Model Predictive Control: Mini-Project

In this project, you will develop an MPC controller to fly a rocket prototype.

The project is worth 40% of your final grade and is due on Friday, January 12st

Report and handing-in 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).
- Report:
  - 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.
- Code:
  - Include a directory for each deliverable containing all the m-files to run the deliverable.
  - Create a file in each directory `Deliverable_xxx.m` which can be run to produce all the the figures for the deliverable.
  - Compress all the code / directories into a single zip file for submission.

Before you start

- Make sure you have installed YALMIP, MPT3, Gurobi and Casadi according to the course exercise setup instructions on Moodle.
- Download and unpack the file `rocket_project.zip` from Moodle.
- Run `rocket = Rocket(1/20);` If this executes correctly, then your setup should be ready to go.

Part 1 | System Dynamics

Building a model of the system dynamics from physical principles is a crucial step in the development of an MPC controller and is a significant part of the task in practice. However, as this process is out of the scope of this course, you are not required to model the rocket by yourself. Instead, we provide you with a nonlinear model.

On the way towards thrust vector control for combustion engine rockets, we study a small-scale prototype where the rocket engine is replaced by high-performance drone racing propellers as depicted in fig. 1. The propellers counter-rotate such that their torques cancel each other out in stationary flight. The propeller pair is mounted on a gimbal and can be tilted by two servos.

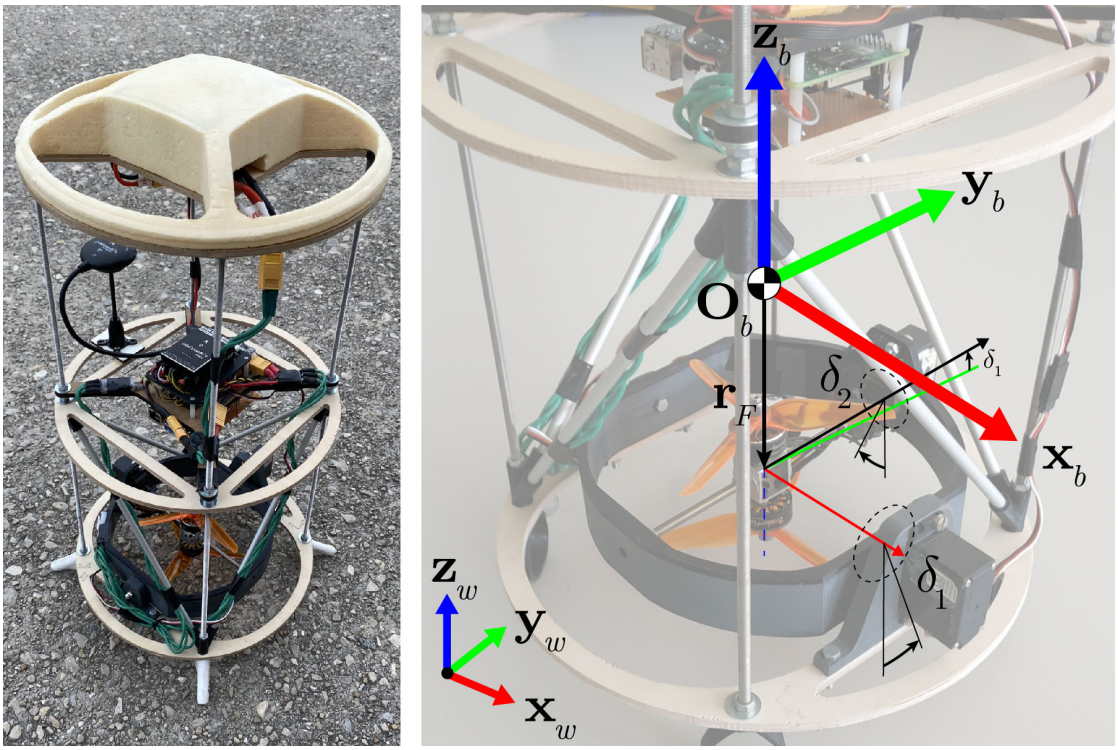

Figure 1: Rocket-shape drone at a glance.

System Definition In order to derive a nonlinear model, consider the following 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 rocket (see fig. 1). The second one is the world frame (subscript w) which is a fixed, inertial frame. We are going to derive a 12-state description of the system with the state vector

$$\mathbf{x} = \begin{bmatrix} \boldsymbol{\omega}^T & \boldsymbol{\varphi}^T & \mathbf{v}^T & \mathbf{p}^T \end{bmatrix}^T, \quad [\mathbf{x}] = \begin{bmatrix} \text{rad/s} & \text{rad} & \text{m/s} & \text{m} \end{bmatrix}^T$$

where $\boldsymbol{\omega} = \begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}^T$ are the angular velocities about the body axes. The Euler angles $\boldsymbol{\varphi} = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^T$ represent the attitude of the body frame with respect to the world frame. The

rotation from world to body frame is obtained by three consecutive rotations about the body axes: 1. α about $\mathbf{x}_b$, 2. β about $\mathbf{y}_b$, 3. γ about $\mathbf{z}_b$. The velocity and position, $\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T$ and $\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$, are expressed in the world frame, i.e., $\mathbf{v} = \dot{\mathbf{p}}$.

The input vector of the model is

$$\mathbf{u} = \begin{bmatrix} \delta_1 & \delta_2 & P_{avg} & P_{diff} \end{bmatrix}^T, \quad [\mathbf{u}] = \begin{bmatrix} \text{rad} & \text{rad} & \% & \% \end{bmatrix}^T$$

where δ_1 and δ_2 are the deflection angles of servo 1 (about $\mathbf{x}_b$) and servo 2 (about rotated $\mathbf{y}_b$), respectively, up to $\pm 15^\circ$ ($= 0.26$ rad).

Although the physical controls are the power settings of motor 1 and 2, we use a convenient abstraction that corresponds to the resulting behavior of the motor pair. $P_{avg} = (P_1 + P_2)/2$ is the average throttle and $P_{diff} = P_2 - P_1$ is the throttle difference between the motors. To hold the throttle of each individual motor within $[0, 100\%]$ while P_{diff} might be up to $\pm 20\%$, we have to limit the valid range for P_{avg} to $[20\%, 80\%]$.

Forces and Moments Simplified, the combined propellers produce a thrust force of magnitude $F(P_{avg})$ and a differential moment of magnitude $M_\Delta(P_{diff})$. They apply along/about the motor axis that is tilted by servo 1 and 2 with deflection angles δ_1, δ_2:

$$ {}_b\mathbf{e}_F(\delta_1, \delta_2) = \begin{bmatrix} \sin\delta_2 \\ -\sin\delta_1 \cos\delta_2 \\ \cos\delta_1 \cos\delta_2 \end{bmatrix} \tag{1}$$

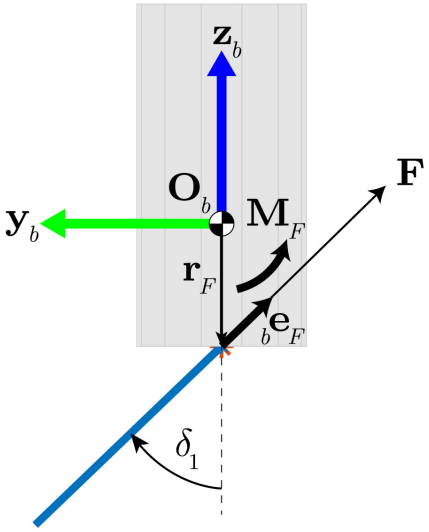

Figure 2: Side view of the rocket.

When the axis is tilted, the thrust vector introduces another moment about the center of mass, ${}_b\mathbf{M}_F = \mathbf{r}_F \times {}_b\mathbf{F}$, so that the resulting force and moment acting on the body are, expressed in body

frame:

$$ {}_b\mathbf{F} = F \cdot {}_b\mathbf{e}_F \tag{2} $$

$$ {}_b\mathbf{M} = M_\Delta \cdot {}_b\mathbf{e}_F + {}_b\mathbf{M}_F \tag{3} $$

where $\times$ is the cross product and $\cdot$ is the dot product.

Linear and Angular Dynamics The acceleration of the center of mass in the inertial (world) frame is given by

$$ \dot{\mathbf{v}} = \mathbf{T}_{wb} \cdot {}_b\mathbf{F}/m - \mathbf{g} \tag{4} $$

where $\mathbf{T}_{wb}(\boldsymbol{\varphi})$ is the direction cosine matrix that transforms a vector from body to world frame (i.e., ${}_w\mathbf{F} = \mathbf{T}_{wb} \cdot {}_b\mathbf{F}$), m is the body mass (`rocket.mass`), and $\mathbf{g} = \begin{bmatrix} 0 & 0 & g \end{bmatrix}^T$ is the gravitational acceleration.

The angular dynamics in the body frame is given by

$$ \dot{\boldsymbol{\omega}} = \mathbf{J}^{-1} \left(-\boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega} + {}_b\mathbf{M} \right), \tag{5} $$

where $\mathbf{J}$ is the inertia matrix of the vehicle.

Attitude Kinematics The rate of change of the Euler angles is a function of the attitude $\boldsymbol{\varphi} = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^T$ expressing the rotating body frame and the angular velocity in the body frame according to the kinematic differential equation

$$ \dot{\boldsymbol{\varphi}} = \frac{1}{\cos\beta} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma\cos\beta & \cos\gamma\cos\beta & 0 \\ -\cos\gamma\sin\beta & \sin\gamma\sin\beta & \cos\beta \end{bmatrix} \boldsymbol{\omega}. \tag{6} $$

When we put all of this together, we get the dynamic equations for the rocket

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

which have been implemented in the function `f` in the Matlab class `Rocket.m`, which is in the directory `src`.

Realistic Simulation What we have seen so far is the *nominal* nonlinear dynamics, i.e., the dynamics we obtain from first principles and with several practical assumptions and simplifications. However, there exist no models that perfectly fit reality. There are always unmodeled effects left, and when we perform system identification, we never obtain perfect model parameters.

In order to mimic some realistic model mismatch between the MPC prediction model and the simulated 'reality', we have implemented a reasonably different dynamics when you run nonlinear simulations. **You will observe that the trajectories of the same simulation will be slightly different each time.**

Todo 1.1 | Study the functions `f` and `getForceAndMomentFromThrust` in the Matlab `Rocket` class to confirm that it implements the dynamics of the system as described above.

To evaluate the functions, you can call them independently:

```
Ts = 1/20;
rocket = Rocket(Ts);
u = [d1, d2, Pavg, Pdiff]'; % (Assign appropriately)
[b_F, b_M] = rocket.getForceAndMomentFromThrust(u)

x = [w,  phi,  v,  p]'; % (Assign appropriately)
x_dot = rocket.f(x, u)
```

Todo 1.2 | Simulate the rocket with various step inputs to confirm that the dynamics responds as expected.

To simulate the nonlinear model for two seconds starting from $\mathbf{x}_0$ with a constant input, you can use:

```
rocket = Rocket(Ts);
Tf = 2.0;                                          % Simulation end time

x0 = [deg2rad([2 -2 0, -2 2 0]),  0 0 0,  0 0 0]'; % (w, phi, v, p) Initial state
u  = [deg2rad([2 0]),  60,   0  ]';                % (d1 d2 Pavg Pdiff) Constant input
[T, X, U] = rocket.simulate(x0, Tf, u);            % Simulate unknown, nonlinear model

rocket.anim_rate = 1.0;                            % Visualize at 1.0x real-time
rocket.vis(T, X, U);
```

A few things to try to see if the rocket is behaving as you think it should. Find input $\mathbf{u}$ that will cause the rocket to:

- Ascend/descend vertically without tipping over.
- Rotate about its body x/y/z axis.
- Fly along the x/y/z axis.
- Hover in space.

Part 2 | Linearization

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

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

```
rocket = Rocket(Ts);

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

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

Note that we have named all the states in the linearized model. Type `sys` and you will see the ordering of the states.

An aside on trimming and linearization

We have computed a trim point (x_s, u_s), which is a steady-state state and input pair such that $0 = f(x_s, u_s)$. We have then linearized our system around this point, $\dot{x} \approxeq A(x-x_s)+B(u-u_s)$. When we design our linear controllers, we do so for the linear system $\dot{x} = Ax + Bu$. This means that if the MPC controller applies an input $u^\star$, the true input applied to the system is $u = u^\star + u_s$, and therefore we will need to take the trim value into account when determining the constraints for our MPC controller.

Note that the `simulate_f` function will add the trim to your MPC controller's input when you provide it with a linear model.

Study the resulting **A**, **B**, **C** and **D** matrices until you recognize that the linearized system about the trim point can be broken into four independent/non-interacting systems.

Deliverable 2.1 | Explain from an intuitive physical / mechanical perspective, why this separation into independent subsystems is possible.

Todo 2.2 | Compute the four independent systems above using the following command

```
[sys_x, sys_y, sys_z, sys_roll] = rocket.decompose(sys, xs, us)
```

Four models are produced:

`sys_x` | Thrust vector angle δ_2 to position x. The system has four states: ω_y, β, v_x, x.

`sys_y` | Thrust vector angle δ_1 to position y. The system has four states: ω_x, α, v_y, y.

`sys_z` | Average throttle P_{avg} to height z. The system has two states: v_z and z.

`sys_roll` | Differential throttle P_{diff} to roll angle γ. The system has two states: ω_z and γ.

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

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

Part 3 | Design MPC Controllers for Each Sub-System

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

We will use a sampling period of $T_s = 1/20$ seconds. The continuous-time models produced in the previous section must be discretized using `sys_d = c2d(sys, Ts)`. This is done for you when using the provided `MpcControl_*` template files.

Constraints

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

$$|\alpha| \leq 10^\circ = 0.1745 \text{ rad}$$
$$|\beta| \leq 10^\circ = 0.1745 \text{ rad}$$

Note that the linearized roll sub-system is valid for any roll angle γ. However, the linearizations of the x and y sub-systems become less accurate as γ moves away from the linearization point. We need to be aware of this when we run the linearized controllers in the nonlinear simulation of Part 4.

In addition to the mechanical input constraints specified in Part 1, another requirement comes up from an engineering perspective. Experiments with the prototype have shown that the rocket descends too quickly when less than 50% average throttle is given. For safety reasons, we want to limit the downward acceleration that can occur to the rocket, and we therefore require a minimum average throttle of 50% at all times:

$$50\% \leq P_{avg} \leq 80\%$$

Design MPC Regulators

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

- Recursive satisfaction of the input and angle constraints.
- Stabilization of the system to the origin (i.e., all states equal to zero).
- Settling time no more than seven seconds when starting stationary at 3 meters from the origin (for x, y and z) or stationary at 40° for roll.

To help you design the controllers, we have created four files: • `MpcControl_x.m` • `MpcControl_y.m` • `MpcControl_z.m` • `MpcControl_roll.m` which you will find in the `templates` directory. Copy these into the `Deliverable_3_1` directory and then make your changes. Your job is to fill in the function `setup_controller` in each file.

You can then get the control from solving the MPC problem via the following code:

```
Ts       = 1/20; % Sample time
rocket   = Rocket(Ts);
[xs, us] = rocket.trim();
sys      = rocket.linearize(xs, us);
[sys_x, sys_y, sys_z, sys_roll] = rocket.decompose(sys, xs, us);

% Design MPC controller
H = ..; % Horizon length in seconds
mpc_x = MpcControl_x(sys_x, Ts, H);

% Get control input (_x is the index of the subsystem here)
u_x = mpc_x.get_u(x_x)
```

Before applying MPC in closed-loop, you should always check first if the optimal open-loop trajectory from a representative state is reasonable. This helps to understand whether the underlying optimal control problem is correctly formulated or, in case of unintended results, how it should be adjusted. You can use the following code to plot the open-loop trajectory:

```
% Evaluate once and plot optimal open-loop trajectory,
% pad last input to get consistent size with time and state
[u, T_opt, X_opt, U_opt] = mpc_x.get_u(x);
U_opt(:,end+1) = NaN;
% Account for linearization point
X_opt = ...
U_opt = ...
ph = rocket.plotvis_sub(T_opt, X_opt, U_opt, sys_x, xs, us); % Plot as usual
```

You can use the `simulate_f` method to simulate a particular dynamics f, which can be a function handle or in this case, a continuous-time linear system that you have obtained from the `Rocket` class. To have the control law evaluated during closed-loop simulation, hand it over as a function handle, and give a zero reference for regulation. The function `plotvis_sub` plots the states and inputs of the sub-system and visualizes its trajectory in 3D by setting all other sub states and inputs to the trim point $(\mathbf{x}_s, \mathbf{u}_s)$.

```
[T, X_sub, U_sub] = rocket.simulate_f(sys_x, x0, Tf, @mpc_x.get_u, 0);
ph = rocket.plotvis_sub(T, X_sub, U_sub, sys_x, xs, us);
```

Where `x0` here is the state of the particular subsystem being simulated.

Note: If you get warnings that the physical limits of your inputs are violated during simulation, then your controller is outputing infeasible inputs and it's incorrect.

Deliverable 3.1 |

- Explanation of design procedure that ensures recursive constraint satisfaction.
- Explanation of choice of tuning parameters. (e.g., $\mathbf{Q}$, $\mathbf{R}$, H, terminal components).
- Plot of terminal invariant set for each of the dimensions, and explanation of how they were designed and how their respective tuning parameters were 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();
```

- Open-loop and closed-loop plots for each dimension of the system starting stationary at 3 meters from the origin (for x, y and z) or stationary at 30° for roll.
- Matlab 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.

For x, y and z the reference is a position, and for $roll$ it is 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, • `MpcControl_x.m` • `MpcControl_y.m` • `MpcControl_z.m` • `MpcControl_roll.m` and fill in the function `setup_steady_state_target` in each one.

Make sure you are editing your code in the directory for Deliverable 3.2. Do not overwrite Deliverable 3.1

You can now get your control input via:

```
u = mpc_x.get_u(x, pos_ref)
```

For showing the reference in the plots, you can enter it to `plotvis_sub`:

```
[T, X_sub, U_sub] = rocket.simulate_f(sys_x, x0, Tf, @mpc_x.get_u, ref_x);
ph = rocket.plotvis_sub(T, X_sub, U_sub, sys_x, xs, us, ref_x);
```

Deliverable 3.2 |

- Explanation of your design procedure and choice of tuning parameters.
- Open-loop and closed-loop plots for each dimension starting at the origin and tracking a reference to -4 meters (for x, y and z) and to 35° for roll.
- Matlab code for the four controllers, and code to produce the plots in the previous step.

[2] It is possible to use a terminal set for tracking here by noticing that there are no constraints on the position of the rocket 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 $\mathcal{X}_f + x_s = \{x \mid G(x - x_s) \leq g\} = \{x \mid Gx \leq g + Gx_s\}$.

Part 4 | Simulation with Nonlinear Rocket

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

Todo 4.1 | Simulate the full nonlinear system with your four controllers from the origin as initial state.

Note that running the linear controllers in the nonlinear simulation results in a model mismatch for the predictive controllers, in addition to the randomized simulation model. As the real (simulated) system behaves a bit differently than predicted in the MPC controller, states may violate the constraints although predicted to be inside.
When you encounter infeasibilities of the linear controllers, look at the trajectory plots up to this point and try to understand which state(s) are leading to constraint violations in the next step. *Hint*: Higher weights on the angular velocities avoid too aggressive maneuvers.

Todo 4.2 | (Bonus) Instead of finding robust tuning weights for any possible randomized model mismatch by trial and error, it is strongly recommended to formulate soft state constraints by introducing slack variables. This will ensure your problem stays feasible under model mismatch.

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

```
% Merge four sub-system controllers into one full-system controller
mpc = rocket.merge_lin_controllers(xs, us, mpc_x, mpc_y, mpc_z, mpc_roll);

% Evaluate once and plot optimal open-loop trajectory,
% pad last input to get consistent size with time and state
x0 = zeros(12,1);
ref4 = [2 2 2 deg2rad(40)]';
[u, T_opt, X_opt, U_opt] = mpc.get_u(x0, ref4);
U_opt(:,end+1) = nan;
ph = rocket.plotvis(T_opt, X_opt, U_opt, ref4); % Plot as usual

% Setup reference function
ref = @(t_, x_) ref_TVC(t_);

% Simulate
Tf = 30;
[T, X, U, Ref] = rocket.simulate(x0, Tf, @mpc.get_u, ref);

% Visualize
rocket.anim_rate = 1; % Increase this to make the animation faster
ph = rocket.plotvis(T, X, U, Ref);
ph.fig.Name = 'Merged lin. MPC in nonlinear simulation'; % Set a figure title
```

Deliverable 4.1 | A plot of your controllers successfully tracking the reference path and reference roll angle. If your tracking performance is not good, explain how you adapt your tuning to improve it.

Part 5 | Offset-Free Tracking

In this section, we assume that the mass of the rocket is significantly different from what we have modeled, and we want to extend the z-controller to compensate.

We assume the mass offset enters the dynamics of the system in the $z-$direction according to

$$\mathbf{x}^+ = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{B}d$$

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 `MpcControl_z.m` file.
- Update the functions `setup_controller` and `setup_steady_state_target` in `MpcControl_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 a mass mismatch by manipulating the `rocket.mass` property **after** having created the controller, i.e., just before simulation.

For demonstrative purposes, we will pick a ≈25% higher initial mass (2.13kg).

```
% Merge four sub-system controllers into one full-system controller
mpc = rocket.merge_lin_controllers(xs, us, mpc_x, mpc_y, mpc_z, mpc_roll);

x0 = ...
ref = ...

% Manipulate mass for simulation
rocket.mass = 2.13;
[T, X, U, Ref] = rocket.simulate(x0, Tf, @mpc.get_u, ref);
```

Once you have written the `setup_estimator` function, you can test it in simulation by replacing the function `rocket.simulate` with `rocket.simulate_est_z`. In the z direction, the controller will now act based on the state estimates from the observer. You can obtain the estimates of the z states from the corresponding rows of the `Z_hat` output in which the last row is the disturbance estimate d.

```
[T, X, U, Ref, Z_hat] = rocket.simulate_est_z(x0, Tf, @mpc.get_u, ref, mpc_z, sys_z);
```

Deliverable 5.1 |

- Explanation of your design procedure and choice of tuning parameters.
- Plot showing the impact of changing the mass on your original controller from Part 4, and then another plot showing that your controller now achieves offset-free tracking. Simulate for 8 seconds from `x0 = [zeros(1, 9), 1 0 3]'` with `ref = [1.2, 0, 3, 0]'`. Scale the plot for z if is necessary to understand the behavior.
- Matlab code for your controllers, and code to produce the plots in the previous step.

Todo 5.2 | Simulate the system with changing mass.

In addition to the different total mass from Part 5.1, we now assume that half of the initial rocket mass is actually fuel, and it will decrease depending on the power consumption of the motor (parameterized by `rocket.mass_rate`). Once all the fuel/energy is consumed, the motor will not produce any thrust anymore. The remaining rocket mass will then be again half of the initial mass.

Let the mass rate coefficient be -0.27, which allows for only a couple of seconds of hovering.

```
% Manipulate initial mass and mass rate coefficient for simulation
rocket.mass = 2.13;
rocket.mass_rate = -0.27;
```

Deliverable 5.2 |

- Plot showing the impact of having a thrust-dependent mass decrease during flight on your controller from Part 5.1 (use the same simulation setup).
- In the first couple of seconds of the simulation, despite the estimator, why is there still a tracking offset in height? Explain how the estimator could be modified to achieve offset-free tracking also for the changing mass case. (Without implementation!)
- With time-varying mass, you might observe multiple distinct properties of the trajectory (depending on your tuning). Briefly describe which different behaviors you can see along the simulation. Towards the end of the simulation, what unexpected behavior can you observe, and why? (If you do not see anything, increase the simulation time a bit.)
- Bonus question: Simulating for even longer (up to ≈20 seconds), you can observe up to two more distinct events in the plots of the closed-loop system. What are they, and how can you explain them?
- Matlab code to produce the plots.

Part 6 | Nonlinear MPC

Todo 6.1 | Develop a nonlinear MPC controller for the rocket using CASADI. Your controller should take the full state as input, and provide four input commands (i.e., we no longer decompose the rocket into four sub-systems here).

The NMPC controller operates on the nonlinear model and can therefore cover the whole state space. However, the Euler angle attitude representation used in the model has a singularity at $\beta = 90°$. Although you should never get to this attitude in normal operation, you should constrain this angle to safe numerical values, i.e., $|\beta| \leq 75°$.

Use the template code `NmpcControl.m`. 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.

Use the NMPC controller to track the given reference like in Part 4 (default maximum roll angle $|\gamma_{ref}| = 15°$). In a second experiment, see how both the linear and nonlinear controllers perform for a maximum roll reference of $|\gamma_{ref}| = 50°$. (See example code below for how to change the maximum roll reference.) You may need to re-tune your linear controllers a little for it to be feasible under this more extreme reference request.

Before applying MPC in closed-loop, you should always check first if the optimal open-loop trajectory from a representative state is reasonable (see code example below).

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

```
Ts = 1/20;
rocket = Rocket(Ts);

H = ..; % Horizon length in seconds
nmpc = NmpcControl(rocket, H);

% MPC reference with default maximum roll = 15 deg
ref = @(t_, x_) ref_TVC(t_);

% MPC reference with specified maximum roll = 50 deg
roll_max = deg2rad(50);
ref = @(t_, x_) ref_TVC(t_, roll_max);

% Evaluate once and plot optimal open-loop trajectory,
% pad last input to get consistent size with time and state
[u, T_opt, X_opt, U_opt] = nmpc.get_u(x, ref);
U_opt(:,end+1) = nan;
ph = rocket.plotvis(T_opt, X_opt, U_opt, ref);

Tf = 30;
[T, X, U, Ref] = rocket.simulate(x0, Tf, @nmpc.get_u, ref);
```

Hint: Because it's likely that you will want a shorter horizon here than ideal in order to keep the computation time under control, you may find a terminal cost very helpful. A common approximate terminal cost is to linearize your system and compute a terminal cost based on this (don't forget to discretize).

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

Deliverable 6.1 |

- Explanation of your design procedure and choice of tuning parameters.
- Discuss the pros and cons of your nonlinear controller vs the linear ones you developed earlier.
- Plots showing the performance of your controller.
- Matlab code for your controllers, and code to produce the plots in the previous step.

NMPC with Delay / Delay Compensation

The computation time to solve nonlinear MPC problems is generally significantly higher than for linear MPC problems. In practice, this poses a major problem for the deployment of such controllers. In this last part, we will see how computational delay (as any other delay) can destabilize any closed-loop system, and implement a simple countermeasure.

Todo 6.2 | Simulate your NMPC controller in closed-loop with increasing values for the delay (`rocket.delay`). Simulate for 2.5 seconds from the origin to a constant reference (see code example below). Simulate with a custom mass of 1.75 in this part.

For this part, we will use a sampling period of $T_s = 1/40$ seconds, i.e., `rocket.delay = 1` corresponds to a delay of 25 milliseconds. For comparison, your Matlab NMPC implementation needs several hundred milliseconds to compute one solution, which demonstrates why real-world applications need highly efficient implementations. We have reduced the sampling period here in order to more clearly show the impact of the computational delay.

Figure 3 shows the problem. We see that the measurement is taken at time k, but that since it takes more than six sample periods to solve the MPC optimization problem, the optimal input is not computed until time $k + 7$ (Note that seven is just an example here). As a result, we find ourselves applying the input $u^\star(x_k)$ at time $k + 7$.

One solution to this is (approximate) delay compensation shown in Figure 4. At time k, we simulate the system forward in time to time $k + 7$ to get the estimated state $\hat{x}_{k+7}$. We then can apply the input $u^\star(\hat{x}_{k+7})$ at time $k + 7$.

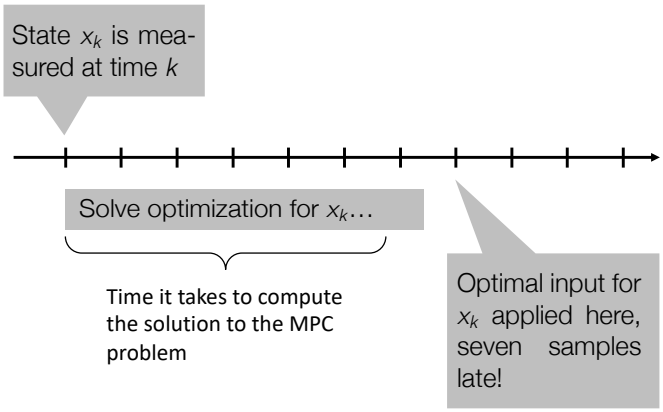

Figure 3: Problem with computational delay

Implement a simple delay compensation algorithm in the `get_u()` function in `NmpcControl.m` by

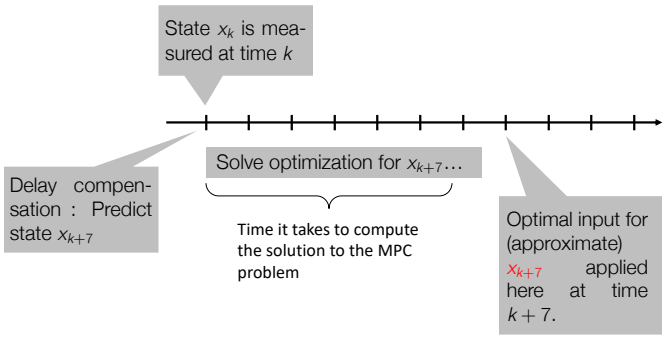

Figure 4: Idea behind delay compenstation

simulating the current measurement forward to the moment when the MPC solution will actually be applied to the system. Use a simple Euler integration scheme. Remember to buffer the current input in `obj.mem_u` for later usage. Think of a meaningful initialization for the buffer at the end of the `NmpcControl()` constructor.

Simulate again and observe how the closed-loop performance gets better as the delay compensation steps meet the actual delay steps in the system.

```
Ts = 1/40; % Higher sampling rate for this part!

... Define NMPC ...

x0 = zeros(12, 1);
ref = [0.5, 0, 1, deg2rad(65)]';

Tf = 2.5;
rocket.mass = 1.75;
rocket.delay = ..;   % 0 if not specified
[T, X, U, Ref] = rocket.simulate(x0, Tf, @nmpc.get_u, ref);
```

Deliverable 6.2 |

- How much delay is needed to observe a drop in closed-loop performance? How much delay to make the closed-loop system unstable?
- Plot the closed-loop trajectory for a partially delay-compensating controller (compensated delay $<$ actual delay), and a fully delay-compensating controller (compensated delay $=$ actual delay).