Final Report
Smoother Transportation of Container with Water Using Robot Arm and Double Pendulum Control

[Ketian Wang], UNI [kw3019]

09/05/2023

1 Introduction

This project initially focuses on the smoother transportation of containers with water. The problem will first be demonstrated as a double cart and double pendulum control issue, followed by the development of a mathematical description of the system. Subsequently, the system's linearization will be performed to generate the state space expression. The observability and controllability of the system will then be introduced, followed by the construction of the model in MATLAB Simulink and the design of an LQR controller.

However, the linearization of the system has resulted in a limitation, where the joint angle cannot exceed 5 degrees, greatly restricting the controller's application scenarios. Therefore, in addition to traditional algorithm orientation control methods, this project also explores data-oriented approaches. Specifically, the MPC controller will first be utilized to generate training data for the forward dynamic. A Neural Network will then be trained using the generated data to handle the considerable non-linearity of the forward system. Subsequently, a Deep Q-Network will be employed as the controller. DQN does not require training data and can be applied to various problems. Finally, the PPO algorithm from the Reinforcement Learning domain will be implemented to further demonstrate the power of data-oriented methods.

2 Initial Motivation

The initial aim of this project is to develop a control mechanism for a robot arm that is tasked with transporting a water-filled container. To achieve this, the container and the water surface inside it must oscillate in sync with the gripper's movement. Thus, a sophisticated control system is required to minimize the oscillation of both the container and the water. A basic schematic of the proposed configuration can be found in the following figure:

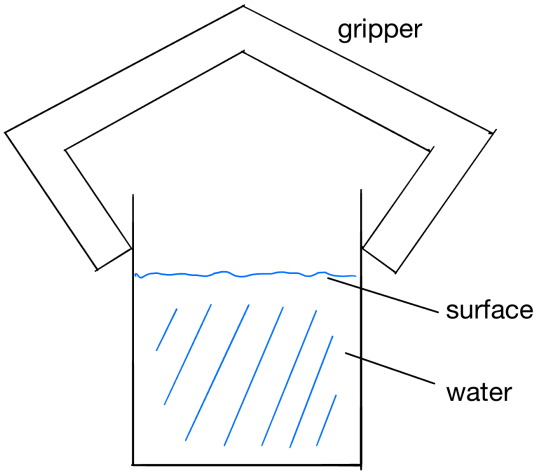

Figure 1: Scratch of the configuration

The system proposed in this project has numerous potential applications in the automation industry, particularly in scenarios where the transportation of liquid containers is required. Oscillations during transport can have negative effects on certain types of liquids, such as soda, which may lose flavor if exposed to excessive vibration. Similarly, in automated chemistry labs, the oscillation of sensitive lab vessels can result in hazardous liquids being spilled. By implementing an effective control mechanism, these issues can be mitigated and avoided.

3 Schematic and System Description

Firstly, based on fluid dynamics, the oscillation of the water surface can be approximated reasonably well by a simple pendulum [1]. Further details on this proof will be covered in the final report. Additionally, if the gripper operates only in a linear fashion, the entire system can be simplified into a 2D plane, greatly reducing the number of parameters. Furthermore, the water container itself can be modeled as a simple pendulum. As a result, the system can be represented as a double pendulum in series connected to a cart on a rail. This configuration is illustrated in the following figure.

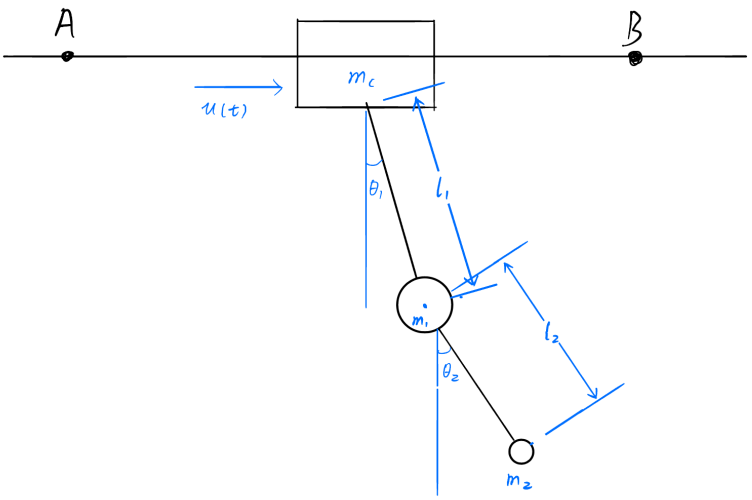

Figure 2: Scratch of the model

As can be seem in the above figure, the target of the actuation is to move the system from point A to point B, while maintaining a low oscillation to the connected double pendulum. To be more specific, the input and actuator of the system is the position of the cart $u(t)$. Ideally, $u(t)$ should start at A and stop at B after a reasonable time interval. 6 states of the system can be monitored, which are the position and speed of the cart $x,\ \dot{x}$; and the angle and angler speed of the two pendulum $\theta_1;\ \theta_2;\ \dot{\theta}_1;\ \dot{\theta}_2$. The system dose not have a output, as the objective is the transportation of the system. The sensor shall sensor the position of the cart and angle of the two pendulum. The actuator shall move the cart to any assigned position. According to the model, the mathematical representation will be developed in e following section.

4 Mathematical model

Using the Lagrangian method to obtain the dynamic model of the system. The total kinetic energy of the system could be expressed as:

$$\begin{aligned} T = &\frac{1}{2}m_c\dot{x}^2 + \frac{1}{2}m_1[(\dot{\theta}_1 l_1\cos\theta_1 + \dot{x})^2 + (\dot{\theta}_1 l_1\sin\theta_1)^2] \\ &+ \frac{1}{2}m_2[(\dot{\theta}_1 l_1\cos\theta_1 + \dot{\theta}_2 l_2\cos\theta_2 + \dot{x})^2 + (\dot{\theta}_1 l_1\sin\theta_1 + \dot{\theta}_2 l_2\sin\theta_2)^2] \end{aligned} \tag{1}$$

The expression can be rearranged as:

$$\begin{aligned} T = &\frac{1}{2}(m_c + m_1 + m_2)\dot{x}^2 + \frac{1}{2}(m_1 + m_2)(\dot{\theta}_1^2 l_1^2 + 2\dot{x}\dot{\theta}_1 l_1\cos\theta_1) \\ &+ \frac{1}{2}m_2[\dot{\theta}_2^2 l_2^2 + 2\dot{\theta}_1 l_1\dot{\theta}_2 l_2\cos(\theta_1 - \theta_2) + 2\dot{x}\dot{\theta}_2 l_2\cos\theta_2] \end{aligned} \tag{2}$$

Suppose that the lowest point of the pendulum is the zero potential energy point, the total potential energy of the system could be obtained as:

$$V = (m_1 + m_2)gl_1(1 - \cos\theta_1) + m_2 g l_2(1 - \cos\theta_2) \tag{3}$$

Therefore, the Lagrangian variable is the difference of the kinetic energy and potential energy.

$$L = T - V \tag{4}$$

Therefore, according the Lagrangian method, taking derivative with respect to each state variable and time, the following equations could be obtained:

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) = (m_c + m_1 + m_2)\ddot{x} + (m_1 + m_2)\ddot{\theta}_1 l_1 \cos\theta_1 - (m_1 + m_2)\dot{\theta}_1^2 l_1 \sin\theta_1 \\ + m_2\ddot{\theta}_2 l_2 \cos\theta_2 - m_2\dot{\theta}_2^2 l_2 \sin\theta_2\end{aligned} \tag{5}$$

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) = (m_1 + m_2) l_1 (\ddot{\theta}_1 l_1 + \ddot{x}\cos\theta_1 - \dot{\theta}_1 \dot{x}\sin\theta_1) \\ + m_2 l_1 l_2 (\ddot{\theta}_2 \cos(\theta_1 - \theta_2) - \dot{\theta}_2(\dot{\theta}_1 - \dot{\theta}_2)\sin(\theta_1 - \theta_2))\end{aligned} \tag{6}$$

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_2}\right) = m_2 l_2 (\ddot{\theta}_2 l_2 + \ddot{\theta}_1 l_1 \cos(\theta_1 - \theta_2) - \dot{\theta}_1 l_1 (\dot{\theta}_1 - \dot{\theta}_2)\sin(\theta_1 - \theta_2) \\ + \ddot{x}\cos\theta_2 - \dot{x}\dot{\theta}_2 \sin\theta_2)\end{aligned} \tag{7}$$

$$\frac{\partial L}{\partial x} = 0 \tag{8}$$

$$\frac{\partial L}{\partial \theta_1} = -(m_1 + m_2) l_1 \sin\theta_1 (g + \dot{\theta}_1 \dot{x}) - m_2 \dot{\theta}_1 l_1 \dot{\theta}_2 l_2 \sin(\theta_1 - \theta_2) \tag{9}$$

$$\frac{\partial L}{\partial \theta_2} = m_2 l_2 [-\sin\theta_2 (g + \dot{\theta}_2 \dot{x}) + \dot{\theta}_1 l_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2)] \tag{10}$$

Therefore, the function for displacement could be represent as:

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = (m_c + m_1 + m_2)\ddot{x} + (m_1 + m_2)\ddot{\theta}_1 l_1 \cos\theta_1 - (m_1 + m_2)\dot{\theta}_1^2 l_1 \sin\theta_1 \\ + m_2\ddot{\theta}_2 l_2 \cos\theta_2 - m_2\dot{\theta}_2^2 l_2 \sin\theta_2 = u(t)\end{aligned} \tag{11}$$

As the joint is assumed to no torque input, the torque of the two pendulum is zero. Therefore:

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) - \frac{\partial L}{\partial \theta_1} = (m_1 + m_2) l_1 (\ddot{\theta}_1 l_1 + \ddot{x}\cos\theta_1 - \dot{\theta}_1 \dot{x}\sin\theta_1 + \sin\theta_1 (g + \dot{\theta}_1 \dot{x})) \\ + m_2 l_1 l_2 (\ddot{\theta}_2 \cos(\theta_1 - \theta_2) - \dot{\theta}_2(\dot{\theta}_1 - \dot{\theta}_2)\sin(\theta_1 - \theta_2) + \dot{\theta}_1\dot{\theta}_2 \sin(\theta_1 - \theta_2)) = 0\end{aligned} \tag{12}$$

$$\begin{aligned}\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_2}\right) - \frac{\partial L}{\partial \theta_2} = m_2 l_2 [\ddot{\theta}_2 l_2 + \ddot{\theta}_1 l_1 \cos(\theta_1 - \theta_2) - \dot{\theta}_1 l_1 (\dot{\theta}_1 - \dot{\theta}_2)\sin(\theta_1 - \theta_2) \\ + \sin\theta_2 (g + \dot{\theta}_2 \dot{x}) - \dot{\theta}_1 l_1 \dot{\theta}_2 \sin(\theta_1 - \theta_2)]\end{aligned} \tag{13}$$

After simplification, the system is governed by the three following equations:

$$\begin{aligned}(m_c+m_1+m_2)\ddot{x}+(m_1+m_2)\ddot{\theta}_1 l_1\cos\theta_1-(m_1+m_2)\dot{\theta}_1^2 l_1\sin\theta_1\\ +m_2\ddot{\theta}_2 l_2\cos\theta_2-m_2\dot{\theta}_2^2 l_2\sin\theta_2=u(t)\end{aligned}$$

$$\begin{aligned}(m_1+m_2)(\ddot{\theta}_1 l_1+\ddot{x}\cos\theta_1-\dot{\theta}_1\dot{x}\sin\theta_1+\sin\theta_1(g+\dot{\theta}_1\dot{x}))\\ +m_2 l_2(\ddot{\theta}_2\cos(\theta_1-\theta_2)-\dot{\theta}_2(\dot{\theta}_1-\dot{\theta}_2)\sin(\theta_1-\theta_2)+\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2))=0\end{aligned} \tag{14}$$

$$\begin{aligned}\ddot{\theta}_2 l_2+\ddot{\theta}_1 l_1\cos(\theta_1-\theta_2)-\dot{\theta}_1 l_1(\dot{\theta}_1-\dot{\theta}_2)\sin(\theta_1-\theta_2)\\ +\sin\theta_2(g+\dot{\theta}_2\dot{x})-\dot{\theta}_1 l_1\dot{\theta}_2\sin(\theta_1-\theta_2)=0\end{aligned}$$

5 Model Simplify and Linearized

The linearization of this system can be somewhat contentious, as it encompasses trigonometric terms that, when linearized, yield satisfactory accuracy only for small angles. In theory, the double double pendulum's angle should not be restricted; however, this project aims to transfer the container and its liquid content with minimal oscillation. Consequently, the angles of both pendulums may be deemed to fall within a 5-degree range. With this in mind, the system can be simplified as follows:

$$\begin{aligned}\sin\theta_{1,2}&\approx\theta_{1,2}\\ \cos\theta_{1,2}&\approx 1-\frac{\theta_{1,2}^2}{2}\approx 1\\ \cos(\theta_1-\theta_2)&\approx 1-\frac{(\theta_1-\theta_2)^2}{2}\approx 1\\ \sin(\theta_1-\theta_2)&\approx(\theta_1-\theta_2)\end{aligned} \tag{15}$$

Accordingly, the system can be firstly simplified as:

$$\begin{aligned}(m_c+m_1+m_2)\ddot{x}+(m_1+m_2)\ddot{\theta}_1 l_1-(m_1+m_2)\dot{\theta}_1^2 l_1\theta_1\\ +m_2\ddot{\theta}_2 l_2-m_2\dot{\theta}_2^2 l_2\theta_2=u(t)\end{aligned}$$

$$\begin{aligned}(m_1+m_2)(\ddot{\theta}_1 l_1+\ddot{x}-\dot{\theta}_1\dot{x}\theta_1+\theta_1(g+\dot{\theta}_1\dot{x}))\\ +m_2 l_2(\ddot{\theta}_2-\dot{\theta}_2(\dot{\theta}_1-\dot{\theta}_2)(\theta_1-\theta_2)+\dot{\theta}_1\dot{\theta}_2(\theta_1-\theta_2))=0\end{aligned} \tag{16}$$

$$\begin{aligned}\ddot{\theta}_2 l_2+\ddot{\theta}_1 l_1-\dot{\theta}_1 l_1(\dot{\theta}_1-\dot{\theta}_2)(\theta_1-\theta_2)\\ +\theta_2(g+\dot{\theta}_2\dot{x})-\dot{\theta}_1 l_1\dot{\theta}_2(\theta_1-\theta_2)=0\end{aligned}$$

Eliminating the trigonometric terms, the state matrix still can not be computed due to the existing of cross terms. Therefore, further linearization of the system should be calculated by to Eliminate the cross terms:

$$\begin{aligned}\mathbf{A}&=\left.\frac{\partial\mathbf{f}(\mathbf{x},\mathbf{u})}{\partial\mathbf{x}}\right|_{(\mathbf{x},\mathbf{u})=(0,0)}\\ \mathbf{B}&=\left.\frac{\partial\mathbf{f}(\mathbf{x},\mathbf{u})}{\partial\mathbf{u}}\right|_{(\mathbf{x},\mathbf{u})=(0,0)}\end{aligned} \tag{17}$$

The result is :

$$\begin{aligned}
\ddot{x} &= \frac{-(m_2+m_1)g}{m_c}\theta_1 + \frac{1}{m_c}u(t) \\
\ddot{\theta}_1 &= \frac{(m_2m_c+m_1m_2+m_cm_1+m_1^2)g}{l_1m_cm_1}\theta_1 + \frac{m_2g}{m_1l_1}\theta_2 - \frac{1}{m_cl_1}u(t) \\
\ddot{\theta}_2 &= -\frac{(m_2+m_1)g}{m_1l_2}\theta_1 + \frac{(m_2+m_1)g}{m_1l_2}\theta_2 - \frac{1}{m_cl_2}u(t)
\end{aligned} \tag{18}$$

Therefore, the Simplify and Linearized model can be expressed as follow:

$$\frac{d}{dt}x(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{19}$$

Where $x(t)$, A and B matrix are:

$$\begin{aligned}
x(t) &= [x, \dot{x}, \theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]^T \\
A &= \begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & -\frac{(m_2+m_1)g}{m_c} & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & \frac{(m_2m_c+m_1m_2+m_cm_1+m_1^2)g}{l_1m_cm_1} & 0 & -\frac{m_2g}{m_1l_1} & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & -\frac{(m_2+m_1)g}{m_1l_2} & 0 & \frac{(m_2+m_1)g}{m_1l_2} & 0
\end{bmatrix} \\
B &= \begin{bmatrix}
0 \\ \frac{1}{m_c} \\ 0 \\ -\frac{1}{m_cl_1} \\ 0 \\ -\frac{1}{m_cl_2}
\end{bmatrix}
\end{aligned} \tag{20}$$

6 Controllability and Observability

Firstly, since the controller will takes the position of cart and the position of the two link as the feedback input to design a LQR controller, the system is fully observed. For the controllability, as given the matrix A and B, the controllability criteria matrix can be also proved to be full rank. Since this project dose not emphasize on investigation some noval system, the Controllability and Observability proveness will not be unfolded.

7 Simulink Model and LQR Controller

7.1 Simulink Model

In this project, Matlab Simulink is utilized to define the system's geometry and execute simulations. Working in conjunction with Matlab scripts that carry out calculations, various controllers can be designed to ensure smooth transportation. At the mid-term stage, the basic geometry of a cart and double-pendulum system has been established, and a feedback control loop has been defined. The model requires further refinement and rendering, while the feedback parameters still need to be computed based on the selected method.The Simulink model is depicted in the figure below:

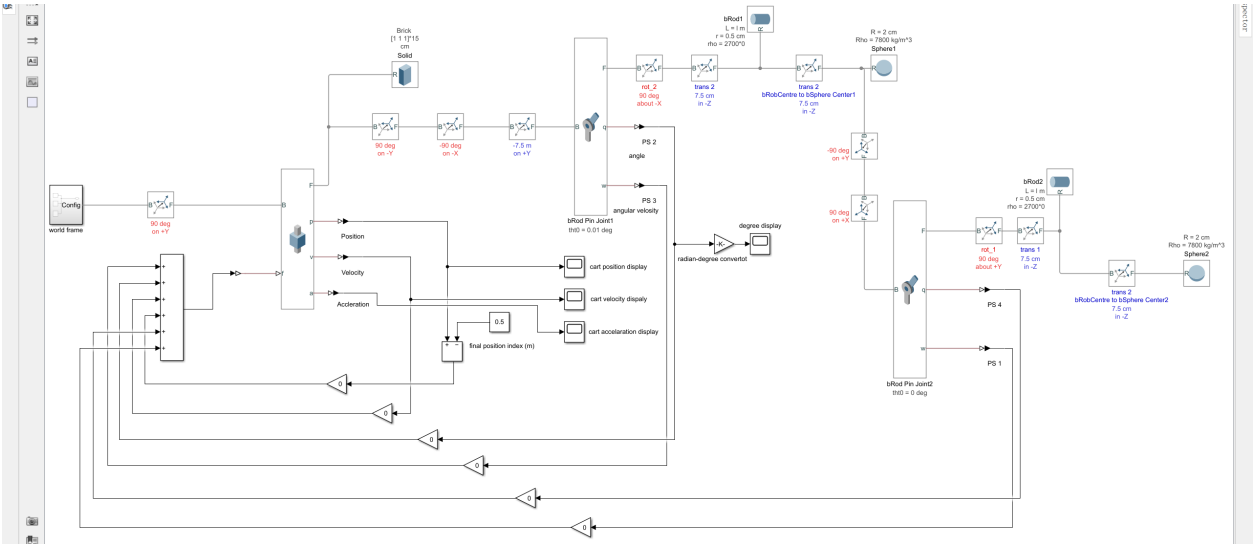

Figure 3: Simulink model configuration

The model features a prismatic joint that characterizes the cart's movement, along with two revolute joints that define the double-pendulum's motion. The system employs the cart's position, speed, and the double-pendulum's angular position and speed as feedback inputs. The controller processes these six feedback parameters and applies force to the cart accordingly.

7.2 LQR controller

The LQR feedback parameters can be conveniently calculated using the $[K, S, P] = lqr(sys, Q, R, N)$ function in MATLAB. The Q matrix here is chosen to be $diag([1, 1, 5, 5, 5, 5])$ and the weight for the input is 0.01 (the input is really not that important here) .And once the feedback gain is obtained and forward to the Simulink, the performace can be obtained by the scope in the above model, which are shown as follows:

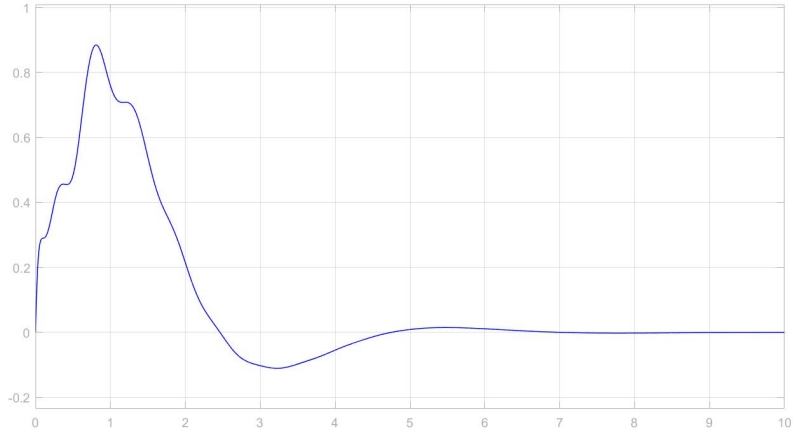

Figure 4: Cart position

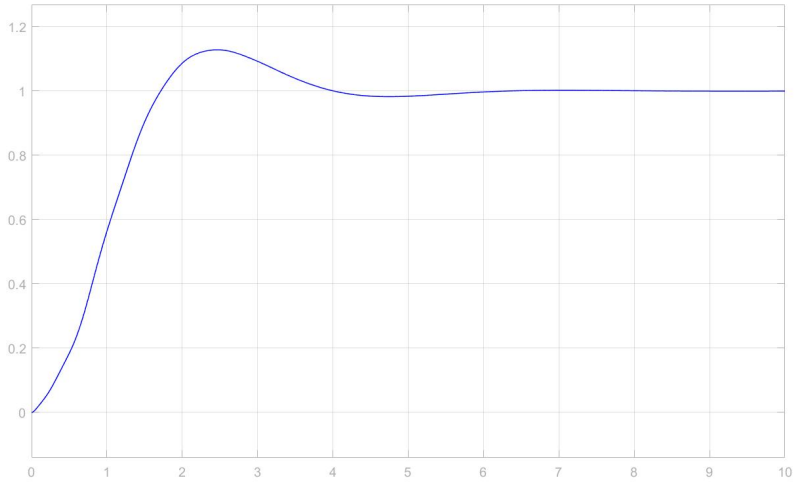

Figure 5: Cart speed

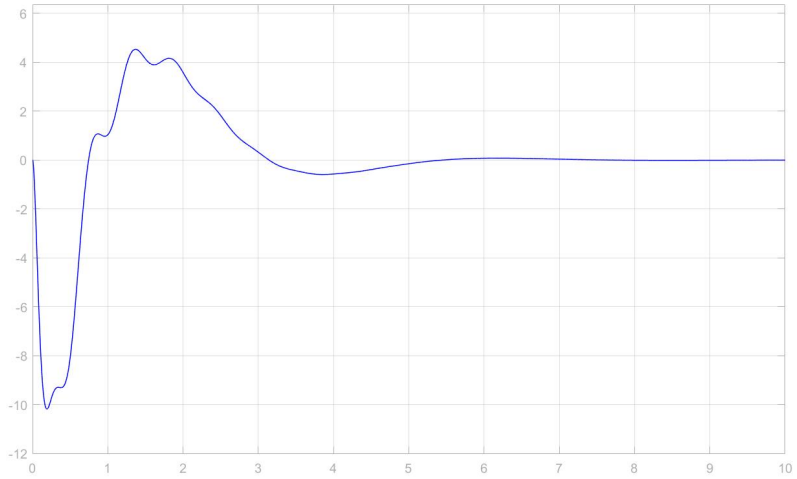

Figure 6: Joint 1 position

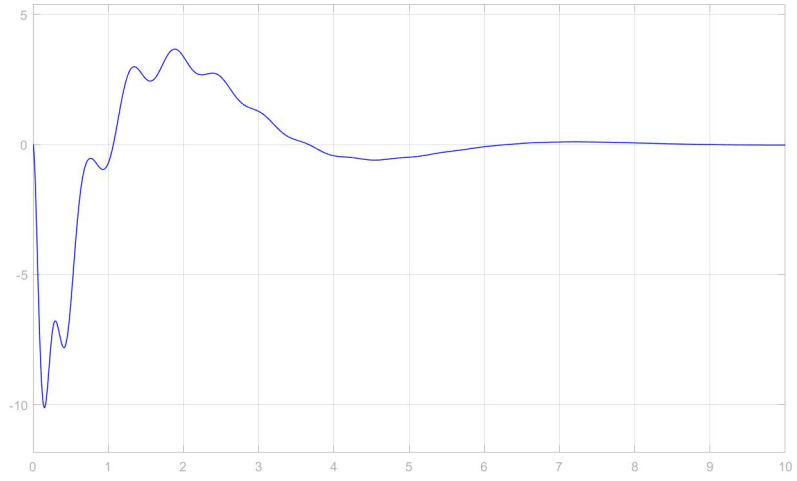

Figure 7: Joint 2 position

As can be seen form the above figure, the system will remain stable after about 7 second. The Q and R matrix can be further modified if certain performance is desired.

8 Double Pendulum control with MPC and Neural Network

In this section, we will explore controlling the double pendulum system using a Model Predictive Control (MPC) and learning the forward kinetics through a Neural Network. Initially, the MPC controller will be implemented and tested within the teacher dynamic (real simulation environment). Following this, the MPC will be employed to generate training data for the Neural Network. After collecting the data, the Neural Network will be trained and the MPC controller, combined with the learned forward dynamic, will be evaluated at various random points. The primary objective of using a Neural Network to learn the forward dynamic is to address its highly non-linear nature. In the previous section, limiting the angle within 5 degrees achieved only modest accuracy, rendering it impractical for real-world applications. By employing a Neural Network, we can more effectively manage the non-linearity of the forward dynamic. This section will demonstrate that even a small-scale Neural Network can yield remarkable accuracy in this context.

8.1 MPC controller design

The controller algorithm that being employed in this project is described as follows:

Algorithm 1: MPC controller

input : Current state x_k
output: Optimized control input sequence $U_k^* = [u_k^*, \ldots, u_k^*]$
Given an action u_k, use sequence $U_k^0 = [u_k, u_k, \ldots, u_k]$ (u_k repeated N times);
"Roll out" the trajectory $X_k^0 = [x_k, x_{k+1}, \ldots, x_{k+N}]$ and compute its cost;
Set $u_k^* = u_k$;
begin
  Initialize x_k;
  while *Improving* **do**
    Compute cost of $U_k^* = [u_k^*, \ldots, u_k^*]$;
    for $i = 0$ *to num_joints* **do**
      $U_k^i = u_k^* \pm \Delta u$ applied to the i-th element;
      Roll out $U_k^i = [u_k^i, u_k^i, \ldots, u_k^i]$;
      if *best* U_k^i *has lower cost than* U_k^* **then**
        $u_k^* \leftarrow$ best u_k^i;
      end
    end
    Execute u_k H times, end up in state x_{k+H};
    Update x_k and u_k^*;
  end
end

Please notice that in the MPC controller a constant action is maintained in the loop of one horizon in order to achieve a higher efficiency. Specifically, the control horizon is chosen to be 10 and u_k will be repeated 40 times. An decayed delta is also introduced in this controller to make small adjustments of towards the end of the control section. To define the cost calculation, a trajectory is firstly recorded for the 40 repeated u_k, the position error to the target and the speed error (since the target is to stop at the goal point, the absolute speed value will be the speed error) will be accumulated throughout the 40 steps. For detailed definition, please refer to the code in

appendix.
The controller is then manual tested on 400 random generated goal point and the position error for each test are plotted as follows:

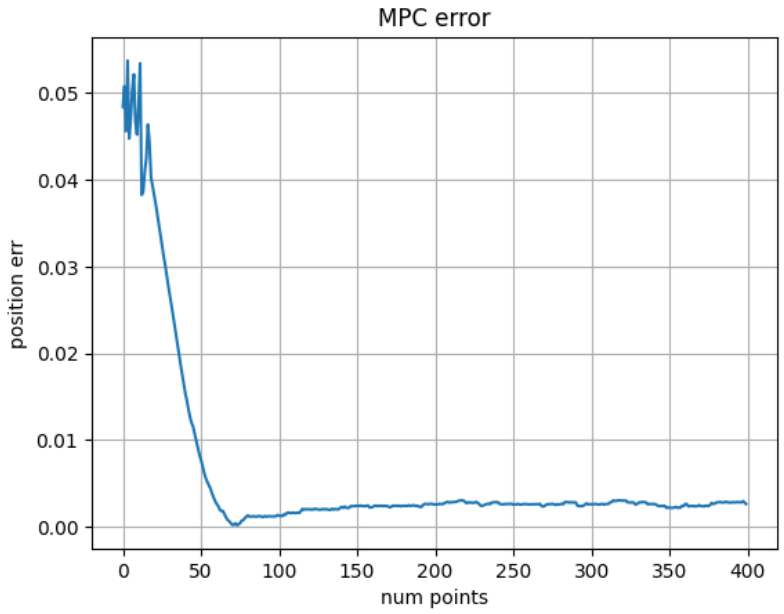

Figure 8: MPC controller manual test

As can be seen, the controller have a very decent accuracy as low as 4×10^{-4} as the training goes. Therefore, this controller is then used to collect the training data.

8.2 Training data collection

To train the NN, a large amount of data with sufficient diversity should be collected. In this project, the data will be collected by the following steps:

1. Generate a random point and set as goal
2. Using MPC controller with real world dynamic to perform the simulation
3. In each step, collected the state of links, and MPC control action (two joints' torque) as the training features; collected the next step's state as the training target. (remember that the NN is trained to learn the forward dynamic.)

By collecting the data in this manner, the training data, especially the action can be more specified at my designed controller, which is of great help when train the forward dynamic.

8.3 Neural network

As previously mentioned, we collected the state of the links (position and speed), and the MPC control action (torque of both joints) as the training features, while using the subsequent step's state as the training target. However, since joint acceleration can be easily calculated if the action is assumed constant within a single time step, the Neural Network can be trained to predict acceleration. Subsequent step states can then be determined based on the predicted acceleration and compared to the actual values to calculate the loss. This method can be illustrated in the following figure:

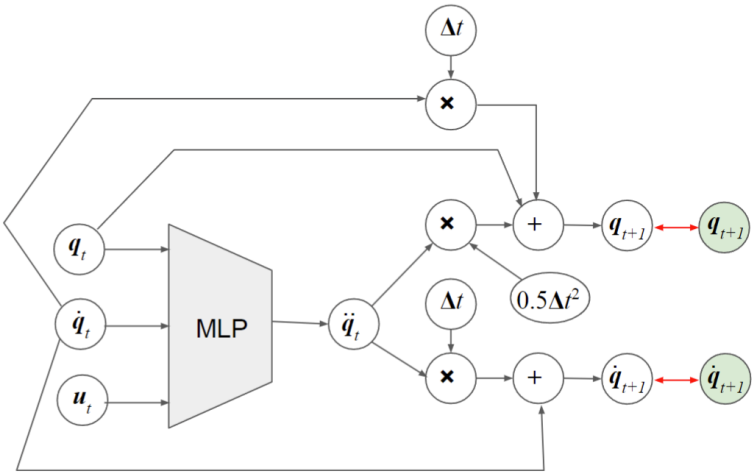

Figure 9: NN train method

The trained loss are illustrated in the following figure:

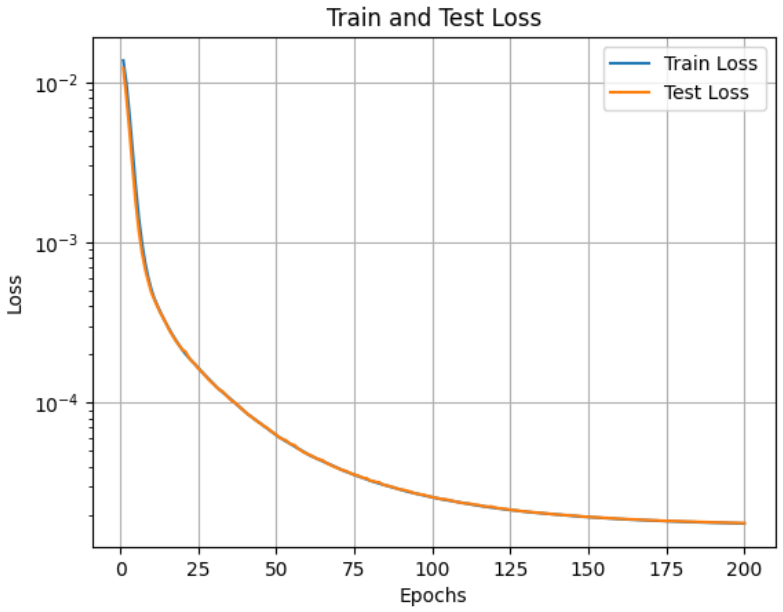

Figure 10: NN train and test loss, 'log' scale y-axis

As can be seem the train and test loss both decrease efficiently without very intense over fitting. The learning forward dynamic will then be applied to the MPC controller on random points.

8.4 MPC with learning dynamic-testing

The testing results with model in each epoch are illustrated in the following two figures: (Please notice the error in the x label, which should be "epoched". Apologize for any confusion.)

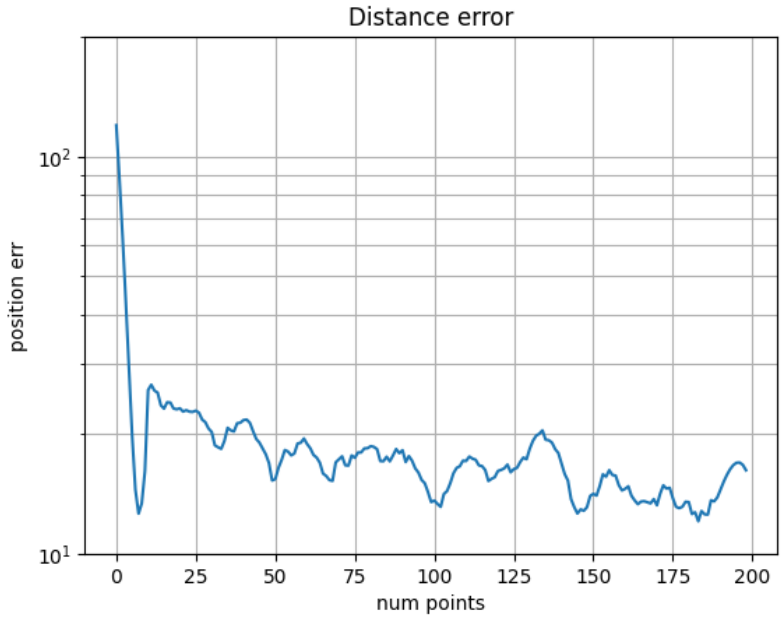

Figure 11: Accumulated position error, 'log' scale y-axis

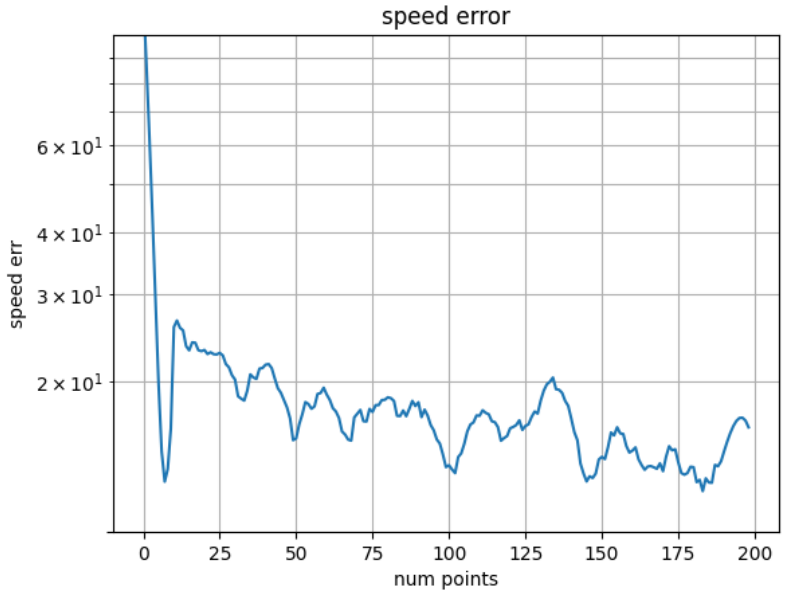

Figure 12: Accumulated speed error, 'log' scale y-axis

Please note that both the position error and speed error represent accumulated errors over 200 time steps, with average position and speed errors falling below 0.1. Taking into account the time required for the MPC to move the endpoint to the goal, the MPC controller, combined with the Neural Network learning the forward dynamic, can effectively control the joint to reach the target point.

9 Deep Q-learning with neural network

In contrast to the previous MPC + NN method, which required the collection of training data, the Deep Q-Learning with Neural Network (DQN) approach does not rely on pre-collected data. Instead, a set of actions (joint torque) will be defined, and the DQN will determine the best action based on the current states. In this section, we will first introduce the DQN class, emphasizing action design. Subsequently, the DQN will be trained specifically for the two-link environment. Finally, the training results will be evaluated to demonstrate the effectiveness and adaptability of the DQN method in achieving accurate and efficient control of the double pendulum system.

9.1 Q network and action definition

The general idea of DQN can be represented by the following figure:

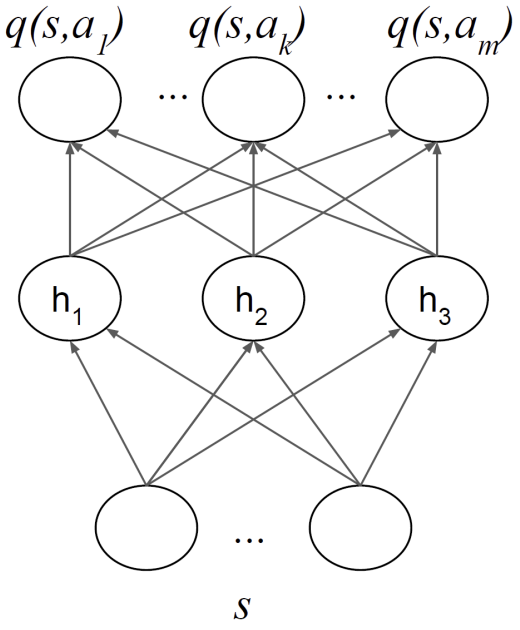

Figure 13: DQN network layout

The net work will input the current state and the goal to decided the most likely action the agent should take (The output is a probability distribution).

Specifically, the actions are designed following the concept of ”up, down, left, right, and stay.” For the two-link system, they can simultaneously add torque, subtract torque, move in opposite directions, or maintain the same action. Furthermore, to enable the system to converge faster and allow for minimal adjustments once the goal is reached, a ”larger” action force is defined for rapid adaptations, while a smaller action force is designated for fine-tuning within close range. Therefore, in this project, the actions are defined to be a list

```
        self.action_mapping = {0: [0, 0],
                               1: [-2, -2],
                               2: [2, -2], 3: [-2, 2], 4: [2, 2],
                               5: [-0.05, -0.025], 6: [0.05, -0.025],
                               7: [-0.05, 0.025], 8: [0.05, 0.025]}
```

When selecting an action, a policy must choose the one that has the highest estimated Q-value for the current state. As part of the QNetwork class, the function: $select\_discrete\_action(...)$ does exactly that. The arm environment itself however expects a 2-dimensional, continuous action vector. Therefore, when it comes to send an action to the environment, the kind of action the environment expects should be provided. To determine how to convert between the discrete action space of your Q-Network and the continuous action space of the arm is achieved by $action\_discrete\_to\_continuous(...)$ function in the QNetwork

9.2 DQN training and reward engineering

In this project, the following procedure are executed to train the DQN. Specifically, reward en-

Algorithm 2: Train a DQN

input : Action network Q^A, target network Q^T, replay memory D, capacity N, episodes M, time steps T, exploration rate ϵ, discount factor γ
output: Trained action network Q^A
begin
  Initialize action network Q^A with random weights; target network $Q^T = Q^A$;
  Initialize replay memory D with capacity N;
  for *episode* $= 1$ *to* M **do**
    Initialize: episode_reward $= 0$; $s_1 =$ random start state; reset robot to match s_1;
    for $t = 1$ *to* T **do**
      With probability ϵ, $a_t =$ random; otherwise $a_t = \max_a Q^A(s_t, a)$;
      Execute a_t and observe r_{t+1}, s_{t+1}; If s_{t+1} is terminal: break (end episode);
      episode_reward $+= r_{t+1}$;
      Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in D (remove old data if needed);
      Train network:;
      Sample random minibatch of transitions from D;
      Perform batch gradient descent on Q^A with
       $L = F(Q^A(s_t, a_t) - [r_t + \gamma \max_a Q^T(s_{t+1}, a)])$;
      Set $s_t = s_{t+1}$;
    end
    Report episode_reward;
    Every k episodes: set target network $Q^T = Q^A$;
  end
end

gineering is further refined to promote faster training convergence. Unlike the default negative reward based on the square of the final position error, which diminishes as the terminal approaches the goal point and may result in a gradient vanishing problem, two periodic rewards are defined to encourage the agent to converge more rapidly, effectively moving closer to the target goal. When the agent reaches an area within a range of 0.3 from the target, a reward of 10 will be granted. If the agent moves even closer, within 0.1 of the target, an additional reward of 10 will be provided. In this manner, the agent will initially learn to swiftly move the terminal within a range of 0.3 and subsequently attempt to maintain it within the 0.1 range. The train loss and reward are plotted in the following figures:

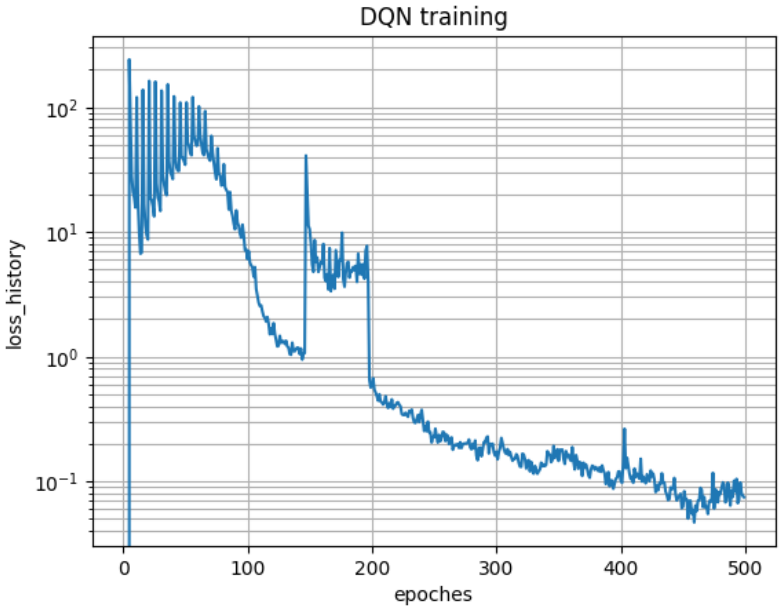

Figure 14: DQN training loss

The initial several epochs does not calculate the loss in order to prevent the larger gradient variation. As can be seen, the train process functions effectively, and the train loss can decent to a low value.

9.3 DQN testing

The model of every 5 epoch is saved and tested on several random points. The result are shown as follows:

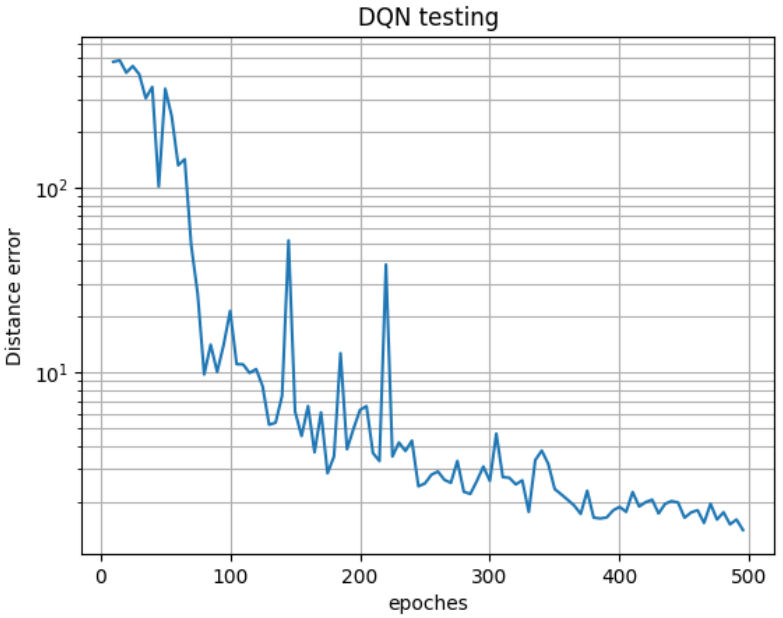

Figure 15: DQN training loss

as can be seen, the testing accumulative position error largely have the same tendency as the

training loss. And the method of DQN have proved to be effective in set as a controller of this project.

10 Proximal Policy Optimization

In this section, the PPO algorithm in stable baselines 3 is deployed directly. Here, a brief process introduction of PPO will be demonstrated.

1. Problem formulation: The task should be framed as a reinforcement learning problem. A continuous state space and action space is chosen for this problem. The state includes the angles and angular velocities of the two joints, while the action space comprise the torque applied to each joint.
2. Reward function: A negative distance between the terminal and the target is defined as the reward function
3. Environment: 8 parallel simulation environment are created. These environment provide the agent with information of state, the reward, and whether the episode has ended.
4. Policy network: A policy network, represented by a neural network, ought to be employed to map states to actions. The input to this network is the state, and the output corresponds to the actions taken by the agent. The neural network will learn a policy for controlling the two-joint link through the PPO algorithm.
5. PPO algorithm: The PPO algorithm should be applied to optimize the policy network. PPO is designed to balance exploration and exploitation by limiting the policy update to avoid significant deviations from the old policy. Two main components are involved in this algorithm: the surrogate objective and the clipping mechanism. The surrogate objective is calculated as the product of the advantage estimate and the probability ratio between the new and old policies, while the clipping mechanism constrains the policy updates within a pre-defined range to maintain stability.

As the training of the PPO is not modifiable, several selected intermediate models are saved and tested. The result are shown as follow: (Please notice that the X axis should be the list $range(500, 63500, 500)$)

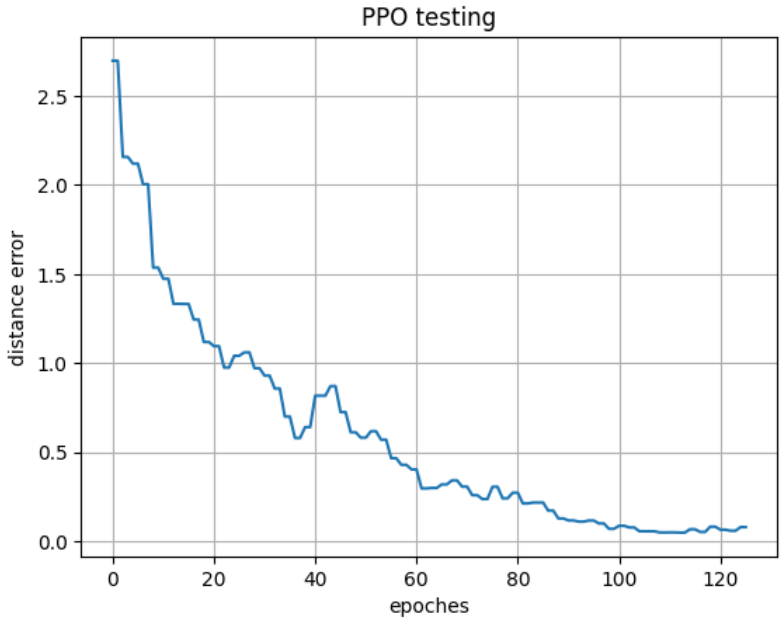

Figure 16: DQN training loss

As can be seen, the position error decreases effectively with the train process.

11 Conclusion

This project has successfully focused on improving the transportation of containers with water. The problem was initially demonstrated as a double cart and double pendulum control issue, followed by the development of a mathematical description of the system. Subsequently, the system was linearized to generate the state space expression. The observability and controllability of the system were introduced, followed by the construction of the model in MATLAB Simulink and the design of an LQR controller.

However, the linearization of the system resulted in a limitation, where the joint angle could not exceed 5 degrees, greatly restricting the controller's application scenarios. Therefore, in addition to traditional computational orientation control methods, this project also explored data-oriented approaches. Specifically, the MPC controller was utilized to generate training data for the forward dynamic. A Neural Network was then trained using the generated data to handle the considerable non-linearity of the forward system. Subsequently, a Deep Q-Network was employed as the controller. DQN does not require training data and can be applied to various problems. Finally, the PPO algorithm from the Reinforcement Learning domain was implemented to further demonstrate the power of data-oriented methods.

a key feature of this project was to compare algorithm-oriented control methods and data-oriented approaches, both of which demonstrated significant potential. In the future, these two sets of methods could be combined to allow the algorithm-oriented approach to guide the direction, while the Neural Network handles the computationally intensive tasks. This hybrid approach could further enhance the performance and applicability of control systems for smoother transportation of containers with water and other related applications.

Overall, though the initial objectives are slightly shifted, the project has achieved its goal.

References

[1] B. I. Rabinovich, V. G. Lebedev, and A. I. Mytarev, *Vortex Processes and Solid Body Dynamics*, ser. Fluid Mechanics and its Applications, R. Moreau, Ed. Dordrecht: Springer Netherlands, 1994, vol. 25. [Online]. Available: http://link.springer.com/10.1007/978-94-011-1038-9

Appendix

In the appendix, related code will be provided

LQR in Matlab:

```
close all
clear all
clc

l1 = 0.15;
l2 = 0.15;
mc = 1;
m1 = 0.1;
m2 = 0.1;
g = 9.8 ;

A=[0 1 0 0 0 0;
   0 0 (m1+m2)*g/mc 0 0 0;
   0 0 0 1 0 0;
   0 0 (m2*mc+m1*m2+mc*m1+m1^2)*g/(l1*mc*m1) 0 -m2*g/(m1*l1) 0;
   0 0 0 0 0 1;
   0 0 -(m2+m1)*g/(m1*l2) 0 (m2+m1)*g/(m1*l2) 0 ];

B=[0 ; 1/mc ; 0 ; -1/mc/l1; 0; -1/mc/l2];

Q=[1 0 0 0 0 0;
     0 1 0 0 0 0;
     0 0 5 0 0 0;
     0 0 0 5 0 0;
     0 0 0 0 5 0;
     0 0 0 0 0 5] ;

  R= 0.01 ;

  K=lqr(A,B,Q,R) ;

  k1 = K(1);
  k2 = K(2);
  k3 = K(3);
  k4 = K(4);
  k5 = K(5);
  k6 = K(6);
```

MPC in python

```
class MPC:

  def __init__(self):
       self.control_horizon = 10
       self.repeat_n = 40
       self.Δ_u_max = 0.1
       self.Δ_u_min = 0.01
       self.Δ_u_decay = 0.99
       self.position_weight = 80
       self.velocity_weight = 1
       self.update_max = 40



  def compute_cost(self, dynamics, trajectory, goal):
       position_error = 0
       velocity_error = 0
       # print(trajectory)
       for state in trajectory:
          pos_ee = dynamics.compute_fk(state)
          position_error += np.linalg.norm(goal - pos_ee)**2
          vel_ee = dynamics.compute_vel_ee(state)
```

```
            velocity_error += np.linalg.norm(vel_ee)**2

        # vel_ee = dynamics.compute_vel_ee(state)
        # velocity_error = np.linalg.norm(vel_ee)**2
        # print('position_error', position_error)
        # print('velocity_error', velocity_error)
        cost = position_error * self.position_weight + velocity_error * ...
          self.velocity_weight
        return cost


    def rollout_trajectory(self, dynamics, state, U):
        num_steps = len(U)
        trajectory = [np.zeros_like(state) for _ in range(num_steps+1)]
        # trajectory[0] = state
        for i in range(len(U)):
            state = dynamics.advance(state, U[0])
            trajectory[i] = state
        return trajectory

    def compute_action(self, dynamics, state, goal, action):
      # Put your code here. You must return an array of shape (num_links, 1)
        N = self.repeat_n
        H = self.control_horizon
        num_joints = dynamics.get_action_dim()
        # print('action', action)
        best_action = np.copy(action).reshape(-1)
        U_star = np.tile(best_action, (N, 1))

        for i in range(num_joints - 1, -1, -1): # update from the largest joint
            best_cost = float('inf')
            update_flag = True
            update_counter = 0

            while update_flag and update_counter < self.update_max:


                update_counter += 1
                Du = self.△_u_max * self.△_u_decay ** update_counter
                Du = np.max([Du, self.△_u_min])
                U_i_plus = np.copy(U_star)
                U_i_minu = np.copy(U_star)
                U_i_plus[:, i] += Du
                U_i_minu[:, i] -= Du

                trajectory_i_plus = self.rollout_trajectory(dynamics, state, U_i_plus)
                cost_i_plus = self.compute_cost(dynamics, trajectory_i_plus, goal)
                trajectory_i_minu = self.rollout_trajectory(dynamics, state, U_i_minu)
                cost_i_minu = self.compute_cost(dynamics, trajectory_i_minu, goal)

                min_cost = np.min([best_cost, cost_i_plus, cost_i_minu])


                update_flag = False
                if min_cost < best_cost:
                    update_flag = True
                    # print('update')
                    best_cost = min_cost
                    index = np.argmin([cost_i_plus, cost_i_minu])
                    combi = [U_i_plus, U_i_minu]
                    U_star = combi[index]

        result = U_star[0].reshape([-1, 1])

        return result
```

DNN model

```
class Model(nn.Module):
```

```
    def __init__(self, num_links, time_step):
        super().__init__()
        self.num_links = num_links
        self.time_step = time_step

    def forward(self, x):
        qddot = self.compute_qddot(x)
        state = x[:, :2*self.num_links]
        next_state = self.compute_next_state(state, qddot)
        return next_state

    def compute_next_state(self, state, qddot):
        pos = state[:, :self.num_links]
        vel = state[:, self.num_links: 2 * self.num_links]

        new_vel = vel + qddot * self.time_step
        new_pos = pos + vel * self.time_step + qddot * 0.5 * self.time_step ** 2

        output = torch.cat([new_pos, new_vel], dim=1)
        return output

    def compute_qddot(self, x):
        pass

class Model2Link(Model): #
  def __init__(self, time_step):
    super().__init__(2, time_step)
    # Your code goes here
    self.layer1 = nn.Linear(6, 512)
    self.layer2 = nn.Linear(512,64)
    self.layer3 = nn.Linear(64,2)

  def compute_qddot(self, x):
     y = F.relu(self.layer1(x))
     y = F.relu(self.layer2(y))
     y = self.layer3(y)
     # print(y)
     # y = self.compute_next_state(state, y)
     return y

  def predict(self, features):
     self.eval()
     features = torch.from_numpy(features).float()
     features = features.unsqueeze(0)
     return self.forward(features).detach().numpy()
```

Q-network model

```
class QNetwork(nn.Module):
    def __init__(self, env):
        super(QNetwork, self).__init__()

        # torque_range_1 = np.arange(-1, 1.5, 0.5)
        # torque_range_2 = np.arange(-0.5, 0.75, 0.25)
        # torque_combinations = [np.array([torque_1, torque_2]) for torque_1 in ...
          torque_range_1 for torque_2 in torque_range_2]
        # # random.shuffle(torque_combinations)

        # action_mapping = {}
        # for action_idx, torque_pair in enumerate(torque_combinations):
        #    action_mapping[action_idx] = torque_pair
        self.action_mapping = {0: [0, 0],
                               1: [-2, -2],
                               2: [2, -2], 3: [-2, 2], 4: [2, 2],
                               5: [-0.05, -0.025], 6: [0.05, -0.025],
                               7: [-0.05, 0.025], 8: [0.05, 0.025]}

        self.obs_dim = env.observation_space.shape[0]
        self.action_dim = len(self.action_mapping)
```

```
21
22          self.fc1 = nn.Linear(self.obs_dim, 128)
23          self.fc2 = nn.Linear(128, 32)
24          self.fc3 = nn.Linear(32, self.action_dim)
25
26
27
28      def forward(self, x, device):
29          x = F.relu(self.fc1(x))
30          x = F.relu(self.fc2(x))
31          x = self.fc3(x)
32          return x
33
34      def select_discrete_action(self, obs, device):
35          # Put the observation through the network to estimate q values for all possible ...
              discrete actions
36          obs = torch.Tensor(obs)
37          est_q_vals = self.forward(obs.reshape((1,) + obs.shape), device)
38          # Choose the discrete action with the highest estimated q value
39          discrete_action = torch.argmax(est_q_vals, dim=1).tolist()[0]
40          return discrete_action
41
42      def action_discrete_to_continuous(self, discrete_action):
43          continuous_action = self.action_mapping[discrete_action]
44          return continuous_action
```

DQN training model

```
1   class TrainDQN:
2
3
4       def __init__(self, env, seed=0):
5           np.random.seed(seed)
6           torch.manual_seed(seed)
7           torch.backends.cudnn.deterministic = False
8           self.env = env
9           self.device = torch.device('cpu')
10          self.q_network = QNetwork(env).to(self.device)
11          self.target_network = QNetwork(env).to(self.device)
12          self.target_network.load_state_dict(self.q_network.state_dict())
13
14          self.env.seed(seed)
15          self.env.observation_space.seed(seed)
16
17          self.num_action = 9
18
19          self.epsilon = 0.8
20          self.epsilon_dacay = 0.95
21          self.epsilon_min = 0.2
22
23          self.replay_buffer = ReplayBuffer(10000)
24
25          self.batch_size = 128
26          self.gamma = 0.95
27          self.criterion = nn.MSELoss()
28          self.lr = 0.001
29          self.optimizer = optim.Adam(self.q_network.parameters(), lr=self.lr)
30          self.num_episodes = 500
31
32          self.q_update_num = 5
33
34
35      def to_tensor(self, states, action_indexs, rewards, next_states):
36          states = torch.from_numpy(states).float()
37          action_indexs = torch.from_numpy(action_indexs).long()
38          rewards = torch.from_numpy(rewards).float()
39          next_states = torch.from_numpy(next_states).float()
40          return states, action_indexs, rewards, next_states
41
```

```
    def train_batch(self):
        self.optimizer.zero_grad()
        states, action_indexs, rewards, next_states, dones = ...
          self.replay_buffer.sample(self.batch_size)
        states, action_indexs, rewards, next_states = self.to_tensor(states, ...
          action_indexs, rewards, next_states)

        q_a = self.q_network.forward(states, self.device).gather(1, ...
          action_indexs.view(-1, 1)).squeeze()

        q_t = self.target_network.forward(next_states, self.device)

        q_t_max = torch.max(q_t.detach(), dim=1)[0]
        q_tar = rewards + self.gamma * q_t_max

        loss = self.criterion(q_a, q_tar)
        loss.backward()

        # clamp
        for para in self.q_network.parameters():
            para.grad.data.clamp_(-0.05, 0.05)

        self.optimizer.step()

        return loss.item()

    def get_reward(self, target_goal, info):

      positiona_error = np.linalg.norm(info['pos_ee'] - target_goal)
      reward_temp = - positiona_error ** 2
      if positiona_error < 0.3:
        reward_temp += 10
       # print('positiona_error', positiona_error)
      if positiona_error < 0.1:
        reward_temp += 10
      #   # print('positiona_error', positiona_error)
      # elif positiona_error < 0.1:
      #   reward_temp += 10
      #   # print('positiona_error', positiona_error)
      # elif positiona_error < 0.05:
      #   reward_temp += 10
      #   # print('wowowo_positiona_error', positiona_error)


      return reward_temp


    def train(self):
        best_reawrd = -np.inf
        deacied_epsilon = self.epsilon
        loss_history = []
        reward_history = []
        for episode in range(self.num_episodes):

            # goal = np.array([[0.5], [-1.5]])
            episode_reward = 0
            state = self.env.reset()
            # print(state)
            target_goal = self.env.goal
            # print(target_goal)
            loss = 0

            deacied_epsilon = np.max([self.epsilon_min, deacied_epsilon * ...
              self.epsilon_dacay])
            # print(deacied_epsilon)
            while True:

                if random.random() < deacied_epsilon:
                    index = np.random.randint(0, self.num_action)
                else:
```

```
                index = self.q_network.select_discrete_action(state, self.device)

            action = self.q_network.action_discrete_to_continuous(index)

            state_next, reward, done, info = self.env.step(action)
            if done:
                break

            self.replay_buffer.put((state, index, reward, state_next, done))

            reward_temp = self.get_reward(target_goal, info)

            episode_reward += reward_temp

            if len(self.replay_buffer.buffer) ≥1000:
                loss += self.train_batch()
            state = state_next

        if episode % self.q_update_num == 0:
            self.target_network.load_state_dict(self.q_network.state_dict()) # update num
            self.save_model(episode)



        loss_history.append(loss)

        # print('loss', loss)
        reward_history.append(episode_reward)
        # if episode_reward > best_reawrd:
        #   best_reawrd = episode_reward
        #   print("Episode:", episode, " reward:", episode_reward)
        #   self.save_model(episode)

        # if episode > (self.num_episodes - 10):
        #   print("Episode:", episode, " reward:", episode_reward)
        #   self.save_model(episode)

        # if episode % 5 == 0:
        #   self.save_model(episode)

        # if episode_reward < 1500:
        #   print('bad point', target_goal)
        #   print("Episode:", episode, " reward:", episode_reward)

    return loss_history, reward_history




def save_model(self, episode_num, save_dir='models'):

    timestr = time.strftime("%Y-%m-%d_%H-%M-%S")
    model_dir = os.path.join(save_dir)
    if not os.path.exists(os.path.join(model_dir)):
        os.makedirs(os.path.join(model_dir))
    savepath = os.path.join(model_dir, f'q_network_ep_{episode_num:04d}.pth')
    torch.save(self.q_network.state_dict(), savepath)
    print(f'model saved to {savepath}\n')
```

PPO model:

```
# Default parameters
timesteps = 500000
nenv = 8
seed = 8
batch_size = 1024
save_interval = 500

```

```
save_dir = f'ppo_models_4'
set_random_seed(seed)

arm = make_arm()

vec_env = make_vec_env(arm=arm, nenv=nenv, seed=seed)

model = PPO("MlpPolicy", vec_env, n_steps=batch_size)

class SaveIntermediateModelCallback(BaseCallback):
    def __init__(self, save_interval, save_dir):
        super(SaveIntermediateModelCallback, self).__init__()
        self.save_interval = save_interval
        self.save_dir = save_dir

    def _on_step(self) -> bool:
        if self.n_calls % self.save_interval == 0:
            save_path = os.path.join(self.save_dir, f"ppo_network_step_{self.n_calls}.zip")
            self.model.save(save_path)
            print(f"Saved model at time step {self.n_calls}")
        return True

if not os.path.exists(save_dir):
    os.makedirs(save_dir)

save_callback = SaveIntermediateModelCallback(save_interval, save_dir)

model.learn(total_timesteps=timesteps, callback=save_callback)
```