Model Predictive Control
Prof. M. Zeilinger

Spring 2024

Programming Exercise
Climate Control

Preliminaries

MATLAB

For this project, you need to have MATLAB **R2023b** installed, which you can download from the ETH IT Shop[1]. Furthermore, the Control Systems Toolbox, the Optimization Toolbox, Simulink and **no other toolboxes** ought to be be installed.

Note: If you are using an Apple laptop with M1 chipset, please install MATLAB R2023a instead to avoid compatibility issues with the MPT toolbox.

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. Download the file `install_mpt3.m`.
4. Run `install_mpt3.m` in MATLAB.

MPT automatically installs Yalmip.

Provided Files

The provided files of this project are structured in three subdirectories: `templates/`, `testing/` and `utilities/`. 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. A complete list of deliverables with and their input-output signature can be found in table 3. More information about the functions can be found in the templates and their respective task description.

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.

Below you find some important information regarding the provided template and utility functions:

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

- `templates/generate_params_cc.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.

- `utilities/plot_trajectory_cc.m` (Function) — This function takes as input arguments trajectory data in terms of the states and inputs, X_{sim} and U_{sim} as defined in eq. (13), feasibility information `ctrl_feas` of the controller, see Task 7, and the `params` struct returned by `generate_params_cc` to create a basic plot. Feel free to modify it as you need.

- `utilities/plot_trajectory_gt_est_cc.m` (Function) — Similarly to `plot_trajectory_cc`, this function plots the closed-loop trajectories X_{sim} and U_{sim}. Additionally, takes as inputs a disturbance sequence D_{sim} as well as estimated state and disturbance trajectories X_{est} and D_{est} as defined in eq. (24), and feasibility information `ctrl_feas` to test your implementations in Task 27 and 29.

- `utilities/traj_abs2delta.m` (Function) — This function can be used to transform trajectory data X_{sim}, U_{sim} into delta-coordinates by shifting them according to the steady-state $\boldsymbol{x}_s$, $\boldsymbol{u}_s$.

- `utilities/traj_delta2abs.m` (Function) — This function can be used to transform trajectory data X_{sim}, U_{sim} into absolute coordinates by shifting them according to the steady-state $\boldsymbol{x}_s$, $\boldsymbol{u}_s$.

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, most 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("all");
```

The testing framework is available for all submitted functions and scripts. To run the tests for one specific function called `function_name` (`function_name` is a string, e.g., `"generate_system_cont_cc"`), run

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

The output `test_struct` is a MATLAB struct whose fields are described in Table 2. Note that when running "all" tests, the framework first checks that the tests for any prerequisite functions pass. 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 (generated by `generate_params_cc`) in your implementation and do not recompute or hard-code them.

In case we (or you) find a bug inside the software 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 within the two weeks preceding the hand-in date. Afterwards, if the testing framework flags parts of your submission as

Field	Value
model.a1o	$\alpha_{1,o}$
model.a2o	$\alpha_{2,o}$
model.a3o	$\alpha_{3,o}$
model.a12	$\alpha_{1,2}$
model.a23	$\alpha_{2,3}$
model.m1	m_1
model.m2	m_2
model.m3	m_3
model.nx	n_x
model.nu	n_u
model.nd	n_d
model.ny	n_y
model.A	A
model.B	B
model.Bd	B_d
model.C	C
model.Cd	C_d
model.C_ref	C_{ref}
model.TimeStep	Δt

Field	Value
constraints.InputMatrix	H_u
constraints.InputRHS	$\boldsymbol{h}_u$
constraints.P1Max	$p_{1,\max}$
constraints.P1Min	$p_{1,\min}$
constraints.P2Max	$p_{2,\max}$
constraints.P2Min	$p_{2,\min}$
constraints.StateMatrix	H_x
constraints.StateRHS	$\boldsymbol{h}_x$
constraints.T1Max	$T_{1,\max}$
constraints.T1Max	$T_{1,\min}$
constraints.T2Max	$T_{2,\max}$
constraints.T2Min	$T_{2,\min}$
constraints.T3Max	$T_{3,\max}$
constraints.T3Min	$T_{3,\min}$
exercise.d0_est	$d_{0,\text{est}}$
exercise.InitialConditionA	$\boldsymbol{x}_0^A$
exercise.InitialConditionB	$\boldsymbol{x}_0^B$
exercise.InitialConditionC	$\boldsymbol{x}_0^C$
exercise.MPCHorizon	N
exercise.QdiagOpt	$\boldsymbol{q}_{\text{opt}}$
exercise.RadiationA	$\boldsymbol{\eta}_A$
exercise.RadiationB	$\boldsymbol{\eta}_B$
exercise.RdiagOpt	$\boldsymbol{r}_{\text{opt}}$
exercise.SimHorizon	N_{sim}
exercise.T_ref	T_{ref}
exercise.T_o	T_{o}
exercise.x0_est	$x_{0,\text{est}}$

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

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[3]. You are not allowed to make this project description, template, or your solution publicly available.

Hand in a single zip-file, where the filename contains the ETH ID numbers ("xx-xxx-xx", including dashes) of all team-members according to this template (note that there are no spaces in the filename):

`IDNumber1_IDNumber2_IDNumber3.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.

[3] `https://ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf`

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`.

- A MAT-file which contains the `test_struct` that is returned by the testing framework (see the description of the testing framework above).
- A PDF file of a scan of the Declaration of Originality signed by all team-members.
- If applicable, a PDF file of 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; in particular, the files `testing/` and `utilities/` should **not** be submitted. The directory structure and naming of the files **must** follow this template:

```
IDNumber1_IDNumber2_IDNumber3.zip
├── Declaration_of_Originality.pdf
├── test_struct.mat
├── Report.pdf (optional)
└── <All template/*.m and template/*.mat files>
```

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 **19.05.24, 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 questions 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.

Deliverable summary

Task	Function	Inputs	Outputs	Pt.
1	`generate_system_cont_cc`	`params`	A^c, B^c, B_d^c	2
2	`discretize_system_dist`	A^c, B^c, B_d^c, `params`	A, B, B_d	2
3	`generate_constraints_cc`	`params`	$H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$	2
4	`generate_params_cc` (modify)		`params`	1
5	`compute_steady_state`	`params`, $\boldsymbol{d}$	$\boldsymbol{u}_s, \boldsymbol{x}_s$	2
6	`generate_params_delta_cc`	`params`	`params_delta`	2
7	`LQR`	Q, R, `params`	`ctrl` (`LQR` object)	2
7	`LQR/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
8	`simulate`	$\boldsymbol{x}(0)$, `ctrl`, `params`	X_t, U_t, `ctrl_info`	2
9	`traj_cost`	X_t, U_t, Q, R	$J_{N_{\text{sim}}}$	1
10	`traj_constraints_cc`	X_t, U_t, `params`	$T^t_{1,\max}$, $T^t_{2,\min}$, $T^t_{2,\max}$, $p^t_{1,\min}$, $p^t_{1,\max}$, $p^t_{2,\min}$, $p^t_{2,\max}$, J_u, `cstr_viol`	3
11	`lqr_tuning_script_cc`	-	`lqr_tuning_script_cc.mat`	2
12	`lqr_maxPI`	Q, R, `params`	$\boldsymbol{H}_{\text{LQR}}$, $\boldsymbol{h}_{\text{LQR}}$	3
14	`MPC`	Q, R, N, `params`	`ctrl` (`MPC` object)	5
14	`MPC/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
16	`MPC_TE`	Q, R, N, `params`	`ctrl` (`MPC_TE` object)	2
16	`MPC_TE/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
18	`MPC_TS`	Q, R, N, H, $\boldsymbol{h}$, `params`	`ctrl` (`MPC_TS` object)	2
18	`MPC_TS/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
21	`MPC_TS_SC_ET`	Q, R, N, H, $\boldsymbol{h}$, S, v, S_t, v_t, `params`	`ctrl` (`MPC_TS_SC_ET` object)	4
21	`MPC_TS_SC_ET/eval`	$\boldsymbol{x}$	$\boldsymbol{u}$, `ctrl_info`	0
22	`MPC_TS_SC_ET_script_cc`	-	`MPC_TS_SC_ET_script_cc.mat`	2
24	`generate_params_aug_obs`	`params`	`params_aug`	1
25	`compute_observer_gain`	$\boldsymbol{p}$, `params_aug`	L	1
26	`Linear_Observer`	L, `params_aug`	`obsv` (`Linear_Observer` object)	0
26	`Linear_Observer/eval`	$\tilde{\boldsymbol{x}}(k)$	$\tilde{\boldsymbol{x}}(k+1)$	1
27	`simulate_observer`	$\boldsymbol{x}(0), \boldsymbol{x}_{\text{est}}(0)$, $\boldsymbol{d}_{\text{est}}(0)$, `ctrl`, `obsv`, D_{sim}, `params`	X_{sim}, U_{sim}, X_{est}, D_{est}, `ctrl_info`	3
28	`obsv_tuning_script_cc`		`obsv_tuning_script_cc.mat`	2
29	`MPC_TS_offsetfree`	$Q, R, N, H, \boldsymbol{h}$, `params`	`ctrl` (`MPC_TS_offsetfree` object)	3
29	`MPC_TS_offsetfree/eval`	$\boldsymbol{x}$, $\boldsymbol{d}$, $\boldsymbol{x}_s$, $\boldsymbol{u}_s$	$\boldsymbol{u}$, `ctrl_info`	1

Table 3: Deliverable summary

System description

In this project you will implement an MPC controller for the temperature regulation of a delivery truck with three temperature zones, two of which are equipped with cooling units to regulate the desired temperatures. The truck is illustrated in Figure 1 and consists of the following zones.

Zone 1 Deep freeze temperature zone at $-21°\mathrm{C}$ for frozen goods actuated by *Cooling Unit 1*. In order to ensure food safety, the temperature may not exceed $-15°\mathrm{C}$ at any time.

Zone 2 Cooled temperature zone at $0.3°\mathrm{C}$ for fresh goods actuated by *Cooling Unit 2*. The temperature shall never exceed $4°\mathrm{C}$. Additionally, freezing temperatures below $0°\mathrm{C}$ must be avoided at all times.

Zone 3 Unregulated temperature zone for non-perishable goods.

Each zone is equipped with a sensor providing the current temperature of the respective zone. The system is actuated by cooling units in Zone 1 and 2 with different capacities.

Cooling Unit 1 Provides cooling power between $-2500\,\mathrm{W}$ and $0\,\mathrm{W}$.

Cooling Unit 2 Provides cooling power between $-2000\,\mathrm{W}$ and $0\,\mathrm{W}$.

Your task is to design a temperature controller for the delivery truck, which tracks the desired temperature and satisfies the food safety constraints at all times.

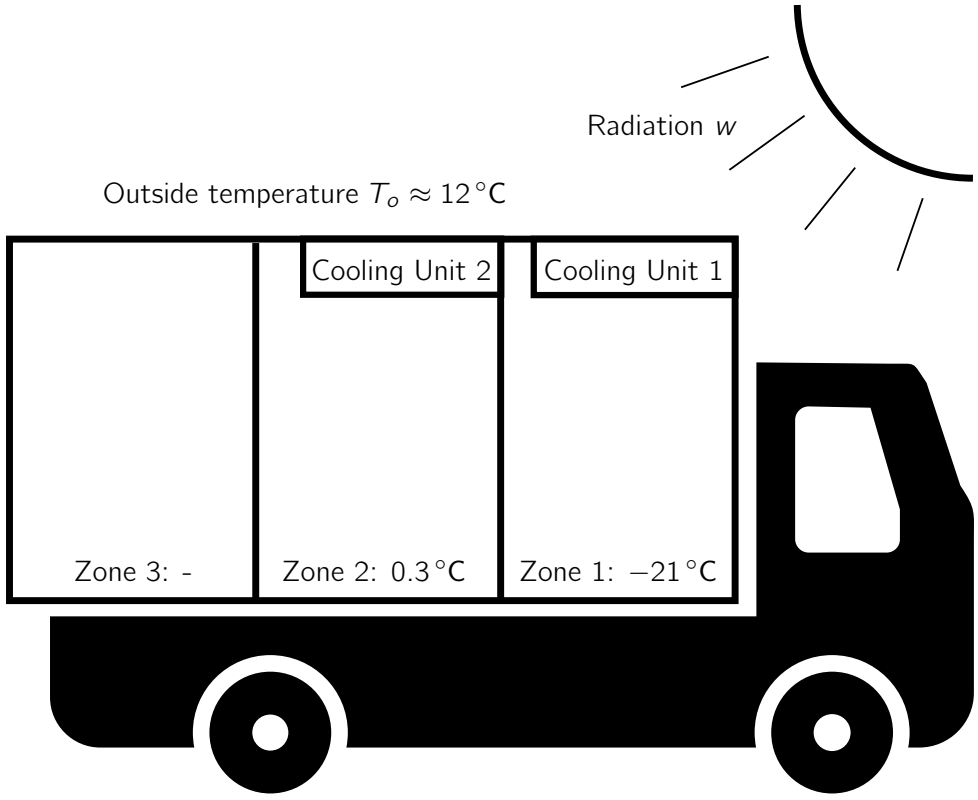

Figure 1: Delivery truck with different climate zones. Zones 1 and 2 are equipped with an integrated cooling unit regulating the respective temperatures.

Modeling

We model the system using simplified heat flows, consisting of flows between the zones, as well as from the outside and additional heat sources, e.g., by radiation. The outside temperature T_o and external heat flux $\boldsymbol{\eta} = \left[\eta_1, \eta_2, \eta_3\right]^\top$ are, for now, considered to be constant and known such that the system dynamics can be modelled using the following differential equations

$$\begin{aligned} m_1 \dot{T}_1^c(t) &= \alpha_{1,2}(T_2^c(t) - T_1^c(t)) + \alpha_{1,o}(T_o - T_1^c(t)) + p_1^c(t) + \eta_1 \\ m_2 \dot{T}_2^c(t) &= \alpha_{1,2}(T_1^c(t) - T_2^c(t)) + \alpha_{2,3}(T_3^c(t) - T_2^c(t)) + \alpha_{2,o}(T_o - T_2^c(t)) + p_2^c(t) + \eta_2 \\ m_3 \dot{T}_3^c(t) &= \alpha_{2,3}(T_2^c(t) - T_3^c(t)) + \alpha_{3,o}(T_o - T_3^c(t)) + \eta_3, \end{aligned} \tag{1}$$

with initial condition $\boldsymbol{x}_0 = [T_1^c(0), T_2^c(0), T_3^c(0)]^\top$, where $T_i^c(t)$ is the temperature of zone i at time $t \geq 0$, $p_i(t)$ is the cooling power of unit i, m_i is its thermal mass, d_i an external heat flux and $\alpha_{i,j}$ the thermal conductivity between zone i and j, or the outside (o).

Your first task is to bring the system in standard form for regulation to a steady-state. Therefore, denote by

$$\boldsymbol{x}(t) := \begin{bmatrix} T_1^c(t) \\ T_2^c(t) \\ T_3^c(t) \end{bmatrix} \in \mathbb{R}^{n_x}, \qquad \boldsymbol{u}(t) := \begin{bmatrix} p_1^c(t) \\ p_2^c(t) \end{bmatrix} \in \mathbb{R}^{n_u}, \qquad \boldsymbol{d} := \begin{bmatrix} \alpha_{1,o} \\ \alpha_{2,o} \\ \alpha_{3,o} \end{bmatrix} T_o + \boldsymbol{\eta} \in \mathbb{R}^{n_d}, \tag{2}$$

the system state, control input and disturbance, respectively.

Deliverables

1. Rewrite ODE (1) in the following form:

   $$\dot{\boldsymbol{x}}(t) = A^c \boldsymbol{x}(t) + B^c \boldsymbol{u}(t) + B_d^c \boldsymbol{d},$$

   where $B_d^c = \text{diag}([\frac{1}{m_1}, \frac{1}{m_2}, \frac{1}{m_3}])$ is the disturbance input matrix. Implement a function called `generate_system_cont_cc` that takes the `params` struct as input and returns the matrices A^c, B^c, B_d^c. *2 pt.*

2. Discretize the continuous-time ODE with sampling time $\Delta T = 60$ such that the resulting discrete-time dynamics are given by the difference equation

   $$\boldsymbol{x}_d(k+1) = A\boldsymbol{x}_d(k) + B\boldsymbol{u}_d(k) + B_d\boldsymbol{d}_d, \tag{3}$$

   where $\boldsymbol{x}_d(k) := \boldsymbol{x}(k\Delta T)$, $\boldsymbol{u}_d(k) := \boldsymbol{u}(k\Delta T)$ and $\boldsymbol{d}_d := \boldsymbol{d}$ for $\boldsymbol{x}_d(0) = \boldsymbol{x}(0)$. Assume piece-wise constant input signals within sampling intervals, i.e., $\boldsymbol{u}(k\Delta T + \tau) = \boldsymbol{u}_k = \text{const.}$ for all $\tau \in [0, \Delta T)$. Implement your solution as a function `discretize_system_dist`, that computes and returns A, B, B_d based on the continuous-time matrices A^c, B^c, B_d^c and the `params` struct as inputs.
   Hint: You can use the MATLAB function c2d. *2 pt.*

3. Find matrices $H_x \in \mathbb{R}^{6\times n_x}$, $H_u \in \mathbb{R}^{4\times n_u}$, as well as vectors $\boldsymbol{h}_x \in \mathbb{R}^6$, $\boldsymbol{h}_u \in \mathbb{R}^4$ such that

   $$H_x \boldsymbol{x}_d(k) \leq \boldsymbol{h}_x, \tag{4}$$
   $$H_u \boldsymbol{u}_d(k) \leq \boldsymbol{h}_u, \tag{5}$$

   encode box constraints $x_{i,\min} \leq x_i \leq x_{i,\max}$, $u_{j,\min} \leq u_j \leq u_{j,\max}$, for all state and input components $i = 1, \ldots, n_x$, $j = 1, \ldots, n_u$, respectively. For unconstrained directions of the state constraints, set the maximum and minimum values to $+1000$ and -1000, respectively, which are stored as `constraints.T{i}Max` and `constraints.T{i}Min` in the `params` struct. Implement a function `generate_constraints_cc`, that takes as input the parameter struct `params` and outputs $H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$. *2 pt.*

4. Modify the function `generate_params_cc` to also compute A, B, B_d, 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_cc` should now exactly give a `struct` as specified in Table 1.
   Important: When using $H_u, \boldsymbol{h}_u, H_x, \boldsymbol{h}_x$ in the following exercise, avoid calling the function `generate_constraints_cc` to re-generate them; instead, always use the values saved in the `params` struct. 1 *pt.*

5. Let a measurement vector $\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{d}) \in \mathbb{R}^{n_y}$ be defined by

$$\boldsymbol{y}(\boldsymbol{x}, \boldsymbol{d}) := C\boldsymbol{x} + C_d\boldsymbol{d}, \tag{6}$$

   where $C \in \mathbb{R}^{n_y \times n_x}$, $C_d \in \mathbb{R}^{n_y \times n_d}$, are defined in the `params` struct. Implement a function `compute_steady_state` that takes as input the parameter struct `params` (with the given temperature reference T_{ref}) and a disturbance estimate $\boldsymbol{d}$, and outputs the corresponding steady state $\boldsymbol{x}_s$ and steady-state input $\boldsymbol{u}_s$ of (3), such that offset-free tracking may be achieved, i.e., $C_{\text{ref}}\boldsymbol{y}(\boldsymbol{x}_s, \boldsymbol{d}) = T_{\text{ref}}$. 2 *pt.*

Given the steady-state of the system, the system dynamics can be expressed in the form of a discrete-time "delta-formulation" model:

$$\Delta\boldsymbol{x}_d(k+1) = A\Delta\boldsymbol{x}_d(k) + B\Delta\boldsymbol{u}_d(k), \tag{7}$$

which can be used for regulating the system to the computed steady-state.

6. Implement a function `generate_params_delta_cc` that takes as input the `params` struct, based on which it computes a disturbance estimate

$$\boldsymbol{d} = \begin{bmatrix} \alpha_{1,\text{o}} \\ \alpha_{2,\text{o}} \\ \alpha_{3,\text{o}} \end{bmatrix} T_{\text{o}} + \boldsymbol{\eta}_A \tag{8}$$

   and the corresponding steady-state $\boldsymbol{x}_s, \boldsymbol{u}_s$. Based on these values, it should adjust and *overwrite the constraints* (4),(5) *and initial conditions* $\boldsymbol{x}_0^A, \boldsymbol{x}_0^B, \boldsymbol{x}_0^C$ of the system, such that they are equivalently expressed in the delta-coordinates

$$\Delta\boldsymbol{x}_d(k) = \boldsymbol{x}_d(k) - \boldsymbol{x}_s, \tag{9}$$
$$\Delta\boldsymbol{u}_d(k) = \boldsymbol{u}_d(k) - \boldsymbol{u}_s, \tag{10}$$

   as well as add fields for the steady state $\boldsymbol{u}_s$, $\boldsymbol{x}_s$. The corresponding fields in the `params_delta` struct can be found in Table 4. The function should should return the modified struct `params_delta`.

   2 *pt.*

Field	Value
`exercise.InitialConditionA`	$\Delta \boldsymbol{x}_0^A$
`exercise.InitialConditionB`	$\Delta \boldsymbol{x}_0^B$
`exercise.InitialConditionC`	$\Delta \boldsymbol{x}_0^C$
`constraints.InputMatrix`	ΔH_u
`constraints.InputRHS`	$\Delta \boldsymbol{h}_u$
`constraints.StateMatrix`	ΔH_x
`constraints.StateRHS`	$\Delta \boldsymbol{h}_x$
`exercise.u_s`	$\boldsymbol{u}_s$
`exercise.x_s`	$\boldsymbol{x}_s$

Table 4: Added/modified fields in the `params_delta` struct (other fields same as `params`).

Unconstrained Optimal Control

Notation: For brevity, and with a slight abuse of notation, we drop the subscript in the following and write $x(k)$ and $u(k)$ to refer to the discrete-time state and input variables, $x_d(k)$ and $u_d(k)$, respectively. Additionally, in the following we consider the regulation to the computed steady-state and omit Δ, referring to, e.g., $\Delta x(k)$ by writing $x(k)$. Analogously, we denote by `params` the struct `params_delta` in delta-coordinates.

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

$$\boldsymbol{u}(k) := F_\infty \boldsymbol{x}(k) \tag{11}$$

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

$$J_\infty(\boldsymbol{x}(0)) := \sum_{k=0}^{\infty} \boldsymbol{x}(k)^\top Q \boldsymbol{x}(k) + \boldsymbol{u}(k)^\top R \boldsymbol{u}(k), \tag{12}$$

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 control 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 $\boldsymbol{q} \in \mathbb{R}^{n_x \times 1}$, $\boldsymbol{q} := \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix}^\top$ as well as $\boldsymbol{r} \in \mathbb{R}^{n_u \times 1}$, $\boldsymbol{r} := \begin{bmatrix} r_1 & r_2 \end{bmatrix}^\top$ choose the parametrization $Q := \mathrm{diag}(\boldsymbol{q})$ and $R = \mathrm{diag}(\boldsymbol{r})$.

Tasks

7. 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 (11) 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.*

8. Simulate the closed-loop system for a given initial condition $\boldsymbol{x}(0)$, controller object `ctrl` and number of time steps N_{sim}. Implement your solution in the function `simulate`, which takes $\boldsymbol{x}(0)$, `ctrl`, `params` as inputs and outputs the closed-loop trajectory according to equation (3) in terms of the matrices $X_t \in \mathbb{R}^{n_x \times (N_{\text{sim}}+1)}$, $U_t \in \mathbb{R}^{n_u \times N_{\text{sim}}}$, with

$$X_t := \begin{bmatrix} \boldsymbol{x}(0) & \dots & \boldsymbol{x}(N_{\text{sim}}) \end{bmatrix}, \qquad U_t := \begin{bmatrix} \boldsymbol{u}(0) & \dots & \boldsymbol{u}(N_{\text{sim}}-1) \end{bmatrix}, \tag{13}$$

as well as an N_{sim}-dimensional array of the `ctrl_info` structs described in Task 7. 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.*

9. 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 (13) and outputs the closed-loop quadratic cost for a given trajectory, i.e.,

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

1 *pt.*

10. Check the satisfaction of the constraints for a given trajectory. Implement a function called `traj_constraints_cc`, that takes as input the trajectory data X_t, U_t in **absolute** coordinates and computes as outputs

    - the maximum temperature in Zone 1, $T^t_{1,\max} = \max_{k \in [0, N_{\text{sim}}]} T^c_1(k\Delta T)$,
    - the minimum temperature in Zone 2, $T^t_{2,\min} = \min_{k \in [0, N_{\text{sim}}]} T^c_2(k\Delta T)$,
    - the maximum temperature in Zone 2, $T^t_{2,\max} = \max_{k \in [0, N_{\text{sim}}]} T^c_2(k\Delta T)$,
    - the minimum cooling power in Zone 1, $p^t_{1,\min} = \min_{k \in [0, N_{\text{sim}}]} p^c_1(k\Delta T)$,
    - the maximum cooling power in Zone 1, $p^t_{1,\max} = \max_{k \in [0, N_{\text{sim}}]} p^c_1(k\Delta T)$,
    - the minimum cooling power in Zone 2, $p^t_{2,\min} = \min_{k \in [0, N_{\text{sim}}]} p^c_2(k\Delta T)$,
    - the maximum cooling power in Zone 2, $p^t_{2,\max} = \max_{k \in [0, N_{\text{sim}}]} p^c_2(k\Delta T)$,
    - the closed-loop finite-horizon input cost, $J_u := \sum_{k=0}^{N_{\text{sim}}-1} \boldsymbol{u}(k)^\top \boldsymbol{u}(k)$,
    - a boolean flag `cstr_viol` indicating if the trajectory contains any constraint violations, i.e., `cstr_viol` = false if and only if all constraints provided in the task description on page 1 are satisfied.

3 *pt.*

11. Design an LQR controller for the system such that for the initial condition $\boldsymbol{x}^A_0$, the following properties hold for the resulting, simulated closed-loop system:

    - Input and state constraints are satisfied for all time steps $k \in [0, 60]$.
    - The closed-loop system approaches the reference reasonably fast, more precisely, it holds that

$$|\Delta \boldsymbol{x}_1(30)| \leq 3 \cdot 10^{-1}, \qquad |\Delta \boldsymbol{x}_1(60)| \leq 3 \cdot 10^{-2}, \tag{14}$$

$$|\Delta \boldsymbol{x}_2(30)| \leq 2 \cdot 10^{-2}, \qquad |\Delta \boldsymbol{x}_2(60)| \leq 2 \cdot 10^{-3}. \tag{15}$$

    Provide a script `lqr_tuning_script_cc.m` that runs a closed-loop simulation based on your tuning of Q and R and saves the following information as a MAT-file `lqr_tuning_script_cc.mat`:

2 *pt.*

Variable	Value
`Q`	Q
`R`	R
`X`	X_t (in relative coordinates)
`U`	U_t (in relative coordinates)

Table 5: Variables contained in `lqr_tuning_script_cc.mat`.

A first model predictive controller

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 $Q^* := \text{diag}(\boldsymbol{q}^*)$ and $R^* := \text{diag}(\boldsymbol{r}^*)$, with

$$\boldsymbol{q}^* := \begin{bmatrix} q_1^* \\ q_2^* \\ q_3^* \end{bmatrix} := \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \qquad \boldsymbol{r}^* := \begin{bmatrix} r_1^* \\ r_2^* \end{bmatrix} := \begin{bmatrix} 2.4 \\ 0.5 \end{bmatrix} \cdot 10^{-5}$$

Nevertheless, feel free to also experiment with your best values obtained from the LQR controller tuning in Task 11 and see how it affects the MPC closed-loop performance.

Tasks

12. Explicitly compute the maximum positively invariant set $\mathcal{X}_{LQR}$ under application of the LQR controller. 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 (11) to system (3), 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 returns $H_{\text{LQR}} \in \mathbb{R}^{n_H \times n_x}$ and $\boldsymbol{h}_{\text{LQR}} \in \mathbb{R}^{n_H}$ such that $\mathcal{X}_{LQR} = \{\boldsymbol{x} \mid H_{\text{LQR}} \boldsymbol{x} \leq \boldsymbol{h}_{\text{LQR}}\}$.
    Hint: You can use the MPT toolbox[4] for this task. 3 *pt.*

13. *[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?*

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

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

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

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

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

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

[4] `https://www.mpt3.org/`; in particular, see `https://www.mpt3.org/UI/Filters` for constraint specification.

at each time step, where $l_f(\boldsymbol{x}) = J_\infty(\boldsymbol{x})$ is equal to the LQR infinite-horizon cost (12). 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.
Hint 3: Consider the YALMIP documentation for setting up the model predictive controller: `https://yalmip.github.io/example/standardmpc/`. 5 *pt.*

15. *[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?*

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

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

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

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

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

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

$$\min_{U} \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i + l_f(\boldsymbol{x}_N) \tag{19a}$$
$$\text{s.t. } \boldsymbol{x}_0 = \boldsymbol{x}(k) \tag{19b}$$
$$\boldsymbol{x}_{i+1} = A\boldsymbol{x}_i + B\boldsymbol{u}_i, \qquad i = 0, \dots, N-1 \tag{19c}$$
$$H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x, \qquad i = 0, \dots, N-1 \tag{19d}$$
$$H_u \boldsymbol{u}_i \leq \boldsymbol{h}_u, \qquad i = 0, \dots, N-1 \tag{19e}$$
$$H\boldsymbol{x}_N \leq \boldsymbol{h}, \tag{19f}$$

    in the class template `MPC_TS`, where $l_f(\boldsymbol{x}) := J_\infty(\boldsymbol{x})$ is the LQR infinite-horizon cost and H, $\boldsymbol{h}$, define a polytopic terminal set. 2 *pt.*

19. *[Self-study]: How can the terminal set be chosen for the given system to ensure that the closed-loop dynamics is stable and satisfies constraints when initialized with an initial condition for which* (19) *is feasible?*

20. *[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* $N = 40 \dots 20$. *What do you observe?*

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

21. Introduce slack variables $\boldsymbol{\epsilon}_i \in \mathbb{R}^{n_{\text{constr},x}}$, $i = 0, \dots, N$, $\boldsymbol{\epsilon}_\text{t} \in \mathbb{R}^{n_\text{t}}$, into the MPC problem (19) to restore feasibility in the case of state and terminal constraint violations. Implement a model predictive controller based on the MPC problem

$$\min_{U,X,E} \quad \sum_{i=0}^{N-1} \boldsymbol{x}_i^\top Q \boldsymbol{x}_i + \boldsymbol{u}_i^\top R \boldsymbol{u}_i + l_\text{f}(\boldsymbol{x}_N) + \sum_{i=0}^{N} \boldsymbol{\epsilon}_i^\top S \boldsymbol{\epsilon}_i + v\|\boldsymbol{\epsilon}_i\|_1 + \boldsymbol{\epsilon}_\text{t}^\top S_\text{t} \boldsymbol{\epsilon}_\text{t} + v_\text{t}\|\boldsymbol{\epsilon}_\text{t}\|_1 \tag{20a}$$

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

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

$$H_x \boldsymbol{x}_i \leq \boldsymbol{h}_x + \boldsymbol{\epsilon}_i, \quad i = 0, \dots, N-1, \tag{20d}$$

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

$$\boldsymbol{H}\boldsymbol{x}_N \leq \boldsymbol{h} + \boldsymbol{\epsilon}_\text{t} \tag{20f}$$

$$\boldsymbol{\epsilon}_i \geq 0, \quad i = 0, \dots, N-1, \tag{20g}$$

$$\boldsymbol{\epsilon}_\text{t} \geq 0 \tag{20h}$$

in the class `MPC_TS_SC_ET`. The class shall be initialized with the same parameters as `MPC_TS`, plus the additional values for $S \in \mathbb{R}^{n_{\text{constr},x} \times n_{\text{constr},x}}$, $v \in \mathbb{R}$ and $S_\text{t} \in \mathbb{R}^{n_H \times n_H}$, $v_\text{t} \in \mathbb{R}$ for the constraint violation penalty with $v, v_\text{t} \gg 0$. Thereby $n_{\text{constr},x}$ denotes the number of state constraints as given by the state constraint definition in the `params` struct.
Hint: To avoid numerical errors, it might be necessary to switch to an MPC implementation without *substitution, see the "Implicit prediction form" YALMIP tutorial:* `https://yalmip.github.io/example/standardmpc/`. 4 *pt.*

22. Choose $S, v, S_\text{t}, v_\text{t}$ such that the controller based on the soft-constrained MPC problem (20) returns the same control inputs 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 providing a script called `MPC_TS_SC_ET_script_cc.m`, which verifies that, with your choice of $S, v, S_\text{t}, v_\text{t}$, the soft-constrained controller `MPC_TS_SC_ET` returns the same solution as `MPC_TS` for initial condition $\boldsymbol{x}_0^B$ while returning a solution for initial condition $\boldsymbol{x}_0^C$ (which is infeasible for `MPC_TS`). Provide the script and save your selected values in the MAT-file `MPC_TS_SC_ET_script_cc.mat`, using the following naming convention:

Variable	Value
`v`	v
`S`	S
`v_t`	v_t
`S_t`	S_t

Table 6: Variables contained in `MPC_TS_SC_ET_script_cc.mat`.

2 *pt.*

23. *[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?*

Offset-free MPC

Notation: In this section, we denote by $x(k)$, $u(k)$, `params` again the state, input and `params` struct in **absolute coordinates**.

The heat transfer due the outside air temperature and radiation is typically not known exactly. Therefore, the task is to estimate the resulting disturbance such that offset-free tracking is ensured. More precisely, consider (3) with time-invariant disturbance $d(k) = \bar{d}$, where $\bar{d} \in \mathbb{R}^{n_d}$ is unknown but constant.

Deliverables

24. Augment the discrete-time system (3) such that the constant disturbance $\bar{d}$ can be observed, i.e., determine the matrices $\tilde{A}$, $\tilde{B}$, and $\tilde{C}$, for the augmented state $\tilde{\boldsymbol{x}}_{\text{est}}(k) \in \mathbb{R}^{n_{\tilde{x}}}$, $\tilde{\boldsymbol{x}}_{\text{est}}(k) = \begin{bmatrix} \boldsymbol{x}_{\text{est}}(k)^\top & \boldsymbol{d}_{\text{est}}(k)^\top \end{bmatrix}^\top$ and its dynamics

$$\tilde{\boldsymbol{x}}_{\text{est}}(k+1) = \tilde{A}\tilde{\boldsymbol{x}}_{\text{est}}(k) + \tilde{B}\boldsymbol{u}(k), \tag{21}$$

$$\boldsymbol{y}(k+1) = \tilde{C}\tilde{\boldsymbol{x}}_{\text{est}}(k+1). \tag{22}$$

Thereby, the measurement vector for the augmented state should coincide with that for the original state and disturbance as given in eq. (6), i.e., $\boldsymbol{y}(k) = \boldsymbol{y}(\boldsymbol{x}_{\text{est}}(k), \boldsymbol{d}_{\text{est}}(k))$. Write a function `generate_params_aug_obs` that takes as input the `params` struct and returns a struct `params_aug`, where, according to the new system definition, the following fields have been modified:

Field	Value
`model.A`	$\tilde{A}$
`model.B`	$\tilde{B}$
`model.C`	$\tilde{C}$
`model.nx`	$n_{\tilde{x}}$
`constraints.StateMatrix`	ΔH_x
`constraints.StateRHS`	$\Delta \boldsymbol{h}_x$

Table 7: Added/modified fields in the `params_aug` struct (other fields same as `params`).

1 *pt.*

25. Design the observer gain matrix L in the linear observer

$$\tilde{\boldsymbol{x}}(k+1) = \tilde{A}\tilde{\boldsymbol{x}}(k) + \tilde{B}\boldsymbol{u}(k) + L\left(\tilde{C}\tilde{\boldsymbol{x}}(k) - \boldsymbol{y}(k)\right). \tag{23}$$

Write a function `compute_observer_gain`, which takes as input a vector $\boldsymbol{p} \in \mathbb{R}^{n_{\tilde{x}}}$ of poles, as well as the `params_aug` struct, and returns a matrix $L \in \mathbb{R}^{n_{\tilde{x}} \times n_y}$, such that the eigenvalues of the autonomous estimation-error dynamics, i.e., the dynamics of $(\boldsymbol{x}(k) - \tilde{\boldsymbol{x}}(k))$ for $\boldsymbol{u}(\cdot) \equiv 0$, are given by $\boldsymbol{p}$.

1 *pt.*

26. Implement a class `Linear_Observer` that is instantiated with a given observer gain L and the augmented `params_aug` struct, which are stored as properties of the class. Add an `eval` method to the class, which takes as input the current state $\tilde{\boldsymbol{x}}(k)$, and outputs the updated state estimate $\tilde{\boldsymbol{x}}(k+1)$ for the next time step based on the stored values, as given in equation (23).

27. Simulate the closed-loop system *including the observer* for a given true initial condition $\boldsymbol{x}(0)$, estimated initial condition $\boldsymbol{x}_{\text{est}}(0)$ and disturbance $\boldsymbol{d}_{\text{est}}(0)$, controller object `ctrl`, observer

object `obsv`, an array of disturbances $D_{\text{sim}} := \begin{bmatrix}\boldsymbol{d}(0) & \dots & \boldsymbol{d}(N_{\text{sim}-1})\end{bmatrix}$, and parameter struct `params` (**not** `params_aug`!). Implement your solution in the function `simulate_observer`, which takes $\boldsymbol{x}(0)$, $\boldsymbol{x}_{\text{est}}(0)$, `ctrl`, `obsv`, `params`, D_{sim} as inputs, and outputs the closed-loop trajectory according to equation (3) in terms of the the true state and input $X_t \in \mathbb{R}^{n_x \times (N_{\text{sim}}+1)}$, $U_t \in \mathbb{R}^{n_u \times N_{\text{sim}}}$ as in equation (13), the estimated states and disturbances $X_{\text{est}} \in \mathbb{R}^{n_x \times (N_{\text{sim}}+1)}$, $D_{\text{est}} \in \mathbb{R}^{n_d \times (N_{\text{sim}}+1)}$ with

$$X_{\text{est}} := \begin{bmatrix}\boldsymbol{x}_{\text{est}}(0) & \dots & \boldsymbol{x}_{\text{est}}(N_{\text{sim}})\end{bmatrix}, \qquad D_{\text{est}} := \begin{bmatrix}\boldsymbol{d}_{\text{est}}(0) & \dots & \boldsymbol{d}_{\text{est}}(N_{\text{sim}})\end{bmatrix}, \tag{24}$$

as well as an N_{sim}-dimensional array of the `ctrl_info` structs described in Task 7. The simulation is to be executed as follows:

(a) The current disturbance estimate $\boldsymbol{d}_{\text{est}}(k)$ is used to update the set point $\boldsymbol{x}_s$, $\boldsymbol{u}_s$.

(b) The control input $\boldsymbol{u}(k)$ is evaluated based on the estimated state $\boldsymbol{x}_{\text{est}}(k)$, estimated disturbance $\boldsymbol{d}_{\text{est}}(k)$ and set point $\boldsymbol{x}_s$, $\boldsymbol{u}_s$. Note that this requires to adapt the input-output signature of the `eval` function of the controller, such that `[u, ctrl_info] = ctrl.eval(x, d, x_sp, u_sp)`; we have provided a `Controller_setpoint` class that can be used for testing.

(c) The true system state is advanced based on the control input $\boldsymbol{u}(k)$ and actual value $\boldsymbol{d}(k)$ of the disturbance.

(d) The estimated state $\boldsymbol{x}_{\text{est}}(k+1)$ and disturbance $\boldsymbol{d}_{\text{est}}(k+1)$ are estimated based on the instantaneous measurement at time $(k+1)$; at time $k = 0$, no measurement update is executed, since it is assumed to already be included in the initial state and disturbance estimate.

3 *pt.*

28. Tune the poles $\boldsymbol{p}$ of the closed-loop estimation error dynamics such that accurate estimation of the autonomous system trajectory (with zero control input) can be achieved from initial condition $\boldsymbol{x}_0^A$ for the (unknown) disturbance given by eq. (8) with the radiation value $\boldsymbol{\eta}_B =$ `params.exercise.RadiationB`; in particular, fast convergence should be assured by satisfying the following convergence criteria:

$$\|\boldsymbol{x}_{\text{sim}}(1) - \boldsymbol{x}_{\text{est}}(1)\|_2 \leq 4, \quad \|\boldsymbol{x}_{\text{sim}}(10) - \boldsymbol{x}_{\text{est}}(10)\|_2 \leq 1, \quad \|\boldsymbol{x}_{\text{sim}}(30) - \boldsymbol{x}_{\text{est}}(30)\|_2 \leq 0.2.$$

Provide a script `obsv_tuning_script_cc`, which simulates the system using your chosen poles $\boldsymbol{p}$ and saves the following information as a MAT-file `obsv_tuning_script_cc.mat`:

Variable	Value
`poles`	$\boldsymbol{p}$
`X`	X_{sim} (abs. coordinates)
`X_est`	X_{est} (abs. coordinates)

Table 8: Variables contained in `obsv_tuning_script_cc.mat`.

Hint: For simulation with zero control input, you can obtain `[0, ctrl_info] = ctrl.eval(x)` by using the provided `Controller_constant` controller class. 2 *pt.*

29. Implement a class `MPC_TS_offsetfree` similar to `MPC_TS`, which either

(a) solves problem (19) in delta-coordinates shifted by the current (estimate of) the set point $\boldsymbol{x}_s$, $\boldsymbol{u}_s$, or

(b) solves problem (19) in absolute coordinates according to Lecture 7, slide 31,

such that offset-free steady-state tracking can be ensured. Crucially, note that the `eval` method of the controller takes as inputs the (estimates of the) current state $\boldsymbol{x}_{\text{est}}$, disturbance $\boldsymbol{d}_{\text{est}}$ and set point $\boldsymbol{x}_s$, $\boldsymbol{u}_s$ to compute the control input $\boldsymbol{u}$ in **absolute** coordinates. 3 *pt.*

30. *[Self-Study]: Experiment with the following settings for closed-loop simulation:*

    (a) different true initial conditions,

    (b) different estimated initial conditions,

    (c) different observer poles and (noisy) disturbance sequences.

    Does the controller always guarantee offset-free tracking? If not, why not? How could the reliability of the controller be improved?

31. *[Self-Study]: In order to deploy your controller on the real system you are usually required to implement the MPC on low-cost embedded hardware. To this end, it is important to ensure computational efficiency and to implement the MPC 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. Investigate if the computation times of your controllers can be improved by switching to FORCES Pro for solving the MPC problems. For your own investigation, you may sign up at* `https://my.embotech.com/auth/sign_up`.