

UNIVERSITY OF COLORADO AT BOULDER

COEN 5830 - INTO. TO ROBOTICS

FALL 2024

PROFESSOR: DR. LEO BEUKEN

TEACHING ASSISTANT: SRIKRISHNA RAGHU

Final Project

TEAM MEMBERS:

IAN MCCONACHIE

MICHAEL BERNABEI

December 14, 2024

CONTENTS

1	Problems Attempted	3
2	Perception	4
2.1	Lidar Occupancy Grid - 2 Points	4
2.1.1	Lidar Occupancy Algorithm High-level Steps	4
2.2	IMU Sensor Fusion - 1 Point	4
2.2.1	The Complimentary Filter	4
2.2.2	The Complimentary Filter Implementation	4
2.3	Canny Edge Detection - 1 + 1 Bonus Point	6
3	Object Oriented Programming	7
3.1	OO Programming - Warehouse Robot Management - 3 points + 1 Bonus	7
4	Path Planning	7
4.1	RRT - 3 points	7
4.2	Deploying the solution - 1 bonus points	8
4.2.1	Deploying the Solution Noticed Issues	8
5	Dynamics & Control	9
5.1	Dynamic System Modeling - 2 Points	9
5.2	Inverted Pendulum Control - 1 point	11
5.3	Inverted Pendulum State Control - 1 point	12

LIST OF ALGORITHMS

1	Lidar Occupancy Grid Algorithm	4
2	Complimentary Filter Implementation	5
3	Simple Canny Edge Implementation	6
4	OOP Warehouse Robot Management	7
5	Webots RRT Path Planning	8
6	RRT Path Planning Replay Velocities	8
7	Feedback Control	12

1 PROBLEMS ATTEMPTED

- Perception - Lidar Occupancy Grid - 2 Points
- Perception - IMU Sensor Fusion - 1 Points
- Perception - Canny Edge Detection - 1 Point + 1 Bonus
- Object Oriented Programming - Warehouse Robot Management - 3 Points + 1 Bonus
- Path Planning - RRT - 3 Points
- Path Planning - RRT Deploying the solution - 1 bonus point
- Dynamics & Control - 5.1 Part A - 2 Points
- Dynamics & Control - 5.1 Part B - 1 Points
- Dynamics & Control - 5.1 Part C - 1 Points

Total points attempted: 17

2 PERCEPTION

2.1 LIDAR OCCUPANCY GRID - 2 POINTS

For the lidar occupancy grid problem we elected to use the **Bresenham Line Algorithm** to rasterize the rays throughout the environment. We chose this method after Dr. Leo highlighted the benefits of the line algorithm for rasterizing rays. One particularly amazing feature of the algorithm is the use of only integer arithmetic in the calculation of occupancies. Using only integer calculations in the occupancies reduces computationally complexity and therefore increases speed. Now since Bresenham's line algorithm is the heart of our work, we will describe the steps in our implementation that lead up to and after the Bresenham portion of our code.

2.1.1 LIDAR OCCUPANCY ALGORITHM HIGH-LEVEL STEPS

In this section we will highlight the main steps in our algorithm for this part of the project. We will try to stick to plain old english when describing these steps.

Algorithm 1 Lidar Occupancy Grid Algorithm

- 1: Read in lidar, heading, and position data then store in internal data structures
 - 2: Create Grid based off passed in lidar CSV data
 - 3: Start animation loop
 - 4: **while** True **do**
 - 5: Convert incoming lidar readings and robot position data to grid coordinates
 - 6: Mark current position of robot on grid
 - 7: Rasterize lidar readings using Bresenham's line algorithm and update grid
 - 8: Rescale grid x,y axis to real world x,y axis
 - 9: display animation, i.e. show state of lidar scan to user
 - 10: **end while**
-

The above algorithm highlights the main steps in the occupancy grid algorithm. As stated above, the details of the code will be highlighted on request at the oral examination.

2.2 IMU SENSOR FUSION - 1 POINT

2.2.1 THE COMPLIMENTARY FILTER

After some research, we elected to implement a simple **Complimentary filter** for fusing accelerometer and gyro data. The Complimentary filter works well for determining the orientation of an object because it filters out high frequency noise from the gyro estimates and the low frequency noise from the accelerometer and then blends the measurements. Essentially, the filter reduces the White Gaussian noise from the accelerometer measurements and reduces the noise introduced from integrating the gyro measurements, i.e. performing dead reckoning.

2.2.2 THE COMPLIMENTARY FILTER IMPLEMENTATION

We extracted the equations for things such as accelerometer tilt angle, gyro integration procedure, and the Complimentary filter blending equation from this [Complimentary Filter Website](#). The filter is

depicted in the equation below.

$$\begin{aligned}\theta_{roll} &= \alpha\theta_{integrated\ roll\ gyro} + (1 - \alpha)\theta_{accelerometer} \\ \theta_{roll} &= \alpha\theta_{integrated\ pitch\ gyro} + (1 - \alpha)\theta_{accelerometer}\end{aligned}$$

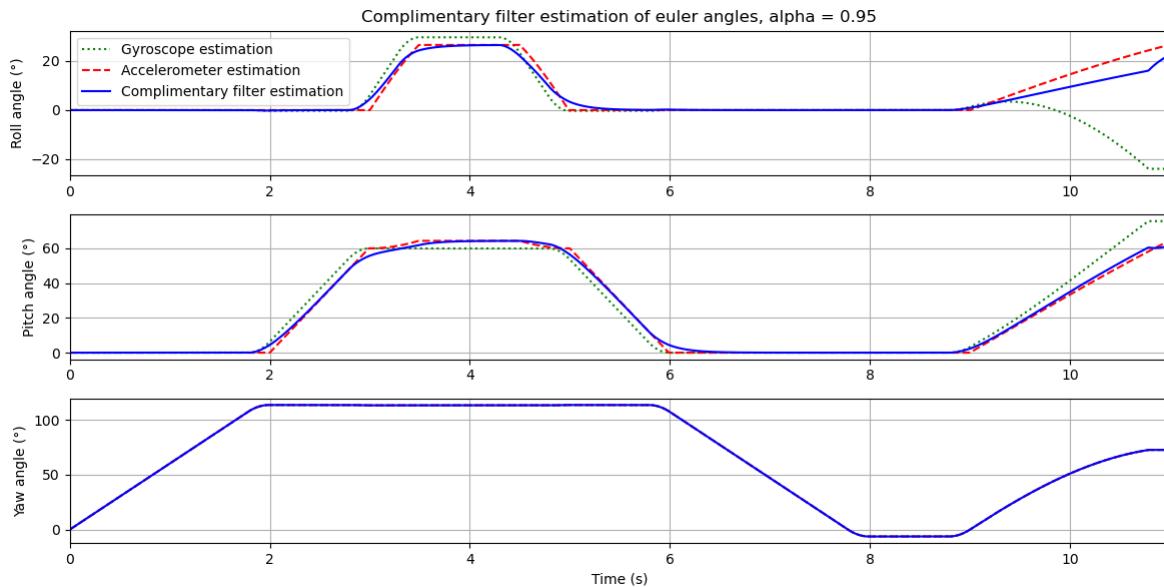
Our main take away in studying this filter is that it is a **poor man's Kalman filter**. Where α is similar to a Kalman gain, but instead of blending dynamics and measurements we are blending measurements from the accelerometer and gyro. α helps us decide how much we should trust the gyro estimates and the accelerometer updates. That is, a large α indicates trusting gyro measurements over accelerometer measurements. This is easily seen by setting $\alpha = 1$, then,

$$\begin{aligned}\theta_{roll} &= \theta_{integrated\ roll\ gyro} + (1 - 1)\theta_{accelerometer} = \theta_{integrated\ roll\ gyro} \\ \theta_{roll} &= \theta_{integrated\ pitch\ gyro} + (1 - 1)\theta_{accelerometer} = \theta_{integrated\ pitch\ gyro}\end{aligned}$$

Hence, when $\alpha = 1$, no accelerometer measurements are fused. The high level description of our implemented Complimentary filter follows:

Algorithm 2 Complimentary Filter Implementation

- 1: Read in IMU values
 - 2: Parse values
 - 3: Calculate time difference for gyro scope integration
 - 4: **while** samples available **do**
 - 5: Integrate Roll, Pitch, and Yaw gyro values
 - 6: Calculate accelerometer angles
 - 7: Filter accelerometer and gyro values based on α value
 - 8: Save filtered values
 - 9: **end while**
 - 10: Plot result
-



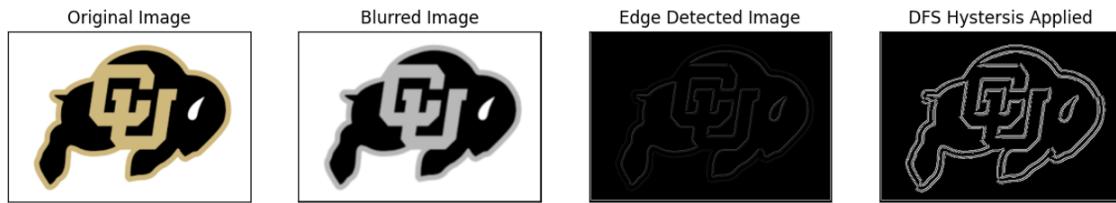
2.3 CANNY EDGE DETECTION - 1 + 1 BONUS POINT

For Canny edge detection, we implemented each of the steps necessary for the algorithm and then compared our results with those of OpenCV. We noticed that OpenCV's implementation of the Canny edge detection algorithm was much faster and yielded better results. We attribute this to years of optimization under the hood and more eye's looking at their Canny edge implementation code. We noticed that tuning our low and high thresholds in the hysteresis step would yield better or worse results depending on how we tuned the thresholds. We feel OpenCV has a clever way to select these values and therefore yield better results.

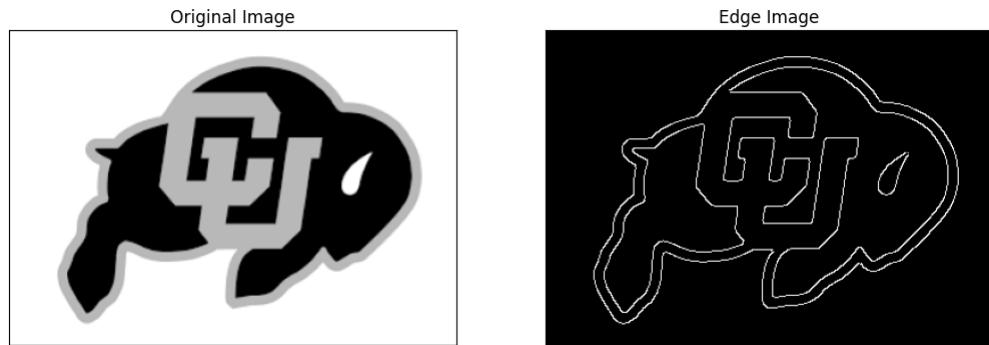
Algorithm 3 Simple Canny Edge Implementation

- 1: Load color image
 - 2: Convert loaded image to gray scale image
 - 3: Apply Gaussian blur to gray scale image to reduce noise
 - 4: Apply non-maximum suppression to then edges
 - 5: Perform depth first search hysteresis thresholding
-

In the image below, we have our implementation of Canny. Each of the sub-images correspond to the steps in the previously mentioned algorithm. Our implementation does a fairly good job of detecting the lines in the image.



However, in the image below, we see OpenCV's Canny implementation and notice their final image looks better. Again, we believe this is some optimizations made in their hysteresis step.



Finally, to ensure we get credit for the bonus point. **It is important to note that we implemented a DFS using a stack and while loop to perform the hysteresis step in the Canny edge detection algorithm.**

3 OBJECT ORIENTED PROGRAMMING

3.1 OO PROGRAMMING - WAREHOUSE ROBOT MANAGEMENT - 3 POINTS + 1 BONUS

For this part of the project, we implemented a robot parent class and the child classes as instructed. We then created two custom classes for our project. The first of these classes is the Warehouse class. Its responsibilities include managing the state of the map, conducting movement across the map, and showing the state of the map with robots on it to the user. The second class is an implementation of Dijkstra's search algorithm. We know there are more optimized search algorithms such as A^* ; however, we only need a simple solution for finding the shortest path. In short, the project required implementation of the [Mediator Design Pattern](#) that utilizes a shortest path search algorithm for movement from the start to the goal.

Algorithm 4 OOP Warehouse Robot Management

```
1: Randomly create child robots
2: Generate random start points for child robots
3: Generate random goals for child robots
4: while not all robot goals reached do
5:   Warehouse requests all robots advance towards their goals
6:   Each robot uses Dijkstra's search algorithm to find the shortest path
7:   Generated paths are inspected by the Warehouse
8:   if no collisions occur then
9:     Warehouse gives the go-ahead for robots to move 1 spot
10:  else
11:    Resolve collisions using methods described in the project file
12:  end if
13:  if robot reached goal then
14:    Add to goal reached
15:  end if
16: end while
```

The warehouse management code is also generalizable to an $n \times n$ grid. The user can specify the grid size and the robot number independently.

4 PATH PLANNING

4.1 RRT - 3 POINTS

Srikrishna provided the pseudo code for this assignment in the RRT puck and supervisor Python skeleton code. The only real algorithm we had to implement for this program besides implementing Srikrishna's pseudo code was a communication protocol between the Supervisor and the E-Puck.

Algorithm 5 Webots RRT Path Planning

```
1: Start Webots and load RRT Puck and Supervisor files
2: Webots automatically initializes global variables for RRT Puck and Supervisor
3: while timestep != 1 do
4:   Randomly sample map with 50 percent goal biasing
5:   Search position node tree for closest node CN to randomly sampled node
6:   teleport robot to node CN
7:   Inform RRT puck that it should perform 5 Monte Carlo propagations
8:   if Five Monte Carlo propagations successful then
9:     Add the propagation that is closest to randomly sampled node CN to the tree
10:    Set closet propagation node parent to CN
11:   end if
12:   if latest propagation within collision bounds of goal then
13:     stop simulation and reply velocities
14:   end if
15: end while
```

4.2 DEPLOYING THE SOLUTION - 1 BONUS POINTS

For this part of the project we simply added on to the work we did in the previous section by saving the actions. Actions are data structures that hold the velocities and the duration for which those velocities were executed. After the goal is reached, we simply send these actions to the RRT Puck process and inform the puck to replay the velocities.

Algorithm 6 RRT Path Planning Replay Velocities

```
1: if goal reached then
2:   Have Supervisor send saved actions to E-puck
3: end if
4: if actions received by E-puck then
5:   Replay actions on E-puck
6: end if
```

4.2.1 DEPLOYING THE SOLUTION NOTICED ISSUES

In replaying the actions back to the E-puck we noticed that the longer the simulation ran before finding its collision threshold, the more off target the E-puck would be when replaying the velocities. To test our theory we lowered our GOAL COLLISION THRESHOLD to .7 and compared the final path of the E-puck with the replayed path. We did the same for the GOAL COLLISION THRESHOLD to .1. The results are depicted below.

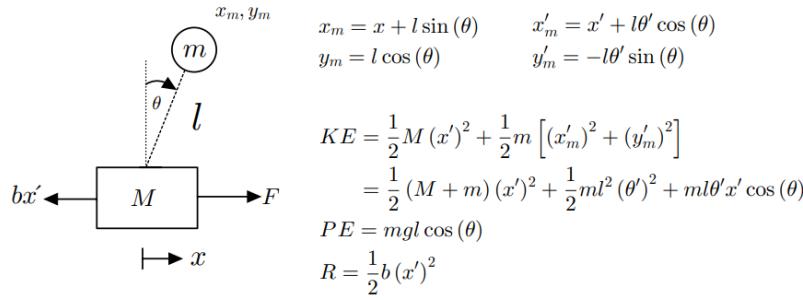


In the image above, we see that the shorter the simulation runs the more closely the final path matches the replayed path. And the longer the simulation runs, the more inaccurate the replayed path is. We believe this is because when replaying the velocities and durations we are essentially doing **DEAD RECKONING**. In other words, we are integrating our velocities and durations in the hope that they lead to our final goal. However, small errors in the replayed durations and velocities accumulate over the course of the simulation and lead to more and more drift from our final goal.

5 DYNAMICS & CONTROL

5.1 DYNAMIC SYSTEM MODELING - 2 POINTS

In this project, our goal is to control an inverted pendulum on a cart. We begin by developing a simulation to model the cart's dynamics over time. First, we define the states of interest. Then, using Lagrangian mechanics, we calculate the system's energy and derive the equations of motion by minimizing the Lagrangian. To account for non-conservative forces, we incorporate a Rayleigh dissipation function into the Lagrange equations to represent damping. The detailed mathematical process is outlined below.



$$L = KE - PE = \frac{1}{2}(M+m)(x')^2 + \frac{1}{2}ml^2(\theta')^2 + ml\theta'x' \cos(\theta) - mgl \cos(\theta)$$

for x

$$\frac{d}{dt}\left(\frac{\partial L}{\partial x'}\right) - \frac{\partial L}{\partial x} + \frac{2R}{\partial x'} = F$$

$$\frac{d}{dt}[(M+m)x' + ml\theta' \cos(\theta)] + bx' = F$$

$$(M+m)x'' + ml\theta'' \cos(\theta) - ml(\theta')^2 \sin(\theta) + bx' = F \quad (1)$$

for θ

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \theta'}\right) - \frac{\partial L}{\partial \theta} + \frac{2R}{\partial \theta'} = 0$$

$$\frac{d}{dt}(ml^2\theta' + mlx' \cos(\theta)) + ml\theta'x' \sin(\theta) - mgl \sin(\theta) = 0$$

$$ml(l\theta'' + x'' \cos(\theta) - x'\theta' \sin(\theta)) + ml\theta'x' \sin(\theta) - mgl \sin(\theta) = 0$$

$$l\theta'' + x'' \cos(\theta) - g \sin(\theta) = 0 \quad (2)$$

Non-linear model

solve coupled system (equations 1 & 2) for x'' & θ''

$$x'' = \frac{ml(\theta')^2 \sin(\theta) - gm \cos(\theta) \sin(\theta) - bx' + F}{(m+M) - m \cos^2(\theta)}$$

$$\theta'' = \frac{(M+m)g \sin(\theta) - ml(\theta')^2 \cos(\theta) \sin(\theta) - F \cos(\theta) + bx' \cos(\theta)}{l(M+m) - ml \cos^2(\theta)}$$

$$\frac{d}{dt} \begin{bmatrix} x \\ x' \\ \theta \\ \theta' \end{bmatrix} = \begin{bmatrix} x' \\ \frac{ml(\theta')^2 \sin(\theta) - gm \cos(\theta) \sin(\theta) - bx' + F}{(m+M) - m \cos^2(\theta)} \\ \theta' \\ \frac{(M+m)g \sin(\theta) - ml(\theta')^2 \cos(\theta) \sin(\theta) - F \cos(\theta) + bx' \cos(\theta)}{l(M+m) - ml \cos^2(\theta)} \end{bmatrix}$$

Once the equations of motion are derived, they can be implemented in Python using an ODE solver. The solver outputs the states of the system at any given time within the simulation interval. In addition to the inherent non-linearity in the equations of motion, we introduce further complexity by applying force saturation, limiting the applied force to a maximum of 40 N. This evaluates the controller's ability to handle real-world non-linearities. The next step involves designing a feedback controller to stabilize the system by calculating the required force to apply.

5.2 INVERTED PENDULUM CONTROL - 1 POINT

This system is nonlinear, unstable, and highly coupled. To design an optimal linear state-space controller, such as LQR, we first need to linearize the system. The linearization is performed about the equilibrium point $x^* = \theta^* = 0$ resulting in the linear model shown below.

Linear model for controller design:

$$\begin{aligned} \text{Linearized around: } & \theta^* = 0 \\ & x^* = 0 \end{aligned}$$

$$l\theta'' + x'' - g(\theta) = 0$$

$$(M+m)x'' + ml\theta'' + bx' = F$$

$$\frac{d}{dt} \begin{bmatrix} x \\ x' \\ \theta \\ \theta' \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{b}{M} & \frac{-mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{b}{Ml} & \frac{(M+m)g}{Ml} & 0 \end{bmatrix} \begin{bmatrix} x \\ x' \\ \theta \\ \theta' \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{M} \\ 0 \\ -\frac{1}{Ml} \end{bmatrix} F$$

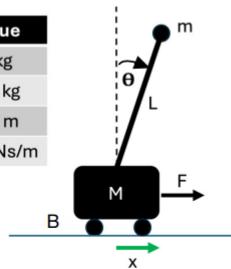
Using the A and B matrices from the linear model, along with Q and R matrices that define the state cost and input cost, we can compute the controller. A linear quadratic regulator (LQR) controller is designed by minimizing the cost integral, as shown below. This process yields the gain matrix k, which is then used in a feedback loop with the measured state to calculate the new applied force.

$$Q = \begin{bmatrix} q_x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & q_\theta & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad R = [r_{cost}]$$

LQR:

$$\min \left\{ \int_0^{\infty} (x^T Q x + u^T R u) dt \right\} \rightarrow k \quad \text{such that:} \quad u = F = -kx$$

Parameter	Value
M	1 kg
m	0.5 kg
L	0.4 m
B	0.01 Ns/m



$$q_x = 1500$$

$$q_\theta = 1200$$

$$r_{cost} = 0.1$$

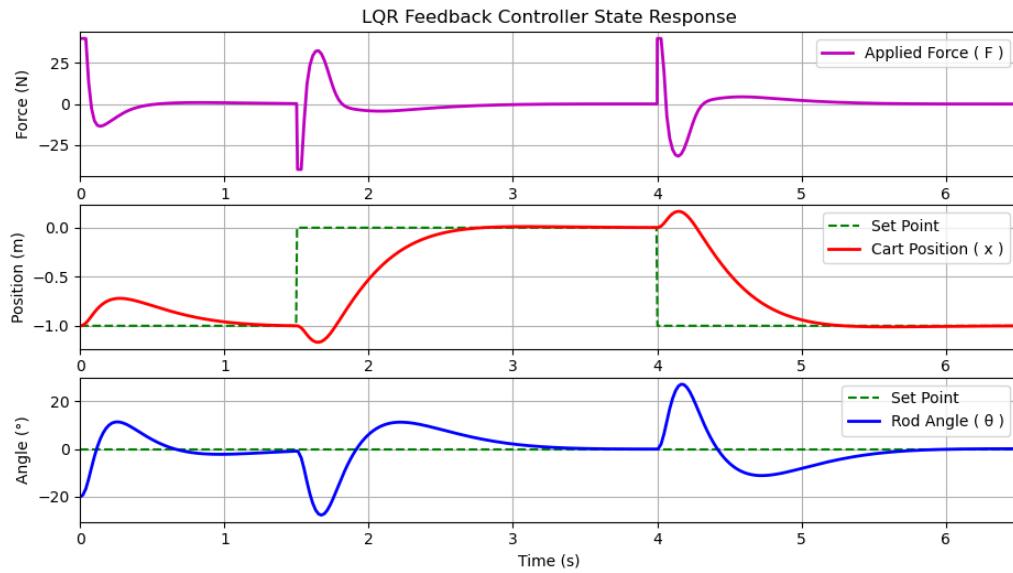
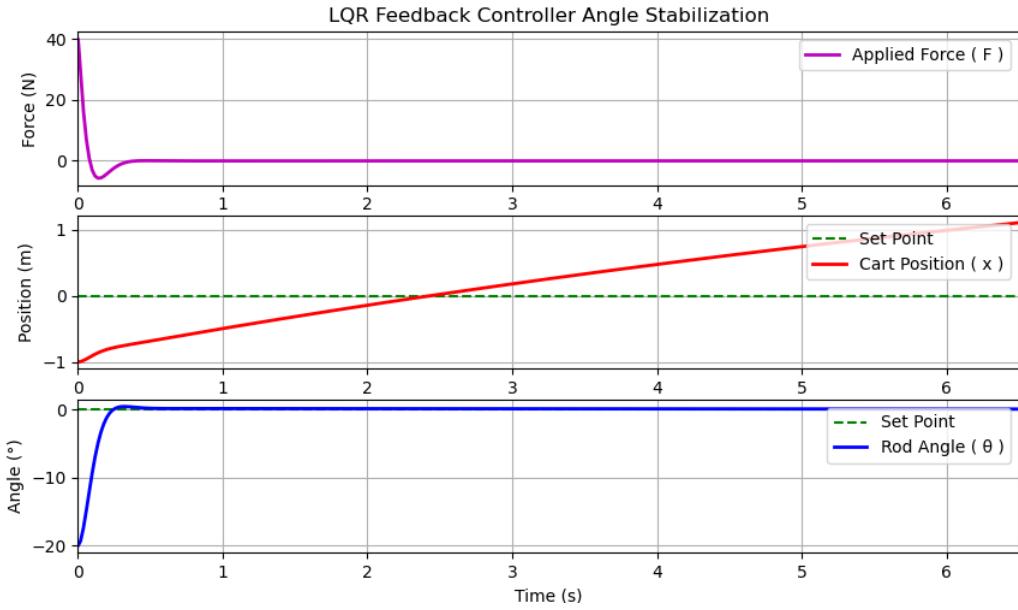
$$k = [-1225 \quad -75.5 \quad -242.9 \quad -42.6]$$

Algorithm 7 Feedback Control

- 1: **while** System is running **do**
 - 2: Get the state vector X at the current iteration
 - 3: Subtract the set point from the state vector
 - 4: Calculate the dot product of the control gain and the state vector
 - 5: Set the output force equal to the negative dot product
-

5.3 INVERTED PENDULUM STATE CONTROL - 1 POINT

When implementing the controller, we introduce deviations to the known parameters to account for measurement inaccuracies. The most significant deviation is applied to the damping parameter, as it is the most challenging to measure accurately in a real system. These deviations are incorporated into the linear model when calculating the gain matrix k for the LQR controller. Despite the system's non-linearities and model deviations, the feedback controller remains robust, effectively stabilizing disturbances and maintaining system stability.



The two plots demonstrate the system's capability to stabilize the pendulum's angle and simultaneously control both the angle and the cart's position.