Model Predictive Control: Mini-Project

In this project, you will develop an MPC controller to land a rocket under thrust.

The project is worth 40% of your final grade and is due on Sunday, January 11th, 2026

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 Python 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 files to run the deliverable.
  - Create a Jupyter Notebook for each deliverable, `Deliverable_xxx.ipynb`, which can be run to produce all the figures for the deliverable.
  - Compress all the code into a single zip file for submission.

Before you start

- Make sure you have setup your computer according to the course exercise setup instructions on Moodle.
- The files are located in the same Git repository as the course exercises: `MPC-Course-EPFL`.
- Run `Todo_1/Todo_1.ipynb`. If this file runs 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.

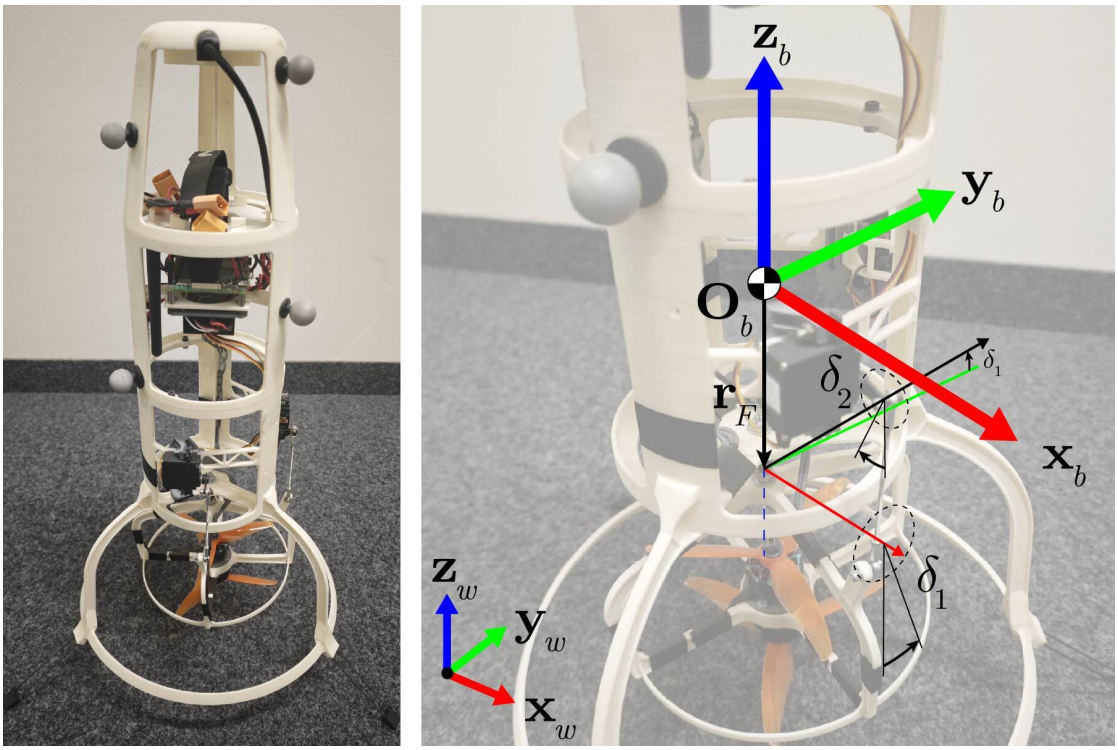

Figure 1: Rocket-shape drone at a glance.

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 ± 20, we have to limit the valid range for P_{avg} to $[10, 90]$.

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:

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

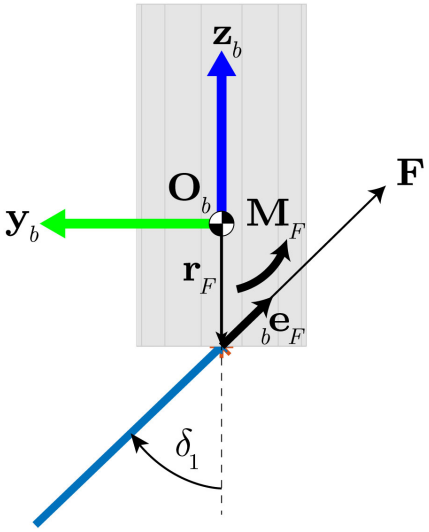

Figure 2: Side view of the rocket.

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 $\boldsymbol{r} = \begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix}^T$ expressing the rotating body frame and the angular velocity in the body frame according to the kinematic differential equation

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

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

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

which have been implemented in the method `f` in the file `src/rocket.py` and will constitute the rocket model for the project.

Todo 1.1 | Starting from the `Todo_1/Todo_1.ipynb` notebook, study the methods `f` and `getForceAndMomentFromThrust` in the Python `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=Ts, model_params_filepath=rocket_params_path)
u = np.array([d1, d2, Pavg, Pdiff]) # (Assign appropriately)
b_F, b_M = rocket.getForceAndMomentFromThrust(u)

x = np.array([w(3),  phi(3),  v(3),  p(3)]) # (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:

```
Tf = 2.0                                                    # Simulation duration
x0 = np.array([w0(3),  r0(3),  v0(3),  p0(3)])             # (w, phi, v, p) initial state
u  = np.array([2*np.pi/180, 0, 60, 0])                      # (d1 d2 Pavg Pdiff) input
T, X, U = rocket.simulate(x0, Tf, u, method='nonlinear') #  nonlinear model

vis = RocketVis(rocket, rocket_obj_path)                    # Visualization utility
vis.anim_rate = 1.0
vis.animate(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 axes.
- Fly along the x/y/z axes.
- 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 and linearized version of the rocket:

```
xs, us = rocket.trim() # Compute steady-state for which 0 = f(xs,us)
sys = rocket.linearize_sys(xs, us) # Linearize the nonlinear model about trim point
sys.info()
```

Here, trimming a system $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ means to find a state and input equilibrium pair $\mathbf{x}_s, \mathbf{u}_s$ such that $f(\mathbf{x}_s, \mathbf{u}_s) = 0$. We have then linearized our system around this point

$$\dot{\mathbf{x}} \approx A(\mathbf{x} - \mathbf{x}_s) + B(\mathbf{u} - \mathbf{u}_s). \tag{7}$$

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

Note that we have named all the states in the linearized model. Type `sys.info()` and you will see the information, like ordering of the states and the inputs, $\mathbf{x}_s$, and $\mathbf{u}_s$, .etc. You can go through the class `Sys` in `rocket.py` to see more details.

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

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

```
sys_x, sys_y, sys_z, sys_roll = sys.decompose()
sys_x.info() # check information e.g. sys_x here
```

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.

Deliverable 2.1 | Explain in the report why it is possible to separate this system into independent subsystems from an intuitive physical/mechanical perspective.

Todo 2.3 | Discretization

In the following parts, you will implement discrete MPC controllers, i.e., the continuous-time models have to be discretized. **In this project, we always use a sampling period of $T_s = 1/20$ seconds.** Discretizing the continuous system (7) results in the following discrete-time delta dynamics:

$$\begin{aligned} &\Delta\mathbf{x} := \mathbf{x} - \mathbf{x}_s,\ \Delta\mathbf{u} := \mathbf{u} - \mathbf{u}_s \\ &\Delta\mathbf{x}^+ = A_d\Delta\mathbf{x} + B_d\Delta\mathbf{u}. \end{aligned} \tag{8}$$

This results in the standard linear dynamics that you have seen in the lecture for MPC design. In addition, the same discretization can be applied to each subsystem. You can use *scipy.signal cont2discrete* function to obtain the matrices of the discretized dynamics:

```
# e.g. sys x here
from scipy.signal import cont2discrete
Ts = 1/20;
Ad, Bd, Cd, _, _ = cont2discrete(system=(sys_x.A,sys_x.B,sys_x.C,sys_x.D), dt=Ts)

Ad, Bd, Cd = sys_x.discretize(Ts) # This has been integrated into the Sys class
```

Using the dynamics (8) in the MPC means that if the MPC computes an optimal input $\Delta u^\star$, the true input applied to the system is $u = \Delta u^\star + \mathbf{u}_s$. Additionally, we will need to take the trim value into account when determining the constraints for our MPC controller.

Part 3 | Design MPC Velocity Controllers for Each Sub-System

The project task is to implement a controller for landing a rocket into a mechanical arm recovery system (SpaceX style). The landing is divided into two phases. The first task is to design a controller for reaching close to the landing platform.

Your goal is to design a recursively feasible, stabilizing MPC controller that can track **velocity** step references for each of the dimensions x, y and z, and that can track **orientation** step references for the roll dimension.

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 the `scipy.signal cont2discrete` function.

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.

In addition to the mechanical input constraints specified in Part 1, another requirement comes from an engineering perspective. Experiments with the prototype have shown that the rocket descends too quickly when less than 40% 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 40% at all times:

$$40 \leq P_{avg} \leq 80$$

Design MPC Controller for Stabilization

Todo 3.1 | Design four MPC controllers for x, y, z and roll. The velocity controllers for x, y, z should not contain the position states, as we only aim to control the velocities. They need to have the following properties:

- Recursive satisfaction of the state and input constraints.
- Stabilization of the system to zero x, y, z velocities and zero roll angle.
- Settling time no more than seven seconds when starting at speed 5 m/s for each dimension (for x, y and z) or stationary at 40° for roll (γ).

To help you design the controllers, we have created five files inside the `LinearMPC_template` directory: `MPCControl_base.py` and its subclasses `MPCControl_xvel.py`, `MPCControl_yvel.py`, `MPCControl_zvel.py`, `MPCControl_roll.py`.

Your job is to fill in the methods `setup_controller()` and `get_u()` in the classes. Use the templates provided and make your changes. You can either fill separate methods for each subclass, or implement it generally in the base class with specific methods in the subclasses for costs and constraints.

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

```
Ts = 0.05
rocket = Rocket(Ts);

H = ...  # MPC horizon in seconds
mpc = MPCVelControl().new_controller(rocket, Ts, H)
u, x_ol, u_ol, t_ol, _ = mpc.get_u(t0, x0)
```

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:

```
vis = RocketVis(rocket, rocket_obj_path)
vis.anim_rate = 1.0
vis.animate(t_ol[:-1], x_ol[:,:-1], u_ol);
```

You can use the `rocket.simulate_step` method to simulate a step for the *linear* or *nonlinear* system. To have the control law evaluated during closed-loop simulation, you can use the `rocket.simulate_control` method giving a zero reference for regulation.

```
sim_time = ...      # Simulation length
H = ...             # MPC horizon in seconds
x1 = rocket.simulate_step(x0, Ts, u, method='nonlinear')
t_cl, x_cl, u_cl, t_ol, x_ol, u_ol = rocket.simulate_control(mpc, sim_time, H, x0, \
                                                             method='linear')
```

You can use the following code to plot the closed-loop trajectory together with open-loop predictions:

```
vis.animate(t_cl[:-1], x_cl[:,:-1], u_cl, T_ol=t_ol[...,:-1], X_ol=x_ol, U_ol=u_ol);
```

For debugging and tuning purposes, it can be useful to verify the subsystem's controllers separately. A single MPC subsystem can be verified with the following commands:

```
from LinearMPC.MPCControl_xvel import MPCControl_xvel
xs, us = rocket.trim()
A, B = rocket.linearize(xs, us)
mpc_x = MPCControl_xvel(A, B, xs, us, Ts, H) # Full 12x12 A and 12x4 B matrices
u0, x_traj, u_traj = mpc_x.get_u(x0)         # x and u for single subsystem
```

The example is for the x subsystem, but analogous can be obtained for the others.

Deliverable 3.1 | Satisfy and explain the following points:

- Explanation of design procedure that ensures recursive constraint satisfaction.
- Explanation of choice of tuning parameters. (e.g., Q, 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 a `mpt4py` `Polyhedron` higher than two dimensions, you can plot its projections with:

```
Xf.projection(dims=(0,1)).plot(ax)
Xf.projection(dims=(1,2)).plot(ax)
```

- Open-loop and closed-loop plots for each dimension of the system starting at 5 m/s (for x, y and z) or stationary at 30° for roll.
- Python code for the four controllers, and code to produce the plots in the previous step.

Design MPC Controllers for Tracking

Todo 3.2 | Extend your controllers so that they can track constant velocity references. For x, y and z the reference is a velocity, and for roll it is an angle in radians.

To implement your controllers, modify the four controllers from the previous section.

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, x_ol, u_ol, t_ol = mpc.get_u(t0, x0, x_target=ref)
```

For showing the reference in the plots, you can enter it to the animation code:

```
vis.animate(t_cl[:-1], x_cl[:,:-1], u_cl, Ref=ref)
```

Deliverable 3.2 | Satisfy and explain the following points:

- Explanation of your design procedure and choice of tuning parameters.
- Open-loop and closed-loop plots starting at the origin and tracking a velocity reference of 3 m/s (for x, y and z) and to 35° for roll.
- Python code for the four controllers and code to produce the plots in the previous step.

PID Position Tracking Controller

Todo 3.3 | The control objective is to take the rocket from a high altitude and velocity to a stationary point in the sky from which a controlled landing can be executed. To do this, you will extend your controller with a PID position tracking controller.

A PI controller for computing velocity setpoints can be added to the control loop by passing it to the `simulate_control` method in the `rocket` object with the following notation. The varying velocity reference in time is returned by the method.

```
pos_target = np.array([0,0,10.0])
pos_controller = PIControl(pos_target)
..., ref = rocket.simulate_control(..., pos_control=pos_controller, ...)
```

The PID controller is already implemented and tuned in the template files, it only needs to be passed to the rocket simulator.

Deliverable 3.3 | Satisfy and explain the following points:

- Explanation of your design procedure and choice of tuning parameters.
- Closed-loop plots starting at pos $= [50, 50, 100]$m tracking a position reference at pos $= [0, 0, 10]$m and to 0° for roll.
- Python code for PID and MPC controllers and code to produce the plots in the previous step.

Part 4 | Simulation with Nonlinear Rocket

In this section, you will use your controllers on the nonlinear rocket simulator.

Todo 4.1 | Simulate the full nonlinear system with your four controllers on the same task as the previous point. 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 | (Optional) 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.

Deliverable 4.1 | A plot of your controllers successfully completing the maneuver. If your tracking performance was not good, explain how you adapted your tuning to improve it.

Part 5 | Offset-Free Tracking

As seen in the section above, fuel consumption influences the weight of the rocket. 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, design an offset-free tracking controller.

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

You can simulate your system with mass decreasing due to fuel consumption. The initial mass and fuel rate can be manipulated with the parameters `rocket.mass` and `rocket.fuel_rate`. Change the rocket mass after computing the controllers to test the disturbance estimator.

Hint | Treat the value `x0` input into your function `get_u(x0)` as a measurement, and then run your estimator inside the `get_u` function and use your estimated state and disturbance when computing your control law.

Deliverable 5.1 | Satisfy and explain the following points:

- Explanation of your design procedure and choice of tuning parameters.
- Show the impact of simulating with `rocket.mass = 1.5` and `rocket.fuel_rate = 0` on your controller from Part 4, compared to your controller in Part 5, that should now achieve offset-free tracking. Simulate for 15 seconds from $\text{pos}_0 = [0, 0, 1]$ and $v_0 = [5, 5, 10]$ with velocity reference $v_{\text{ref}} = [0, 0, 0]$.
- Discuss the estimation of the disturbance together with the estimation error over time. Is the disturbance constant for a constant rocket mass? Why or why not?
- Python 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 decreases linearly with thrust. Once all the fuel is consumed, the motor will not produce any thrust and the simulation will end.

Deliverable 5.2 | Satisfy and explain the following points:

- Show the impact of simulating with `rocket.mass = 2.0` and `rocket.fuel_rate = 0.1` on your controller. Simulate for 15 seconds from $\text{pos}_0 = [0, 0, 1]$ and $v_0 = [5, 5, 10]$ with velocity reference $v_{\text{ref}} = [0, 0, 0]$.

- 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.)
- Python code to produce the plots.

Part 6 | Robust Tube MPC for landing

In this section, you will implement a robust tube MPC controller for the second flight phase, from descent to a soft landing on the 'chopsticks'. We aim to control the rocket from a low height to very close to the ground, where you can imagine robotic arms could be used to capture the rocket, just as SpaceX does. Specifically, the landing phase is to move from $x = 3$, $y = 2$, $z = 10$, roll$= 30°$ to $x = 1$, $y = 0$, $z = 3$, roll$= 0°$.

Now that we're close to the ground, we add a constraint on z to avoid collision:

$$z \geq 0 \tag{9}$$

The rocket now has less fuel, and so we re-trim and re-linearize it around the target point $x = 1$, $y = 0$, $z = 3$:

```
x_ref = np.array([0.]*9 + [1., 0., 3.])
xs, us = rocket.trim(x_ref)
A, B = rocket.linearize(xs, us)
```

Check the new $\mathbf{x}_s$ and $\mathbf{u}_s$, and you'll see that the required steady-state thrust of the rocket is lower.

In the part 6.1, Your task is to design a tube-MPC **position** controller for the $z-$dimension only. (You will add nominal controllers for the other dimensions in the part 6.2). To help you design the controllers, we have created files inside the `LandMPC_template` directory: `MPCLandControl.py`, `MPCControl_base.py` and its subclasses `MPCControl_x.py`, `MPCControl_y.py`, `MPCControl_z.py`, `MPCControl_roll.py`.

Todo 6.1 | Design a robust tube MPC controller so that (9) is satisfied robustly for the subsystem `sys_z` subject to disturbances.

We consider a disturbance term w to capture the model mismatch for the subsystem `sys_z`. Define $\mathbf{x}_z$, $\mathbf{u}_z$, $\mathbf{x}_{s,z}$ and $\mathbf{u}_{s,z}$ as the corresponding state, input and steady-state pair in `sys_z`. Similar to the delta dynamics in (8), define $\Delta\mathbf{x}_z := \mathbf{x}_z - \mathbf{x}_{s,z}, \Delta\mathbf{u}_z = \mathbf{u}_z - \mathbf{u}_{s,z}$. Then we have the following discrete-time uncertain linear system:

$$\begin{aligned} &\Delta\mathbf{x}_z^+ = A_d\Delta\mathbf{x}_z + B_d\Delta\mathbf{u}_z + B_d w \ , \\ &w \in \mathbb{W} := [-15, 5] \ . \end{aligned} \tag{10}$$

where A_d and B_d are the corresponding discrete-time matrices in `sys_z`. This results in the standard uncertain constrained linear dynamics that you have seen in the lecture for tube MPC design. Using the above formulation, your task is to stabilize the delta-system to a neighbourhood of the origin, which will cause your rocket to reach close to $z = 3$.

Complete the function `get_u` in the object `MPCControl_z` and simulate your system. The simulator can be run with either no noise, random noise, or an extreme disturbance to help with your tuning:

```
mpc_z_rob = MPCControl_z(A, B, xs, us, Ts, H)
w_type = ...        # 'random', 'no_noise', 'extreme'
sim_time = ...      # Simulation length
t_cl, x_cl, u_cl = rocket.simulate_subsystem(mpc_z_rob, sim_time, x0, w_type)
```

Note that the trajectories produced are for the entire rocket, but the dimensions except `sys_z` are set to zero. Apart from the `.animate` function, you can plot the static trajectories for just the z-dimension as:

```
plot_static_states_inputs(t_cl[:-1], x_cl[:,:-1], u_cl, xs, 'sys_z')
```

Deliverable 6.1 |

- Explanation of your design procedure and choice of tuning parameters. Settling time is no more than four seconds when landing from $z = 10$ to $z = 3$.
- Plots of the minimal invariant set $\mathcal{E}$ and the terminal set $\mathcal{X}_f$. Give the vertices of the tightened input constraint $\tilde{\mathbb{U}}$.
- Two closed-loop plots (`plot_static_states_inputs`) showing your robust controller working under 'random' and 'extreme' disturbances.
- Python code for your robust tube controller, and code to produce the plots.

Todo 6.2 | Create nominal position controllers for $x-$, $y-$ and $roll-$ and then merge them with your robust MPC controller.

For the other three subsystems, `sys_x`, `sys_y`, `sys_roll`, derive their linearized systems, similar to (10), and design nominal MPC **position** controllers for them to track the position targets $x = 1$, $y = 0$, $roll = 0$. You will be simulating against the nonlinear model and so here you should use soft constraints and no terminal constraint.

Merge your robust MPC $z-$controller and three nominal MPC controllers using the `MPCLandControl` class.

```
x_ref = np.array([0.]*9 + [1., 0., 3.])  # Trim point
mpc = MPCLandControl().new_controller(rocket, Ts, H, x_ref=x_ref)
```

Simulate your rocket with the nonlinear model.

```
sim_time = ...      # Simulation length
t_cl, x_cl, u_cl, t_ol, x_ol, u_ol = rocket.simulate_land(mpc, sim_time, H, x0)
```

Apart from the `.animate` function, you can plot the static trajectory as:

```
plot_static_states_inputs(t_cl[:-1], x_cl[:,:-1], u_cl, xs)
```

Deliverable 6.2 |

- Explain your design procedure and choice of tuning parameters in. In the simulation with the nonlinear model, settling time is no more than four seconds when landing from $x = 3$, $y = 2$, $z = 10$, roll$= 30°$ to $x = 1$, $y = 0$, $z = 3$, roll$= 0°$.
- Closed-loop plots (`plot_static_states_inputs`) showing the performance of your merged MPC controller.
- Python code for your controllers, and code to produce the plots.

Part 7 | Nonlinear MPC

In this section, you will implement a nonlinear MPC controller for the same rocket landing task studied in Part 6. Your job is to complete the given Python file `nmpc_land.py` in the `LandMPC_template` directory, following the instructions below. Then you need to test your controllers in `Deliverable_7_1.ipynb`.

Todo 7.1 | Develop a nonlinear MPC controller 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 subsystems here).

NMPC controller design

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.,

$$|\beta| \leq 80°.$$

In addition, do not forget the constraints (9).

You should design the NMPC controller with a `get_u(x0)` function, i.e.:

```
nmpc = ...  # Your NMPC
u, x_ol, u_ol, t_ol = nmpc.get_u(t0, x0) # get_u should provide this functionality
```

Simulation with the nonlinear rocket

In this part, we only simulate NMPC with the nonlinear model based on the first principle shown in Part 1. You can test your controller with the following code:

```
sim_time = ...      # Simulation length
t_cl, x_cl, u_cl, t_ol, x_ol, u_ol = rocket.simulate_land(nmpc, sim_time, H, x0)
```

Apart from the `.animate` function, you can plot the trajectory as:

```
plot_static_states_inputs(t_cl[:-1], x_cl[:,:-1], u_cl, xs)
```

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

Hint | As in our NMPC exercise, you can evaluate the dynamics of the rocket using CASADI variables `x` and `u` via the call `self.f` in the `nmpc_land.py` template, i.e., `x_dot = self.f(x, u)`. Note it is the continuous-time dynamics.

Hint | If you are using the `Opti` class, you can use the following option to speed up the computation.

```
opts = {
    'expand': True,
    ... # other settings
```

```
}
opti.solver('ipopt', opts)
```

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. To save computation time, you should only compute the linearization and the terminal cost matrix once.

Deliverable 7.1 |

- Explain your design procedure and choice of tuning parameters. Settling time is no more than four seconds when landing from $x = 3$, $y = 2$, $z = 10$, roll$= 30^\circ$ to $x = 1$, $y = 0$, $z = 3$, roll$= 0^\circ$ in the simulation using the original model.
- Open-loop and closed-loop plots (`plot_static_states_inputs`) showing the performance of your controller in the simulation using the original model.
- Compare your NMPC controller vs your "robust MPC and three nominal MPCs" (in Todo 6.2). Discuss the pros and cons.
- Python code for your controllers, and code to produce the plots.