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 11, 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 . . . 5

3 Object Oriented Programming **7**
3.1 OO Programming - Warehouse Robot Management - 3 points . . . 7

4 Path Planning **8**
4.1 RRT - 3 points . . . 8
4.2 Deploying the solution - 1 bonus points . . . 9
4.2.1 Deploying the Solution Noticed Issues . . . 9

5 Dynamics and Control **10**
5.1 Part A - 2 Points . . . 10

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 . . . 8
5 Webots RRT Path Planning . . . 8
6 RRT Path Planning Replay Velocities . . . 9

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
- Path Planning - RRT - 3 Points
- Path Planning - RRT Deploying the solution - 1 bonus point
- Dynamics And Control - 5.1 Part A - 2 Points

Total points attempted: 13

2 Perception

2.1 Lidar Occupancy Grid - 2 Points

For the lidar occupancy grid problem we elected to use the **Brensenham 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 Brensenham's line algorithm is the heart of our work, we will describe the steps in our implementation that lead up to and after the Brensenham 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 Brensenham'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 gryo 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.

$$\theta_{roll} = \alpha\theta_{integrated\ roll\ gryo} + (1-\alpha)\theta_{accelorometer}$$
$$\theta_{roll} = \alpha\theta_{integrated\ pitch\ gryo} + (1-\alpha)\theta_{accelorometer}$$

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,

$$\theta_{roll} = \theta_{integrated\ roll\ gryo} + (1-1)\theta_{accelorometer} = \theta_{integrated\ roll\ gryo}$$
$$\theta_{roll} = \theta_{integrated\ pitch\ gryo} + (1-1)\theta_{accelorometer} = \theta_{integrated\ pitch\ gryo}$$

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

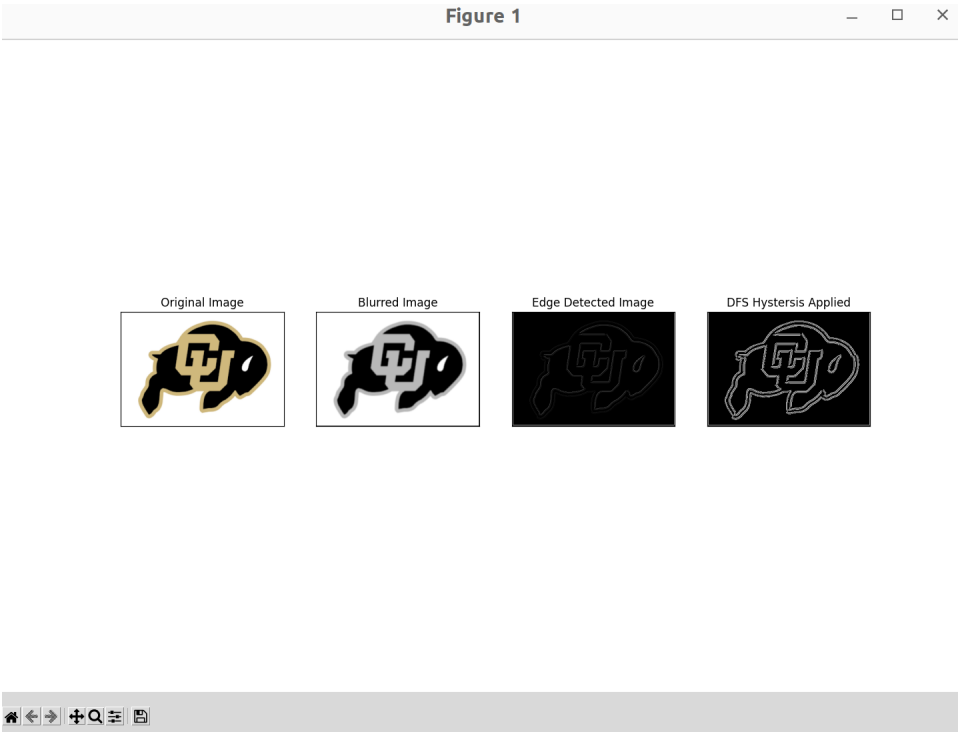
2.3 Canny Edge Detection - 1 + 1 Bonus Point

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 hystersis 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.

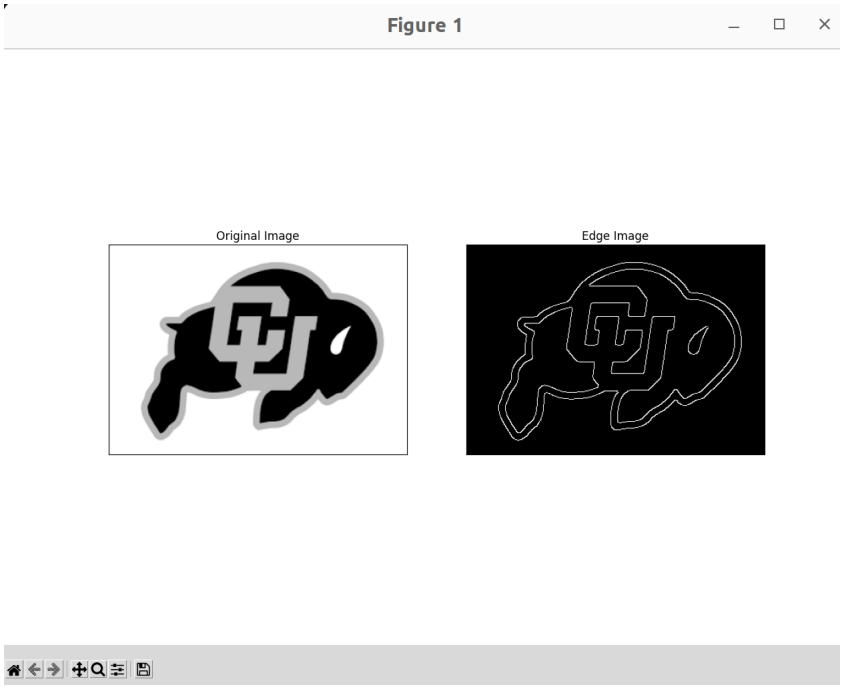
Algorithm 3 Simple Canny Edge Implementation

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 hystersis 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 hystersis step in the Canny edge detection algorithm.**

3 Object Oriented Programming

3.1 OO Programming - Warehouse Robot Management - 3 points

For this part of the project we implemented a robot parent class and implemented the child classes as instructed. We then created two custom classes for our project. The first of these classes, is the Warehouse class. It's 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 are aware 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

```
Randomly create child robots
Generate random start points for child robots
Generate random goals for child robots
while not all robot goals reached do
    Warehouse request all robots advance towards their goals
    Each robot uses Dijkstra's search algorithm to find shortest path
    Generated paths are inspected by the Warehouse
    if no collisions occur then
        Warehouse gives the go ahead for robots to move 1 spot
    else
        Resolve collisions using methods described in project file
    end if
    if robot reached goal then
        Add to goal reached
    end if
end while
```

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

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

4.2 Deploying the solution - 1 bonus points

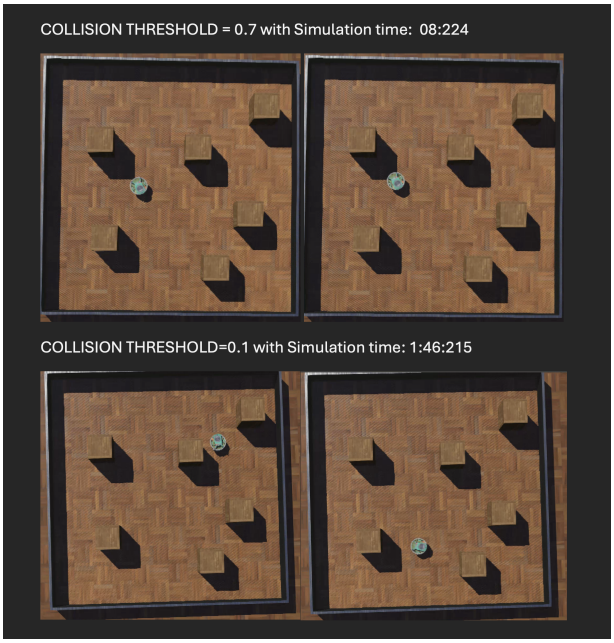
For this part of the project we simply added on to the work we did in the previous section by saving the actions. Actions are data structures that hold the velocities and the duration for which those velocities were executed. After the goal is reached, we simply send these actions to the RRT Puck process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

Algorithm 6 RRT Path Planning Replay Velocities

```
if goal reached then
    Have Supervisor send saved actions to E-puck
end if
if actions received by E-puck then
    Replay actions on E-puck
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 it's 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 and Control

5.1 Part A - 2 Points