

UNIVERSITY OF COLORADO AT BOULDER

COEN 5830 - ROBO 5000

FALL 2024

PROFESSOR: DR. LEO BEUKEN

TEACHING ASSISTANT: SRIKRISHNA RAGHU

Final Project

TEAM MEMBERS:

IAN MCCONACHIE

MICHAEL BERNABEI

December 8, 2024

CONTENTS

1	Problems Attempted	3
2	Perception	4
2.1	Lidar Occupancy Grid	4
2.1.1	Lidar Occupancy Algorithm High-level Steps	4
2.2	IMU Sensor Fusion	4
2.3	Canny Edge Detection - 1 + 1 Bonus Point	4
3	Path Planning	6
3.1	RRT - 3 points	6
3.2	Deploying the solution - 1 bonus points	7
3.2.1	Deploying the Solution Noticed Issues	7
4	Dynamics and Control	8
4.1	Part A - 2 Points	8

LIST OF ALGORITHMS

1	Lidar Occupancy Grid Algorithm	4
2	Simple Canny Edge Implementation	5
3	Webots RRT Path Planning	7
4	RRT Path Planning Replay Velcities	7

1 PROBLEMS ATTEMPTED

- Perception - Lidar Occupancy Grid
- Perception - IMU Sensor Fusion
- Perception - Canny Edge Detection
- Path Planning - RRT - 3 points
- Path Planning - RRT Deploying the solution - 1 bonus points
- Dynamics And Control - 5.1 Part A

2 PERCEPTION

2.1 LIDAR OCCUPANCY GRID

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 and save coding details for the oral exam portion of this project.

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 previously stated, code details will be highlighted upon request at the oral examination.

2.2 IMU SENSOR FUSION

IMU Sensor Fusion - 1 points

2.3 CANNY EDGE DETECTION - 1 + 1 BONUS POINT

Canny Edge Detection - 1 Point + 1 Bonus Point

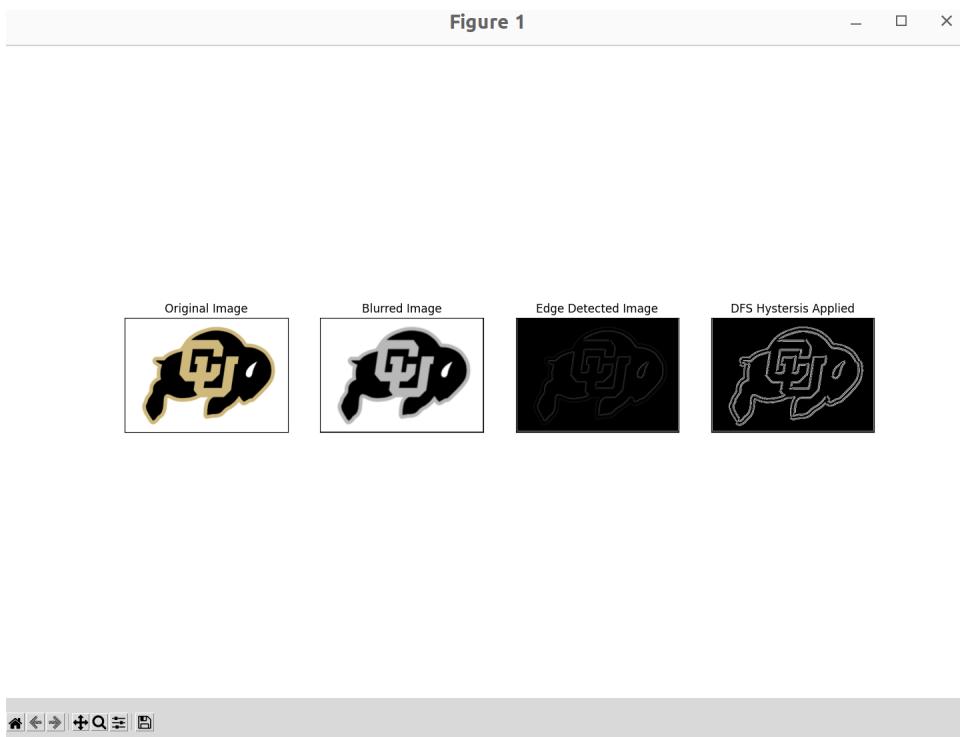
For Canny edge detection, we implemented each of the steps necessary for the algorithm and then

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 yeild better results.

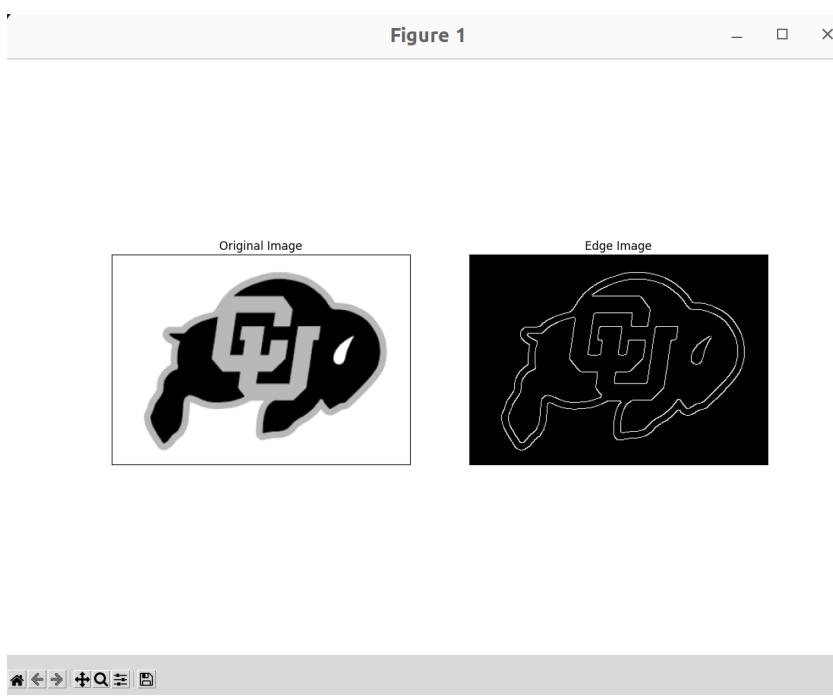
Algorithm 2 Simple Canny Edge Implementation

- 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 PATH PLANNING

3.1 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 psuedo code was a communication protocol between the Supervisor and the E-Puck.

Algorithm 3 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
```

3.2 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 4 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
```

3.2.1 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.

4 DYNAMICS AND CONTROL

4.1 PART A - 2 POINTS