Model Predictive Control (H0E76A)

Exercise Session 5 | State estimation

leander.hemelhof@kuleuven.be
renzi.wang@kuleuven.be
brecht.evens@kuleuven.be

2024–2025

1 Introduction

In this session, we will be implementing and comparing a few different state estimation methods (*Extended Kalman Filter, Moving Horizon Estimation*). To this end, we will be using *CasADi*[1], which is a very useful tool for solving optimization problems – especially optimal control problems.

The dynamics under investigation are based on the example given in the slides of lecture 7. We consider the *batch chemical reactor*, which is described by the following equations

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\begin{bmatrix} C_A \\ C_B \\ C_C \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 1 & -2 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} \kappa_1 C_A - \kappa_{-1} C_B C_C \\ \kappa_2 C_B^2 - \kappa_{-2} C_C \end{bmatrix},$$

where C_i is the concentration of reactant $i \in \{A, B, C\}$ at a given time. These concentrations make up the state vector $x = (C_A, C_B, C_C)$. As an output, one can only measure the sum of the three concentrations: $y = 32.83(x_1 + x_2 + x_3)$. Since concentrations cannot be negative, the system must satisfy the state constraint $x \geq 0$ at all times.

A simulation of the trajectories starting from $x_0 = (0.5, 0.05, 0.0)$ with the associated MHE estimates is depicted in fig. 1

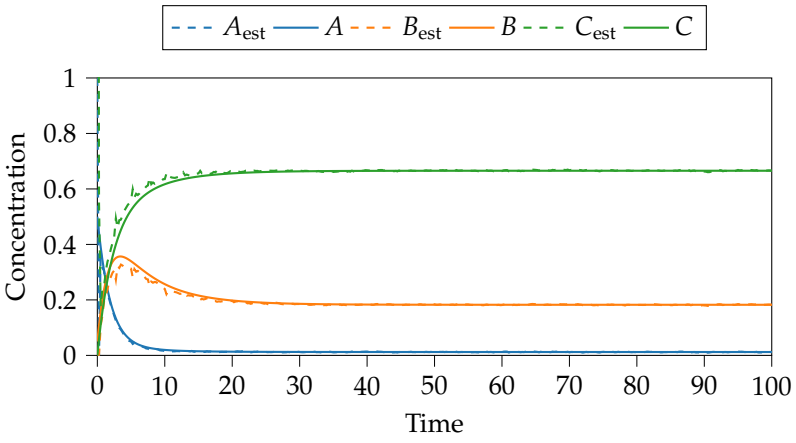

Figure 1: Concentration trajectories over time and their MHE estimates.

1.1 Simulation in Python

You will require two independent implementations of the dynamics. One using `numpy` and another using `casadi` for simulation and optimization respectively.

To generate the noise, we use `numpy.random` to create a realization of the model error w and measurement noise v. For the remainder of the session, we will use $\sigma_w = 0.002$ and $\sigma_v = 0.25$ for the noise standard deviations. We discretize the dynamics using `rk4`.

[1] https://web.casadi.org/

The `numpy` implementation is then as follows:

Code 1 | Example implementation of the dynamics using Numpy.

```
import numpy.random as npr
from rcracers.simulator.core import rk4

NX, NY = 3, 1

@rk4(Ts=0.25)
def dynamics(x):
    k1, km1, k2, km2 = 0.5, 0.05, 0.2, 0.01
    ca, cb, cc = x[0], x[1], x[2]
    r1, r2 = k1 * ca - km1 * cb * cc, k2 * cb**2 - km2 * cc
    res = (-r1, r1 - 2 * r2, r1 + r2)
    return np.array(res)

def measure(x):
    return 32.84 * (x[0] + x[1] + x[2])

f = lambda x: np.maximum(
    dynamics(x) + 0.002 * rg.normal(size=(NX,)),
    np.zeros(NX)
)
h = lambda x: measure(x) + 0.25 * rg.normal(size=(NY,))
```

We can turn the `dynamics` and `measure` methods into symbolic ones using `casadi` as follows.

Code 2 | Example implementation of the dynamics using Casadi.

```
import casadi as cs

x, w = cs.SX.sym("x", NX), cs.SX.sym("w", NX)
fs = cs.Function("f", [x, w], [dynamics(x) + w])
hs = cs.Function("h", [x], [measure(x)])
return fs, hs
```

The code depicted above is implemented in `problem.get_system_equations`. You can reproduce the above behavior using:

Code 3 | Getting the dynamics.

```
from problem import get_system_equations
f, h = get_system_equations(noise=(0.002, 0.25))
fs, hs = get_system_equations(symbolic=True, noise=True)
```

The `numpy` dynamics are used in the provided `simulate` method, which can be executed as follows (see also `template.py`).

Code 4 | Simulating the dynamics.

```
from problem import simulate, ObserverLog, default_config

# mhe (place holder)
mhe = lambda y, log: None

# problem setup
cfg = default_config()

# prepare log
log = ObserverLog()
log.append("x", cfg.x0_est)  # add initial estimate

# simulate
f, h = get_system_equations(noise=(0.0, cfg.sig_v), Ts=cfg.Ts, rg=cfg.rg)
x = simulate(cfg.x0, f, n_steps=400, policy=mhe, measure=h, log=log)
```

Now you can complete the following exercise

Exercise 1 | Extend `template.py` with plotting code to depict the trajectories returned by the `simulate` method. Make sure to include labels and a legend. You can use `matplotlib` to do so. Compare the result with fig. 1.

2 Exercises

2.1 Extended Kalman Filtering

As a first state estimation scheme we consider the *Extended Kalman Filter (EKF)*, the update equations of which are described in the slide. As a first step write out the computations required to do the update:

Exercise 2 | Write down the computations that need to be done in the different steps that make up the Extended Kalman Filter.

You should have realized that you will require linearizations of the discrete-time dynamics. Even though this could be done by hand, we provide `get_linear_dynamics` for convenience:

Code 5 | Linearizing the dynamics.

```
from problem import get_system_equations, get_linear_dynamics
fs, hs = get_system_equations(symbolic=True, noise=True)
dfdx, dfdw, dhdx = get_linear_dynamics(fs, hs)
```

Now it is time for the real deal:

Exercise 3 | We will now implement the basic EKF.

1. Implement the equations you worked out in Exercise 2 into the `__call__` method of the `EKF` class in `template.py`. You can use `__init__` to pass parameters like the initial guess and Q, R and P matrices. These can be initialized as $\sigma_i^2 I$ with $\sigma_p = 0.5$, $\sigma_w = 0.002$ and $\sigma_v = 0.25$. As the initial state use the bad guess $\bar{x}_0 = (1, 0, 4)$. Adjust the `part_1` method in `template.py` to pass these values to your implementation.

2. Compare the estimated states with the actual ones. Do they converge?

   To record the state estimates you will need to use the `log` method passed to the `__call__` method by the simulator. Some dummy code is already provided which you can replace.

   ```
   # log the state estimate and the measurement for plotting
   log("y", y)
   log("x", np.zeros(3))
   ```

   Pass the state estimate you compute instead of `np.zeros(3)` in this last line. The resulting state trajectories will be present in your `log` instance and can be accessed:

   ```
   print(log.x.shape)
   ```

   Look for a similar line in the `part_1` method and pass `log.x` to your plotting method.

Exercise 4 | Since the state constraints are quite simple, we may attempt to improve the result by enforcing them after the fact. That is, we can clip the state estimates to zero to make sure that they are never negative.

Check if this improves the results. You may want to switch to a logarithmic y-axis or longer simulation horizon. To do so you can alter the value of `n_steps`:

```
x = simulate(cfg.x0, f, n_steps=400, policy=mhe, measure=h, log=log)
```

2.2 Moving Horizon Estimation

A second, more advanced approach we will consider is *Moving Horizon Estimation (MHE)*.

Exercise 5 | Write down the optimization problem that needs to be solved each time a new measurement arrives.

1. What are the problem parameters you will need to pass to the MHE?
2. How many of the past measurements should you keep? What should happen when you don't have enough yet?

To save time, a method to build the optimization problem using `casadi` is already provided. You can use it as follows:

Code 6 | Generating the MHE problem

```
fs, hs = get_system_equations(symbolic=True, noise=True)
loss = lambda w, v: w.T @ w + v.T @ v
solver = build_mhe(loss, fs, hs, 10, lbx=0.0, ubx=10.0)

# you can solve the optimization problem as follows:
x, w = solver(y=np.zeros((10, 1)))
```

Verify whether the implementation matches your optimization problem.

Exercise 6 | Now we can implement the MHE:

1. Change the example implementation of `loss` given above to the correct one.
2. Similarly to before you will have to extend the `MHE` class provided in `template.py`. You can set the horizon length to $N = 10$ at first. Use `log` to store your estimates.
3. Keep track of the measurements correctly, keeping in mind that you start off with just one.
4. Run the simulation and verify whether the states are tracked correctly. What happens when you change the horizon length to $N = 25$? Does it perform better than the EKF?