Robotic Systems II: Homework II

Instructor: Konstantinos Chatzilygeroudis (`costashatz@upatras.gr`)

November 2 2024

Introduction

The homework concerns the modeling and control of non-linear systems using the Linear Quadratic Regulator (LQR) and Convex Model Predictive Control (MPC). In particular, we are going to model, simulate and control a double pendulum system (Fig. 1). The homework is split into three parts/sub-questions: 1) modeling and discretization of the system, 2) performing linearization and LQR on the linearized system, and 3) controlling the system under disturbances via convex MPC.

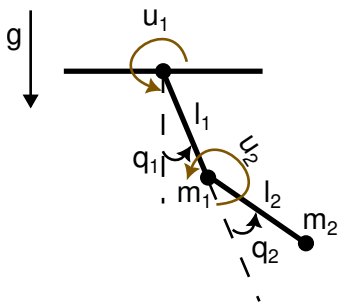

Figure 1: Double Pendulum System

1 Modeling (20%)

For this homework we are going to use the double pendulum system (Fig. 1). The system is a two-link (masses m_1 and m_2, and lengths l_1 and l_2 respectively), robot (see Fig. 1). Both joints exert torque (u_1 and u_2 respectively). The system has four continuous state variables: two joint positions (q_1, q_2) and two joint velocities $(\dot{q}_1, \dot{q}_2)$.
The state of the system is as follows:

$$\boldsymbol{x} = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \in \mathbb{R}^4$$

And the control inputs:

$$\boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in \mathbb{R}^2$$

The continuous dynamics of the system are given by the following equations:

$$\dot{\boldsymbol{x}} = f(\boldsymbol{x}, \boldsymbol{u}) = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} \in \mathbb{R}^4$$

By defining $\boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}$, we can derive the following equation:

$$\ddot{\boldsymbol{q}} = \boldsymbol{M}^{-1}\Big(\boldsymbol{u} - \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{G}(\boldsymbol{q})\Big)$$

where

$$\boldsymbol{M} = \begin{bmatrix} l_1^2 m_1 + l_2^2 m_2 + l_1^2 m_2 + 2 l_1 m_2 l_2 \mathrm{cos} q_2 & l_2^2 m_2 + l_1 m_2 l_2 \mathrm{cos} q_2 \\ l_2^2 m_2 + l_1 m_2 l_2 \mathrm{cos} q_2 & l_2^2 m_2 \end{bmatrix} \in \mathbb{R}^{2\times 2}$$
$$\boldsymbol{C} = \begin{bmatrix} -2\dot{q}_2 l_1 m_2 l_2 \mathrm{sin} q_2 & -\dot{q}_2 l_1 m_2 l_2 \mathrm{sin} q_2 \\ \dot{q}_1 l_1 m_2 l_2 \mathrm{sin} q_2 & 0 \end{bmatrix} \in \mathbb{R}^{2\times 2}$$
$$\boldsymbol{G} = \begin{bmatrix} -g m_1 l_1 \mathrm{sin} q_1 - g m_2 (l_1 \mathrm{sin} q_1 + l_2 \mathrm{sin}(q_1 + q_2)) \\ -g m_2 l_2 \mathrm{sin}(q_1 + q_2) \end{bmatrix} \in \mathbb{R}^{2\times 1}$$

and $g = 9.81\, m/s^2$, $m_1 = m_2 = 1\, kg$, $l_1 = l_2 = 0.5\, m$.

In this part, you are asked to:

1. Write a function in Python called `dynamics()` that takes as input the state $\boldsymbol{x} \in \mathbb{R}^{4\times 1}$ and controls $\boldsymbol{u} \in \mathbb{R}^{2\times 1}$ and returns the $\dot{\boldsymbol{x}} \in \mathbb{R}^{4\times 1}$
2. Create a simple visualization of the system (e.g. by using the `matplotlib` library); you are free to use any library for the visualization
3. Discretize the system using **Runge-Kutta 4th Order integration**
4. Make sure that the integration works and is correct

For all the above (apart from the visualizations), you are allowed to use only native Python functions/classes, and the `numpy` and `jax` libraries.

2 Linearization and LQR (40%)

2.1 Linearization around a fixed point

What does it mean we "linearize around a fixed point"?

$$\begin{aligned} \boldsymbol{x}_{k+1} &= f_{\text{discrete}}(\boldsymbol{x}_k, \boldsymbol{u}_k) && \text{discrete dynamics} \\ \bar{\boldsymbol{x}} &= f_{\text{discrete}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) && \text{fixed point} \end{aligned}$$

Taylor Series around $\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}$:

$$\boldsymbol{x}_{k+1} = f_{\text{discrete}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) + \underbrace{\frac{\partial f_{\text{discrete}}}{\partial \boldsymbol{x}}\Big|_{\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}}}_{\boldsymbol{A}}(\boldsymbol{x}_k - \bar{\boldsymbol{x}}) + \underbrace{\frac{\partial f_{\text{discrete}}}{\partial \boldsymbol{u}}\Big|_{\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}}}_{\boldsymbol{B}}(\boldsymbol{u}_k - \bar{\boldsymbol{u}})$$

We can set $\boldsymbol{x}_k = \bar{\boldsymbol{x}} + \Delta\boldsymbol{x}_k, \boldsymbol{u}_k = \bar{\boldsymbol{u}} + \Delta\boldsymbol{u}_k$ and thus:

$$\begin{aligned} \bar{\boldsymbol{x}} + \Delta\boldsymbol{x}_{k+1} &= f_{\text{discrete}}(\bar{\boldsymbol{x}}, \bar{\boldsymbol{u}}) + \boldsymbol{A}\Delta\boldsymbol{x}_k + \boldsymbol{B}\Delta\boldsymbol{u}_k \\ \Delta\boldsymbol{x}_{k+1} &= \boldsymbol{A}\Delta\boldsymbol{x}_k + \boldsymbol{B}\Delta\boldsymbol{u}_k \end{aligned}$$

2.2 Task

In this part, you are asked to linearize the system around $\bar{\boldsymbol{x}} = \begin{bmatrix} \pi \\ 0 \\ 0 \\ 0 \end{bmatrix}$ and $\bar{\boldsymbol{u}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and perform LQR on the linearized version of the system. In particular:

1. Write a function in Python called `linearize()` that takes as input the state $\bar{\boldsymbol{x}}$ and controls $\bar{\boldsymbol{u}}$ and returns the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ for the linearized system (you should use the function `dynamics()` you created in the first part)

2. Write a function in Python called `lqr()` that takes as input the linearized dynamics matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ along with matrices $\boldsymbol{Q}_N$, $\boldsymbol{Q}$ and $\boldsymbol{R}$, solves the infinite LQR problem and returns the $\boldsymbol{P}$ and $\boldsymbol{K}$ matrices. Define your own $\boldsymbol{Q}_N$, $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices for the double pendulum task

3. Write a simulation loop that controls the system using the infinite LQR controller. **Note that we need to simulate with the actual non-linear dynamics!** The simulation timestep should be $dt = 0.05$. The control inputs need to be **limited** to the $[-10, 10]$ range. The initial state should be $\bar{\boldsymbol{x}}$

4. Evaluate how far from the linearization point, the LQR controller is able to control the system

For all the above (apart from the visualizations), you are allowed to use only native Python functions/classes, and the `numpy` and `jax` libraries.

3 Controlling via Convex MPC (40%)

3.1 Convex MPC

Reminder of the **"Convex Model Predictive Control"**:

$$\begin{aligned} \min_{\boldsymbol{x}_{1:H}, \boldsymbol{u}_{1:H-1}} & \sum_{k=1}^{H-1} \left(\frac{1}{2} \boldsymbol{x}_k^T \boldsymbol{Q}_k \boldsymbol{x}_k + \frac{1}{2} \boldsymbol{u}_k^T \boldsymbol{R}_k \boldsymbol{u}_k \right) + \frac{1}{2} \boldsymbol{x}_H^T \boldsymbol{P}_H \boldsymbol{x}_H \\ \text{s.t.} \quad & \boldsymbol{x}_{k+1} = \boldsymbol{A}\boldsymbol{x}_k + \boldsymbol{B}\boldsymbol{u}_k \\ & \boldsymbol{u}_k \in \mathcal{U} \end{aligned}$$

where $\boldsymbol{P}_H$ is the infinite LQR $\boldsymbol{P}$ matrix.

3.2 Task

In this part, you are asked to implement a Convex MPC controller for the system. In particular:

1. Write a function in Python called `mpc_controller()` that takes as input the current state of the system and the target state (reference), and outputs the controls/torques to be applied to the system. The function should implement Convex MPC as a Quadratic Programming. In other words, you should use the linearized dynamics

2. Choose the horizon length $H = 20$ and add noise to the observations (Gaussian with zero mean and diagonal covariance). Compare the LQR controller and the MPC one. Evaluate how far from the linearization point each controller can go

3. Experiment with different horizon lengths. Is it possible to perform a *swing-up* motion (i.e. start from the bottom and balance at the top)?

For all the above (apart from the visualizations), you are allowed to use only native Python functions/classes, and the `numpy`, `jax` and `proxsuite` libraries.

4 Deliverables

- 3 python files with commented code (one for each subquestion)[1]
- A short report

Bonus Points

1. There is a 5% bonus if one derives the equations of the system (exactly as given), $f(\boldsymbol{x}, \boldsymbol{u})$, analytically.
2. There is a 10% bonus if one writes the derivatives by hand and does not use `jax` (autodiff).

[1] Jupyter notebooks are accepted as well.