Nonlinear Control of a Two-link Planar Manipulator

Addy Park

Last edit: October 15, 2024

1 Introduction

The dynamics of robotic manipulators contain trigonometric nonlinearities and quadratic nonlinearities, making them challenging to control. In this project, I aim to design an optimal controller for a two-link manipulator in two dimensions. The optimal controller is to track a desired final state constraint as optimally as possible. I plan to take multiple approaches. In the first approach, I will use feedback linearization to linearize the system and design a linear quadratic regulator (LQR) for the linearized system. In a second appraoch, I plan to use the control policy given by LQR (from the first approach) as an initial guess for optimization using Pontryagin's maximum principle. The rationale for the second approach is that, while the LQR control policy is optimal with respect to the transformed control input defined by feedback linearization, it is not necessarily optimal with respect to the physical control input $\mathbf{u}$. In the third approach, I will use control parameterization, which involves approximating the control input $\mathbf{u}$ by a linear combination of basis function, effectively reducing the decision vector down to just a few variables.

2 Dynamics of a Two-link Planar Manipulator

First, we need to model the dynamics of the two-link manipulator in two dimensions. This can be done from first principle using from Newton's laws. The problem setup is as follows:

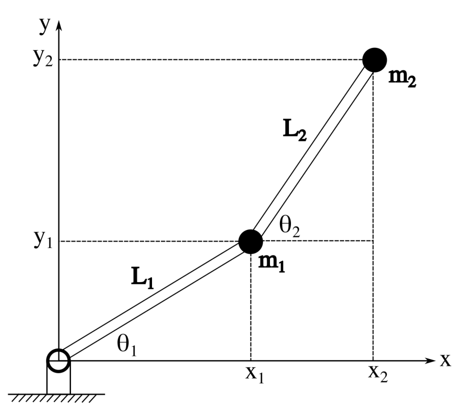

Figure 1: Diagram of two-link manipulator in 2D

Let us define three terms:

$$M(\boldsymbol{\theta}) = \begin{bmatrix} (m_1+m_2)L_1^2 & m_2L_1L_2(\theta_1-\theta_2) \\ m_2L_1L_2cos(\theta_1-\theta_2) & m_2L_2^2 \end{bmatrix}$$

$$C(\boldsymbol{\theta},\dot{\boldsymbol{\theta}}) = \begin{bmatrix} m_2L_1L_2\dot{\theta_2}^2 sin(\theta_1-\theta_2) \\ -m_2L_1L_2\dot{\theta_1}^2 sin(\theta_1-\theta_2) \end{bmatrix}$$

$$G(\boldsymbol{\theta}) = \begin{bmatrix} (m_1+m_2)gL_1cos(\theta_1) \\ m_2gL_2cos(\theta_2) \end{bmatrix}$$

where $\boldsymbol{\theta} = \begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}^T$ and $\dot{\boldsymbol{\theta}} = \begin{bmatrix}\dot{\theta}_1 & \dot{\theta}_2\end{bmatrix}^T$. $M(\boldsymbol{\theta})$ is called the mass matrix, $C(\boldsymbol{\theta},\dot{\boldsymbol{\theta}})$ is called the Coriolis term, and $G(\boldsymbol{\theta})$ is the gravity term.

Using these three terms, the dynamics of the manipulator can be written compactly as the following:

$$\ddot{\boldsymbol{\theta}} = \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} = -M^{-1}(\boldsymbol{\theta}) \left[C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) + G(\boldsymbol{\theta}) - \mathbf{u} \right] \tag{1}$$

where $\mathbf{u} = \begin{bmatrix} \tau_1 & \tau_2 \end{bmatrix}^T$. τ_1 and τ_2 are the applied torques on each of the two joints.

3 Feedback Linearization

Feedback linearization can be used to linearize the system. Let us define a new variable $\mathbf{v} = \ddot{\boldsymbol{\theta}}$. This change of variables allows us to obtain the following *linear* state space system:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\theta}_1 \\ \ddot{\theta}_1 \\ \dot{\theta}_2 \\ \ddot{\theta}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{2}$$

$$\mathbf{y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_2 \end{bmatrix} \tag{3}$$

With the system linearized, we can design a control policy for $\mathbf{v}$ to control $\mathbf{x} = \begin{bmatrix} \theta_1 & \dot{\theta}_1 & \theta_2 & \dot{\theta}_2 \end{bmatrix}^T$ using techniques from linear control theory. Once we have found $\mathbf{v}$, we can invert the mapping to get the physical control input $\mathbf{u}$ (which is the one we care about).

4 Linear Quadratic Regulator

To control the linearized system defined by Equations 1 and 2 using the transformed control input $\mathbf{v}$, we can design a LQR controller with a reference input. But first, consider the system without a reference input (i.e. a regulator). Assuming that the full state $\mathbf{x}$ can be observed (which is a reasonable assumption for a manipulator with encoders) with no additive sensor noise, the block diagram for LQR control looks like the following:

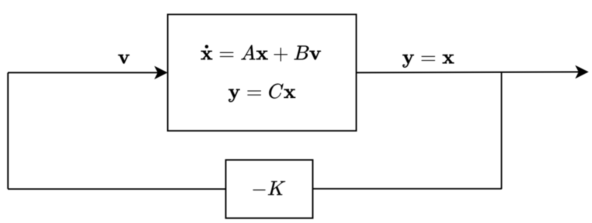

Figure 2: Block diagram of LQR on linearized system without reference input

where matrices A,B, and C are the system matrices of the linear system defined by Equations 2 and 3. K is a gain matrix which can be computed using the MATLAB command `lqr()`. The penalty weights used for the state and control effort are `Q=diag([10 1 10 1])` and `R=diag([2 2])`. Note that R is the penalty weighting on the control input $\mathbf{v}$, not $\mathbf{u}$. Therefore, the LQR solution is optimal with respect to $\mathbf{v}$ but not necessarily optimal with respect to the physical control input $\mathbf{u}$. Because this controller is a regulator, it seeks to steer the system to the origin. To steer the system towards some reference state $\mathbf{r}$ instead of the origin, we require a slightly different formulation. Intuitively, we might expect a valid control policy to be $\mathbf{v} = K(\mathbf{x} - \mathbf{r})$ (this effectively "shifts" the tracking point from the the origin to $\mathbf{r}$). But it turns out we also need to add a nominal input $\bar{\mathbf{v}}$ that makes the reference state $\mathbf{r}$ an equilibrium point. That is, the correct control policy is $\mathbf{v} = \bar{\mathbf{v}} + K(\mathbf{x} - \mathbf{r})$ Also, because the system has two poles at the origin, the steady-state error is guaranteed to converge to zero. The block diagram with the reference input included is given in Figure 3 on the next page.

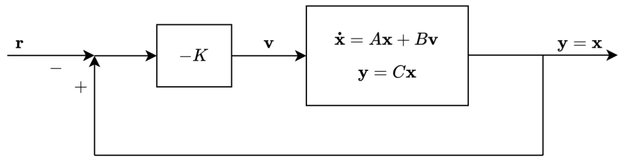

Figure 3: Block diagram of LQR on linearized system with reference input

To simulate the step response (i.e. response to unit reference), I defined a new linear system with system matrices $\tilde{A} = A - BK$, $\tilde{B} = BK$, and $\tilde{C} = C$ with $\mathbf{r}$ as the input and $\mathbf{y}$ as the output and used the `step()` command. Here is the resulting step response with a reference input of $\mathbf{r} = [\pi, 0, \pi/2, 0]$:

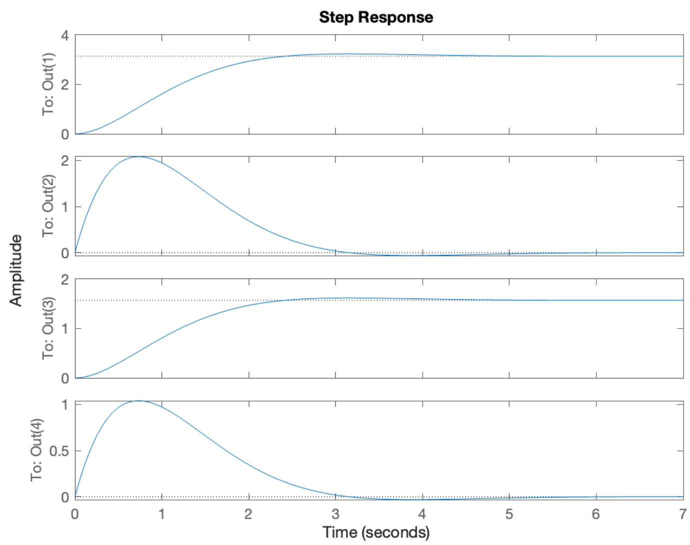

Figure 4: LQR step response of linearized system with reference input

As expected, the steady-state error is zero. The control law that we solved for is a control law for $\mathbf{v}$, not a control law for the the physical control input $\mathbf{u}$. To get the corresponding control law for $\mathbf{u}$, invert the mapping we used to linearize the system in Section 3. Recall that the mapping used to linearize the nonlinear system was the following:

$$\mathbf{v} = M^{-1}(\boldsymbol{\theta}) \left[C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) + G(\boldsymbol{\theta}) \right] + \mathbf{u}$$

To invert the mapping, solve for $\mathbf{u}$:

$$\mathbf{u} = -C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) - G(\boldsymbol{\theta}) + M(\boldsymbol{\theta})\mathbf{v} \tag{4}$$

where $\mathbf{v} = -K(\mathbf{x} - \mathbf{r})$. Now we have $\mathbf{u}$ as a function of $\mathbf{x}$ only, which is the control law for $\mathbf{u}$ that we were seeking. To gain intuition behind what feedback linearization is doing, we can substitute the expression for $\mathbf{u}$ in Equation 4 into Equation 1. $\mathbf{u}$ as we have designed it essentially cancels out any nonlinearities in the system and uses $\mathbf{v}$ to control the resulting linear system.

However, feedback lineraization comes with some caveats. First, to perfectly cancel out nonlinearities, the controller needs perfect knowledge of the state $\mathbf{x}$. Second, it requires that the system be fully-actuated − that is, we require $\dim(\mathbf{u}) = \dim(\mathbf{x})$ such that the control $\mathbf{u}$ can affect all states. It is also required that $\mathbf{u}$ is unconstrained or at least able to take on all values that $C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) + G(\boldsymbol{\theta}) + M(\boldsymbol{\theta})\mathbf{v}$ takes. The first caveat can especially become a problem when there is significant noise in your sensor measurements. In such cases, you should incorporate multiplicative uncertainties into your model for robustness.

Here is a video of the controller successfully tracking a reference input $\mathbf{r} = [\pi, 0, \pi/2, 0]$. A plot of θ_1 and θ_2 plotted over time is shown in Figure 5 is shown below. As expected, the plots in Figures 4 and 5 match.

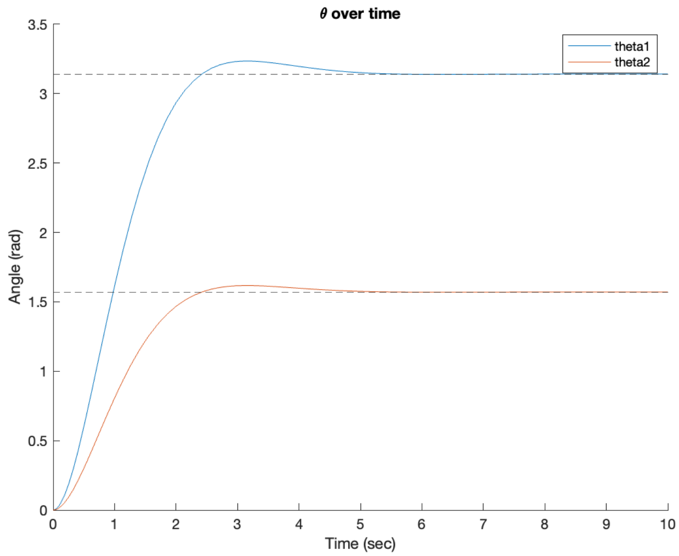

Figure 5: Plot of θ_1 and θ_2 over time

The control effort has also been plotted in the figure below:

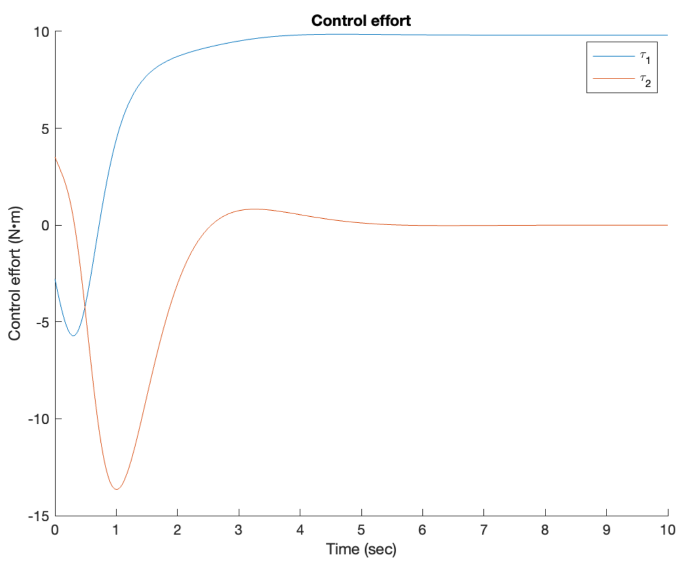

Figure 6: Plot of control effort over time

5 Optimization Using Pontryagin's Maximum Principle

While the system is able to track the reference using feedback linearizaton and LQR, the control policy in Figure 6 is not optimal with respect to $\mathbf{u}$. To optimize the control policy with respect to $\mathbf{u}$, we can use Pontryagin's maximum principle.

5.1 Necessary Conditions for Optimality and Boundary Conditions

Pontryagin's maximum principle provides a set of necessary optimality conditions that must be satisfied by an optimal solution of optimization problems involving dynamic constraints (i.e. constraints that are functions of time). In addition to handling dynamic constraints, this framework can also accommodate state and input constraints, making it applicable to a broad class of control problems. This approach to solving optimal control problems is sometimes referred to as the "indirect" method because it solves the optimization problem without directly discretizing the control and state trajectories and then formulating an optimization problem over these discretized variables. The solution method involves solving for set of first-order differential equations for the state variables and costate variables. The costate variables are a collection of Lagrange multipliers (which are continuous functions of time). But before solving these equations, a cost functional to optimize the control policy with respect to must be defined:

$$J = \int_0^{t_f} L(\mathbf{x}, \mathbf{u})\, dt = \int_0^{t_f} \mathbf{u}^T R\, \mathbf{u}\, dt \tag{5}$$

where $R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. In other words, we will solve a minimal control effort problem. Note that t_f is a free variable that we can optimize over. To steer the system to a desired final state as we did in LQR, we also define a final state constraint that the optimal control policy $\mathbf{u}^*$ must satisfy:

$$\mathbf{x}(t_f) = \begin{bmatrix} \pi \\ 0 \\ \pi/2 \\ 0 \end{bmatrix} \tag{6}$$

The dynamic constraint associated with satisfying the system dynamics is given by Equation 1. We will assume that there are no constraints on the control input $\mathbf{u}$. Next, we construct the Hamiltonian:

$$H = L + \boldsymbol{\lambda}^T \mathbf{f} = L + \sum_{i=1}^{n} \lambda_i f_i$$

where L is defined in the cost function in Equation 4, $\mathbf{f}$ is given by the dynamics $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ in Equation 2, and $\boldsymbol{\lambda}$ is a vector containing the Lagrange multipliers λ_i corresponding to the dynamic constraints f_i on each state variable. With the cost and constraints defined, we seek to find an optimal control policy $\mathbf{u}^*$ that minimizes the cost function J while satisfying the dynamic constraint given by Equation 1 and final state constraint given in Equation 6. Pontryagin's principle states that the optimal state trajectory $\mathbf{x}^*$ and optimal control policy $\mathbf{u}^*$ must satisfy the following necessary conditions:

$$\text{(I)}\;\; \dot{\mathbf{x}}^*(t) = \tfrac{\partial H}{\partial \boldsymbol{\lambda}}(\dot{\mathbf{x}}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t), t)$$

$$\text{(II)}\;\; \dot{\boldsymbol{\lambda}}^*(t) = -\tfrac{\partial H}{\partial \mathbf{x}}(\dot{\mathbf{x}}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t), t)$$

$$\text{(III)}\;\; 0 = \tfrac{\partial H}{\partial \mathbf{u}}(\dot{\mathbf{x}}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t), t)$$

where $V(\mathbf{x}(t_f), t_f)$ is the terminal cost function. Conditions (I) and (II) give us $2n$ differential equations (where n is the dimension of $\mathbf{x}$) and (III) gives us m algebraic equations (where m is the dimension of $\mathbf{u}$). This implies that we will have $2n$ constants of integration to solve for. The constants of integration can be found from the following boundary conditions:

$$\text{(i)}\;\; \mathbf{x}^*(t_0) = \mathbf{x}_0$$

$$\text{(ii)}\;\; \mathbf{x}^*(t_f) = \mathbf{x}_f$$

$$\text{(iii)}\;\; \left[\tfrac{\partial V}{\partial \mathbf{x}}(\dot{\mathbf{x}}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t), t)\right]^T \delta \mathbf{x}_f + \left[H(\dot{\mathbf{x}}^*(t), \mathbf{u}^*(t), \boldsymbol{\lambda}^*(t), t) + \tfrac{\partial V}{\partial t}(\mathbf{x}^*(t_f), t_f)\right] \delta t_f = 0$$

(i) and (ii) are the initial conditions and final conditions, respectively. (iii) is called the transversality condition, which determines the optimal terminal conditions (i.e. optimal t_f and/or $\mathbf{x}(t_f)$, depending on which are allowed to vary). For this problem, since the final state is fixed (but final time is free), the first term vanishes and only the second term remains. The boundary conditions give us $2n + 1$ equations to solve for the $2n$ constants of integration and the final time t_f. Note that, by introducing final constraints, we introduce additional Lagrange multipliers by adding an extra term to the augmented cost function. This has implications on being able to find the Lagrange multipliers at the final time t_f as we will see in the next section.

5.2 Numerical Solution Using Gradient Descent

A straightforward way to solve equations (I)–(III) numerically is by following the iterative procedure outlined below:

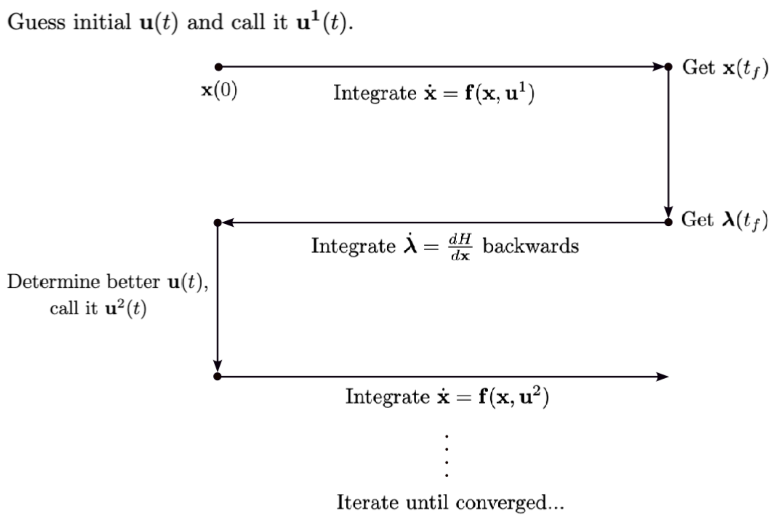

Figure 7: Numerical algorithm for solving state and costate equations.

The problem with this approach is that it assumes that the final costate $\boldsymbol{\lambda}(t_f)$ is not a free parameter and can be determined from knowing $\mathbf{x}(t_f)$. In the case of having final state constraints, we introduce additional Lagrange multipliers associated with the final state constraints. These additional Lagrange multipliers will be constants, not functions of time, since they correspond to constraints at a single point in time rather than a dynamic constraint that must hold for all time. In this case, the original Lagrange multipliers associated with the dynamic constraints must be free at the final time t_f; if they weren't, then the problem would be overconstrained at the final time. Therefore, the procedure in Figure 7 does not work for the cases with final state constraints.

In an attempt to solve for $\boldsymbol{\lambda}(t)$ another way, I used the optimality condition $\frac{\partial H}{\partial \mathbf{u}} = 0$ which is just an algebraic equation and not a differential equation. This is possible in general when $dim(\mathbf{u}) = dim(\boldsymbol{\lambda})$. But for this problem, $dim(\mathbf{u}) < dim(\boldsymbol{\lambda})$. Yet, we are still able to find all four Lagrange multipliers $\lambda_1(t)$, $\lambda_2(t)$, $\lambda_3(t)$, and $\lambda_4(t)$ because $\frac{\partial H}{\partial \mathbf{u}} = 0$ gives us two equations for λ_2 and λ_4 (λ_1 and λ_3 do not appear). One can solved for λ_2 and λ_4 using MATLAB's symbolic toolbox (see symbolic.mlx in the Appendix A). Moreover, the costate equations for $\dot{\lambda}_1(t)$ and $\dot{\lambda}_3(t)$ depend on $\lambda_2(t)$ and $\lambda 4(t)$ only, which allows us to find $\lambda_1(t)$ and $\lambda_3(t)$ by integrating. The resulting plots for all four Lagrange multipliers are plotted below:
However, this doesn't give us any useful information about how we could improve our initial guess for $\mathbf{u}^*(t)$ since we found $\boldsymbol{\lambda}$ using the optimality condition $\frac{\partial H}{\partial \mathbf{u}} = 0$; this means that we assumed our initial guess was already optimal and the $\boldsymbol{\lambda}$ we found is the corresponding optimal $\boldsymbol{\lambda}^*$. As a result, $\frac{\partial H}{\partial \mathbf{u}}$ is just zero for all t:

Ideally, we would know our state trajectory $\mathbf{x}(t)$ and costate trajectory $\boldsymbol{\lambda}(t)$ corresponding to our initial guess for $\mathbf{u}(t)$. Then we can calculate what the corresponding $\frac{\partial H}{\partial \mathbf{u}}$ is and use that information to perform a classic gradient descent algorithm $\mathbf{u}^{i+1} = \mathbf{u}^i + \alpha \frac{\partial H}{\partial \mathbf{u}}\Big|_{\mathbf{u}=\mathbf{u}^i}$ to nudge your control input $\mathbf{u}$ to the optimal value with an appropriate step size α.
An approach like this (which corresponds to the procedure in Figure 7) would be possible if there were no final constraints on the system. In such a case, we would be able to integrate the state equation forward in time from the

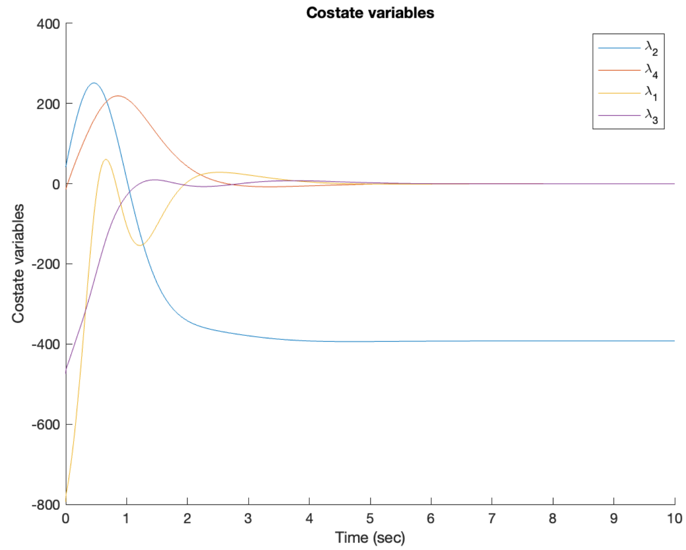

Figure 8: Plot of $\boldsymbol{\lambda}$ over time.

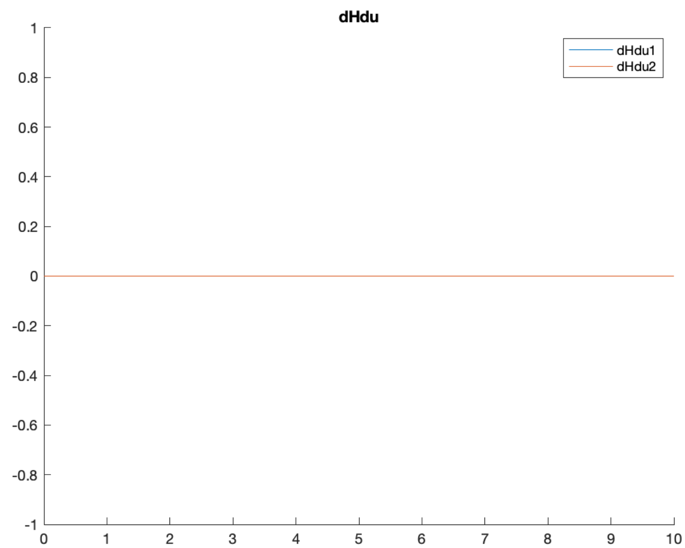

Figure 9: Plot of $\frac{\partial H}{\partial \mathbf{u}}$ over time.

initial state to find $\mathbf{x}(t_f)$, then find $\boldsymbol{\lambda}(t_f)$ from the transversality condition, and then integrate the costate equation backwards in time to find $\boldsymbol{\lambda}(t)$. This would give trajectories for $\mathbf{x}(t)$ and $\boldsymbol{\lambda}(t)$ corresponding to a guess for $\mathbf{u}^*(t)$. Then we would be able to calculate $\frac{\partial H}{\partial \mathbf{u}}$ for the current guess and use gradient descent to improve our guess iteratively until convergence.

A possible workaround this problem would be to enforce a terminal cost instead of a terminal constraint. Whereas a terminal constraint is a hard constraint that must be satisfied, a terminal cost merely penalizes deviations from a desired terminal state. Since this approach does not introduce additional Lagrange multipliers to the problem, the gradient descent method described in this section should work. Moreover, by using this approach, we remove the need for a feasible initial guess to satisfy the desired final state. Instead, any initial guess becomes is a feasible initial guess (it'll just have a high final cost).

6 Optimization by Parameterizing Control Input $\mathbf{u}(t)$

Another approach to solving the optimal control problem is by parameterizing the control input using a set of basis functions. This effectively turns the infinite-dimensional control problem (where we must find control input for every single point in continuous time) to a finite-dimensional one where we can use functions like `fmincon` in MATLAB to search for the optimal set of coefficients for the basis functions. For example, one option is to use polynomials to parameterize $\mathbf{u}(t)$:

$$\mathbf{u}(t) = \begin{bmatrix} \alpha_1 t^n + \beta_1 t^{n-1} xw + \gamma_1 t^{n-2} + \cdots + c_1 \\ \alpha_2 t^n + \beta_2 t^{n-1} + \gamma_2 t^{n-2} + \cdots + c_2 \end{bmatrix} \tag{7}$$

which is parameterized by a set of parameters $[\alpha_1, \beta_1, \gamma_1, ..., c_1, \alpha_2, \beta_2, \gamma_2, ..., c_2]$. These are the decision variables of the optimization problem. As in any iterative method, having a good initial guess can expedite convergence. To obtain a good initial guess for these parameters, we can take our initial guess for $\mathbf{u}(t)$ and use polynomial interpolation to obtain a set of coefficients that approximately matches our initial guess for $\mathbf{u}(t)$ (Figure 6). Using too high of a degree of a polynomial leads to Runge's phenomenon so a polynomial of degree 3 was chosen. Here are plots of the interpolated polynomials:

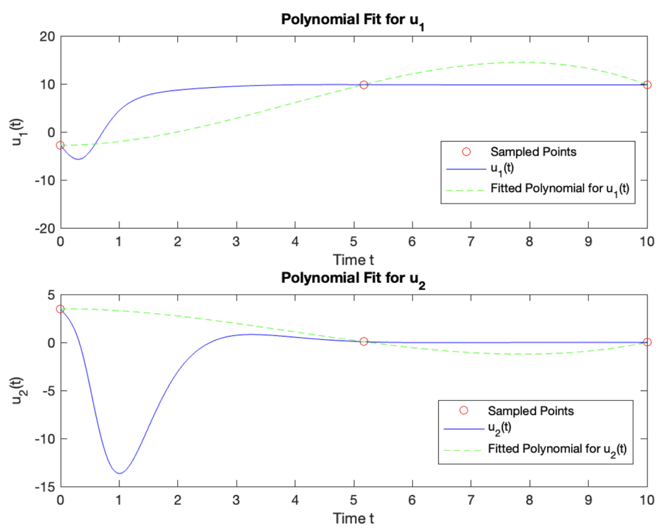

Figure 10: Polynomial fit to initial guess for $\mathbf{u}(t)$.

The coefficients of the polynomial fit are the following:

$$\alpha_1 = -0.0718,\ \beta_1 = 0.8435,\ \gamma_1 = 0,\ c_1 = -2.7852$$

$$\alpha_2 = 0.0194,\ \beta_2 = -0.2289,\ \gamma_2 = 0,\ c_2 = 3.5124$$

While the interpolated polynomial is not a great approximation of our initial guess for $\mathbf{u}(t)$, it has the correct order of magnitude and may serve as a good starting point. But upon running `fmincon`, it was clear that the results of this interpolation leads to a vastly different state trajectory as shown in Figure 11 on the next page.

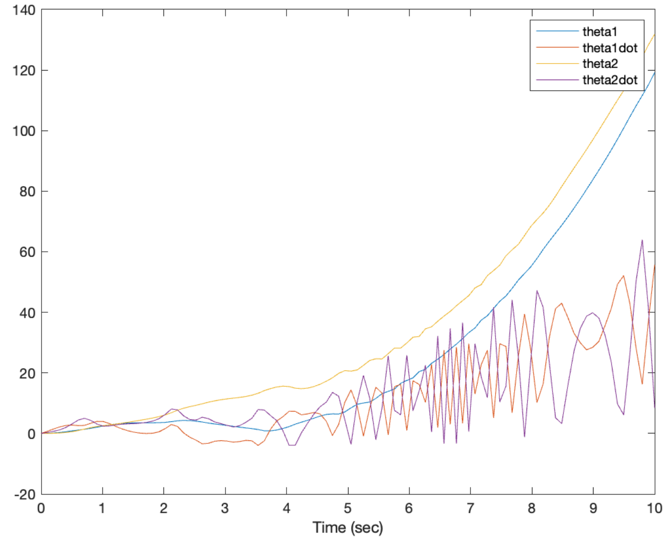

Figure 11: Polynomial fit to initial guess for $\mathbf{u}(t)$.

This poses a problem. If we try to get our initial guess for the parameters to approximate our initial guess $\mathbf{u}(t)$ as closely as possible by increasing the number of points we interpolate through, the polynomial suffers severely from Runge phenomenon. On the other hand, if we sample too few points from our initial guess $\mathbf{u}(t)$, the trajectory is nowhere close to being a feasible solution and the solver has a difficult time converging to an optimal solution. It seems that other basis functions may be more effective.

7 Summary of Code

This section describes how the code for this project is structured. I wrote three different main scripts, one for each approach:

- `main_fdbklin.m`: This is the main script for control using feedback lineaization + LQR.
- `main_pontryagin.m`: This is the main script for optimal control using Pontryagin's principle (using result of feedback lineraization + LQR as an initial guess for iterative procedure).
- `main_fmincon.m`: This is the main script for optimal control using parameterization and `fmincon`.

There is an additional script, `polynomial_fit.m` which fits a polynomial to the control input $\mathbf{u}(t)$ from feedback linearization + LQR. The coefficients of this fit gets used in `main_fmincon.m` as an initial guess for the optimal polynomial coefficients.

The function `TwoLinkArmDynamics.m` is shared by all of the three main scripts above:

- `TwoLinkArmDynamics.m`: This function encodes the dynamics of the two link manipulator. It is meant to be passed into `ode45` to solve for the state trajectory $\mathbf{x}(t)$.

The following functions are used by `main_fdbkln.m`:

- `FLController.m`: This function computes the control input for a given time t according to the control policy in Equation 4.
- `CalcGain.m`: This function computes the optimal gain K for the linearized system in Equations 2 and 3.

The following functions are used by `main_fmincon.m`:

- `control_input.m`: This function computes the control input corresponding to the polynomial control policy in Equation 7 given the polynomial coefficients.

- `cost_function.m`: This function computes the cost $J = \sum_{k=1}^{N} \left(\|\mathbf{x}_{desired} - \mathbf{x}(t_k)\|^2 + \|\mathbf{u}(t_k)\|^2 \right)$ given the polynomial coefficients of your control policy. It uses the coefficients you pass in to generate the control policy as a function of time and uses `ode45` to get the corresponding state trajectory $\mathbf{x}(t)$ in order to calculate J.

- `final_state_error.m`: This function encodes the final state constraint that you pass into `fmincon`.

There is also a livescript (easier to visualize outputs that are symbols in livescripts), `symbolic.mlx`, that uses symbolic toolbox to solve for the costate variables λ_i and convert them to MATLAB functions for use in the `main_pontryagin.m` script.

8 Takeaways

Although I wasn't able to solve for a truly optimal control policy using Pontryagin's principle or by parameterizing the control input $\mathbf{u}(t)$, this project gave me a deeper understanding of optimal control theory. In addition to lecture material covered by Prof. MacMartin in MAE 6780, a useful reference for this project was *Optimal Control Theory: An Introduction* by Kirk, which I used to study Pontryagin's principle in greater detail. This project also gave me a lot of experience with structuring code for complex optimization problems. As I continue to study optimal control, I plan to revisit this problem and improve my approach as I deepen my understanding.

A Appendix

A.1 MATLAB Code: main_fdbkln.m

```
% Two link robot arm control simulation
% Author: Addy (Jin Hyun) Park
% Main script for control using feedback linearization.
clc
clear
close all
global u_global

% System parameters
m1 = 10;
m2 = 10;
L1 = 1;
L2 = 1;
param = [m1; m2; L1; L2]; % parameter vector
u_global = []; % global array to store solution for u and time for u

% Penalty weights and LQR gain
Q = diag([10 1 10 1]);
R = diag([2 2]);
K = CalcGain(Q,R);

% Reference input and control law
r = [pi 0 pi/2 0]; % reference input
u = @(x) FLController(x,K,r,param); % function for control policy (fdbk. lin.)

% Solve for dynamics using ode45
T = 10; % terminal time (just needs to be big enough to reach target pose)
x0 = [0;0;0;0]; % initial conditions
tspan1 = [0 T];
fun1 = @(t,x) TwoLinkArmDynamics(t,x,u(x),param); % robot arm dynamics
[t,x] = ode45(fun1,tspan1,x0); % solve using ode45

% Plots for theta, theta_dot
theta1 = x(:,1);
theta2 = x(:,3);
theta1dot = x(:,2);
theta2dot = x(:,4);

figure(1)
hold on
plot(t,theta1)
plot(t,theta2)
yline(r(1),'--')
yline(r(3),'--')
legend("theta1","theta2")
title("\theta over time")
xlabel("Time (sec)")
ylabel("Angle (rad)")

figure(2)
hold on
plot(u_global(:,1),u_global(:,2))
plot(u_global(:,1),u_global(:,3))
```

```matlab
title("Control effort")
xlabel("Time (sec)")
ylabel("Control input (N*m)")
legend("\tau_1","\tau_2")

% Animate solution
tanimation = linspace(0,T,500);
theta1 = interp1(t,theta1,tanimation); % interpolate solution onto evenly-spaced time vector for smooth animation
theta2 = interp1(t,theta2,tanimation);

videoFile = 'animation.mp4';
v = VideoWriter(videoFile, 'MPEG-4'); % Specify the file name and format
v.FrameRate = 80; % Set the frame rate
open(v); % Open the file for writing

figure(3)
hold on ; grid on
set(gca,'XLim',[-2.5 2.5])
set(gca,'YLim',[-2.5 2.5])
title("Two-link Planar Manipulator")
xlabel("x (m)")
ylabel("y (m)")
axis equal;
for i = 1:length(tanimation)
    %cla
    h1 = plot([0 L1*cos(theta1(i))],[0 L1*sin(theta1(i))],'Color',[0 0.4470 0.7410], 'LineWidth', 5);
    h2 =plot([L1*cos(theta1(i)) L1*cos(theta1(i))+L2*cos(theta2(i))],[L1*sin(theta1(i)) L1*sin(theta1(i))+L2*sin(theta2(i))],'Color',[0.4660 0.6740 0.1880], 'LineWidth', 5);
    pause(0.01);
    if i == length(tanimation) % Break the loop if we've reached the end of the time span
        break
    end
    drawnow; % Render the frame
    frame = getframe(gcf); % Capture the frame
    writeVideo(v, frame); % Write the frame to the video
    delete(h1)
    delete(h2)
end
close(v);
```

A.2 MATLAB Code: TwoLinkArmDynamics.m

```matlab
function xdot = TwoLinkArmDynamics(t,x,u,param)
%State equations (i.e. eqns of motion) for two link robot arm
%   t: time
%   x: state vector [theta1, theta1_dot, theta2, theta2_dot]
%   tvec: time of applied torque
%   u: applied torque vector (the entire time history) [u1(t), u2(t)]
%   param: vector containing system parameters [m1, m2, L1, L2]
% Constants
g = -9.81;
% Extract system parameters
```

```
m1 = param(1);
m2 = param(2);
L1 = param(3);
L2 = param(4);
% Extract state variables
theta1 = x(1);
theta1_dot = x(2);
theta2 = x(3);
theta2_dot = x(4);
% Matrices
M = [(m1+m2)*L1^2 m2*L1*L2*(cos(theta1-theta2));
     m2*L1*L2*cos(theta1-theta2) m2*L2^2];
C = [m2*L1*L2*theta2_dot^2*sin(theta1-theta2);
     -m2*L1*L2*theta1_dot^2*sin(theta1-theta2)];
G = [(m1+m2)*g*L1*cos(theta1);
     m2*g*L2*cos(theta2)];
% Equations of motion
theta_ddot = -inv(M)*(+C+G)+u;
xdot = [theta1_dot;theta_ddot(1);theta2_dot;theta_ddot(2)];
end
```

A.3 MATLAB Code: `FLController.m`

```
function K = CalcGain(Q,R)
%Calculates optimal LQR gain given penalty weights Q and R
% OL system dynamics
A = [0 1 0 0;
     0 0 0 0;
     0 0 0 1;
     0 0 0 0];
B = [0 0;
     1 0;
     0 0;
     0 1];
C = eye(4);
D = zeros(4,2);

sys1 = ss(A,B,C,D);

% LQR
[K,S,P] = lqr(sys1,Q,R);
end
```

A.4 MATLAB Code: `CalcGain.m`

```
function K = CalcGain(Q,R)
% Calculates optimal LQR gain given penalty weights Q and R
% OL system dynamics
A = [0 1 0 0;
     0 0 0 0;
     0 0 0 1;
     0 0 0 0];
B = [0 0;
     1 0;
     0 0;
```

```
     0 1];
C = eye(4);
D = zeros(4,2);

sys1 = ss(A,B,C,D);

% LQR
[K,S,P] = lqr(sys1,Q,R);
end
```

A.5 MATLAB Code: main_pontryagin.m

```
% Two link robot arm control simulation
% Author: Addy (Jin Hyun) Park
% Main script for control using Pontryagin principle.
clc
clear
close all
global u_global

% System parameters
m1 = 10;
m2 = 10;
L1 = 1;
L2 = 1;
param = [m1;m2;L1;L2]; % pack into vector
u_global = []; % global array to store solution for u and time for u

%% Initial guess for u(t) using feedback linearization
% Penalty weights and LQR gain
Q = diag([10 1 10 1]);
R = diag([2 2]);
K = CalcGain(Q,R);

% Reference and control law
r = [pi 0 pi/2 0]; % reference input
u = @(x) FLController(x,K,r,param); % function for controller (fdbk. lin.)

% Solve for dynamics using ode45
T = 10; % terminal time just needs to be big enough to reach target pose
x0 = [0;0;0;0]; % initial conditions
tspan1 = [0 T];
options = odeset('RelTol', 1e-10, 'AbsTol', 1e-10);
fun1 = @(t,x) TwoLinkArmDynamics(t,x,u(x),param); % differential equation to
    solve
[t,x] = ode45(fun1,tspan1,x0,options); % solve using ode45

% Some plots
theta_1 = x(:,1);
theta_2 = x(:,3);
theta_dot_1 = x(:,2);
theta_dot_2 = x(:,4);
t_u = u_global(:,1); % time vector that corresponds to tau
tau1 = u_global(:,2);
tau2 = u_global(:,3);

```

```
figure(1)
hold on
plot(t,theta_1)
plot(t,theta_2)
yline(r(1),'--')
yline(r(3),'--')
legend("theta1","theta2")
title("\theta over time")
xlabel("Time (sec)")
ylabel("Angle (rad)")

figure(2)
hold on
plot(t_u,tau1)
plot(t_u,tau2)
title("Control effort")
xlabel("Time (sec)")
ylabel("Control effort (N*m)")
legend("\tau_1","\tau_2")

% Also compute thetaddot for later
theta_ddot_1 = diff(theta_dot_1)./diff(t);
theta_ddot_2 = diff(theta_dot_2)./diff(t);

%% Compute costate variables
% Interpolate all state and costate variables to the same time vector
[t_u,index,~] = unique(t_u); % no duplicates are allowed for interp1
tau1 = tau1(index);
tau2 = tau2(index);
tau1 = interp1(t_u,tau1,t);
tau2 = interp1(t_u,tau2,t);
% Find lambda_2 and lambda_4 by solvoing dH/du=0
lambda2 = L1.*(L1.*m1.*tau1+L1.*m2.*tau1+L2.*m2.*tau2.*cos(theta_1-theta_2)).*-2.0;
lambda4 = L2.*m2.*(L2.*tau2+L1.*tau1.*cos(theta_1-theta_2)).*-2.0;
% Plot lambda_2 and lambda_4
figure(3)
hold on
plot(t,lambda2)
plot(t,lambda4)
title("Costate variables")
xlabel("Time (sec)")
ylabel("Costate variables")

%%% Solve for lambda_1 and lambda_3 %%%
% First compute lambda_dot_2 and lambda_dot_4
lambda_dot_2 = diff(lambda2)./diff(t);
lambda_dot_4 = diff(lambda4)./diff(t);
% Adjust size to match size of lambda_dot_2 and lambda_dot_4
theta_1 = theta_1(2:end);
theta_2 = theta_2(2:end);
theta_dot_1 = theta_dot_1(2:end);
theta_dot_2 = theta_dot_2(2:end);
lambda2 = lambda2(2:end);
lambda4 = lambda4(2:end);
% Compute lambda_1 and lambda_3
```

```
lambda1 = -lambda_dot_2-(L1.*L2.*lambda4.*m2.*theta_dot_1.*sin(theta_1-theta_2)
    .*(m1+m2).*2.0)./(L2.^2.*m2.^2-L2.^2.*m2.^2.*cos(theta_1-theta_2).^2+L2.^2.*
    m1.*m2)+(L1.*L2.*lambda2.*m2.*theta_dot_1.*cos(theta_1-theta_2).*sin(theta_1-
    theta_2).*2.0)./(L1.*L2.*m1+L1.*L2.*m2-L1.*L2.*m2.*cos(theta_1-theta_2).^2);
lambda3 = -lambda_dot_4+(L1.*L2.*lambda2.*m1.*theta_dot_2.*sin(theta_1-theta_2)
    .*2.0)./(L1.^2.*m1+L1.^2.*m2-L1.^2.*m2.*cos(theta_1-theta_2).^2)-(L1.*L2.*
    lambda4.*m1.*theta_dot_2.*cos(theta_1-theta_2).*sin(theta_1-theta_2).*2.0)./(
    L1.*L2.*m1+L1.*L2.*m2-L1.*L2.*m2.*cos(theta_1-theta_2).^2);
% lambda_1 and lambda_3 are a little jittery so smoothen it out first
lambda1 = smoothdata(lambda1, "sgolay");
lambda3 = smoothdata(lambda3, "sgolay");
% Plot lambda_1 and lambda_3
figure(3)
hold on
plot(t(2:end), lambda1)
plot(t(2:end), lambda3)
legend("\lambda_2","\lambda_4","\lambda_1","\lambda_3")

%% Find new guess for u(t) using gradient descent u' = u + alpha*dH/du
%%% Compute Hamiltonian %%%
% Adjust size of tau1 and tau2
tau1 = tau1(2:end);
dtau1 = gradient(tau1);
dtau1 = smoothdata(dtau1, "sgolay");

tau2 = tau2(2:end);
dtau2 = gradient(tau2);
dtau2 = smoothdata(dtau2, "sgolay");

Hamiltonian = tau1.^2+tau2.^2+lambda1.*theta_dot_1+lambda2.*theta_ddot_1+lambda3
    .*theta_dot_2+lambda4.*theta_ddot_2;
dH = gradient(Hamiltonian);
dH = smoothdata(dH, "sgolay");

dHdu1 = tau1.*2.0+lambda2./(L1.^2.*m1+L1.^2.*m2-L1.^2.*m2.*cos(theta_1-theta_2)
    .^2)-(lambda4.*cos(theta_1-theta_2))./(L1.*L2.*m1+L1.*L2.*m2-L1.*L2.*m2.*cos(
    theta_1-theta_2).^2);
dHdu2 = tau2.*2.0-(lambda2.*cos(theta_1-theta_2))./(L1.*L2.*m1+L1.*L2.*m2-L1.*L2
    .*m2.*cos(theta_1-theta_2).^2)+(lambda4.*(m1+m2))./(L2.^2.*m2.^2-L2.^2.*m2
    .^2.*cos(theta_1-theta_2).^2+L2.^2.*m1.*m2);

% Plot dH/du
figure(4)
title("dHdu")
hold on
plot(t(2:end),dHdu1)
plot(t(2:end),dHdu2)
legend("dHdu1","dHdu2")
ylim([-1,1])
```

A.6 MATLAB Code: `symbolic.mlx`

```
clc
clear
close all

```

```
% Create symbols
syms g m1 m2 L1 L2 theta_1 theta_2 theta_dot_1 theta_dot_2 theta_ddot_1...
    theta_ddot_2 tau1 tau2 lambda1 lambda2 lambda3 lambda4 lambda5 real

M = [(m1+m2)*L1^2 m2*L1*L2*(cos(theta_1-theta_2));
     m2*L1*L2*cos(theta_1-theta_2) m2*L2^2]; % mass matrix
c_vec = [m1*L1*L2*theta_dot_2^2*sin(theta_1-theta_2);
     -m2*L1*L2*theta_dot_1^2*sin(theta_1-theta_2)]; % coriolis term
g_vec = [0;0]; % gravity term
u_vec = [tau1;tau2]; % control input

thetaddot = inv(M)*(u_vec-c_vec-g_vec);

theta1ddot = thetaddot(1);
theta2ddot = thetaddot(2);

Hamiltonian = tau1^2+tau2^2+lambda1*theta_dot_1 + lambda2*theta1ddot + lambda3*
    theta_dot_2 + lambda4*theta2ddot

lambda1dot = -diff(Hamiltonian,theta_1)
lambda2dot = -diff(Hamiltonian,theta_dot_1)
lambda3dot = -diff(Hamiltonian,theta_2)
lambda4dot = -diff(Hamiltonian,theta_dot_2)
dHdu1 = diff(Hamiltonian,tau1)
dHdu2 = diff(Hamiltonian,tau2)

% Solve for lambda2 and lambda4
solution1 = solve([dHdu1==0 dHdu2==0],[lambda2 lambda4]);
solution1.lambda2
solution1.lambda4

% Solve forl lambda1 and lambda3
syms lambda_dot_2 lambda_dot_4
solution2 = solve([lambda2dot-lambda_dot_2==0 lambda4dot-lambda_dot_4==0],[
    lambda1 lambda3]);
solution2.lambda1
solution2.lambda3

% Convert symbolic functions to MATLAB functions for use in other
% scripts/functions
fun1 = matlabFunction(lambda1dot);
fun2 = matlabFunction(lambda2dot);
fun3 = matlabFunction(lambda3dot);
fun4 = matlabFunction(lambda4dot);
fun5 = matlabFunction(solution1.lambda2);
fun6 = matlabFunction(solution1.lambda4);
fun7 = matlabFunction(solution2.lambda1);
fun8 = matlabFunction(solution2.lambda3);
```

A.7 MATLAB Code: main_fmincon.m

```
% Two link robot arm control simulation
% Author: Addy (Jin Hyun) Park
% Main script for control using fmincon.
clc
clear
```

```
close all

global u_global
u_global = []; % global array to store solution for u and t

poly_degree = 1;

% Initial guess for the control parameters (e.g., random or zeros)
n_params = poly_degree + 1; % Number of parameters for each control input
%initial_params = [-0.0718, 0.8435, 0, -2.7852, 0.0194, -0.2289, 0, 3.5124];
initial_params = [-0.2449, 3.7090, -2.7852, 0.0651, -1.0018, 3.5124];
% Time span and initial conditions
t_span = [0, 10]; % Define time span for optimization
x0 = [0; 0; 0; 0]; % initial state

% Define constraints (e.g., final state must be [0, 0])
final_state_constraint = @(params) final_state_error(params, t_span, x0);

% Set up optimization options
options = optimoptions('fmincon', 'Display', 'iter', 'Algorithm', 'sqp');

% Solve optimization problem
optimal_params = fmincon(@(params) cost_function(params, t_span, x0), ...
                         initial_params, [], [], [], [], [], [],
                            final_state_constraint, options);
```

A.8 MATLAB Code: polynomial_fit.m

```
% Script for finding polynomial interpolation that approximates initial
% guess for u(t) found using feedback linearization & LQR.
% Author: Addy (Jin Hyun) Park
clc
clear
close all

% Load initial guess
load("u_global.mat")
u_initial = u_global;

% Define the number of sample points and the degree of the polynomial
num_samples = 3;
poly_degree = 3;

% Generate time vector 't' associated with the data points in 'u'
N = size(u_initial, 1); % Number of total data points
t_original = u_initial(:, 1);

% Select 10 evenly spaced sample points
sample_indices = round(linspace(1, N, num_samples));
samples = u_initial(sample_indices, :);
t_samples = samples(:,1); % Sampled time points
u_samples = samples(:,2:3); % Corresponding sampled u(t) points

% Perform polynomial fitting for each dimension of u
% Initialize matrices to store polynomial coefficients for both dimensions
poly_coeffs = zeros(2, poly_degree+1);
```

```matlab

% Perform polynomial fitting for each dimension of u
for dim = 1:2
    % Fit a 9th-degree polynomial for the sampled data in the current dimension
    poly_coeffs(dim, :) = polyfit(t_samples, u_samples(:, dim), poly_degree);
end

% Display the polynomial coefficients for each dimension
disp('Polynomial coefficients for dimension 1:');
disp(poly_coeffs(1, :));

disp('Polynomial coefficients for dimension 2:');
disp(poly_coeffs(2, :));

% Optional: Plot original data and fitted polynomial for visualization
t_fine = linspace(0, 10, 100); % Time points for plotting the fitted polynomial

% Evaluate the fitted polynomials for both dimensions at the fine time points
u_fit_1 = polyval(poly_coeffs(1, :), t_fine);
u_fit_2 = polyval(poly_coeffs(2, :), t_fine);

% Plot for u1
figure;
subplot(2, 1, 1);
plot(t_samples, u_samples(:, 1), 'ro', 'DisplayName', 'Sampled Points'); % Sampled u1 points
hold on;
plot(t_original, u_initial(:, 2), 'b-', 'DisplayName', 'u_1(t)'); % Original u1 data
plot(t_fine, u_fit_1, 'g--', 'DisplayName', 'Fitted Polynomial for u_1(t)'); % Fitted polynomial for u1
title('Polynomial Fit for u_1');
legend;
xlabel('Time t');
ylabel('u_1(t)');
ylim([-20,20])
hold off;

% Plot for u2
subplot(2, 1, 2);
plot(t_samples, u_samples(:, 2), 'ro', 'DisplayName', 'Sampled Points'); % Sampled u2 points
hold on;
plot(t_original, u_initial(:, 3), 'b-', 'DisplayName', 'u_2(t)'); % Original u2 data
plot(t_fine, u_fit_2, 'g--', 'DisplayName', 'Fitted Polynomial for u_2(t)'); % Fitted polynomial for u2
title('Polynomial Fit for u_2');
legend;
xlabel('Time t');
ylabel('u_2(t)');
hold off;
```

A.9 MATLAB Code: cost_function.m

```matlab
function J = cost_function(params, t_span, x0)
```

```matlab
% Function that calculates cost given coefficients for the polynomial
% control input ('params'). It uses ode45 to get the state trajectory
% corresponding to this polynomial control input and calculates the cost
% for this control & state trajectory.
    % Define time points for evaluation
    t_eval = linspace(t_span(1), t_span(2), 100); % Adjust based on time
        resolution

    % System parameters
    m1 = 10;
    m2 = 10;
    L1 = 1;
    L2 = 1;
    sysparams = [m1; m2; L1; L2]; % pack into vector

    desired_final_state = [pi 0 pi/2 0]';

    % Solve system dynamics over the time span using the given control input
        parameterization
    [t_sol, x_sol] = ode45(@(t, x) TwoLinkArmDynamics(t, x, control_input(t,
        params), sysparams), t_eval, x0);

    % Cost function: for example, minimize final state deviation and control
        effort
    J = 0; % Initialize
    for i = 1:length(t_sol)
        u_i = control_input(t_sol(i), params);

        % Add terms to the cost function, for example:
        J = J + norm(x_sol(i, :) - [desired_final_state])^2 + norm(u_i)^2; %
            Quadratic penalty

        if i == 1
            figure
            plot(t_sol, x_sol)
        end
    end
    J = J / length(t_sol); % Normalize
end
```

A.10 MATLAB Code: control_input.m

```matlab
function u_t = control_input(t, params)
% Function for generating control input as a function of time for
% polynomial control input u(t) = alpha*t+beta*t^2+gamma*t^3+...
% 'params' is a vector containing the polynomial coefficients.
    % params is a vector containing the parameters for both u_1(t) and u_2(t)
    % First half of params is for u_1, second half is for u_2
    n_params = length(params) / 2;
    u1_params = params(1:n_params);
    u2_params = params(n_params+1:end);

    % Polynomial representation of control inputs u_1(t) and u_2(t)
    u1_t = polyval(u1_params, t); % Polynomial for u_1(t)
    u2_t = polyval(u2_params, t); % Polynomial for u_2(t)

```

```
    % Return the 2D control input as a vector
    u_t = [u1_t; u2_t];
end
```

A.11 MATLAB Code: final_state_error.m

```
function [c, ceq] = final_state_error(params, t_span, x0)
% This function contains the final state constraint which will be passed to
% fmincon.
    % Simulate the system
    % System parameters
    m1 = 10;
    m2 = 10;
    L1 = 1;
    L2 = 1;
    sysparams = [m1; m2; L1; L2]; % pack into vector

    [~, x_sol] = ode45(@(t, x) TwoLinkArmDynamics(t, x, control_input(t, params)
        , sysparams), linspace(t_span(1), t_span(2), 100), x0);

    % Final state
    x_final = x_sol(end, :);

    % Constraints: Ensure final state matches desired final state
    desired_final_state = [pi 0 pi/2 0]; % Example desired final state
    ceq = x_final - desired_final_state; % Equality constraint

    % No inequality constraints in this case
    c = [];
end
```