Model Predictive Control (H0E76A)

Exercise Session 4 | Nonlinear model predictive control

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

2024–2025

1 Introduction

In this session, we will consider the problem of autonomous parking, see fig. 1. This is a relatively complex maneuver which is very difficult to carry out using linear control strategies. However, as we will see, nonlinear MPC is a very effective tool to tackle this challenge.

Due to the nonlinearity of the controller, the optimal control problems we obtain will no longer be convex and therefore, we can no longer use cvxpy to build our optimal control problem. For this reason we will move to CasADi for this (and future) sessions.

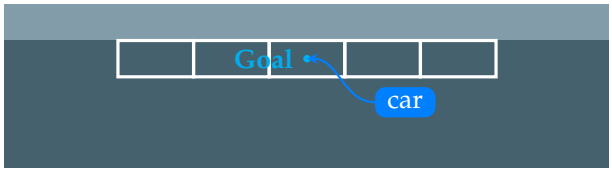

Figure 1: Control task.

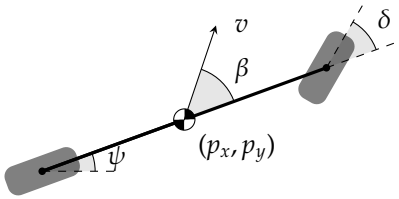

Figure 2: Bicycle model quantities.

We will use the so-called *kinematic bicycle model* to describe the motion of the vehicle. We describe the state of the vehicle by

$$x = (p_x, p_y, \psi, v)$$

with (p_x, p_y) the position of the vehicle's center of mass in a fixed cartesian coordinate system, ψ the heading angle of the vehicle and v the longitudinal velocity, as illustrated in fig. 2. The inputs the vehicle are the drive d and the steering angle δ in radians.

The kinematic bicycle model is given as:

$$\begin{cases} \dot{p}_x = v\cos(\psi + \beta) \\ \dot{p}_y = v\sin(\psi + \beta) \\ \dot{\psi} = \frac{v}{l_r}\sin(\beta) \\ \dot{v} = ad - \mu v \end{cases}$$

with

$$\beta = \arctan\left(\frac{l_f}{l_f + l_r}\tan(\delta)\right).$$

a, μ, l_r and l_f are model parameters, which we have already identified, among with some other data of the vehicle (see `given/parameters.py`).

Notice that the dynamics consists of a system of ordinary differential equations (ode). The first step is thus to discretize the system in a manner that suitably trades off accuracy with computational complexity. We will implement two widely used numerical integration schemes, which will result in a discrete-time dynamical system, which we can use to design our controller.

Then, we will build the optimal control problem and experiment with some different variants – first in a simple case without any nonlinear constraints, and then extending the setting to a full parking maneuver with obstacle avoidance constraints.

2 Discretization

We have already made extensive use of the forward Euler integration method. However, as you probably know, forward Euler may be quite inaccurate. To this end, we will implement the Runge-Kutta order 4 (RK4) scheme:

$$\begin{aligned} x_{t+1} &= x_t + \tfrac{T_s}{6}(s_1 + 2s_2 + 2s_3 + s_4), \text{ with} \\ s_1 &= f(x, u) \\ s_2 &= f(x + \tfrac{T_s}{2} s_1, u) \\ s_3 &= f(x + \tfrac{T_s}{2} s_2, u) \\ s_4 &= f(x + T_s s_3, u) \end{aligned}$$

and compare the quality of the solutions.

Exercise 1 | Implement the Forward Euler and RK4 scheme, and run a closed-loop simulation with the given policy (see `exercise 1` in the template file). Plot the predicted positions over time using the different integrators and plot the norm of the error (w.r.t. `odeint` from scipy as a reference solution). Which problem parameters play an important role in determining the accuracy of an integrator and why?

Solution

See the solution code for the implementation. We can observe a few things from the results:

- The further we predict, the worse the accuracy becomes. This is to be expected since every next state is computed based on the (erroneous) prediction of the current state. Hence, the errors compound.
- The smaller the sampling time, the better the predictions become.
- The larger the acceleration, the larger the errors. The larger the acceleration, the more the velocity will change over the sampling period. As a result, the error in the Forward Euler method resulting from keeping the velocity constant will also be larger as the acceleration increases.
- Finally, it's interesting to note that the regularity of the dynamics plays an important role, although this is not directly observable from the current experiment. This is discussed extensively in other courses, so we will not get into detail about this here.

3 Optimal control problem

We're now ready to build our optimal control problem.

We consider again the standard quadratic cost function

$$V(x, u) = \sum_{k=0}^{N-1} x_k^\top Q x_k + u_k^\top R u_k + x_N^\top Q_N x_N,$$

with $x_0 = x$, $x_{k+1} = f(x_k, u_k)$ for $k = 0, \dots, N-1$. For convenience, we will take $Q_N = 5Q$.

There are physical limits on the actuation of the vehicle, and we assume that the vehicle state stay within bounds as well:

$$\begin{aligned} u_{\min} \leq u_k \leq u_{\max}, \quad k = 0, \dots, N-1 \\ x_{\min} \leq x_k \leq x_{\max}, \quad k = 0, \dots, N-1, \end{aligned}$$

see the given parameters for the numerical values.

We will implement our controller using a *sequential approach* (a.k.a. single-shooting) so that x_t will be represented as a nonlinear function of the initial state (which is a parameter) and the inputs up to time step k, namely

$$x_{k+1} = \bar{f}_{k+1}(x_0, u_0, \dots, u_k) = f(\cdot, u_{k-1}) \circ \cdots \circ f(x_0, u_0).$$

Exercise 2 | We will select $Q = \text{diag}(1, 3, 0.1, 0.01)$ and $R = \text{diag}(1, 0.01)$. Suppose we did not particularly care more about certain state variables than others, what other motivations do you see for choosing nonuniform values in Q and R?

Solution

The state variables are expressed in different units. An entry that takes very large values will tend to dominate the MPC cost, even for relatively small deviations. Q and R can be used to compensate for this. Normalizing all the variables through Q and R also tends to help numerical stability. In this case, we are using a miniature race cars (which account for the small values in positions) and we have states that are in the following ranges

Positions	0.5	[m]
Velocities	0.5	[m/s]
Angles	$\frac{\pi}{4} \approx 0.78$	[rad]

These values are quite well-balanced, so tuning is intuitive in this case. However, considering a full-sized car, positions typically span the orders of 10m, whereas angles remain in the same range as they are here. This typically necessitates the tuning to start from a prescaled version of Q and R that compensates the differences in the units.

Exercise 3 | Implement the `build_ocp` method of the `MPCController` class in the provided template. Use forward Euler discretization and set $T_s = 0.05$ and $N = 60$. The method `build_ocp` is in charge of creating symbolic CasADi variables, constructing the cost function and setting the constraints. See the inline documentation for more details on the expected inputs and outputs. See App. A for some background on how to use CasADi. Plot the control actions of the solution and run a simulation to obtain the resulting state sequence. Simulate the same sequence of control actions using the more accurately discretized model and compare the open-loop state trajectories.

Solution

See solution code. The open loop prediction error increases over the horizon to a value of around 40%, which is not taking any model mismatch into account. This illustrates the risks of open-loop control over long horizons.

Exercise 4 | What would happen if we additionally overestimated the roll friction (this may fluctuate based on the temperature, the road conditions, etc.)? Repeat the previous exercise but for the simulation using the true model, also reduce the `friction` field in `VehicleParameters` by 20%.

Solution

See solution code. The prediction errors now increase to almost 70%. Furthermore, severe overshooting occurs, causing the vehicle to partially drift outside the parking spot.

Exercise 5 | Simulate the MPC controller in closed-loop, both using the dynamics with and without model mismatch. What do you observe?

Solution

The difference between the trajectories is reduced by roughly half as compared to the open-loop case. Furthermore, the overshoot has been reduced significantly.

A Nonlinear programming using CasADi

We will use CasADi as a modelling language for our problem. CasADi is an automatic differentiation library with powerful C-code generation capabilities, which provides bindings to many popular optimization solvers. CasADi expects an optimization problem of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x,p) \\ \text{s.t.} \quad & \underline{x} \le x \le \bar{x} \\ & \underline{g} \le g(x,p) \le \bar{g}, \end{aligned}$$

with x the decision variable, which may be bounded above and below using $\underline{x}$ and $\bar{x}$, and g a possibly nonlinear constraint function with given lower and upper bounds.

To represent this in the code, x is an object of the CasADi `SX.sym` class and f and g are expressions (which can be built from regular mathematical functions that are available in Python). p is a also and object of the `SX.sym` class, but it is not considered a decision variable. Instead, p is a parameter. It is only symbolic during the construction of the problem, but gets assigned a numerical value just before the problem gets solved. This allow us to precompile the solver so that we can efficiently reuse it to solve multiple problems of the same form. In MPC, parameters are usually the initial state.

```
import casadi as cs
import numpy as np

nx, ny = 4, 4
# Make a symbolic variable with name "x" and dimension nx
x = cs.SX.sym("x", (nx,))

# Add upper- and lower bounds to x.
# Note that the absence of a bound is represented by a value of infinity.
ubx = 10 * np.ones(nx)
lbx = -np.inf * np.ones(nx)

# Make a symbolic parameter
p = cs.SX.sym("p", (ny,))

# Define some cost
cost = cs.sum((x - p)**2)

# Define a nonlinear constraint  a <= g(x) <= b
g = 0.25 * x**3.
a = -1.
b = 3.

# Create the solver
nlp = dict(f = cost, x = x, p = p, g = g)

# We will use ipopt as a solver (usually a good bet)
# Set some options to make it less verbose
# (it will otherwise flood your console with printouts)
opts = {"ipopt": {"print_level": 1}, "print_time": False}
solver = nlpsol("my_nlp", "ipopt", nlp)

p_num = np.ones((ny,))
solution = solver(
    p=p_num,   # Set the parameter
    ubg=b,     # Set the bounds for g
    lbg=a,
    lbx=lbx,   # Set the upper and lower bounds on x
    ubx=ubx,
)
# Extract the numerical value of the solution.
x_sol = solution["x"]
```