Programming Exercise
Control for Spacecraft Rendezvous

Introduction

In this programming exercise, you are in charge of designing a controller for the approaching stage of an orbital rendezvous maneuver, where one satellite approaches another object in space. During the approaching stage, the control system shall steer the chaser satellite into the vicinity of the target. Thereby, the control system should be robust against deviations of the chaser satellite's initial starting position, respect state and input constraints, as well as minimize overall fuel consumption.

Equations of motion

The dynamics of the chaser satellite are described in a moving coordinate frame centered at the target object, which moves on a circular orbit. The resulting second-order ordinary differential equations (ODEs) are the *Clohessy-Wiltshire-Hill (CWH) equations*,

$$\begin{bmatrix} \ddot{x}(t) \\ \ddot{y}(t) \\ \ddot{z}(t) \end{bmatrix} = \begin{bmatrix} 2\omega_n \dot{y}(t) + 3\omega_n^2 x(t) \\ -2\omega_n \dot{x}(t) \\ -\omega_n^2 z(t) \end{bmatrix} + \frac{1}{m} \begin{bmatrix} u_x(t) \\ u_y(t) \\ u_z(t) \end{bmatrix}, \tag{1}$$

where $x(t), y(t), z(t) \in \mathbb{R}$ are the spatial coordinates radially outward, along the orbit track, and along the orbital angular momentum vector of the chaser satellite relative to the target body, respectively, as can be seen in Figure 1. The thrust of the chaser satellite in the relative coordinate frame is given by the control inputs $u_x(t), u_y(t), u_z(t) \in \mathbb{R}$.

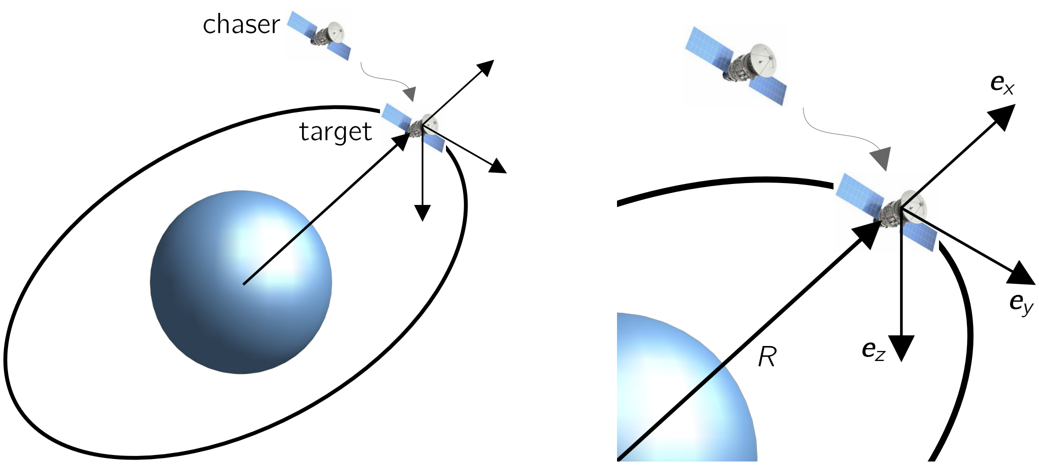

Figure 1: Chaser and target satellite in moving CWH frame.

The model is parametrized by the mass $m := 300\text{kg}$ of the chaser and the *orbital rate* $\omega_n := \sqrt{\frac{\mu}{R^3}}$, where $\mu := 3.986 \cdot 10^{14} \frac{\text{m}^3}{\text{s}^2}$ is the standard gravitational parameter and $R = 7 \cdot 10^6\text{m}$ is the radius of the target orbit.

Constraints

Let

$$\boldsymbol{u}(t) := \begin{bmatrix} u_x(t) & u_y(t) & u_z(t) \end{bmatrix}^\top \in \mathbb{R}^{n_u}. \tag{2}$$

The maximum absolute thrust that can be provided by the satellite in each component is limited, i.e.,

$$\|\boldsymbol{u}(t)\|_\infty := \max\{|u_x(t)|, |u_y(t)|, |u_z(t)|\} \leq u_{\max}, \tag{3}$$

where $u_{\max} := 1\text{N}$. This set of polytopic constraints can be conveniently written as

$$H_u \boldsymbol{u}(t) \leq \boldsymbol{h}_u, \tag{4}$$

for some $H_u \in \mathbb{R}^{6 \times n_u}$ and $\boldsymbol{h}_u \in \mathbb{R}^6$.

Mission objective

The satellite is ejected from the carrier rocket at time $T_0 := 0\text{s}$. We will investigate the approach of the satellite from three different initial conditions which will be defined later. The controller employed for the approaching stage should adhere to the following requirements:

- The fuel consumption should be minimized. As a surrogate for the fuel consumption in the time interval $[0, T]$, we use the $\mathcal{L}_2$ norm; for a piecewise constant input function $\boldsymbol{u}$, with $\boldsymbol{u}(\tau) = \boldsymbol{u}(k\Delta T)$ on the interval $\tau \in [k\Delta T, (k+1)\Delta T)$ for all $k = 0, \dots, N_t - 1$ and $\Delta T \cdot N_t = T$, this results in

$$\|\boldsymbol{u}\|_{\mathcal{L}_2(0,T)} = \Delta T \left(\sum_{k=0}^{N_t - 1} \boldsymbol{u}(k\Delta T)^\top \boldsymbol{u}(k\Delta T) \right)^{\frac{1}{2}}.$$

- The satellite should stay within its track limits to avoid collision risk and ensure the validity of the linearized model. The state constraints are given as

$$-s_{\max} \leq x(t) \leq s_{\max} \qquad -y_{\max} \leq y(t) \leq y_{\max} \qquad -s_{\max} \leq z(t) \leq s_{\max},$$

  where $s_{\max} := 10^5\text{m}$ confines the chaser perpendicular to the target's orbital velocity and $y_{\max} := 10^6\text{m}$ confines it along the orbital track. Analogously to the input constraints, the 6 constraints above can be compactly expressed as

$$H_x \boldsymbol{x}(t) \leq \boldsymbol{h}_x \tag{5}$$

  for some $H_x \in \mathbb{R}^{6 \times n_x}$, $\boldsymbol{h}_x \in \mathbb{R}^6$ where

$$\boldsymbol{x}(t) := \begin{bmatrix} x(t) & y(t) & z(t) & \dot{x}(t) & \dot{y}(t) & \dot{z}(t) \end{bmatrix}^\top \in \mathbb{R}^{n_x}. \tag{6}$$

- At final time $T_f := 172800\text{s}$ (2 days), the distance and absolute velocity difference of the chaser should not exceed $d_{f,\max} := 100\text{m}$ and $v_{f,\max} := 1\text{m/s}$, respectively, i.e.,

$$d_f := \sqrt{x(T_f)^2 + y(T_f)^2 + z(T_f)^2} \leq d_{f,\max}, \tag{7}$$
$$v_f := \sqrt{\dot{x}(T_f)^2 + \dot{y}(T_f)^2 + \dot{z}(T_f)^2} \leq v_{f,\max}. \tag{8}$$

Preliminaries

MATLAB

For this project, you need to have MATLAB R2020b or newer installed. You can download the latest version from the ETH IT Shop[1]. Furthermore, the Control Systems Toolbox and Optimization Toolbox need to be installed.

Installation of MPT & Yalmip

We rely on the Multi-Parametric Toolbox (MPT) for set computations and Yalmip[2] to setup MPC controllers in Matlab.

To install MPT, complete the following instructions:

1. Go to `https://www.mpt3.org/` and click on "Installation & updating instructions".
2. Download the file `install_mpt3.m`.
3. Run `install_mpt3.m` in MATLAB.

MPT automatically installs Yalmip.

Provided Files

The provided files of this project are structured in two subdirectories: `templates/` and `testing/`. Make sure to add the whole provided folder including all files and subdirectories to the **MATLAB path**.

The subdirectory `templates/` contains template files that serve as a basis for your implementation. Do **not** change the input-output structure of the provided functions when implementing your solution. While most of the files are empty, below you find a more detailed description of a few important files.

- `generate_params.m` (Function) — This function returns a struct (called `params` in the following) containing parameters that are shared across most of the functions you implement as part of this project. The fields of the `params` struct and their corresponding parameters are detailed in Table 1.
- `plot_trajectory.m` (Function) — This function plots the state trajectories and constraints of the satellite model. The inputs are given as state and input trajectories X_t, U_t, the boolean array F_t as defined in Task 7, as well as `params`.
- `plot_trajectory_z.m` (Function) — This function plots the state trajectories and constraints of the decoupled z-subsystem of the satellite model. The inputs are as above, with the redefined states and inputs from Task 26.
- You are provided with template files for the deliverables to be implemented by you. Below each task described in the following, you will find a table that summarizes the corresponding deliverables. More information about the functions can be found in the templates and their respective task description.

[1] `https://itshop.ethz.ch/`
[2] `https://yalmip.github.io/`

Field	Value
model.Mass	m
model.GravitationalParameter	μ
model.TargetRadius	R
model.nx	n_x
model.nu	n_u
model.A	A
model.B	B
model.InitialConditionA	$\boldsymbol{x}_0^A$
model.InitialConditionB	$\boldsymbol{x}_0^B$
model.InitialConditionC	$\boldsymbol{x}_0^C$
model.HorizonLength	N_t
model.ScalingMatrix	V
model.TimeStep	ΔT
constraints.InputMatrix	H_u
constraints.InputRHS	$\boldsymbol{h}_u$
constraints.MaxAbsPositionXZ	s_{max}
constraints.MaxAbsPositionY	y_{max}
constraints.MaxAbsThrust	u_{max}
constraints.MaxFinalPosDiff	$d_{\mathrm{f,max}}$
constraints.MaxFinalVelDiff	$v_{\mathrm{f,max}}$
constraints.StateMatrix	H_x
constraints.StateRHS	$\boldsymbol{h}_x$

Table 1: `params` struct of system parameters returned by the function `generate_params`.

The folder `testing/` contains a number of encrypted MATLAB p-files to test your solution, as well as the `run_tests` function. Usage of the testing framework is described in more detail in the following.

Testing framework

In this programming exercise we make use of a testing framework. The framework serves two main purposes: to provide you with feedback about your solution during development, and to automatically grade your submission.

To facilitate testing of your submission, all deliverables are given in terms of *functions* whose input-output behavior is specified in the task description and verified by the testing framework.

To run the tests for all functions, run

```
test_struct = run_tests();
```

The testing framework is available for all submitted functions. We do not provide testing functions for tasks where you have to write a script, i.e., Tasks 11, 23 and 31. To run the tests for one specific function called `"function_name"` (`"function_name"` is a string, e.g., `"generate_system_cont"`), run

```
test_struct = run_tests("function_name");
```

The output `test_struct` is a MATLAB struct whose fields are described in Table 2. Note that the testing framework may vary **all** of the inputs of the function to be tested, so it is important to **avoid hard-coding parameters** such as input and state dimensions etc. As an emphasizing remark, always use the parameters available in the `params` struct in your implementation and do not or recompute or hard-code them.

Field	Value
`Deliverable`	The name of the tested function.
`Info`	Short summary of the diagnosis. Passed tests are labeled `"OK"`. If an error is caught during the execution of your function, it will display `"ERROR: <message>"`. If the output of your function does not match the expected value, it shows `"Failure: <message>"`.
`TestResults`	An array of MATLAB `TestResult` objects, which you can explore for debugging.

Table 2: `test_struct` struct of test results returned by the function `run_tests`.

As this is the first iteration of the automated testing framework, it might be necessary to update the testing framework while you are working on the project. In this regard, we highly appreciate your feedback and would like to encourage you to report bugs or unexpected behavior of the software at the dedicated forum section "Programming Exercise - Questions" on Moodle. In any case, the version of the testing framework used for grading will not be changed after the **12.05.22**. Afterwards, if the testing framework flags parts of your submission as incorrect and you would like to insist on the correctness of your solution, you may write **max. 2 pages** report documenting your code; that part of your submission will then be checked manually.

What you have to hand in

Up to three students are allowed to work together on the programming exercise. They will all receive the same grade. As a group, you must read and understand the ETH plagiarism policy here: `http://www.plagiarism.ethz.ch/` - each submitted work will be tested for plagiarism. You must download and fill out the Declaration of Originality, available at the same link. You are not allowed to make this project description and template publicly available.

Hand in a single zip-file, where the filename contains the names of all team-members according to this template (note that there are no spaces in the filename):

`MPC22PE_Firstname1Surname1_Firstname2Surname2_Firstname3Surname3.zip`.

The zip-file must contain the following files according to the exercises:

- Files corresponding to the deliverables summarized at the end of each tasks section. You will be notified of missing MATLAB files when running the testing framework (see explanation above).
- A scan of the Declaration of Originality, signed by all team-members.
- If applicable, your **optional** 2-page report (see the description of the testing framework above.)

All files should be placed at the highest level of the zip-folder and **no** other files or sub-folders should be included in the zip-folder. The zip-file should be uploaded to Moodle in the Programming Exercise assignment area by just one of the members of the group. The deadline for submission is **19.05.22, 23:59 h**. Late submissions will not be considered.

Simulation and self-study questions

Some of the questions are marked as simulation or self-study questions. While these types of questions are ungraded, they are intended to guide your learning experience and test your broader understanding beyond the pure implementation of the methods. Simulation methods encourage you to test the developed methods in a setting where the differences between different control methods become apparent. Self-study questions are of a more theoretic and experimental nature; they serve as additional material to prepare you for the exam. We thus strongly recommend to answer and discuss these questions in your group.

Tasks

In the following, you find the tasks and corresponding deliverables for this project.

System Modeling

Tasks

1. Rewrite the second-order ODE (1) as a first-order ODE of the form

$$\dot{\boldsymbol{x}}(t) = A_c \boldsymbol{x}(t) + B_c \boldsymbol{u}(t),$$

   where $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ are defined as in equations (2) and (6). Implement a function called `generate_system_cont` that takes the `params` struct as input and returns A_c, B_c. *2 pt.*

2. Discretize the continuous-time ODE using an exact discretization with a sampling time of $\Delta T = 600$s such that the resulting discrete-time dynamics are given by the difference equation

$$\tilde{\boldsymbol{x}}_d(k+1) = \tilde{A}\tilde{\boldsymbol{x}}_d(k) + \tilde{B}\boldsymbol{u}_d(k), \tag{9}$$

   where $\tilde{\boldsymbol{x}}_d(k) := \tilde{\boldsymbol{x}}(\Delta T \cdot k)$ and $\boldsymbol{u}_d(k) := \boldsymbol{u}(\Delta T \cdot k)$. Implement your solution as a function `generate_system`, that computes $\tilde{A}, \tilde{B}$ based on the continuous-time matrices A_c, B_c and the `params` struct as inputs.
   Hint: You can use the MATLAB function `c2d`. *2 pt.*

3. For the problem to be numerically well-conditioned, the states have to be scaled with the following transformation

$$\boldsymbol{x}_d(k) = V\tilde{\boldsymbol{x}}_d(k) \tag{10}$$

   where $V = \text{diag}([10^{-6}, 10^{-6}, 10^{-6}, 10^{-3}, 10^{-3}, 10^{-3}]) =$ `params.model.ScalingMatrix` is a diagonal scaling matrix. Implement a function `generate_system_scaled`, which takes as inputs the matrices $\tilde{A}, \tilde{B},$ `params` and outputs A, B such that

$$\boldsymbol{x}_d(k+1) = A\boldsymbol{x}_d(k) + B\boldsymbol{u}_d(k) \tag{11}$$

   describes the same dynamics as equation (9). Note that this transformation amounts to changing the units of the state from [m; m/s] to [Mm; km/s]. All further exercises are expressed in this new transformed units. For brevity, and with a slight abuse of notation, we drop the subscript in the following and write $\boldsymbol{x}(k)$ and $\boldsymbol{u}(k)$ to refer to the discrete state and input variables, respectively. *2 pt.*

4. Find matrices $H_u, H_x \in \mathbb{R}^{6\times n_x}$, as well as vectors $\boldsymbol{h}_u, \boldsymbol{h}_x \in \mathbb{R}^6$ to express the state and input constraints in the form of equations (4) and (5), respectively (in the transformed coordinates corresponding to equation (10)). Implement a function `generate_constraints`, that takes as input the parameter struct `params` and outputs $H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$. *2 pt.*

5. Modify the function `generate_params` to also compute $A, B, H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$ as part of its execution (using the functions you implemented above) and add the corresponding fields as defined in Table 1 to the output `struct`. The output of the function `generate_params` should now exactly give a `struct` as specified in Table 1. *1 pt.*

Task	Function	Inputs	Outputs	Pt.
1	`generate_system_cont`	`params`	A_c, B_c	2
2	`generate_system`	$A_c, B_c,$ `params`	$\tilde{A}, \tilde{B}$	2
3	`generate_system_scaled`	$\tilde{A}, \tilde{B},$ `params`	A, B	2
4	`generate_constraints`	`params`	$H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$	2
5	`generate_params` (modify)		`params`	1

Table 3: Deliverable summary "System modeling"

Unconstrained Optimal Control

The aim of the following tasks is to design a discrete-time infinite-horizon linear quadratic regulator (LQR) that satisfies the mission requirements. The infinite-horizon LQR controller

$$\boldsymbol{u}(k) := F_\infty \boldsymbol{x}(k) \tag{12}$$

is defined such that it minimizes the infinite-horizon quadratic cost

$$J_\infty(\boldsymbol{x}(0)) := \sum_{k=0}^{\infty} \boldsymbol{x}(k)^\top Q \boldsymbol{x}(k) + \boldsymbol{u}(k)^\top R \boldsymbol{u}(k), \tag{13}$$

for some positive definite weighting matrices $Q \in \mathbb{R}^{n_x \times n_x}$ and $R \in \mathbb{R}^{n_u \times n_u}$. As the mission requirements and constraints cannot be encoded directly, but rather have to follow from the choice of the weights, the main difficulty is to find Q and R that lead to a satisfactory system response. Therefore a parameter study is to be conducted. To simplify the parameter study, we define $\boldsymbol{q} \in \mathbb{R}^{n_x \times 1}$, $\boldsymbol{q} := \begin{bmatrix} q_x & q_y & q_z & q_{vx} & q_{vy} & q_{vz} \end{bmatrix}^\top$ and choose[3] the parametrization $Q := \text{diag}(\boldsymbol{q})$ and $R = I_{n_u}$. The parameter study is split into a set of subproblems that form the deliverables for this task.

Tasks

6. Design an infinite-horizon LQR controller for given inputs Q and R. Implement your solution as a *class* in the template file `LQR.m`. Note that the class template has two functions, the constructor `LQR(Q,R)` and `eval(x)`. For numerical efficiency, the feedback matrix F_∞ should only be computed at initialization once (in the constructor of the class) and stored as a class property `K` $:= L_\infty$. The class property can then be accessed at every call to the `eval` function, which computes the feedback of the controller according to equation (12) without additional computational overhead. Note that, in addition to the control action $\boldsymbol{u}$, the `eval` function also returns a struct `ctrl_info` containing additional information about the control output. For the LQR controller, the `ctrl_info` struct has only one field, `ctrl_feas`, indicating the feasibility of the control problem; it suffices to always set

   `ctrl_info.ctrl_feas` $=$ `true`.

   For convenience, we have already added the `eval` function to the template `LQR.m`. More details can be found in the template file. *2 pt.*

7. Simulate the closed-loop system for a given initial condition $\boldsymbol{x}(0)$, controller object `ctrl` and number of time steps N_t. Implement your solution in the function `simulate`, which

[3] Note that the selection of $R = I_{n_u}$ is not restrictive since scaling the infinite-horizon cost does not change its minimizer. The restriction here is mainly introduced by assuming diagonal weight terms, i.e., no coupling of the different states or inputs in the cost.

takes $\boldsymbol{x}(0)$, `ctrl`, `params` as inputs and outputs the closed-loop trajectory according to equation (11) in terms of the matrices $X_t \in \mathbb{R}^{n_x \times (N_t+1)}$, $U_t \in \mathbb{R}^{n_u \times N_t}$, with

$$X_t := \begin{bmatrix} \boldsymbol{x}(0) & \dots & \boldsymbol{x}(N_t) \end{bmatrix}, \qquad U_t := \begin{bmatrix} \boldsymbol{u}(0) & \dots & \boldsymbol{u}(N_t - 1) \end{bmatrix}, \tag{14}$$

as well as an N_t-dimensional array of the `ctrl_info` structs described in Task 6. Note that the simulation should explicitly **not** include saturation of the inputs, i.e., compute the evolution of the linear system according to the raw input of the feedback controller. 2 *pt.*

8. *[Simulation]: Experiment with different values of the parameters in* $\boldsymbol{q}$*. What is their effect on the resulting closed-loop trajectory with initial condition* $\boldsymbol{x}_0^A$ *(see Table 1)? You can use the provided function* `plot_trajectory` *to visualize your results.*

9. Check the satisfaction of the constraints for a given trajectory. Implement a function called `traj_constraints`, that takes as input the trajectory data X_t, U_t and computes as outputs
   - the maximum absolute value of both $x(k)$ and $z(k)$, $s_{\text{max}}^{(i)} := \max_{k \in [0, N_t]} \max\{|x(k)|, |z(k)|\}$,
   - the maximum absolute value of y, $y_{\text{max}}^{(i)} := \max_{k \in [0, N_t]} |y(k)|$,
   - the maximum absolute value of the applied thrust, $u_{\text{max}}^{(i)} := \max_{k \in [0, N_t - 1]} \|\boldsymbol{u}(k)\|_\infty$,
   - the closed-loop finite horizon input cost, $J_u^{(i)} := \sum_{k=0}^{N_t - 1} \boldsymbol{u}(k)^\top \boldsymbol{u}(k)$,
   - the distance from the target position at T_{f}, see equation (7),
   - the absolute difference from the target velocity, see equation (8),
   - a boolean flag $\texttt{traj\_feas}^{(i)}$ indicating the feasibility of the trajectory, i.e., $\texttt{traj\_feas}^{(i)} =$ true if and only if

$$s_{\text{max}}^{(i)} \leq s_{\text{max}}, \quad y_{\text{max}}^{(i)} \leq y_{\text{max}}, \quad u_{\text{max}}^{(i)} \leq u_{\text{max}}, \quad d_{\text{f}}^{(i)} \leq d_{\text{f,max}}, \quad v_{\text{f}}^{(i)} \leq v_{\text{f,max}}. \tag{15}$$

3 *pt.*

10. Perform a parameter study to find a good choice for $\boldsymbol{q}$. Implement a function `lqr_tuning`, which takes as input an initial state $\boldsymbol{x}(0) \in \mathbb{R}^{n_x}$, as well as an array of parameter vectors $\mathcal{Q} \in \mathbb{R}^{n_x \times M}$, $\mathcal{Q} := \begin{bmatrix} \boldsymbol{q}^{(1)} & \dots & \boldsymbol{q}^{(M)} \end{bmatrix}$, and outputs an M-dimensional array of structs `tuning_struct`. Each element `tuning_struct(`i`)` should contain the following fields:

Field	Value
`InitialCondition`	$\boldsymbol{x}_0$
`Qdiag`	$\boldsymbol{q}^{(i)}$
`MaxAbsPositionXZ`	$s_{\text{max}}^{(i)}$
`MaxAbsPositionY`	$y_{\text{max}}^{(i)}$
`MaxAbsThrust`	$u_{\text{max}}^{(i)}$
`InputCost`	$J_u^{(i)}$
`MaxFinalPosDiff`	$d_{\text{f}}^{(i)}$
`MaxFinalVelDiff`	$v_{\text{f}}^{(i)}$
`TrajFeasible`	$\texttt{traj\_feas}^{(i)}$

Table 4: Fields of `tuning_struct(`i`)`

The second output argument of the function shall be the index $i_{\text{opt}} \in \{1, \dots M\}$, which corresponds to the index of the LQR controller that is feasible and requires the lowest amount of fuel, i.e., $i_{\text{opt}} := \arg\min_i \left\{ J_u^{(i)} \,\middle|\, \texttt{traj\_feas}^{(i)} = \texttt{true} \right\}$. If there exists no feasible LQR controller, the function should set $i_{\text{opt}} := \texttt{nan}$. 3 *pt.*

11. Use the function `lqr_tuning` you implemented above to identify a parameter vector $\boldsymbol{q}$, such that the corresponding LQR controller is feasible and has a low input cost $J_u \leq 11$ for the initial condition $\boldsymbol{x}_0^A$. Provide a script file `lqr_tuning_script.m` which performs the parameter study and finally sets the parameter q $:= \boldsymbol{q}$ corresponding to the best controller parametrization. Save the variable q, as well as the `tuning_struct` of your parameter study in the MAT-file `lqr_tuning_script.mat`.
Hint 1: The optimal parameters may span several orders of magnitude. Consider the MATLAB functions `logspace` and `ndgrid` to get efficient parameter samples for the input matrix $\mathcal{Q}$. You might have to perform several rounds of tuning and refine your sampling grid iteratively to arrive at a satisfactory solution.
Hint 2: Note that the "in-plane" dynamics of the x, y-coordinates are decoupled from the "out-of-plane" dynamics of the z-coordinate. This means you can perform the tuning with respect to the parameters q_x, q_y, q_{vx}, q_{vy} separately from the tuning for the parameters q_z, q_{vz}. *2 pt.*

12. *[Simulation]: Simulate the closed-loop system with the LQR controller obtained in Deliverable 11, starting from the initial condition $\boldsymbol{x}_0^B$. Is your controller still feasible?*

Task	Function	Inputs	Outputs	Pt.
6	`LQR`	Q, R, `params`	`ctrl` (`LQR` object)	2
6	`LQR/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
7	`simulate`	$\boldsymbol{x}_0$, `ctrl`, `params`	X_t, U_t, F_t	2
9	`traj_constraints`	X_t, U_t, `params`	$s_{\max}^{(i)}$, $y_{\max}^{(i)}$, $u_{\max}^{(i)}$, $J_u^{(i)}$, $d_{\mathrm{f}}^{(i)}$, $v_{\mathrm{f}}^{(i)}$, `traj_feas`$^{(i)}$	3
10	`lqr_tuning`	$\boldsymbol{x}_0$, $\mathcal{Q}$, `params`	`tuning_struct`	3
11	`lqr_tuning_script`	-	`lqr_tuning_script.mat`	2

Table 5: Deliverable summary "Unconstrained Optimal Control"

From LQR to MPC

After designing the unconstrained optimal controller, in this section you will design a first simple model predictive controller. In MPC, the control sequence $U := \{\boldsymbol{u}_0, \dots \boldsymbol{u}_{N-1}\}$ is computed as the solution of an optimization problem over a prediction horizon of N steps. Feedback is introduced by only applying the first element of the sequence, $\boldsymbol{u}(0) := \boldsymbol{u}_0$; in the next time step, the optimal input sequence is recomputed based on updated state measurements. Note that we write predicted control inputs with a lower subscript, e.g., $\boldsymbol{u}_0$, to differentiate them from the implemented control inputs $u(0)$. The notation of the states is chosen analogously.

For the following exercises, we use $Q^* := \mathrm{diag}(\boldsymbol{q}^*)$ and $R^* := I_{n_u}$, with

$$\boldsymbol{q}^* := \begin{bmatrix} q_x^* \\ q_y^* \\ q_z^* \\ q_{vx}^* \\ q_{vy}^* \\ q_{vz}^* \end{bmatrix} := \begin{bmatrix} 91.5 \\ 0.0924 \\ 248 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Nevertheless, feel free to also experiment with your best values obtained from the LQR controller tuning and see how it affects the MPC closed-loop performance.

Tasks

13. Explicitly compute the maximum positively invariant set $\mathcal{X}_{LQR}$ under application of the LQR controller, i.e., the set of initial conditions for which the closed-loop system under the LQR controller satisfies state and input constraints for all times. Let $\boldsymbol{x}_{LQR}(k)$ and $\boldsymbol{u}_{LQR}(k)$, $k = 0, \dots, \infty$ be the infinite-horizon closed-loop state and input sequence resulting from application of the LQR controller (12) to system (11), for some initial condition $\boldsymbol{x}_{LQR}(0) = \boldsymbol{x}(0)$. Then, the set $\mathcal{X}_{LQR}$ is defined by

$$\mathcal{X}_{LQR} := \{\boldsymbol{x} \mid \boldsymbol{x}_{LQR}(0) = \boldsymbol{x},\ H_x \boldsymbol{x}_{LQR}(k) \leq \boldsymbol{h}_x,\ H_u \boldsymbol{u}_{LQR}(k) \leq \boldsymbol{h}_u \text{ for all } k \geq 0\}. \tag{16a}$$

Implement a function `lqr_maxPI`, which takes as inputs $Q, R,$ `params` and computes $\mathcal{X}_{LQR}$ in terms of polytopic constraints, i.e., returns $H \in \mathbb{R}^{n_H \times n_x}$ and $\boldsymbol{h} \in \mathbb{R}^{n_H}$ such that $\mathcal{X}_{LQR} = \{\boldsymbol{x} \mid H\boldsymbol{x} \leq \boldsymbol{h}\}$.
Hint: You can use the MPT toolbox for this task. 3 *pt.*

14. *[Self-study]: Check whether $\boldsymbol{x}_0^A$, $\boldsymbol{x}_0^B$, and $\boldsymbol{x}_0^C$ are contained in $\mathcal{X}_{LQR}$. What can you conclude from the result with respect to state and input constraint satisfaction under the LQR controller for these initial conditions?*

15. Implement a function `traj_cost` that takes as input arguments $Q \in \mathbb{R}^{n_x \times n_x}$, $R \in \mathbb{R}^{n_u \times n_u}$, as well as trajectory data X_t, U_t as in equation (14) and outputs the closed-loop quadratic cost for a given trajectory, i.e.,

$$J_{N_t} := \sum_{k=0}^{N_t - 1} \boldsymbol{x}(k)^\top Q \boldsymbol{x}(k) + \boldsymbol{u}(k)^\top R \boldsymbol{u}(k)$$

1 *pt.*

16. Implement a model predictive controller that solves the open-loop optimization problem

$$\min_{U} \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i + l_{\mathrm{f}}(\boldsymbol{x}_N) \tag{17a}$$

$$\text{s.t.} \quad \boldsymbol{x}_0 = \boldsymbol{x}(k) \tag{17b}$$

$$\boldsymbol{x}_{i+1} = A\boldsymbol{x}_i + B\boldsymbol{u}_i, \qquad i = 0, \ldots, N-1 \tag{17c}$$

$$H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x, \qquad i = 0, \ldots, N \tag{17d}$$

$$H_u \boldsymbol{u}_i \leq \boldsymbol{h}_u, \qquad i = 0, \ldots, N-1 \tag{17e}$$

at each time step, where $l_{\mathrm{f}}(\boldsymbol{x}) = J_\infty(\boldsymbol{x})$ is equal to the LQR infinite-horizon cost (13). As for the LQR controller, implement the model predictive controller as a class `MPC`, which creates a YALMIP solver object during initialization with Q, R, N, and solves the optimization problem in the `eval` method. Note that in this case, at each call of `eval` the `ctrl_info` struct has two additional fields: `ctrl_info.objective`, which should contain the value of the objective function for the optimizer U^* of (17), and `ctrl_info.solvetime`, which should contain the time required to solve the problem (17). Again, we have provided you with the full `eval` method. More details can be found in the template file.
Hint 1: The LQR infinite-horizon cost is of the form $J_\infty(\boldsymbol{x}) = \boldsymbol{x}^\top P \boldsymbol{x}$, where $P \in \mathbb{R}^{n_x \times n_x}$ is a constant matrix that can be computed during the initialization of the controller.
Hint 2: You can use the MATLAB functions `tic` and `toc` to get an estimate of the required solve time. 5 *pt.*

17. *[Simulation]: Simulate the closed-loop system starting from $\boldsymbol{x}_0^A$ with the LQR and the model predictive controller for the same choice of $Q := Q^*$, $R := R^*$ and with $N := 30$. Do the same with $\boldsymbol{x}_0^B$ as initial condition. How do the controllers perform with respect to closed-loop constraint satisfaction and closed-loop cost?*

Task	Function	Inputs	Outputs	Pt.
13	`lqr_maxPI`	Q, R, `params`	H, h	3
15	`traj_cost`	X_t, U_t, Q, R	J_N	1
16	`MPC`	Q, R, N, `params`	`ctrl` (`MPC` object)	5
16	`MPC/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0

Table 6: Deliverable summary "From LQR to MPC"

MPC with theoretical closed-loop guarantees

In this part, the task is to formulate a model predictive controller that provides guaranteed closed-loop state and input constraint satisfaction and renders the origin an asymptotically stable equilibrium point of the closed-loop system.

Tasks

18. Implement a model predictive controller based on the MPC problem

$$
\begin{aligned}
\min_{U} \; & \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i && && \text{(18a)} \\
\text{s.t.} \; & \boldsymbol{x}_0 = \boldsymbol{x}(k) && && \text{(18b)} \\
& \boldsymbol{x}_{i+1} = A\boldsymbol{x}_i + B\boldsymbol{u}_i, && i = 0, \ldots, N-1 && \text{(18c)} \\
& H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x, && i = 0, \ldots, N && \text{(18d)} \\
& H_u \boldsymbol{u}_i \leq \boldsymbol{h}_u, && i = 0, \ldots, N-1 && \text{(18e)} \\
& \boldsymbol{x}_N = 0, && && \text{(18f)}
\end{aligned}
$$

in the class template `MPC_TE`, with the same input arguments as the `MPC` class. 2 *pt.*

19. *[Self-study]: Why is the origin an asymptotically stable equilibrium point for the resulting closed-loop system, given that* (18) *is feasible for* $\boldsymbol{x}(0)$?

20. Implement another model predictive controller based on the MPC problem

$$
\begin{aligned}
\min_{U} \; & \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i + l_{\mathrm{f}}(\boldsymbol{x}_N) && && \text{(19a)} \\
\text{s.t.} \; & \boldsymbol{x}_0 = \boldsymbol{x}(k) && && \text{(19b)} \\
& \boldsymbol{x}_{i+1} = A\boldsymbol{x}_i + B\boldsymbol{u}_i, && i = 0, \ldots, N-1 && \text{(19c)} \\
& H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x, && i = 0, \ldots, N && \text{(19d)} \\
& H_u \boldsymbol{u}_i \leq \boldsymbol{h}_u, && i = 0, \ldots, N-1 && \text{(19e)} \\
& \boldsymbol{x}_N \in \mathcal{X}_{LQR}, && && \text{(19f)}
\end{aligned}
$$

in the class template `MPC_TS`, where $l_{\mathrm{f}}(\boldsymbol{x}) := J_\infty(\boldsymbol{x})$ is the LQR infinite-horizon cost and $\mathcal{X}_{LQR}$ from equation (16) is given in terms of H and $\boldsymbol{h}$ as outlined in Deliverable 13. Note that the class is initialized with the additional input arguments H and $\boldsymbol{h}$, hence $\mathcal{X}_{LQR}$ has to be computed outside of the class. 2 *pt.*

21. *[Simulation]: Simulate the closed-loop system with the three MPCs* (17), (18), (19) *starting from* $\boldsymbol{x}_0^A$ *for the same choice of* $Q := Q^*$, $R := R^*$ *and horizon length* $N = 30$ *and compare them in terms of the feasibility of the open-loop optimization problems, as well as their constraint satisfaction and cost in closed-loop. Test also different horizon lengths* $N = 40 \ldots 20$. *What do you observe?*

Task	Function	Inputs	Outputs	Pt.
18	MPC_TE	Q, R, N, params	ctrl (MPC_TE object)	2
18	MPC_TE/eval	$\boldsymbol{x}$	$\boldsymbol{u}$, ctrl_info	0
20	MPC_TS	Q, R, N, H, $\boldsymbol{h}$, params	ctrl (MPC_TS object)	2
20	MPC_TS/eval	$\boldsymbol{x}$	$\boldsymbol{u}$, ctrl_info	0

Table 7: Deliverable summary "MPC with theoretical closed-loop guarantees"

Soft constraints

In practical implementations, model predictive controllers such as (19) can become infeasible despite the provided theoretical guarantees, for instance due to unmodeled disturbances or model mismatch. This problem can be addressed by using soft constraints, providing a recovery mechanism given that the original problem is infeasible. Your task is to design a soft-constrained model predictive controller, which provides the same control inputs as (19), if (19) is feasible, but may provide a feasible solution even when (19) is infeasible.

Tasks

22. Introduce slack variables $\boldsymbol{\epsilon}_i \in \mathbb{R}^6$, $i = 0, \dots, N$, into the MPC problem (19) to restore feasibility in the case of state constraint violations. Implement a model predictive controller based on the MPC problem

$$\min_U \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i + l_f(\boldsymbol{x}_N) + \sum_{i=0}^{N} \boldsymbol{\epsilon}_i^\top S \boldsymbol{\epsilon}_i + v\|\boldsymbol{\epsilon}_i\|_\infty \tag{20a}$$

$$\text{s.t.} \quad \boldsymbol{x}_0 = \boldsymbol{x}(k) \tag{20b}$$

$$\boldsymbol{x}_{i+1} = A\boldsymbol{x}_i + B\boldsymbol{u}_i, \qquad i = 0, \dots, N-1 \tag{20c}$$

$$H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x + \boldsymbol{\epsilon}_i, \qquad i = 0, \dots, N \tag{20d}$$

$$H_u \boldsymbol{u}_i \leq \boldsymbol{h}_u \qquad i = 0, \dots, N-1 \tag{20e}$$

$$\boldsymbol{\epsilon}_i \geq 0, \qquad i = 0, \dots, N \tag{20f}$$

$$\boldsymbol{x}_N \in \mathcal{X}_{LQR} \tag{20g}$$

in the class MPC_TS_SC. The class shall be initialized with the same parameters as MPC_TS, plus the additional values for $S \in \mathbb{R}^{6\times 6}$ and $v \in \mathbb{R}$ for the constraint violation penalty with $v \gg 0$. 4 *pt.*

23. Choose S and v such that the controller based on the soft-constrained MPC problem (20) returns the same control input values as the controller based on the MPC problem (19), whenever (19) is feasible, for the same choice of weighting matrices Q^*, R^* and horizon length $N = 30$. Verify your selection in simulation by writing a script called MPC_TS_SC_script.m. Provide the script and save your selected values as variables S := S and v := v in the MAT-file MPC_TS_SC_params.mat. 2 *pt.*

24. *[Simulation]: Repeat the simulation for the initial condition* $\boldsymbol{x}_0^C$ *and experiment with different values for* S *and* v *in the soft-constrained formulation* (20). *How does the choice of* S *and* v *influence the closed-loop behavior?*

Task	Function	Inputs	Outputs	Pt.
22	MPC_TS_SC	Q, R, N, H, $\boldsymbol{h}$, S, v, params	ctrl (MPC_TS_SC object)	4
22	MPC_TS/eval	$\boldsymbol{x}$	$\boldsymbol{u}$, ctrl_info	0
23	MPC_TS_SC_script.mat	-	MPC_TS_SC_params.mat	2

Table 8: Deliverable summary "Soft constraints"

Robust MPC

If we want to ensure satisfaction of constraints even under uncertain model descriptions, such as model mismatch or unmodeled disturbances, we can use a robust MPC approach. In the given robust MPC task, we just consider the (decoupled) subsystem in z-direction of the overall system (11) after applying the transform (10). Consequently, we consider the following uncertain system

$$\boldsymbol{x}_z(k+1) = A_z\boldsymbol{x}_z(k) + B_z\boldsymbol{u}_z(k) + \boldsymbol{w}(k), \tag{21}$$

where $\boldsymbol{x}_z(k) = \begin{bmatrix} z(k) & v_z(k) \end{bmatrix}^\top \in \mathbb{R}^2$, $\boldsymbol{u}_z(k) \in \mathbb{R}$, and the disturbance $\boldsymbol{w}(k) \in \mathbb{R}^2$ models the uncertainty in the system. The disturbances are constrained to $H_w\boldsymbol{w}(k) \le \boldsymbol{h}_w$ for all time steps k with

$$H_w := \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}, \qquad \boldsymbol{h}_w := w_{\max} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \tag{22}$$

where $w_{\max} := 10^{-4}$.

We provided you with the function generate_params_z which takes as an input your params struct and provides the struct params_z, in which A, B, n_x, n_u, H_x, $\boldsymbol{h}_x$, H_u, and $\boldsymbol{h}_u$ are adapted to represent the uncertain subsystem in z-direction (21). To keep notation simple and for your reference in the implementation, we redefine $n_x := 2$, $n_u := 1$; for the updated values of the other variables, please refer to the provided function generate_params_z. Note that the redefined variables should not affect your implementation, which should work for either set of variables (as long as the dimensions match). The initial conditions $\boldsymbol{x}_0^A$, $\boldsymbol{x}_0^B$, and $\boldsymbol{x}_0^C$ are removed, and the initial condition $\boldsymbol{x}_{z,0}^A$ is added. Additionally, the function generate_params_z adds the fields shown in Table 9 to the params_z struct to represent the constraints on the disturbances (22).

Field	Value
model.InitialConditionA_z	$\boldsymbol{x}_{z,0}^A$
constraints.DisturbanceMatrix	H_w
constraints.DisturbanceRHS	$\boldsymbol{h}_w$
constraints.MaxAbsDisturbance	$w_{\max}$

Table 9: Fields added to params_z struct by the function generate_params_z.

Tasks

25. Sample a sequence W_t of N_t disturbance vectors $\boldsymbol{w}(k)$, i.e.,

$$W_t := \begin{bmatrix} \boldsymbol{w}(0) & \dots & \boldsymbol{w}(N_t - 1) \end{bmatrix}.$$

Thereby, each disturbance vector $\boldsymbol{w}(k)$, $k = 0, \dots, N_t - 1$, should be sampled from a uniform probability distribution defined on the polytope $\{\, \boldsymbol{w} \in \mathbb{R}^{n_x} \mid H_w \boldsymbol{w} \leq \boldsymbol{h}_w \}$. Implement a function `generate_disturbances` that takes `params_z` as input and outputs W_t. 1 *pt.*

26. Simulate the closed-loop system (21) in the presence of additive disturbances. Adapt the simulation function developed in Task 7 to implement a function `simulate_uncertain`, which takes as inputs $\boldsymbol{x}_0$, `ctrl`, W_t, `params_z`, and outputs X_t, U_t, F_t as in Task 7, but for the uncertain system model (21). 1 *pt.*

The following tasks lead you through the design of a tube-based robust MPC controller[4] of the form

$$\min_{Z,V} \; l_f(z_N) + \sum_{i=0}^{N-1} z_i^\top Q z_i + \boldsymbol{v}_i^\top R \boldsymbol{v}_i \tag{23a}$$

$$\begin{aligned}
\text{s.t.} \quad & \boldsymbol{x}(k) \in z_0 \oplus \mathcal{E} && && (23b)\\
& z_{i+1} = A z_i + B \boldsymbol{v}_i, && i = 0, \dots, N-1 && (23c)\\
& z_i \in \mathcal{X}_z \ominus \mathcal{E}, && i = 0, \dots, N && (23d)\\
& \boldsymbol{v}_i \in \mathcal{U}_z \ominus K_{\text{tube}} \mathcal{E}, && i = 0, \dots, N-1 && (23e)\\
& z_N \in \mathcal{X}_N, && && (23f)
\end{aligned}$$

where $Z := \{z_0, \dots z_N\}$ and $V := \{\boldsymbol{v}_0, \dots \boldsymbol{v}_{N-1}\}$ define a sequence of nominal states and inputs, $\mathcal{E}$ is a robust positively invariant set[5], K_{tube} a stabilizing linear feedback controller for system (21), and $\mathcal{X}_z = \{\, \boldsymbol{x}_z \in \mathbb{R}^{n_x} \mid H_x \boldsymbol{x}_z \leq \boldsymbol{h}_x \}$ and $\mathcal{U}_z = \{\, \boldsymbol{u}_z \in \mathbb{R}^{n_u} \mid H_u \boldsymbol{u}_z \leq \boldsymbol{h}_u \}$ denote the polytopic state and input constraints. At each time step k, the following control input is applied to the system:

$$\boldsymbol{u}_z(k) = \kappa_{\text{tube}}(\boldsymbol{x}_z(k)) = \boldsymbol{v}_0^* + K_{\text{tube}}(\boldsymbol{x}_z(k) - z_0^*). \tag{24}$$

27. In this task, design a stabilizing linear feedback controller $\boldsymbol{u}_z(k) = K_{\text{tube}} \boldsymbol{x}_z(k)$ with $K_{\text{tube}} \in \mathbb{R}^{n_x \times n_u}$ for system (21) using pole placement. Implement a function `compute_tube_controller` that takes an array of poles $p = \begin{bmatrix} p_1 & p_2 \end{bmatrix}$ with $p_1, p_2 \in \mathbb{C}$ as an input, and outputs the feedback matrix of the tube controller, K_{tube}.
Hint: Use the MATLAB function `place`. 1 *pt.*

28. Compute the polytopic tube
$$\mathcal{E} := \{\, \boldsymbol{x}_z \in \mathbb{R}^{n_x} \mid H_{\text{tube}} \boldsymbol{x}_z \leq \boldsymbol{h}_{\text{tube}} \}$$
used in the MPC problem (23) as the minimal robust positive invariant (RPI) set for system (21) under the tube controller designed in Task (27). In a function `compute_minRPI`, implement the algorithm given in the MPC lecture slides[5] to obtain the polytopic tube $\mathcal{E}$ in dependence of the inputs K_{tube} and `params`. The outputs of the function are given as H_{tube}, $\boldsymbol{h}_{\text{tube}}$, n_{iter}, where n_{iter} is the number of iterations after which the algorithm has converged, i.e., $n_{\text{iter}} := i$, such that $\mathcal{E}_{i+1} = \mathcal{E}_i$.
Hint 1: To check convergence of your algorithm you can use `eq`$(\mathcal{E}_{i+1}, \mathcal{E}_i)$ *after reducing the polytopes to a minimal representation, e.g., by using the function* `Polyhedron.minHRep()`.
Hint 2: If your algorithm takes long to converge, consider choosing different poles p to design your tube controller K_{tube}. 3 *pt.*

29. Obtain the tightened constraints as defined in the MPC problem (23). Let H_x, $\boldsymbol{h}_x$, H_u, and $\boldsymbol{h}_u$ define the polytopic constraints for the system (21) as given in `params_z`. Implement a function `compute_tightening`, which takes `params_z` as input and adapts the parameters H_x, $\boldsymbol{h}_x$, H_u, and $\boldsymbol{h}_u$ to represent the tightened constraints in (23d) and (23e). The output is then given by the modified struct `params_z_tube`. 3 *pt.*

[4] Lec. 9, Robust MPC, Tube-MPC Problem Formulation
[5] Lec. 9, Robust MPC, Minimum Robust Invariant Set

30. Implement the tube-based robust model predictive controller (23) in the class MPC_TUBE. Design the terminal cost $l_f(\boldsymbol{x})$ as the LQR infinite-horizon cost (13) of the nominal system, i.e., system (21) without disturbances $w(k)$. During initialization of the class, a YALMIP solver object representing the optimization problem (23) needs to be created based on the inputs Q, R, N, the polytopic terminal set described by H_N and $\boldsymbol{h}_N$, the polytopic tube $\mathcal{E}$ described by H_{tube}, $\boldsymbol{h}_{\text{tube}}$, the tube controller K_{tube}, as well as the system model with tightened constraints in the struct `params_z_tube`. The `eval` method should solve the optimization problem and obtain the control input according to (24). *4 pt.*

31. Let $Q = \text{diag}(q_z^*, q_{vz}^*)$, $R = 1$, $N = 50$, and choose $p = \begin{bmatrix} 0.1 & 0.5 \end{bmatrix}$ to design the tube controller K_{tube} using the function designed in Task 27. Design the tube $\mathcal{E}$ as outlined in Task 28 and obtain tightened constraints as outlined in Task 29. Using the function `lqr_maxPI` with an appropriate choice of inputs, design the terminal set $\mathcal{X}_N$ such that the resulting controller is recursively feasible and robustly stable. Write the design of all ingredients and the setup of the resulting MPC_TUBE controller in a script called called `MPC_TUBE_script.m`. Provide the script and save the following variables in the MAT-file `MPC_TUBE_params.mat`: `p` $:= p$, `K_tube` $:= K_{\text{tube}}$, `H_tube` $:= H_{\text{tube}}$, `h_tube` $:= \boldsymbol{h}_{\text{tube}}$, `H_N` $:= H_N$, `h_N` $:= \boldsymbol{h}_N$, and `params_z_tube`. *2 pt.*

32. *[Simulation]: Sample different disturbance sequences W_t and simulate the MPC-TUBE controller obtained in Task 31 using your function `simulate_uncertain` for initial condition $\boldsymbol{x}_{z,0}^A$. Compare your MPC-TUBE controller against applying the MPC-TS controller implemented in Task 20 based on the system defined in `params_z` with $N = 50$. Plot the closed-loop trajectories using the function `plot_trajectory_z`. What can you observe? What can you observe if you apply the maximum disturbance at each time step, i.e., $w(k) = \begin{bmatrix} w_{\max} & w_{\max} \end{bmatrix}^\top$ for all time steps k?*

Task	Function	Inputs	Outputs	Pt.
25	`generate_disturbances`	`params_z`	W_t	1
26	`simulate_uncertain`	$\boldsymbol{x}_0$, `ctrl`, W_t, `params_z`	X_t, U_t, F_t	1
27	`compute_tube_controller`	p, `params_z`	K_{tube}	1
28	`compute_minRPI`	K_{tube}, `params_z`	H_{tube}, $\boldsymbol{h}_{\text{tube}}$, n_{iter}	3
29	`compute_tightening`	K_{tube}, H_{tube}, $\boldsymbol{h}_{\text{tube}}$, `params_z`	`params_z_tube`	3
30	`MPC_TUBE`	Q, R, N, H_N, $\boldsymbol{h}_N$, H_{tube}, $\boldsymbol{h}_{\text{tube}}$, K_{tube}, `params_z_tube`	`ctrl` (MPC_TUBE object)	4
30	`MPC_TUBE/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	1
31	`MPC_TUBE_script`	-	`MPC_TUBE_params.mat`	2

Table 10: Deliverable summary "Robust MPC"

FORCES Pro [Bonus]

[Bonus] It is possible to get full points in the programming exercise without solving this question.

In order to deploy your controller on the real system you are usually required to implement the model predictive controller on low-cost embedded hardware. To this end, it is important to ensure computational efficiency and to implement the model predictive controller using a low-level language like C or a code generator. FORCES Pro is a code generator that is compatible with any embedded platform having a C compiler.

Installation: You should have received an email from Embotech.com regarding FORCES Pro. Please contact us if you did not. Please follow the outlined instructions for download and installation. Note that the license will expire after the programming exercise deadline.

In the following tasks, we again consider the 6-dimensional system (11).

Deliverables

33. [Bonus] In the class template `MPC_TE_forces`, implement the model predictive controller (18) from Task 18 using the Forces Pro solver. The input and output arguments should be the same as for the `MPC` class implemented in Task 18. 2 *pt.*

34. *[Simulation]: Simulate the closed-loop system with both,* `MPC_TE_forces` *and* `MPC_TE` *as implemented in Task 18. Use the initial condition* $\boldsymbol{x}_0^A$ *for the same choice of* $Q := Q^*$, $R := R^*$ *and horizon length* $N = 30$ *and compare the results. Investigate the difference in solve time between the two implementations of the MPC controller* (18)*. What do you observe?*

    Hint: The solve time used by the FORCES Pro solver is provided in `info.solvetime`.

Task	Function	Inputs	Outputs	Pt.
33	`MPC_TE_forces`	Q, R, N, `params`	`ctrl` (`MPC_TE_forces` object)	2
33	`MPC_TE_forces/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0

Table 11: Deliverable summary "FORCES Pro [Bonus]"