EXERCISE 7
Adaptive control and MPC

Tue Herlau
tuhe@dtu.dk

18 March, 2022

Objective: Today's exercise will address the case where the dynamics of the model is not known but has to be learned from experience. The dynamics will as a rule be non-linear, and we will therefore consider local regression as our primary learning method due to it's stability/robustness. We will consider several variants of this idea, and then consider a real-world implementation known as LMPC. Please note that the LMPC example is somewhat involved and represents some of the complications getting the methods to work on real-world systems. (35 lines of code)
Material: Obtain exercise material from our gitlab repository at
`https://gitlab.gbar.dtu.dk/02465material/02465students`

Contents

1 Adaptive control and simple agents (`lqr_learning_agents.py`)

In this section, we will consider a suit of agents which all learn the dynamics and plan a solution using the learned dynamics. For simplicity, we will apply all the methods to the boing-problem from [Her21, section 14.1.3].

Recall the Boing problem defines the airspeed y_1 and climb rate y_2 as:

$$\boldsymbol{y}_k = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \underbrace{\begin{bmatrix} 1. & 0. & 0. & 0. \\ 0. & -1. & 0. & 7.74 \end{bmatrix}}_{=P} \boldsymbol{x}_k \tag{1}$$

We will consider the problem of changing the airspeed to $y_1 = 10$ (recall the speed is relative to the current velocity) which can ben implemented using the cost-function

$$J(\boldsymbol{x}_0) = \sum_{k=0}^{N-1} \left(\frac{1}{2} \|\boldsymbol{y}_k - \boldsymbol{y}^*\|^2 + \frac{1}{2} \|\boldsymbol{u}_k\|^2 \right) .0 \tag{2}$$

In the boing-example, the dynamics is assumed to be linear, i.e. of the form $\boldsymbol{x}_{k+1} = A\boldsymbol{x}_k + B\boldsymbol{u}_k$. The concrete form of the matrices A, B can be found in the lecture notes, however, we will consider them to be unknown in the following and therefore consider methods which learn them from data.

Since the boing-problem is linear, and all the methods we consider involve estimation of a linear (possible local approximation) of the dynamics, we expect all the methods to work well and yield roughly equal results. This is of course less interesting (since we will not see a difference in what the models do), however, it makes it easy to verify the implementation. Later we will consider non-linear problems.

For your implementation, I have supplied code which performs the linear regression using a format appropriate for us. You can find it in the file `regression.py`, and the main point is the input matrices are assumed to stack observations horizontally:

```
# regression.py
def solve_linear_problem_simple(Y, X, U, lamb=0):
    """ Implements the solver for the basic linear regression problem.
    The method solves a problem of the form:

    > x_{k+1} = A x_k + B u_k + d

    using the more familiar naming convention
    > Y = A X + B U + d.

    Assuming there are N observations, and x_k is n-dimensional, the conventions are that
    A is n x n dimensional, Y is n x N dimensional, X is n x N dimensional, and so on (i.e. observations are in the horizontal dimension).
```

Similarly, for the buffer-datastructure, I have included a very basic implementation which supplies the following functionality:

```
# lqr_learning_agents.py
class Buffer:
    def __init__(self):
        self.x = []
```

```
        self.u = []
        self.xp = []

    def push(self, x, u, xp):
```

1.1 Part A: Linear MPC

Our first agent is the most simple: We collect all data $\boldsymbol{x}_k, \boldsymbol{u}_k$ in a buffer, use this to estimate A, B using linear regression, and then we plan the actions using LQR. This idea will not work for a non-linear problem, but it should work for the Boing problem.

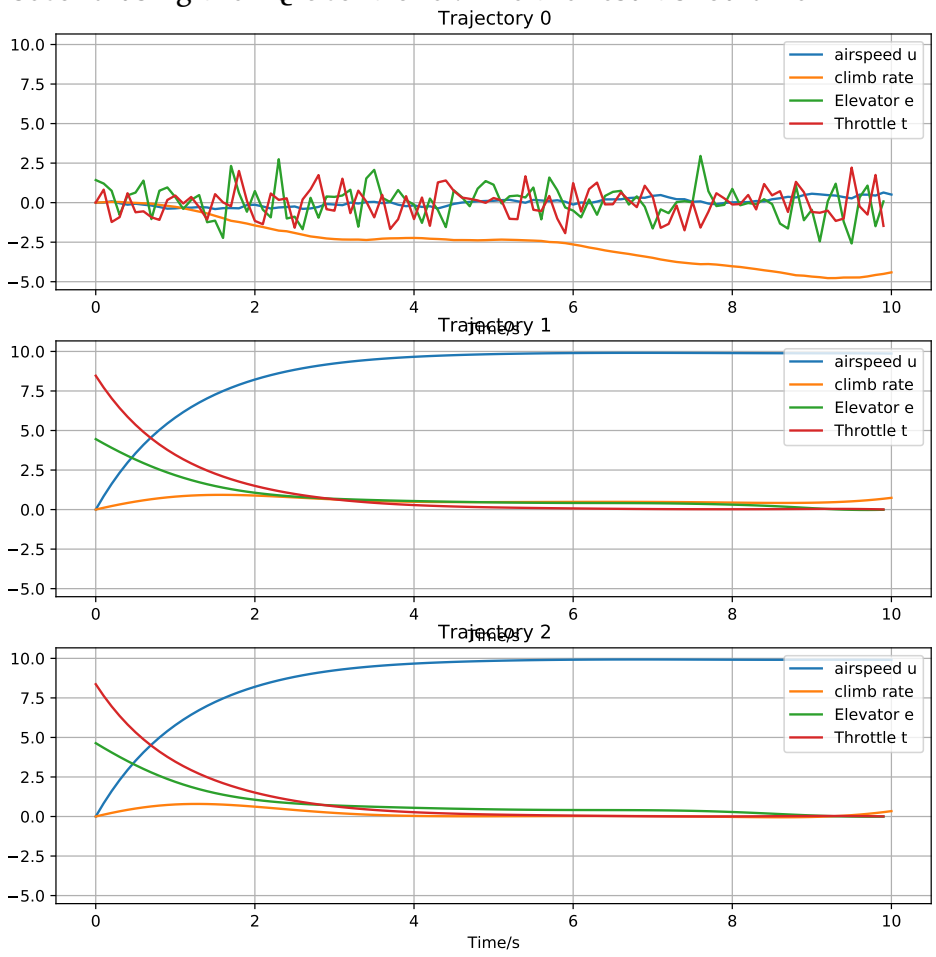
The concrete algorithm is a simplification of [Her21, algorithm 26], and will work similarly to the LQR agent with a few simplifications

- Instead of planning on a short horizon N, we plan on the full horizon consisting of $\frac{T^{\max}}{\Delta}$ steps.
- The agent returns random actions in the first trajectory since it has no control matrices
- At the second (and subsequent) trajectory, the agent re-computes A, B, d at time $t = 0$, then obtain $L_k, \boldsymbol{l}_k$ using LQR

Problem 1 *Basic learning LQR*

Complete the implementation of the basic learning LQR-agent. Your implementation follows [Her21, algorithm 26] but with the simplifications mentioned above. The final implementation will be very similar to the LQR agent from last week, except the matrices A, B, are estimated using linear regression. You should use the function `solve_linear_problem_simple` from the file `regression.py` for this task.

Info: The code runs for two episodes/trajectories. The first using random actions, the second using the LQR controller. The the result should be



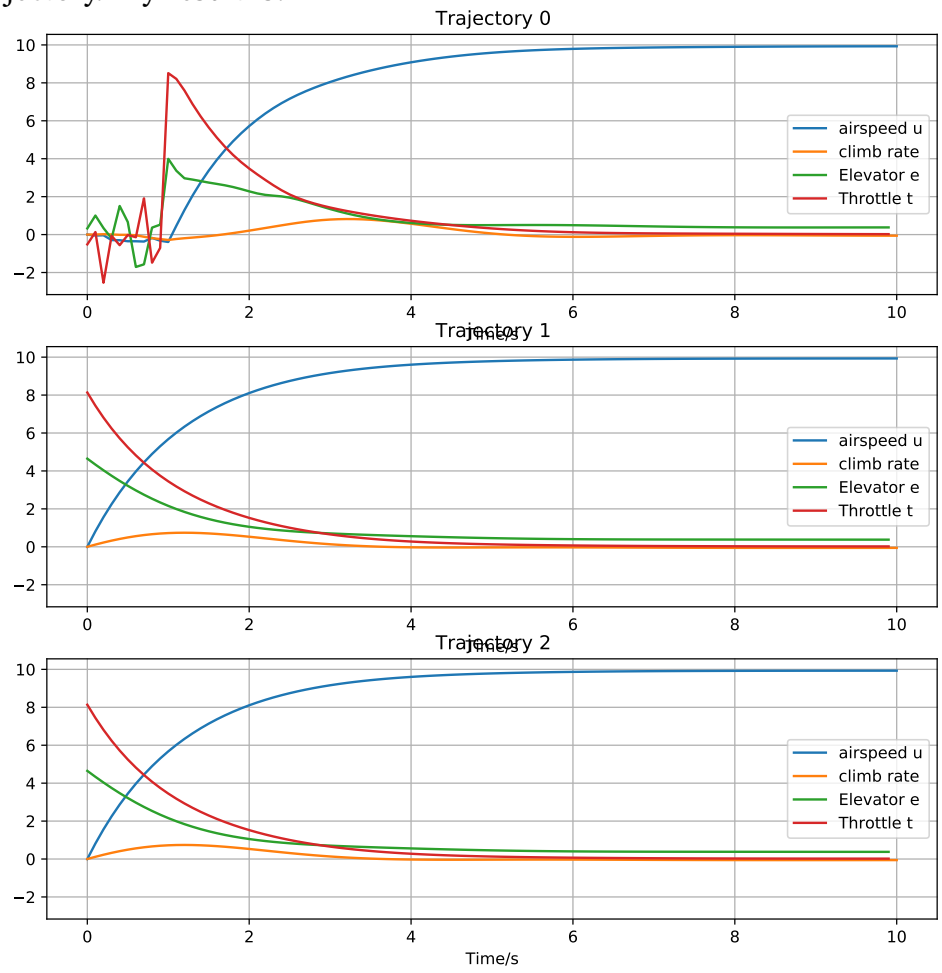
1.2 Part B: Basic LQR and MPC

We will now extend the previous agent to include MPC, i.e. implement [Her21, algorithm 26]. The main point is that instead of computing the control matrices once at the start of the trajectory, they should now be computed at every step, but at a shorter horizon.

Problem 2 *Learning LQR and MPC*

Complete the implementation of the learning LQR-MPC agent, [Her21, algorithm 26]. You can re-use most of the existing code for the basic agent.

Info: The basic MPC-LQR agent now learns online, and could be trained on a single trajectory. My result is:



Detour: The pendulum (`mpc_pendulum_experiment_lqr.py`) Let's check if the method works by applying it to the pendulum problem. Since you already wrote the code, it should just be a matter of instantiating your agent and running it on the problem. Run it using the parameters:

```
# mpc_pendulum_experiment_lqr.py
    L = 12
    neighboorhood_size = 50
    min_buffer_size = 50
```

Problem 3 *Learning LQR, MPC and local approximations*

Instantiate your local MPC/LQR agent and run it with the parameters given above. The agent should swing up the pendulum in a few episodes.

Info: The result will be variable in terms of the number of episodes etc. In my experience, about four episodes is enough.

1.3 Part C: Basic LQR + MPC using local regression

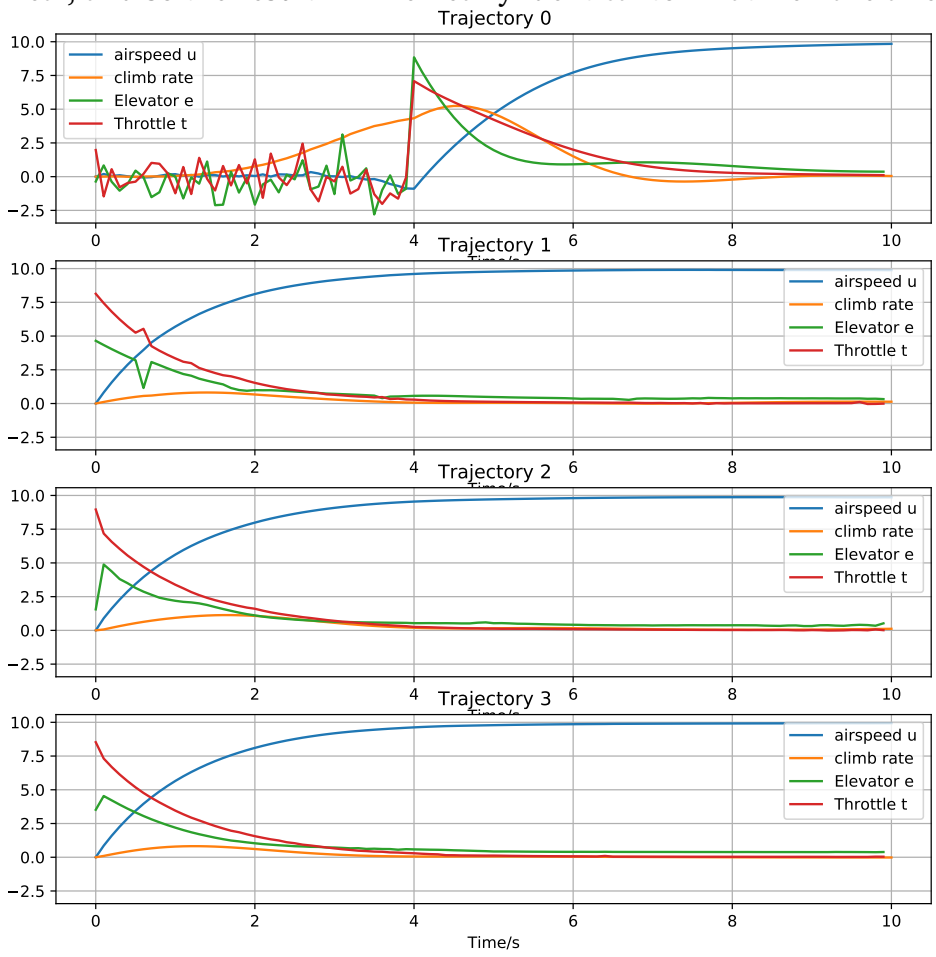
The previous implementation has two defects: It will not scale well with a lot of data and secondly, it will not generalize to non-linear problems. We can fix the first problem (at least partly) by simply selecting the (perhaps) 50 closest observations to the *current* $\boldsymbol{x}_k, \boldsymbol{u}_k$ position, however, for non-linear problems the dynamics at $\boldsymbol{x}_{k+\delta}, \boldsymbol{u}_{k+\delta}$ will change in the future and the local matrices need to be estimated based on where we (approximately) believe the system will be in the future.

This is described in [Her21, algorithm 28], and the method of using local estimation of where the system will be will be quite similar to iterative LQR.

Problem 4 *Learning LQR, MPC and local approximations*

Complete the implementation of the learning LQR-MPC agent which uses local approximations as described in [Her21, algorithm 28].

Info: The outcome will be pretty boring since the dynamics of the boing experiment is linear, and so the result will be nearly identical to what we have already seen:



Detour: The pendulum (`mpc_pendulum_experiment_lqr.py`) You can check your code on the pendulum swingup task which should work. I use the parameters:

```
# mpc_pendulum_experiment_lqr.py
    L = 12
    neighboorhood_size = 50
    min_buffer_size = 50
```

Problem 5 *Learning LQR, MPC and local approximations*

Instantiate your local MPC/LQR agent and run it with the parameters given above. The agent should swing up the pendulum in a few episodes.

Info: The result will be variable in terms of the number of episodes etc. In my experience, about four episodes is enough.

2 Congratulations! You made it through control!

The next problems requires a linear-quadratic optimizer to work. This can be a bit tricky to install, and I think it is perfectly sensible to leave it to the imagination unless you are extra curious!

3 Local MPC and optimization (`learning_agent_mpc_optimize.py`) ✸✸

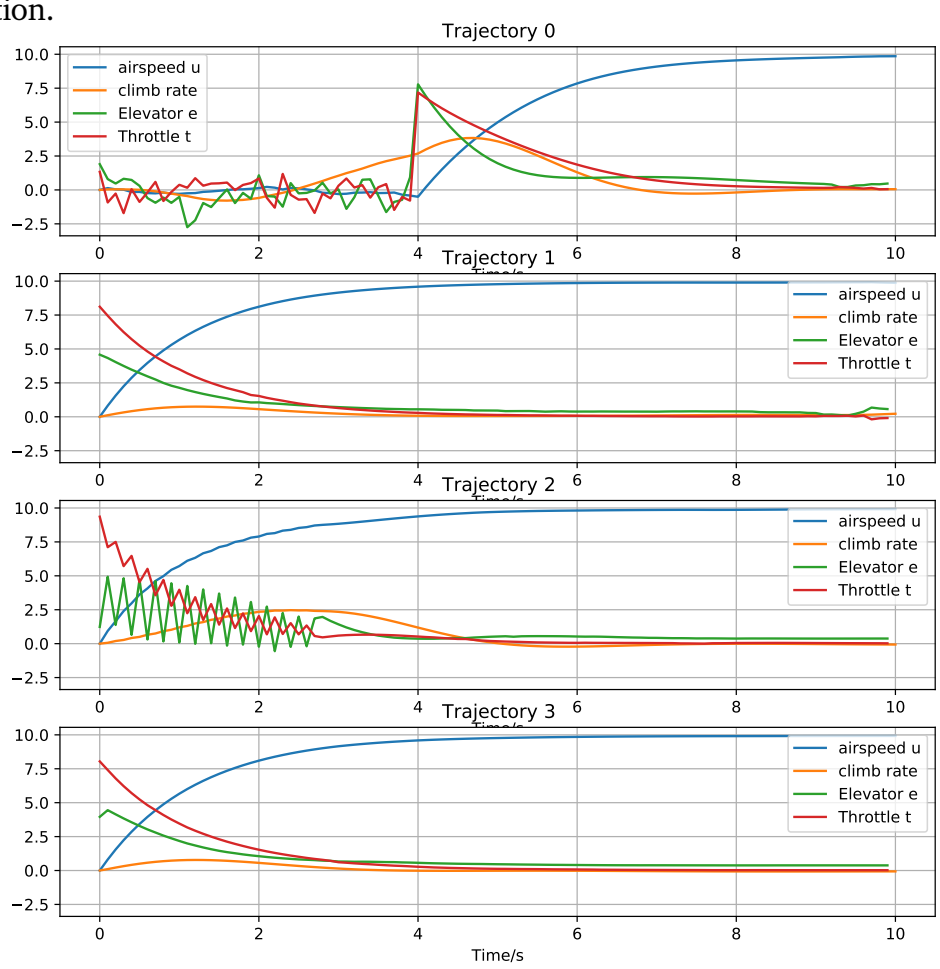
The advantage of using LQR is that it is very fast, the disadvantage is that constraints, especially action-constraints, are not handled well. We can fix that by using the optimization method described in [Her21, algorithm 29]; i.e. we will only change the solve-method in the implementation to optimize rather than use LQR. See `https://colab.research.google.com/github/cvxgrp/cvx_short_course/blob/master/intro/control.ipynb#scrollTo=WeoGIbrpb7zC` for further details.

For the actual solver we will use the simple yet fast CVX library. The CVX library implements the optimization problem using a (symbolic) representation which is very intuitive to use, and then transform the representation into an optimization problem which can be solved efficiently. Unfortunately, it only works for linear-quadratic problems with linear constraints which is why we could not use it two weeks ago. For this problem, I have implemented all functionality except the construction of the cost function.

Problem 6 *Local learning MPC and optimization*

Complete the implementation of the optimization-based approach described in [Her21, algorithm 29]. It should be a matter of adding a single new line (defining the objective) and you are very welcome to inspect the rest of the code.

Info: Once more there is little new to report since it (again) simple find the optimal solution to the Boing problem in roughly the same time as the LQR-based implementation.



Detour: The pendulum (`mpc_pendulum_experiment_optim.py`) You can check your code on the pendulum swingup task with the optimization-based approach you just implemented. I use the parameters:

```
# mpc_pendulum_experiment_optim.py
    L = 12
    neighboorhood_size=50
    min_buffer_size=50
```

Problem 7 *Learning with MPC, optimization and local approximations*

Instantiate your local MPC/optimization agent and run it with the parameters given above. The agent should swing up the pendulum in a few episodes.

Info: The result will be variable in terms of the number of episodes etc. In my experience, about four episodes is enough, and there does not appear to be a substantial difference compared to the LQR-based approach.

4 Learning MPC for controlling a race car (`lmpc_run.py`) ★★

This exercise was included in last years problem set, but has been superseded by the previous exercise. I have included it if some wishes to do a LMPC-based project and you are welcome to try it out, but since the lecture used to contain hints it is likely going to be quite difficult. The solution is available online on gitlab.

In this exercise we will see how LMPC based on safe sets can be used to minimize the lap time of a race car. The code we will be using is based on the actual research code[1] from the BARC project, see [RCB17]. A video of the controller applied to a real car can be found here: `https://www.youtube.com/watch?v=pB2pTedXLpI`.

Our version of the code has been simplified and adapted to the course. That is, the dynamical model is implemented more compactly in sympy, and the data structure in the code has been streamlined. Despite this, note that the code is still fairly complex and we will here provide a reading guide to the main files and functions:

`lmpc_run.py` This file is used for running the whole thing, which includes loading the various scripts, setting parameters and calling the controller.

`sympc/lmpc_agent.py` Contains the actual controller, where the optimization problem is formulated and solved. You will have to implement some of the matrices and vectors used in the optimization problem and edit this file.

`lmpc_sample_safe_set.py` Contains the methods use to find the safe set points, and use them to construct a linearised dynamic model used in the optimization. You will not have to implement anything here, but it might be useful to understand how the safe set is implemented in practice.

`fnc/` This folder contains the reference implementation with a minimum of changes. We will use the reference implementation to compute the matrices A, G, b, h from the optimization problem to allow you to check the correctness of the implementation. That is, when the code does "self-checks", it checks that the output of your program matches the original implementation.

4.1 Theoretical questions

Before we dig into the code, we will do a few theoretical considerations.

[1] see `https://github.com/urosolia/RacingLMPC`

Problem 8

Consider why regular MPC, without safe sets, is not suitable for this task. Under what conditions *would* regular MPC be suitable for the task? Hint: Consider the finite control horizon.

In order to initialise a feasible lap, a simple PID (Porportional integral derivative) controller is employed to drive the car at low speed around the centre of the track. A second lap is performed using a Linear time-varying MPC, for a total of two laps. The obtained states (position, orientation, velocity and steering) are then used as an initial safe set.

Problem 9

What would happen if LMPC with a control horizon of 1 time step was used to control the car? How does increasing the control horizon improve the controller?

4.2 Implementing the LMPC optimization problem

The idea behind the safe sets is that they ensure feasible solutions (Since we end up places from where we have succeeded previously), and that they provide estimates of how far we are from attaining our goal (finishing a lap). However, it is not necessary to end up *exactly* where we have been before, it is enough to just be within something we have tried.

Therefore the safe set restriction can be relaxed, and instead a solution 'inside' the safe set is enough. This really corresponds to requiring that the solution is within an the convec hull of the safe set points, as described by lecture 5 eq. (**??**).

Problem 10

Carefully study the optimization problem which arises from the learning-MPC formulation. Consider the role of $\boldsymbol{x}$, $\boldsymbol{u}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\rho}$ in the formulation given in the notes.

In the end, the problem should be on the form of a general quadratic program:

$$\frac{1}{2}\arg\min_{\boldsymbol{z}} \boldsymbol{z}^\top P\boldsymbol{z} + \boldsymbol{q}^\top \boldsymbol{z} \tag{3}$$

$$\text{s.t.} \tag{4}$$

$$A\boldsymbol{z} = \boldsymbol{b}, \tag{5}$$

$$G\boldsymbol{z} \leq \boldsymbol{h}, \tag{6}$$

where $\boldsymbol{z}$ includes all variables that have to be estimated. In our case, this is the predicted states, control actions, weights on safe set states and slack, $\boldsymbol{z} = (\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{\alpha}, \boldsymbol{\rho})$. The

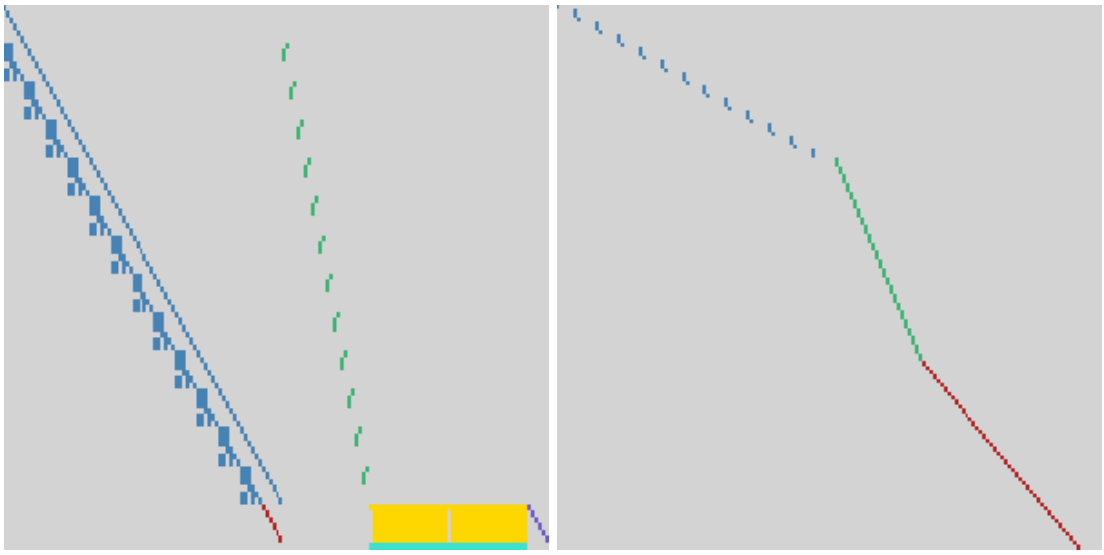


Figure 1: Structures of A- and G-matrix

difficulty lies in computing A, G, $\boldsymbol{b}$ and $\boldsymbol{h}$. Consider Figure 4.2, which shows the block structures of A and G according to

$$\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{u} \\ \boldsymbol{\alpha} \\ \boldsymbol{\rho} \end{bmatrix} = \begin{bmatrix} A_{1,1}\boldsymbol{x} + A_{1,2}\boldsymbol{u} + A_{1,3}\boldsymbol{\alpha} + A_{1,4}\boldsymbol{\rho} \\ A_{2,1}\boldsymbol{x} + A_{2,2}\boldsymbol{u} + A_{2,3}\boldsymbol{\alpha} + A_{2,4}\boldsymbol{\rho} \\ A_{3,1}\boldsymbol{x} + A_{3,2}\boldsymbol{u} + A_{3,3}\boldsymbol{\alpha} + A_{3,4}\boldsymbol{\rho} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b_1} \\ \boldsymbol{b_2} \\ \boldsymbol{b_3} \end{bmatrix}$$

$$\begin{bmatrix} G_{1,1} & G_{1,2} & G_{1,3} & G_{1,4} \\ G_{2,1} & G_{2,2} & G_{2,3} & G_{2,4} \\ G_{3,1} & G_{3,2} & G_{3,3} & G_{3,4} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{u} \\ \boldsymbol{\alpha} \\ \boldsymbol{\rho} \end{bmatrix} = \begin{bmatrix} G_{1,1}\boldsymbol{x} + G_{1,2}\boldsymbol{u} + G_{1,3}\boldsymbol{\alpha} + G_{1,4}\boldsymbol{\rho} \\ G_{2,1}\boldsymbol{x} + G_{2,2}\boldsymbol{u} + G_{2,3}\boldsymbol{\alpha} + G_{2,4}\boldsymbol{\rho} \\ G_{3,1}\boldsymbol{x} + G_{3,2}\boldsymbol{u} + G_{3,3}\boldsymbol{\alpha} + G_{3,4}\boldsymbol{\rho} \end{bmatrix} = \begin{bmatrix} \boldsymbol{h_1} \\ \boldsymbol{h_2} \\ \boldsymbol{h_3} \end{bmatrix}$$

Make sure that you understand all parts of A and G before you continue.

Problem 11

Implement the matrices, A and G, and the vectors $\boldsymbol{b}$ and $\boldsymbol{h}$ in "getQP". Hint: Split the implementation of A into three parts: 1) enforcing the dynamic model and the effect of inputs, 2) forcing the terminal state into a linear combination of safe set points (including slack) and 3) enforcing that the weights on safe set points sums to one. Similarly, the implementation of G can be split into three parts, consisting of the requirements on $\boldsymbol{x}$, on $\boldsymbol{u}$ and on $\boldsymbol{\alpha}$.

Info: The values of the matrices A, G and the vectors $\boldsymbol{b}$, $\boldsymbol{h}$ are compared to those in my implementation, and differences results in the code raising an error. To see were you might have some problems, the code also produces the sparsity pattern of your implementation, which you can compare to those shown in Figure .

Problem 12

Now that you have (hopefully) finished the implementation it is time to run it. Run the script and look at how the race car (driver?) improves over time. Describe what is going on in the animation.

Info: You should get lap times similar to:

```
Lap time at iteration 0 is 30.8 seconds
Lap time at iteration 1 is 31.3 seconds
Lap time at iteration 2 is 16.8 seconds
Lap time at iteration 3 is 11.4 seconds
Lap time at iteration 4 is 9.200000000000001 seconds
Lap time at iteration 5 is 8.3 seconds
Lap time at iteration 6 is 8.200000000000001 seconds
Lap time at iteration 7 is 8.200000000000001 seconds
Lap time at iteration 8 is 7.9 seconds
Lap time at iteration 9 is 7.7 seconds
Lap time at iteration 10 is 7.9 seconds
Lap time at iteration 11 is 8.3 seconds
```

References

[Her21] Tue Herlau. Sequential decision making. (See **02465_Notes.pdf**), 2021.

[RCB17] Ugo Rosolia, Ashwin Carvalho, and Francesco Borrelli. Autonomous racing using learning model predictive control. In *2017 American Control Conference (ACC)*, pages 5115–5120. IEEE, 2017. (See **rosolia2017.pdf**).