Assignment 1: Model Predictive Control of a N-link Manipulator

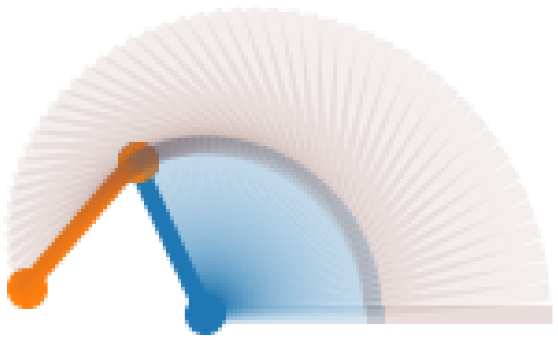
CS 6756, Fall 2023

Due on: 9/22/23

Overview

In this assignment, you are going to build a N-link manipulator, control it to locations of your choice and potentially do fancier things. Your weapon of choice will be Model Predictive Control (MPC), specifically an iterative Linear Quadratic Regulator (iLQR)! On this journey you will:

- Derive a model for N-link planar manipulator
- Design a LQR policy to regulate the manipulator about a set point
- Design an iLQR policy to control the end effector precisely
- Be creative with iLQR!



Building up your Model Predictive Control

Getting Started

The code for this assignment can be found here:
https://github.com/portal-cornell/cs6756-learning-robot-decisions-fa23/blob/main/assignments/assignment1_mpc/assignment1_mpc.ipynb

We recommend using Google Colab for this assignment.

Forward Kinematics and Visualization

Let's visualize the arm in different *configurations*. The *configuration* of the arm is specified by a vector of joint angles. Given these joint angles, one can proceed link by link and compute the position of the link in the world. This computation is referred to as forward kinematics. Once we have the link positions, we can visualize the arm easily.

We have provided some code for both forward kinematics and arm visualization. Try visualizing different configurations so you get some intuition about forward kinematics!

Deriving Dynamics of a 2-link Manipulator [10 points]

Let's begin by modeling our N-link manipulator. We will stick to N=2 to keep things simple (more on N>2 later). The dynamics model is a function with type $x_{t+1} = f(x_t, u_t)$, where x_t is the 2N dimensional state (Angle/Velocity θ, $\dot{\theta}$) and u_t is the N dimensional control input (Torques τ).

Assume:

- **that the mass of the link is concentrated at the end of the link.**
- **the manipulator is flat on a table on the X, Y plane. So gravity does not come into play.**

So, how do we derive equations of motion?

Now, if you were like me, you probably first learnt this the *Newtonian* way, i.e., write down the forces and constraints, apply F=ma and compute accelerations from there. But writing down forces due to constraints between the two links of the manipulator can be quite hairy. Instead, the cool kids listen to *Lagrange*, i.e., write down the energy of the system and apply the principle of least action.

You have two options:
Option 1 [Real dynamics]: Apply Lagrangian mechanics for a 2-link manipulator.

- Apply $\tau_i = \frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}_i} - \frac{\partial L}{\partial \theta_i}$ where τ_i is the torque for the ith joint, L is the lagrangian and so on
- Derive an expression for $\ddot{\theta}_i$, the angular acceleration for the i^{th} joint.
- Apply Euler integration to get angular velocity and position. (Use dt=0.1)

If you are brave, you can derive it all by hand. If you are feeling queasy, check your math against Lynch & Park, Pg 273.

Option 2 [Fake dynamics]: Chicken out and simply apply $\tau_i = ml^2\ddot{\theta}_i$. Apply Euler integration to get angular velocity and position. You would be ignoring a whole bunch of terms that couple the two joints, but whatever! Sometimes a simple fake model is good to start with to debug your code.

Things to include in the writeup

- Do a quick sanity check of your dynamics model and tell us what you did
- What assumptions are you making in your dynamics model?

The Linear Quadratic Regulator (LQR) [10 points]

It's now time to code up your first decision making algorithm – LQR or value-iteration-for-linear-dynamics-and-quadratic-costs! Let's regulate the arm around the rest position $\theta_1^{ref} = 0,\ \theta_2^{ref} = 0.$

We'll help you make your code modular so the move to iLQR is pain-free. There are only 4 functions:

1. *def regulator_cost():* The quadratic cost function.
2. *def lqr_input_matrices():* Use the jacobian and hessian functions provided by PyTorch Autograd to create the A, B, Q, R matrices.
3. *def lqr_gains():* Run value iteration and compute V and K matrices for each time step
4. *def lqr_forward():* Start from initial state and iteratively apply K matrices and dynamics to roll out a trajectory

Once you are done, run the code snippet to see your LQR in action!

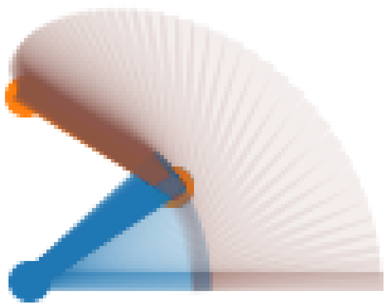
Things to include in the writeup

- Plot showing the trace of the arm starting from some initial condition and reaching the regulating state

- Try a couple different initial states and generate plots of regulator cost vs time. What shape would you say the cost curve takes?
- What happens if you set the control cost to 0? Does the torque go to infinity (since there's nothing penalizing it)?

Iterative Linear Quadratic Regulator (iLQR) [20 points]

We are finally ready to trek up to iLQR! We are going to use iLQR to move the end-effector of the arm to a target position. The cost function will from regulating around a state to $(ee(t) - ee_{goal})^2$ where $ee(t)$ is the end effector position.



You have the freedom to go your own way, but we recommend the key steps from Chap 2 of MACRL

1. Time-varying LQR: Update LQR to handle time varying A_t, B_t, Q_t, R_t matrices
2. Affine LQR: Since we are linearizing about (x^{ref}, u^{ref}) and not the origin, we need to account for offset terms in the dynamics. This is best handled by going to homogeneous coordinates where we augment the state $\tilde{x}_t = [x_t\ 1]$.
3. Change of basis: Since we have a trajectory $(x^{ref}(t), u^{ref}(t))$ that we are linearizing about, it's best to work in the delta space $\delta x_t = x_t - x_t^{ref}$

Let's say you implement this perfectly - it still may fail to converge to the optimal solution. You are going to need some special sauce! The problem isn't an oddity with iLQR, but more fundamental – ensuring the optimization problem is *well conditioned*. There's a couple of considerations at least::

1. Ensuring positive semidefinite Q_t: Since we are quadricizing every iteration, Q_t is not guaranteed to be PSD (think about what that means). We need to project it to PSD in every iteration.
2. Levenberg Marquedt (LM): When computing LQR gains, you need to invert a matrix. If that matrix is close to singular, inversion is going to blow up the gains. LM is a way to prevent this by adding a positive diagonal component.

Things to include in the writeup

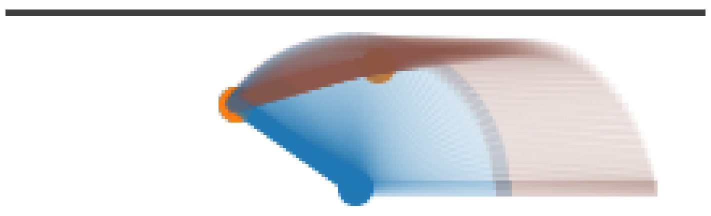
- Plot showing the trace of the arm starting from the initial condition and reaching the target end effector. Also attach a plot of the cost of the trajectory versus time?
- How did you enforce PSD?
- How do you choose lambda for Levenberg Marquedt?

Extra Credit: Choose Your Adventure!

Congratulations, you made it! You now have a working iLQR and can do interesting things with it. We leave it to you to choose one out of the following three adventures:

Adventure 1: Obstacle avoidance

Place a wall such that the arm has to avoid the wall as it goes from rest to goal end effector configuration (as shown in the figure below). How would you change the cost function to capture this? How would you tradeoff obstacle avoidance cost with other cost terms?



Adventure 2: Draw the first initial of your name

Use iLQR to track a curve that draws out the first initial of your name. We recommend using a gui to manually draw the curve and then feeding that as a reference for iLQR to track. Have it trace at multiple speeds: slow, medium, fast.

Adventure 3: N > 2 link manipulator

Implement the dynamics for N > 2 (say 7?) dimensional arm. Then try iLQR on such a system. It will probably be very slow though. What is the time complexity of naive iLQR? Can you make it faster?

Deliverables

Please upload the following deliverables to Gradescope:

- PDF of writeup
- Zip file of the code/jupyter notebook.

These will show up as two separate assignments in Gradescope.