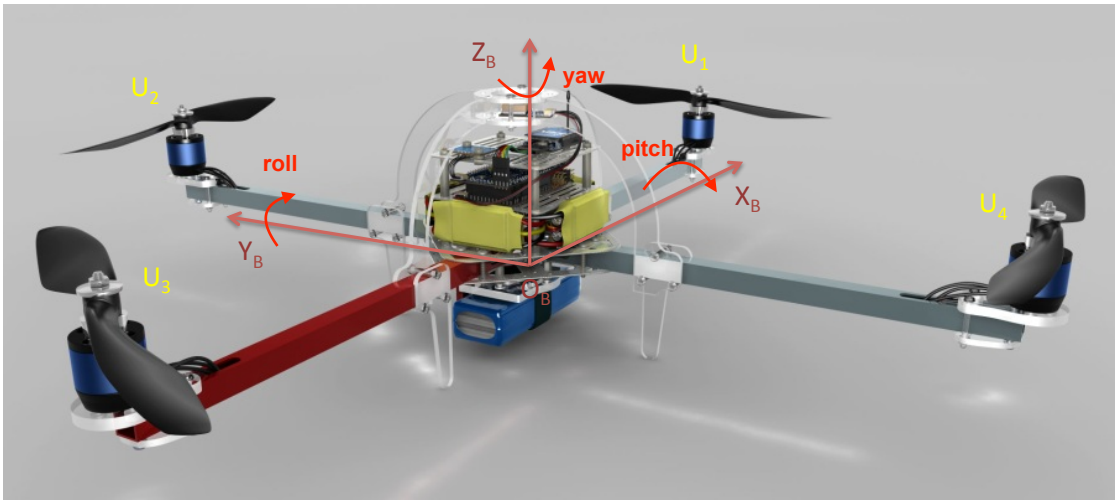
Before you start:

- Make sure you have installed YALMIP, MPT3, Gurobi and Casadi according to the course exercise setup instructions on Moodle
- Download the project file `Quad.m` from Moodle
- Run `quad = Quad();` If this executes correctly, then your setup should be ready to go

Instructions:

- Group sign up and report hand-in is via Moodle
- You can do the project in groups of one, two or three
- Include everyone's name and SCIPER on the title page of your project report
- When you have completed the project, hand in one report (pdf) per group and your matlab code (zip)
  - Your report should contain headings according to the **Deliverables** listed below in the project description
  - **You will be graded on the Deliverables**, and not on the Todos
  - The report should be written in English
  - Explain what you're doing and why for each deliverable, but don't be excessive. The entire report should be less than 20 pages
  - Each m-file should be named according to the convention Deliverable 2.1 = `Deliverable_2_1.m`
    - If you need multiple m-files for a given deliverable, combine them in a zip file and use the name `Deliverable_2_1.zip`

Part 1 | System Dynamics



In order to derive a nonlinear model consider two reference frames. The first one is the body frame (subscript B) with the origin O_B attached to the center of mass of the quadrotor as drawn in the picture. The second one is the world frame which remains fixed. We are going to derive a 12-state description of the robot with the state vector

$$\mathbf{x} = \begin{bmatrix} \dot{\theta} & \theta & \dot{p} & p \end{bmatrix}$$

where $p = \begin{bmatrix} x & y & z \end{bmatrix}$ is the position of the center of mass of the quadcopter and $\theta = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix} = \begin{bmatrix} \text{roll} & \text{pitch} & \text{yaw} \end{bmatrix}$ is the orientation. These angles represent, in order, the rotation around the x_B axis followed by the rotation around the y_B axis followed by rotation around the z_B axis.

The input of the model

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & u_3 & u_4 \end{bmatrix}^T$$

is the thrusts of the four rotors, which are proportional to the square of the rotor angular velocity and hence must be nonnegative. Consequently, each rotor produces a force F_i and moment M_i according to

$$F_i = k_F u_i, \quad M_i = k_M u_i.$$

The net body force F and body moments $(M_\alpha, M_\beta, M_\gamma)$ can then be expressed in terms of $\mathbf{u}$ as

$$\begin{bmatrix} F \\ M_\alpha \\ M_\beta \\ M_\gamma \end{bmatrix} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & k_F L & 0 & -k_F L \\ -k_F L & 0 & k_F L & 0 \\ k_M & -k_M & k_M & -k_M \end{bmatrix} \mathbf{u} \tag{1}$$

where L is the distance from the center of mass to the center of the rotor.

The acceleration of the center of mass is then given by

$$\ddot{O}_B = \begin{bmatrix} 0 \\ 0 \\ -m \cdot g \end{bmatrix} + F\mathbf{z}_B. \tag{2}$$

where g is the acceleration due to gravity, m is mass (`quad.mass`) and $\mathbf{z}_B$ is the unit vector in the z direction of the body coordinate frame expressed in the world coordinates (with similar definitions for $\mathbf{x}_B$ and $\mathbf{y}_B$).

The angular dynamics are given by

$$\dot{\omega} = \mathcal{I}^{-1}\left(-\omega \times \mathcal{I}\omega + \begin{bmatrix} M_\alpha \\ M_\beta \\ M_\gamma \end{bmatrix}\right), \tag{3}$$

where $\omega = \dot{\alpha}\mathbf{x}_B + \dot{\beta}\mathbf{y}_B + \dot{\gamma}\mathbf{z}_B$ is the angular velocity of body coordinate frame with respect to the world coordinate frame, $\mathcal{I}$ is the inertia matrix of the quadrotor and $\times$ is the cross product.

Now take a look at Equation 1, which is the key to the understanding of how a quadrotor works. From the first line we can see that the net body force F (in the direction $\mathbf{z}_\mathrm{B}$) is simply the sum of the four rotor thrusts (times a constant k_F). From the second line we see that the moment around the $\mathbf{x}_\mathrm{B}$ axis (which then gives rise to a change in the roll angle α) is given by the imbalance in the thrusts of rotors 2 and 4. Similarly the pitch angle β is controlled by the imbalance in the thrusts of rotors 1 and 3. Finally, from the fourth line we can see that the yaw angle γ is controlled by the imbalance in the net thrusts of the opposite rotor pairs (1,3) and (2,4). This is because the two pairs rotate in the opposite direction (and therefore generate no net moment around $\mathbf{z}_B$ when the net thrusts of the two pairs are equal; otherwise there is a moment imbalance and the quad rotates).

From Equation 2 we see that in order to fly the quadrotor around we need to control the body direction $\mathbf{z}_B$ (which is a function of α and β) and simultaneously apply the net body force F in that direction, all of this while counteracting gravity.

Finally, Equation 3 says that the body angles (α, β, γ) are controlled by the moments $(M_\alpha, M_\beta, M_\gamma)$, which are directly related to the rotor thrusts through Equation 1.

Putting together Equations 2 and 3, we can write the dynamic equations

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$$

Todo 1.1 | Study the function `f` in the Matlab `Quad` class to confirm that it implements the dynamics of the system as described above.

Todo 1.2 | Simulate the quadcopter with various step inputs to confirm that the dynamics respond as you expect.

To simulate the quadcopter for one second starting from the origin with a constant input, you can use:

```
quad = Quad();
Tf = 1.0; % Time to simulate for

x0 = zeros(12,1);                                % Initial state
u = [1;1;1;1];                                   % Input to apply
sim = ode45(@(t, x) quad.f(x, u), [0, Tf], x0);  % Solve the system ODE
quad.plot(sim, u);                               % Plot the result
```

Try to find inputs u that will keep the quad flat and level, and inputs that will cause it to roll, pitch and yaw. Note that the inputs u take values between 0 and 1.5.

Part 2 | Linearization and Diagonalization

In the first part of the project, we are going to control a linearized version of the quadcopter.

Todo 2.1 | Use the following code to generate a trimmed[1] and linearized version of the quadcopter

```
quad = Quad();

[xs,us] = quad.trim()        % Compute steady-state for which 0 = f(xs,us)
sys = quad.linearize(xs, us) % Linearize the nonlinear model
```

Go through the functions `trim` and `linearize` to see how they work.

Notice that we have named all the states in the linearized model - type `sys` and you will see the ordering of the states.

Consider the input matrix T given in Equation 1, shown again here:

$$\mathbf{v} = \begin{bmatrix} F \\ M_\alpha \\ M_\beta \\ M_\gamma \end{bmatrix} = \begin{bmatrix} k_F & k_F & k_F & k_F \\ 0 & k_F L & 0 & -k_F L \\ -k_F L & 0 & k_F L & 0 \\ k_M & -k_M & k_M & -k_M \end{bmatrix} \mathbf{u} = T\mathbf{u}$$

We now take our linearization and do a change of variables to define a new input $\mathbf{v}$

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} = A\mathbf{x} + BT^{-1}\mathbf{v}$$

Todo 2.2 | Apply this transformation to your linearized system

```
sys_transformed = sys * inv(quad.T)  % New system is  A * x + B * inv(T) * v
```

Study the resulting A, B, C and D matrices until you recognize that this transformation breaks the system into four independent / non-interacting systems.

Deliverable 2.1 | Explain why this occurs, and whether this would occur at other steady-state conditions.

Hint: Think about what properties x_s and u_s must have for $\dot{x}_s = f(x_s, u_s) = 0$

Todo 2.3 | Compute the four independent systems above using the following command

```
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us)
```

Hint: Note that the `decompose` command applies the transformation for you. The `sys` input here is the output of the `linearize` command.

Four models are produced:

`sys_y`	Roll moment input M_α to position y. The system has four states: y, $\dot{y}$, α, $\dot{\alpha}$.
`sys_x`	Pitch moment input M_β to position x. The system has four states: x, $\dot{x}$, β, $\dot{\beta}$.
`sys_z`	Total thrust input F to height z. The system has two states: z and $\dot{z}$.

[1] Trimming a system $\dot{x} = f(x, u)$ means to find a state and input pair $\bar{x}, \bar{u}$ such that $f(\bar{x}, \bar{u}) = 0$.

`sys_yaw` | Yaw moment input M_γ to yaw angle γ. The system has two states γ and $\dot{\gamma}$

Note that these are all **continuous time** models.

Part 3 | Design MPC Controllers for Each Sub-System

For each of the dimensions x, y, z and yaw, your goal is to design a recursively feasible, stabilizing MPC controller that can track step references.

Throughout this section, we consider a sampling period of $1/5$ **seconds.** The continuous time models produced in the previous section must be discretized using `discrete_system = c2d(sys, 1/5);`.

Constraints

Because our linearization is approximate, we must place constraints on the maximum angle that the quad can take so that our approximation is valid:

$$|\alpha| \leq 2^\circ = 0.035 rad$$
$$|\beta| \leq 2^\circ = 0.035 rad$$

Note that the model is in radians.

The quad-copter has constraints on all of the inputs $0 \leq \mathbf{u} \leq 1.5$. However, when we decompose the system, these constraints would couple the four systems through their virtual inputs $\mathbf{u} = T^{-1}\mathbf{v} + u_s$:

$$0 \leq T^{-1}\mathbf{v} + u_s \leq 1.5$$

To avoid this effect, we compute an inner approximation of this constraint which does not couple the systems

$$\{\mathbf{v} \,|\, \underline{v} \leq \mathbf{v} \leq \bar{v}\} \subset \{\mathbf{v} \,|\, 0 \leq T^{-1}\mathbf{v} + u_s \leq 1.5\}$$

The vectors $\underline{v}$ and $\bar{v}$ define a convenient box constraint and have been pre-computed. The value are

$$-0.3 \leq M_\alpha \leq 0.3$$
$$-0.3 \leq M_\beta \leq 0.3$$
$$-0.2 \leq F \leq 0.3$$
$$-0.2 \leq M_\gamma \leq 0.2$$

Design MPC Regulators

Todo 3.1 | Design four MPC controllers for x, y, z and yaw with the following properties:

- Recursive satisfaction of the input and angle constraints given in the previous section
- Stabilization of the system to the origin (i.e., all states equal to zero)
- Settling time of around eight seconds when starting stationary at two meters from the origin (for x, y and z) or stationary at 45 degrees for yaw

To help you design the controllers, we've created four files: • `MPC_Control_x.m` • `MPC_Control_y.m` • `MPC_Control_z.m` • `MPC_Control_yaw.m` Your job is to fill in the function `setup_controller` in each file.

You can then evaluate your control via

```
Ts       = 1/5;
quad     = Quad(Ts);
[xs, us] = quad.trim();
sys      = quad.linearize(xs, us);
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us);

% Design MPC controller
mpc_x   = MPC_Control_x(sys_x, Ts);

% Get control inputs with
ux   = mpc_x.get_u(x)
```

Deliverable 3.1 |

- Explanation of design procedure that ensures recursive constraint satisfaction
- Explanation of choice of tuning parameters. (e.g., Q, R, N, terminal components)
- Plot of terminal invariant set for each of the dimensions, and explanation of how they were designed and tuning parameters used
  Hint: If X_f is higher than two dimensions, you can plot its projections with

  ```
  Xf.projection(1:2).plot();
  Xf.projection(2:3).plot();
  Xf.projection(3:4).plot();
  ```

  Hint: An LQR controller makes a good terminal controller...
- Plot for each dimension starting stationary at two meters from the origin (for x, y and z) or stationary at 45 degrees for yaw
- m-code for the four controllers, and code to produce the plots in the previous step

Design MPC Tracking Controllers

Todo 3.2 | Extend your controllers so that they can track constant references while maintaining recursive feasibility

For x, y and z the reference is a position, and for yaw it's an angle in radians.

You may drop the requirement of invariance here (i.e., no terminal set is required).[2]

To implement your controllers, modify your four controllers from the previous section • `MPC_Control_x.m` • `MPC_Control_y.m` • `MPC_Control_z.m` • `MPC_Control_yaw.m` and fill in the function `setup_steady_state_target`

[2] It is possible to use a terminal set for tracking here by noticing that there are no constrains on the position of the quad and that all steady states are zero for all non-position states. This means that a shifted version of the terminal invariant set for regulation is still invariant. i.e., if $\mathcal{X}_f = \{x \mid Gx \leq g\}$ is invariant, then so is $\mathcal{X}_f + x_s = \{x \mid G(x - x_s) \leq g\} = \{x \mid Gx \leq g + Gx_s\}$.

in each one.

You can now get your control input via:

```
Ts       = 1/5;
quad     = Quad(Ts);
[xs, us] = quad.trim();
sys      = quad.linearize(xs, us);
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us);

% Design MPC controller
mpc_x   = MPC_Control_x(sys_x, Ts);

% Get control inputs with
ux   = mpc_x.get_u(x, x_position_reference)
```

Deliverable 3.2 |

- Explanation of your design procedure and choice of tuning parameters
- Plot for each dimension of the system starting at the origin and tracking a reference to -2 meters from the origin (for x, y and z) and to 45 degrees for yaw
- m-code for the four controllers, and code to produce the plots in the previous step

Part 4 | Simulation with Nonlinear Quadcopter

In this section, you will use your controllers to have the nonlinear quadcopter track a given path.

Todo 4.1 | Simulate the full nonlinear system with your four controllers.

You can simulate your system to track the given reference path with the command

```
Ts       = 1/5;
quad     = Quad(Ts);
[xs, us] = quad.trim();
sys      = quad.linearize(xs, us);
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us);

mpc_x   = MPC_Control_x(sys_x, Ts);
mpc_y   = MPC_Control_y(sys_y, Ts);
mpc_z   = MPC_Control_z(sys_z, Ts);
mpc_yaw = MPC_Control_yaw(sys_yaw, Ts);

sim = quad.sim(mpc_x, mpc_y, mpc_z, mpc_yaw);
quad.plot(sim);
```

Deliverable 4.1 | A plot of your controllers successfully tracking the path using `quad.plot(sim)`

Part 5 | Offset-Free Tracking

In this section, we assume that the mass of the quadcopter changes, and we want to extend your z-controller to compensate.

The dynamics of the system in the $z-$direction is now

$$x^+ = Ax + Bu + Bd$$

where d is a constant, unknown disturbance. Your goal is to update your controller to reject this disturbance and track setpoint references with no offset.

Todo 5.1 | For the z dimension only, design an offset-free tracking controller.

- Implement the `setup_estimator` function in the `MPC_Control_z.m` file
- Update the functions `setup_controller` and `setup_steady_state_target` in `MPC_Control_z.m` to provide offset-free tracking.

In this section, you will need to use an observer to estimate the offset and the state of the system and therefore, we can no longer ensure constraint satisfaction. So for this section, you can drop the terminal set.

You can simulate your system with an input bias with the code

```
BIAS = -0.1;

Ts       = 1/5;
quad     = Quad(Ts);
[xs, us] = quad.trim();
sys      = quad.linearize(xs, us);
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us);

mpc_x   = MPC_Control_x(sys_x, Ts);
mpc_y   = MPC_Control_y(sys_y, Ts);
mpc_z   = MPC_Control_z(sys_z, Ts);
mpc_yaw = MPC_Control_yaw(sys_yaw, Ts);

sim = quad.sim(mpc_x, mpc_y, mpc_z, mpc_yaw, BIAS);
quad.plot(sim);
```

Deliverable 5.1 |

- Explanation of your design procedure and choice of tuning parameters
- Plot showing that the z-controller achieves offset-free tracking
- m-code for your controllers, and code to produce the plots in the previous step

Part 6 | Nonlinear MPC (Bonus)

This part is optional. You can earn up to an additional 10% beyond the 40% grade of the project by completing this part of the project.

Todo 6.1 | Develop a nonlinear MPC controller for the quadcopter using CASADI. Your controller should take the full state of the quadcopter as input, and provide four propeller speeds (i.e., we do not decompose the quad into four sub-systems here).

Use the code below to compute your controller. Complete the "YOUR CODE HERE" block to setup your controller. The rest of the code is just a wrapper to solve the optimization problem efficiently.

```
function [ctrl, traj] = ctrl_NMPC(quad)

import casadi.*

opti = casadi.Opti(); % Optimization problem

N = *****; % MPC horizon [SET THIS VARIABLE]

% ---- decision variables ---------
X = opti.variable(12,N+1); % state trajectory variables
U = opti.variable(4, N); % control trajectory (throttle, brake)

X0 = opti.parameter(12,1);  % initial state
REF = opti.parameter(4,1); % reference position [x,y,z,yaw]

%%%%%%%%%%%%%%%%%%%%%%%%
%%%% YOUR CODE HERE %%%%
%%%%%%%%%%%%%%%%%%%%%%%%

ctrl = @(x,ref) eval_ctrl(x, ref, opti, X0, REF, X, U);
end


function u = eval_ctrl(x, ref, opti, X0, REF, X, U)
% ---- Set the initial state and reference ----
opti.set_value(X0, x);
opti.set_value(REF, ref);

% ---- Setup solver NLP    ------
ops = struct('ipopt', struct('print_level',0, 'tol', 1e-3), 'print_time', false);
opti.solver('ipopt', ops);

% ---- Solve the optimization problem ----
sol = opti.solve();
assert(sol.stats.success == 1, 'Error computing optimal input');

u = opti.value(U(:,1));

% Use the current solution to speed up the next optimization
opti.set_initial(sol.value_variables());
opti.set_initial(opti.lam_g, sol.value(opti.lam_g));
end
```

You can then simulate and plot the result of your controller with the following code

```
quad = Quad();
CTRL = ctrl_NMPC(quad);

sim = quad.sim(CTRL)
quad.plot(sim)
```

Hint: As in our NMPC exercise, you can evaluate the dynamics of the quadcopter using CASADI variables x and u via the call `quad.f(x,u)`

Deliverable 6.1 |

- Explanation of your design procedure and choice of tuning parameters
- Explanation of why your nonlinear controller is working better than your linear one.
- Plots showing the performance of your controller.
- m-code for your controllers, and code to produce the plots in the previous step

Summary of `Quad.m`

`Quad.m` is a Matlab class captures the behaviours of our quad copter.

You can list its functions with `methods(Quad)`

The following is a list of the functions that you might want to use

```
% Compute the quad dynamics
dx = quad.f(x, u)

% Plot the trajectory of the quad. sim is the output of quad.sim()
quad.plot(sim)

% Compute trim state and input for hovering flights
[xs, us] = quad.trim()

% Return a linearization of the quad around the equilibrium point xs, us
sys = quad.linearize(xs, us)

% Decompose the quad copter into four systems around the given equilibrium
[sys_x, sys_y, sys_z, sys_yaw] = quad.decompose(sys, xs, us)

% Simulate the nonlinear quadcopter to track an MPC reference
sim = quad.sim(ctrl_x, ctrl_y, ctrl_z, ctrl_yaw)

% Split the state into its parts (also works on casadi variables)
[omega, theta, vel, pos] = quad.parse_state(x)
```