Exercises for Lecture Course on Numerical Optimal Control (NOC)
Albert-Ludwigs-Universität Freiburg – Winter Term 2025 / 26

Exercise 10: Model Predictive Control with `acados`

Florian Messerer, Dr. Armin Nurkanović, Prof. Dr. Moritz Diehl

In this exercise, we will implement a model predictive control (MPC) based controller, which repeatedly solves optimal control problems (OCP) as part of a feedback loop. The controller will be implemented using `acados`, and as control task we consider the swing-up of a pendulum mounted on top of a cart. We will then further speed up the controller sampling time by using the real time iteration scheme (RTI).
`acados` is a software package providing fast embedded solvers for nonlinear optimal control. Problems are formulated based on CasADi symbolics and the `acados` OCP interface. Among others, `acados` provides numerical integrators for ordinary and differential algebraic equations (ODE resp. DAE), based on both implicit and explicit Runge-Kutta methods, as well as sequential quadratic programming (SQP)-type OCP solvers. The `acados` core library is implemented in `C`, but can be accessed via interfaces to Python, Octave, and Matlab. Given an OCP description, `acados` generates self-contained C code, which – after compilation – can be executed efficiently to solve the problem. For more information, see `https://docs.acados.org/`.

1. Install acados by following the instruction at `https://docs.acados.org/installation/index.html` and the Python interface by following `https://docs.acados.org/interfaces/index.html`.

   To ensure everything works properly, execute `examples/acados_python/getting_started/minimal_example_ocp.py` from the examples folder. This should run without error.

As control task, we consider the swing-up of a pendulum mounted on a cart, as illustrated below:

The state consists of the horizontal position of the cart center p_x, the angle of the pendulum θ, as well as the respective velocities v_x and ω. The pendulum has length l with a mass m mounted at its tip. The cart has mass M and can be actuated by applying a horizontal force F, i.e., $u = F$. The corresponding dynamics are

$$x = \begin{bmatrix} p_\mathrm{x} \\ \theta \\ v_\mathrm{x} \\ \omega \end{bmatrix}, \quad \dot{x} = \begin{bmatrix} v_\mathrm{x} \\ \omega \\ \frac{-ml\sin\theta\omega^2 + mg\cos\theta\sin\theta + F}{M + m(1-\cos^2\theta)} \\ \frac{-ml\cos\theta\sin\theta\omega^2 + F\cos\theta + (M+m)g\sin\theta}{l(M + m(1-\cos^2\theta))} \end{bmatrix} := f(x,u). \tag{1}$$

where g is the gravitational acceleration.
By penalizing deviations of state and control from the origin, the swing-up task can be expressed by the continuous-time OCP

$$\min_{x(\cdot),\,u(\cdot)} \quad \int_0^T \frac{1}{2}x(t)^\top Q x(t) + \frac{1}{2}u(t) R u(t)\mathrm{d}t + \frac{1}{2}x(T)^\top Q x(T) \tag{2a}$$

$$\text{s.t.} \quad x(0) = \bar{x}_0, \tag{2b}$$

$$\dot{x}(t) = f(x(t), u(t)), \quad t \in [0, T], \tag{2c}$$

$$-F_\mathrm{max} \le u(t) \le F_\mathrm{max}, \quad t \in [0, T], \tag{2d}$$

with cost matrices $Q \succeq 0$, $R \succeq 0$.

2. Formulating and solving the OCP with `acados`.

   (a) Complete the templates `cartpole_model.py` and `cartpole_ocp.py` in order to solve OCP (2) with `acados`. A detailed overview of the `acados` OCP interface can be found at `https://github.com/acados/acados/blob/master/docs/problem_formulation/problem_formulation_ocp_mex.pdf`. Specifically, note that:

      i. In general, `acados` expects dynamics defined in implicit form via $f_{\text{impl}}(x, \dot{x}, u) = 0$. Thus, in our case we have $f_{\text{impl}}(x, \dot{x}, u) = \dot{x} - f(x, u)$.

      ii. We use a stage cost of the form `'NONLINEAR_LS'` (nonlinear least squares), i.e., $l(x, u) = \frac{1}{2}\|y - y_{\text{ref}}\|_W$, with $y = F(x, u)$ a (possibly) nonlinear expression depending on x and u. In our case, $y = (x, u)$ is simply the concatenation of state and controls, and $W = \text{blockdiag}(Q, R)$, i.e., the blockdiagonal concatenation of Q and R. Similar considerations hold for the terminal cost.

   (b) Take a detailed a look at the above templates files. On a high level, describe the steps `acados` takes to discretize and solve the OCP. Refer to the concepts learned in the course.

   The OCP is discretized with direct multiple shooting of the horizon T into N control intervals (piecewise constant controls), with an implicit Runge-Kutta method as integrator. The resulting NLP is solved with an SQP method using a Gauss-Newton Hessian approximation. The resulting QP are (partially) condensed and then solved by `HPIPM`.

3. Model predictive control (MPC). Since we can now successfully solve the OCP, we are able to construct an MPC loop around the OCP. For this purpose, we read the first discretized control value u_0 from the OCP solution and apply it to the simulation of the controlled system (when deploying the controller in the real world, this would be the actual physical system). After one simulation step, this results in a new state. We resolve the OCP for this new state, and again employ the resulting u_0, and so on.

   (a) Complete and run the template `cartpole_closed_loop.py`. How long does it take for the MPC controller to return a control input after receiving the current state?

   The solver reports computation times between 0.917 ms and 12.082 ms with a median of 2.901 ms. (Obviously this will differ between different computers.)

   (b) The provided template uses the exact same dynamics for the simulation of the 'real' system as are used in the OCP. This is clearly unrealistic, since in reality our model will never be perfect. Create some model-plant mismatch by perturbing the simulation. For this purpose, add some noise at every time step of the simulation, e.g., $w \sim \mathcal{N}(0, \Sigma)$ with $\Sigma = \text{diag}(\sigma_{\text{p}}^2, \sigma_\theta^2, \sigma_{\text{v}}^2, \sigma_\omega^2)$ and $\sigma_{\text{p}} = \sigma_\theta = 10^{-3}$ and $\sigma_{\text{v}} = \sigma_\omega = 10^{-2}$. You can also create some bias by using a nonzero mean. Alternatively, or additionally, you can add strong perturbations at specific time points, corresponding to someone suddenly kicking the pendulum tip. For this purpose, after half the simulation horizon has passed, add $\pi/10$ to the state θ. Play around with the disturbances and see how the controller responds.

   Hint: If you perturb the system too strongly, `acados` *will not be able solve the OCP in the given number of maximum iterations. This results in warning referring to 'status 2' (generated by checking the OCP solver return status). The returned control u_0 will then correspond to a nonconverged solution.*

4. Real-time iteration scheme (RTI). Due to model-plant mismatch, it is in general desirable to control systems at a high frequency, in order to be able to quickly react to deviations between prediction and reality. Considering computational delay, there is a trade-off between taking time to fully solve the OCP or quickly returning a non-converged, suboptimal, solution. The main ideas of the real-time iteration scheme (RTI) can be summarized as:

- After receiving the current system state $\bar{x}_0$, only perform a single SQP iteration before returning u_0.
- While waiting for the updated system state $\bar{x}_0$, already perform as many computations as possible without knowing $\bar{x}_0$ ("preparation phase"). Then, only few computations need to be performed after receiving $\bar{x}_0$ ("feedback phase").

(a) Consider a discrete-time OCP in simultaneous formulation (multiple shooting), solved with an SQP method. Assume we are currently at iteration i, with current solution guess y^i, where y collects all optimization variables. Which computations can already be performed while the initial state $\bar{x}_0$ is still unspecified?

- Constructing the QP subproblem, i.e., computing all Jacobians (i.e., the linearization of the dynamics) (and computing the Hessian approximation, if applicable. In our specific case, the Hessian approximation is constant.).
- Condensing the QP.

(b) Run the template `cartpole_closed_loop.py` with RTI turned on. How long does it take for the RTI controller to return a control input after receiving the current state? How does this compare to the fully converged MPC from the previous task? How do the resulting trajectories compare? Why is the latter comparison not fully fair?

For the feedback phase, the solver reports computation times between 0.293 and 4.653 ms with a median of 0.559 ms. RTI-MPC struggles a bit more with performing the swing-up. With respect to stabilization under noise and strong perturbations, there is no clear difference. The comparison is not fully fair, because RTI-MPC could be run at a higher control frequency, i.e., performing a simulation over a shorter time step before updating the control.