UNIVERSITY OF CALIFORNIA SAN DIEGO

Detecting Edges of Roaming Surface with an RGB-Camera for Micro-Processor Robots

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Engineering Sciences (Mechanical Engineering)

by

Narek Geghamyan

2019

EPIGRAPH

”Ask and it will be given to you;

seek and you will find;

knock and the door will be opened to you.”

Matthew 7:7

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to first thank my parents, Arthur and Margarit, for working hard every day of their lives and taking the chance to immigrate to the United States with nothing in their pockets.

Thank you to my friends, brothers, and sisters who have helped and supported me these past few years: Armen, Irma, David, Arin, Ani, Poghos, Nima, Narek, Yonatan, Ali, and, Igor.

Lastly, thank you to professor Thomas Bewley for allowing me to work on the project which would eventually become my thesis. Professor Bewley's doctoral student, Daniel Yang, has also been a great help and guide through the world of robotics.

ABSTRACT OF THE THESIS

Detecting Edges of Roaming Surface with an RGB-Camera for Micro-Processor Robots

by

Narek Geghamyan

Master of Science in Engineering Sciences (Mechanical Engineering)

University of California San Diego, 2019

Professor Thomas Bewley, Chair

Robots have the ability to drive with a wide array of freedom, including the freedom to drive past the surface edge, ending in a possible fatal accident. Ultrasonic sensors may be used on robotics with a constant and steady angle of attack relative to the ground, yet they are not able to detect where the edge of the roaming surface is. Furthermore, robots designed as mobile inverted pendulums require a method to observe the roaming surface using sensors not directed perpendicularly, or close to it. Cameras, a common addition to robotic hardware, are optical sensors which capture light from all angles. Using image processing techniques, we are able to detect where the edges of the roaming surface exist and develop a strategy to avoid falling off the edge of the surface.

Chapter 1

Introduction

1.1 Roaming on a Surface and Accident Prevention

Robots today are able to analyze their surrounding [Ren93], perform intricate maneuvers [RI99], climb stairs [Yan18], and more. Yet, the safety of the robots themselves is often overlooked. Robots are damaged if colliding into objects or falling of the edge of the table.

A robot may avoid colliding into an object using ultrasonic range detector sensors, it may also use a depth camera. However, what if the robot is seeking to avoid is falling off of the roaming surface in advance of reaching the edge? Ultrasonic range sensors must face obstacles perpendicularly to detect their distance, as do depth cameras. An ultrasonic sensor may work if it's placed directly underneath the front of the robot, but won't be able to detect an edge until the robot is dangling above it.

If a robot wishes to observe the environment and plan a trajectory in advance of its movements, then a solution which is able to find potential sources of accidents is needed. This is one reason why cameras useful. They are able to view the environment in front of the robot in advance and are not limited by the obstacles or edge orientation.

1.2 Hardware

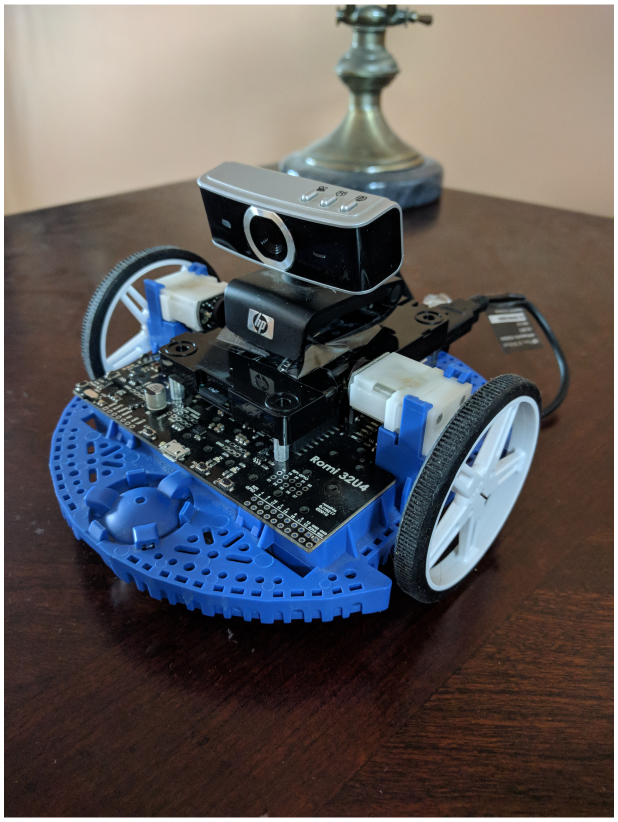

Figure 1.1: Fully assembled robot with Romi control board underneath a hidden Raspberry PI, which is covered by a black case. On top of the case is the RGB web camera.

1.2.1 Robot Build

The robot was built modeling the Pololu Romi-Pi robot. The robot initially consisted of a chassis kit with two plastic gear motors rated at 120:1 HP, two encoders attached to the motor shaft, and a Romi control board. To control the robot, a Raspberry Pi is attached directly to the control board via GPIO (general purpose input and output) pins. In addition to the Pololu Romi-Pi robot, a USB RGB web-camera was connected to and attached above the Raspberry Pi.

It should be noted, the encoders incorporated in the Pololu Romi chassis motor kit did not provide consistent readings and were thus ignored. The challenge became more to utilize the camera to decipher the robots position with respect to the roaming surface edge.

1.2.2 Raspberry Pi

The robots computation power was chosen to run on a Raspberry Pi. The Raspberry Pi is considered a System on a Chip (SoC) as it contains nearly all of the components of a full computer without the hard drive. The Raspberry Pi could also have been replaced with a micro-processor but was chosen for its ease of use. The Raspberry Pi is connects to the Pololu Romi 32U4 Control Board. The control board then controls two motors and numerous visual and audible sensors, and is able to observe the behavior two encoders.

Raspberry Pi model 3 was chosen for the project. It has a quad core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB of RAM, and other useful features, such as GPIO ports, HDMI output, and multiple USB 2.0 ports. While these specifications are greater than the majority of micro-processors, and match that, or exceed, most mobile devices, they still leave the Raspberry Pi with computational limitations as there are computers with more power.

The hardware limitations of the Raspberry Pi are only a concern if the computation time leads to lagged results. If the hardware does not suffice quick processing, then the software must be altered.

1.2.3 Depth and RGB Cameras

The type of camera used makes a tremendous difference. Light strongly effects a cameras ability to capture accurate date along with the cameras shutter speed. Initially the project utilized a depth camera to determine where the edges of the roaming surface are. The depth camera uses Time of Flight (TOF) to determine how far objects are from the camera. TOF cameras have an additional IR camera and sensors: the camera emits IR light and the sensors receives the infrared light waves bouncing back from perpendicularly faced surfaces with respect to the camera angle.

The sensor allows the depth camera to compute how far objects in view are based on the time it takes for the light waves to reach the receiving sensor. After numerous trial and errors, the depth camera was concluded to be promising but not technologically ready to be used well for such a project as the software incorporating the data needs to be advanced by the developers and the size and weight of the cameras are yet too large.

The next logical progression was to use RGB (Red, Green, Blue) cameras. Nearly all commercial cameras found in web-cameras and mobile devices are RGB cameras. An added benefit as those wishing to incorporate the solution developed will have an easier time finding the right hardware to use. Another benefit was the wide availability of software packages able to read data from standard RGB cameras. A standard HP (Hewlett-Packard) web camera, attached via USB 2.0, was used.

Preferred camera used is the Pi camera module (which connects directly to the CSI port on the Raspberry Pi) as it may process the image quicker than a camera attached via USB. However, for this project, we used a standard USB RGB camera as it is the quickest and most convenient method of attaching a camera to a robot. It also allows us in the near future to test the capabilities of the robot with other electronic components.

1.3 Software

1.3.1 ROS versus Python Script

Many robotic applications use ROS (Robot Operating System) to control and observe multiple actions. ROS is a collection of software libraries that help create robots applications. Unfortunately, there is an issue with using ROS on micro-processors. While ROS was able to install on the Raspberry Pi (after many attempts), it did not have full functionality given the limited computation power of the Raspberry Pi. Certain features of ROS for the Raspberry Pi are also still in development, such as utilizing a depth camera. For these reasons, a python scrip was chosen to operate the robot. Simple was better and faster. In the future, ROS will be necessary once the robots operations become more involved and complicated.

1.3.2 Image Processing

An important aspect of the project is to decipher from a captured image where the edge of the surface is. The goal is to develop a quantifiable method of expressing how likely a region is the edge of the roaming surface. Two assumptions are made: First, the surface is of uniform color and texture. Second, the surface is differentiable from the surrounding environment. Now, computationally observing multiple regions and quantifying their similarities becomes easier.

Structural Similarity (SSIM) index is a method of predicting the perceived quality of digital images and videos. SSIM is used for measuring the similarity between two images. Original uncompressed, distortion-free images are used. It's quite a basic calculation, which makes it attractive for a light algorithm. Computing SSIM returns a floating value from 0 to 1, and indication how likely two image windows match.

SSIM is computed using two images, or image segments, of same dimension. The SSIM calculation of the two image windows, x and y, is shown in 1.1 where μ is the average value and σ is the variance of the given pixel window. Furthermore, c is a function of the dynamic range of pixels of the respective image window.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{1.1}$$

SSIM is traditionally best used with the Luma, which represents the brightness of an image. It is also similar to converting an image to black and white, which is what was eventually used for the edge detection algorithm. In addition, the original three layers of an image (whether color such as RGB or alternatively HSV) may be used to locate edges of a roaming surface. Increasing the information available may allow for the algorithm to distinguish different textures or assist with edge detection if the outside environment does not appear vastly different form the roaming surface.

1.4 An Open Versus Closed Loop System

Robots are capable of being programmed to stop immediately but the world is not perfect and wheels may slip, momentum may prevent an immediate stop, and if the robot is moving too quickly, it may not capture an accurate image and miscalculate when to stop. A feedback loop ensures the robots understanding of the environment remains up to date.

An open loop system may be used to demonstrate the behavior of the robot if it is capable of gathering sufficient data to perform its duty of driving forward and stopping short of the edge. Yet, an open loop system has its limitations. The robot is capable of capturing enough information about its environment from an initial data to determine a complete set of actions, but this does not always occur. The data may be inconclusive and thus, not entirely helpful. It will then only be helpful by indicating the need to gather more information. This is the intention of the closed loop system.

Chapter 2

Algorithm

2.1 Algorithm

Developing the algorithm required knowing how long the micro-processor (Raspberry Pi in our case) needs to capture an image and compute the robots distance to the roaming surface edge. The robot may move forwards quickly, therefore, if the user wishes to move the robot quickly, they must be able to detect where the edge of the roaming surface is well in advance of reaching the edge.

The Raspberry Pi is capable to transmitting a desired motor input to the control board. The Romi control board then prompts the motors to function accordingly. While it drives the robot forward, the Raspberry Pi captures additional images and computing the distance to the edge. The ability to perform multiple actions at once, by combining the control board and Raspberry Pi, allows for less time lost due to computation.

The algorithm comprises of two section: Properly segmenting the image captured by the camera and determining where the edge of the surface is. A key component of determine the location of the surface edge will be by searching for the edge efficiently. This is done by observing a selective path on the image, as apposed to the entire image. Algorithms by Boykov [BFL06] display a proper segmentation of an image if the user indicates a fraction of each distinct regions. The approach would later inspire the tree search algorithm developed as the robot indicates the distinct regions autonomously. The tree search approach allows the algorithm to observe a trajectory the robot may take on the captured image. The trajectory may be broad, but a narrow path is much quicker and often sufficient. Inspiration to observe only a portion of the image was confirmed after reviewing a paper by W. Lee, J. Wang, et al [LWLS93] where they selectively observe portions of an image or video window to optimize the search and computation time. The proposed algorithm does not need to observe the entire image. By using a determined path to search for an edge and observing only a portion of the image, the new algorithm was able to shave off nearly 2 seconds of computational time.

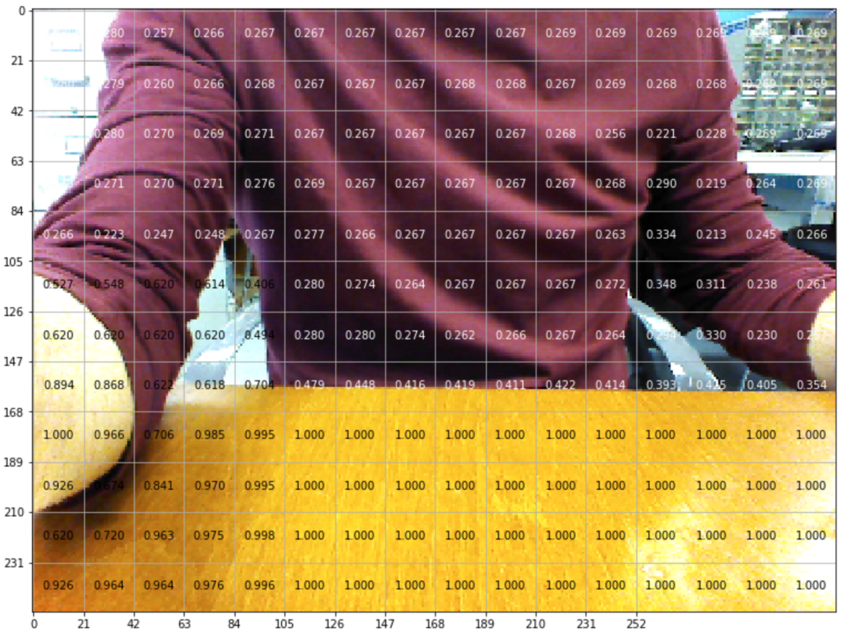

Figure 2.1: How the image processing was envisioned to work.

2.2 Image Processing

Segmenting the captured image of the visible environment is important as it distinguishes between the roaming surface and outside environment. Initially, an ideal method was envisioned, one that would allow the robot to distinguish between the surface it is driving on and everything else. Figure 2.1 displays the goal quite well. Once various methods were explored and tried, Two distinct solutions were developed. Both share strong similarities, but only one is computationally efficient while the other is more robust.

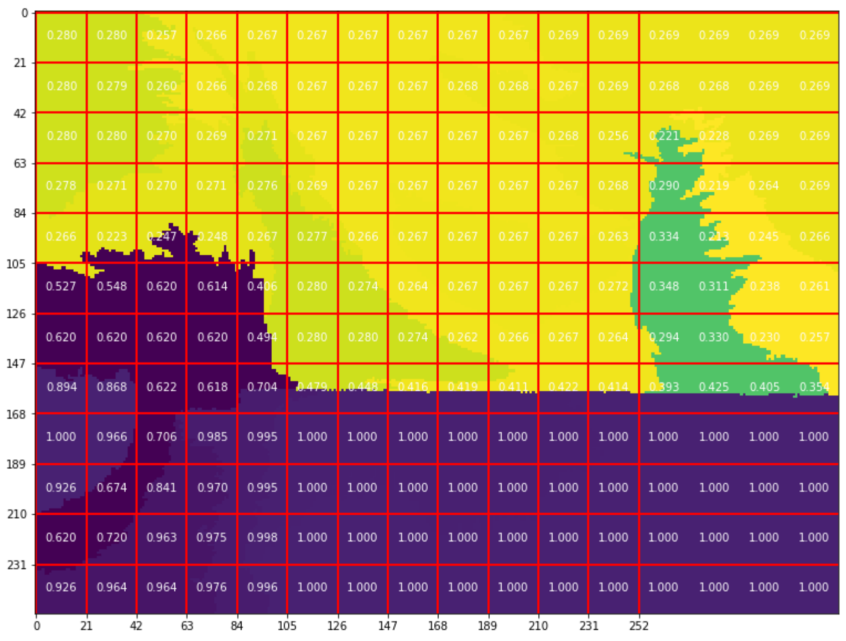

Figure 2.2: The Segmented image using k-means.

Initially a k-means was used to segment the image is three to four Voronoi cells. K-Means is a clustering algorithm which computes the mean of data available, in this case image color values from the RGB channels, and groups the data into the specified number of clusters based on smallest distance error with respect to surrounding data. In other words, k-means groups the image pixels into the groups if it deems the pixels are similar enough (considering the number of desired clusters). The k-means segmentation gave a post-processed image to work with, which was smooth and easily allowed for differentiation of table surface and outside world (beyond the surface edge), as shown by figure 2.2, which displays K-Means segmentation appearance for the robot. However, the Raspberry Pi, although in its third build model, still took on average of 6 seconds to compute where the exact edge of the table surface was. The edge detection sequence may be observed in two figures. First, figure C.1, choosing the initial foot of the robot as the known environment, and in the second figure C.2, comparing other grids blocks, starting from the bottom, to the initial block. While accurate, the k-means methodology required 5 seconds to compute the distance location. Unfortunately, an average of 5 seconds is too long and can lead to inaccurate environment analysis.

Another approach was taken; a more ”quick-and-dirty” method. One that, in hope, would significantly reduce the computation time. First, the image was not smoothed or processed in any way. So long as the table surface was a distinctly different color than the rest of the world and the tables surface color was uniform, the gray version of the image would suffice (single channel image). Second, the area of each grid region was increased by 225 percent. The size of the grid squares are noticeably larger, as shown in Figure 2.3. Covering more area per grid square means shorter for loops. An added benefit to using larger grid squares is the ability to neglect small deviations in the image which will miscalculate the surface properly as permissible or over the edge. The gray image is in part already a segmentation, that is why it saves computation time. The new method works consistently as long as the roaming surface is a distinctly different shade of color intensity than the surrounding environment. The sequence may be observed once again by figures C.3 and C.4. Additionally, the corner edge detection final image of the sequence is shown in C.5.

Having a smarter sweep of the grid squares continued to reduced computation time by a total of 95 percent. The ”smarter” sweep is more strategic as it does not sweep all of the squares, only those in front of the robot. Attempts to further increase grid square sizes showed showed an optimal shape of 60 by 60 pixels. The time to compute the cell scans matched larger squares but with a greater resolution reading. The computation time for the new method ranged from 0.2

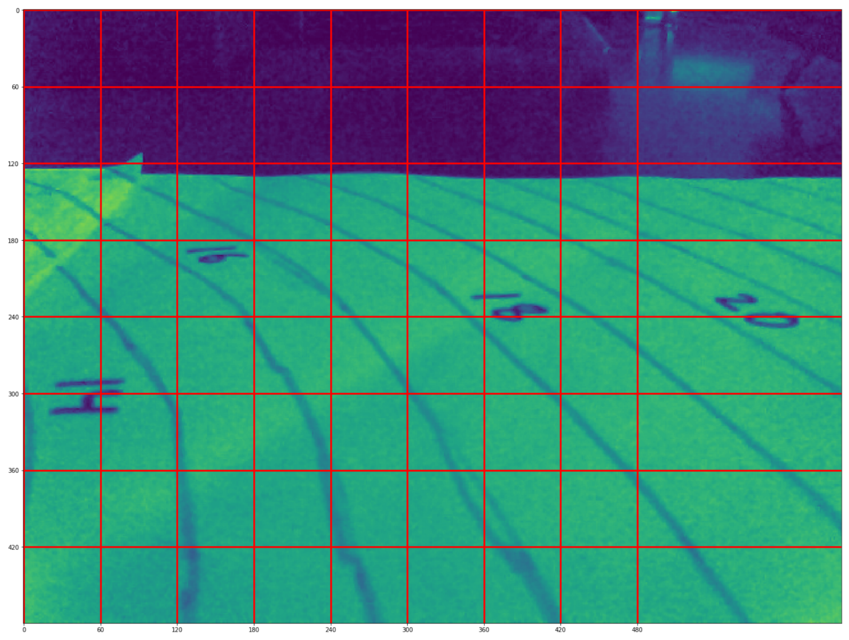

Figure 2.3: Gray scale image displaying segmentation, shown in color for visual contrast. The image also portrays the larger grid lines, allowing for much faster processing.

seconds to nearly 1 second, depending on how far the edge was from the foot of the robot. On average, a half second computation time is much more desirable and quick enough to prevent the robot from having to halt in order to compute the distance to the edge.

2.2.1 No Edge In Sight

If an edge is not found, the algorithm allows the robot to traverse forward in the predetermined direction for a brief moment of time. A quarter of a second is usually sufficient but it may be altered to suit the users needs. Afterwards, the robot captures another image and once again proceeds to search for the surface edge. This process is part of the control loop continuously determining how much farther the robot may travel, which is discussed in chapter 3.

2.2.2 Necessary Conditions

The hardware is not perfect, nor is the software. Interestingly the SSIM values computed on the Raspberry Pi occasionally differ from the values computed on a full scale computer. Consistency is crucial. Lighting must be consistent as the shading and shadows cast on the roaming surface will confuse the algorithm and result in miscalculation. Additionally, the software must be able to distinguish between the roaming surface and outside environment. Therefore, the color and consistency of the color of the roaming surface in reference to the environment is crucial. The gray scale image is able to distinguish differences, but the K-means segmentation and use of the hue image (of Hue-Saturation-Intensity), allows for a much better distinction between different surfaces.

2.3 Calibration

Once the image of the visible environment is captured and processed for an edge within one of the grid cells, the real world distance to that marked grid cell and the robot must be determined. A ”distance” matrix, shown in appendix B.1, was developed to determine exactly how far the discovered edge is from the robot.

The distance matrix is created by first placing a large sheet of paper with marked circles, shown in Figure B.1, each an inch farther in radius from the center, in front of the robot. The foot of the sheet, along with the starting point, is directly beneath the front of the robot. Several images are taken with the robot stationary and the camera is adjusted to what is determined to be the most optimal angle with respect to the surface.

The distance matrix calibrates roughly how far the grid cells are from the robot. The calibration needs to be repeated for different angle of the camera and if the grid cells dimensions change. An additional calibration of the motor control inputs to the output speeds is needed. The robot is driven forward with specified inputs and the distance traveled after one second is recorded. After the rate of travel is recorded for the entire spectrum of motor inputs, an equation is formed using polynomial curve fitting.

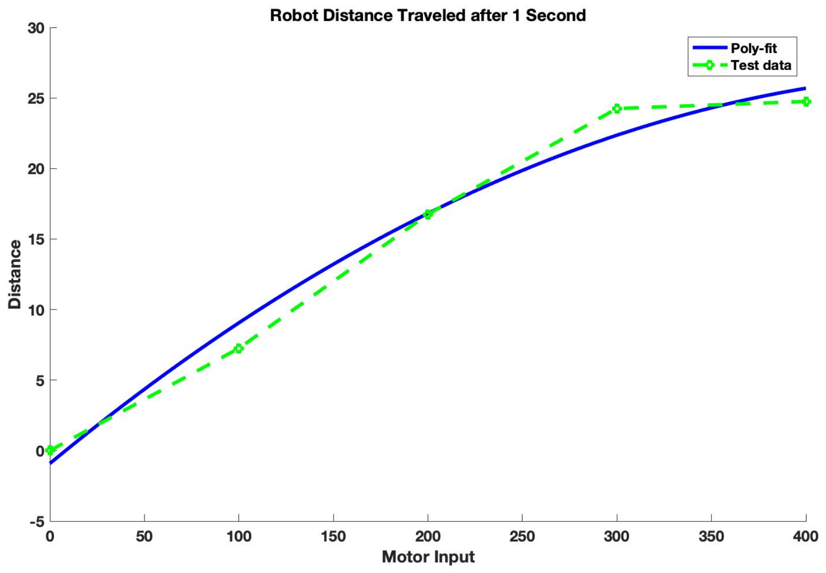

Figure 2.4: Calibration plot. Speed (inches/second) of each motor input.

To calibrate the robots motion in an arc, the robots trajectory at various motor input ratios was observed. The robot traveled on the same large sheet of paper with marked distance radius used previously. As the robot moved in the arc trajectory, its location was marked on the sheet. After multiple ratios were marked, the robot took an image of the same sheet from the base, figure B.2. The image was then used to develop the arcMatrix, as shown in section B.1.

The arcMatrix provides the robot with the ability to know what ratio to use if an edge is detected to the side, and we wish for the robot to travel to the location in an arc trajectory. It was assumed the wheel ratio, where one wheel was held at a fixed speed, and the other wheel was the same speed but increased by 10, 15, 25, etc, percent, is symmetric between both wheels.

2.4 Pseudo Code

Following is a pseudo code of the algorithm's procedure as it captures an image and computes the location of the edge. The entire procedure is part of a continuous loop, which repeats until the robot reaches the edge of the roaming surface. It should be noted the computed time to reach the edge is important as the robot, in this instance, moves forward at a constant speed. Where the speed not constant, the time allocated to driving forward would still be significant as distance covered is a function of speed and time.

```
1: Capture image from RGB web camera
2: Convert image from RGB to Gray
3: Using SSIM, compare individual blocks of the image in a trajectory the robot will follow
4: if Edge is found then
5:     Determine time it takes for robot to reach edge if driven at specified speed
6:     Subtract time it took to compute distance from time to drive forward
7:     Drive forward at specified speed for previously computed time
8: else
       Drive forward 10 inches
9: end if
```

Chapter 3

Using An Open And Closed Loop System

3.1 Open Loop System

The open loop system is used to demonstrate the behavior of the robot during the end state of its trajectory. Demonstrating a successful halt while moving forward at both constant and varying speeds, and moving in an arc, indicate the ability to perform the maneuver when and if the robot is capable of detecting the edge within a reasonable distance from the robot.

3.1.1 Moving Forward

If the robot is informed by the image captured where the edge of the roaming surface is, it may adjust its behavior as it approaches the edge. The exact behavior is determined by the user. For our purposes, we indicate the robot must decrease its speed every nine-tenths of the total time it will take to reach a nine-tenths of the remaining distance. The polynomial fit shown in figure 2.4 is used to compute how long it will take to travel the desired distance.

3.1.2 Moving Towards the Corners

The robot may also be instructed to move towards an edge located off center. If an edge off center is detected, the robot is programmed to move in an arc like trajectory. Moving in an arc is similar to moving forward, with an additional step. The detected edge location is cross referenced with the arcMatrix to determine which wheel ratio to use. The wheel ratio along with the distance to the edge from the distance matrix both allow the robot to approach the edge off center in the same manner as an edge directly in front of the robot.

3.2 Closed Loop System

All of the functionality of the open loop system may be developed for a closed loop system. The closed loop system would capture a new image and update its knowledge of the environment. The closed loop system is however limited by the capture rate of the camera. If the robot is moving too quickly, the images captured may be blurry and thus uninformative. To prevent a blurry image, the robot is programmed to stay stationary for a fraction of a second after traveling a predetermined distance, usually within 12 inches. This particular distance is chosen as the robot's field of view is best when it is within 14 inches directly. The end result is a well functioning robot.

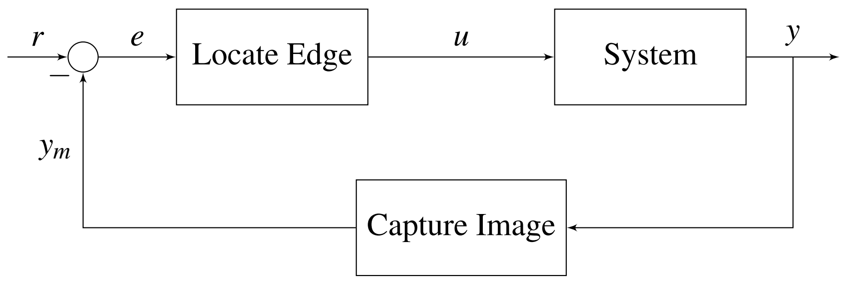

Figure 3.1: Closed Loop System

3.2.1 Moving Towards the Corners

Moving in an arc towards and edge requires traversing over a longer distance than if the robot were to move in a direct path. For this reason, arc like trajectories benefit more from a closed loop system as the robot is able to correct its position. Needing to move slightly more forward by a few inches towards the edge is an example of the benefits of a closed loop feedback system. By applying a feedback system, our robot becomes more robust.

Chapter 4

Conclusion

Throughout the project, we are able to demonstrate the how a robot may further improve its understandings of the surroundings with the use of a single RGB camera. In chapter 1, the decision to use an RGB camera and not incorporate ROS were addressed.

In chapter 2, we introduce the inspiration for the trajectory oriented, tree search algorithm. The algorithm also address the procedure of capturing an image and detecting the location of the roaming surface edge. Furthermore, the two methods of image processing and segmentation, k-means and gray scale, were discussed. Both methods possess an advantage, yet have respective faults. For our purposes, the gray scale was chosen but the k-means method may pose more beneficial in future uses. Chapter 2 also discusses how the robots perception of the world is calibrated, producing the distance and arc matrices.

Chapter 3 introduced the need for both open and closed loop systems. Open loop systems were ideal for observing how the robot behaves once the edge is detected on or off center. Closed loop systems in addition allowed for the robot to utilize the develop open loop system algorithm in an environment where the end behavior was not immediately required.

If edge detection algorithm were incorporated in a more complex robot, with multiple responsibilities, incorporating the algorithm within the ROS architecture is recommended. ROS allows multiple applications to process well simultaneously. Furthermore, future implementations of the project should invest in a well capable RGB camera. The faster a cameras capture rate and better quality, along with ability to capture dark environments well, strongly dictate how accuracy of the segmentation algorithm.

If the computational capabilities of micro-processors and system-on-a-chip platforms rise, future iteration of the robot may be able to determine the likelihood of a surface edge and alter the dynamics of motion, such as the speed of movement, based on the probability of an edge or obstacle along the trajectory. For the time being, the foundation developed is sufficient for future improvements.

Ultimately, the edge detection project has shown promise for robots wishing to understand the environment strictly using a camera.

Appendix A

Script Code

A.1 Moving Forward: Open Loop

```
% code used in python script for moving forward

#!/usr/bin/env python3


# drive robot forward to edge

# by Narek A. Geghamyan of UCSD Jacobs, FCCR Lab


import time

import os

# os.system('clear')

# print('System starting up...')


import cv2

from cv2 import VideoCapture, imwrite

from skimage.measure import compare_ssim as ssim

import numpy as np
```

```
from a_star import AStar
a_star = AStar()

def motors(left, right):
    a_star.motors(-int(left), -int(right))

def timeOfMovement(distanceEdge,motorInput):
        d = (0.95)*distanceEdge
        distanceMotors = motorInput*(-0.0001)**2 + motorInput*(0.1108) -
            0.9143#after 1sec
        timeToDrive = d/distanceMotors
        #newMotorInput
        return timeToDrive

def turnRight():
    # turn robot rightwards
    motorInput = 200
    motors(-motorInput, motorInput)
    time.sleep(0.2)
    motors(0,0)

def turnLeft():
    # turn robot leftwards
    motorInput = 200
    motors(motorInput, -motorInput)
    time.sleep(0.2)
    motors(0,0)


def findEdge(img, height, width, cellSize,numRow, numCol):

    occupancyGrid = np.zeros([numRow,numCol])

    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    if(cellSize%10==0):

        originalCell = img_gray[height-cellSize:height, \

                width//2-cellSize//2:width//2+cellSize//2]

    else:

        originalCell = img_gray[height-cellSize:height, \

                width/2-cellSize/2:width/2+cellSize/2+1]


    # drive forward - tree is formed from bottom of map upwards toward edge

    robotWidth = 2   # number of boxes from mapping

    y0 = numRow -1       # starting y point

    x0 = numCol//2 - robotWidth//2 # starting x point

    row = y0

    loop = True

    while loop :

        for col in range(x0, x0+robotWidth):

            roamCell = img_gray[row*cellSize-cellSize:row*cellSize , \

                                        col*cellSize:col*cellSize+cellSize]

            if(row==0):

                fwdLimitCoord = (0,5)

                loop = False

                break
            occupancyGrid[row,col]
                =ssim(originalCell,roamCell,multichannel=True,win_size=5)
            if(occupancyGrid[row,col] < 0.6):
                fwdLimitCoord = (row,col) # x,y coordinate location
                loop = False
                #break
        row = row -1
    return fwdLimitCoord

# initialize distance matrix - my version of calibration matrix (incehs)
distanceMatrix = [[22, 22, 22, 20, 25, 25, 20, 22, 22, 22],
                  [14, 14, 13, 13, 13, 13, 13, 13,14, 14],
                  [9.5, 9, 9, 9, 9, 9, 9, 9, 9, 9.5],
                  [7, 7, 6.8, 6.4, 6, 6, 6.4, 6.4, 6.8, 7, 7],
                  [5.5, 5.2, 5, 4.8, 4.6, 4.6, 4.8, 5, 5.2, 5.5],
                  [4.2, 4, 3.8, 3.5, 3.5, 3.5, 3.5, 3.8, 4, 4.2],
                  [3.5, 3, 2.8, 2.7, 2.7, 2.7, 2.7, 2.8, 3, 3.5],
                  [2.5, 2.3, 2, 2, 1.8, 1.8, 2, 2, 2.3, 2.5]]

cam = VideoCapture(0)                # initialize the camera
cellSize = 60
#height, width = np.shape(img)[:2]
height = 480             # hard coding in order to reduce comp. time
width = 640              # hard coding in order to reduce comp. time
numRow = height//cellSize
numCol = width//cellSize
```

```
closeToEdge = False

motorInput = 100



while(closeToEdge == False):           # while away from edge

    motors(0,0)

    print('motors stopping & capturing new image...')

    time.sleep(0.1)                    # may be omited

    s, img = cam.read()                # capture image

    startTime = time.time()

    motors(motorInput,motorInput)

    edgeCoord = findEdge(img, height, width, cellSize,numRow, numCol)

    dist = distanceMatrix[edgeCoord[0]][edgeCoord[1]]

    dist = dist+6 # calibration offset

    timeElapsed = time.time() - startTime

    print('distance to edge= ', dist, '. Time elapsed = ', timeElapsed)

    distTraveled = timeElapsed*(motorInput*(-0.0001)**2 + motorInput*(0.1108) -
        0.9143)

    # compute distance traveled, compare that to distance from edge,

    # then determine if the robot needs to move anymore forward

    if(distTraveled < dist):

        distRemaining = dist - distTraveled

        timeRemaining = timeOfMovement(distRemaining, motorInput)

        print('Sleeping for ',timeRemaining, ' seconds...')

        time.sleep(timeRemaining)

    if(dist<19):

        closeToEdge = True
        break

motors(0,0)
```

A.2 Moving Towards the Corner: Open Loop

```
# driving in an arc

import time
import os
# os.system('clear')
# print('System starting up...')

import cv2
from cv2 import VideoCapture, imwrite
from skimage.measure import compare_ssim as ssim
import numpy as np

from a_star import AStar
a_star = AStar()

def motors(left, right):
    a_star.motors(-int(right), -int(left)) # flipped right/left

def timeOfMovement(distanceEdge,motorInput):
    d = (0.95)*distanceEdge
    distanceMotors = motorInput*(-0.0001)**2 + motorInput*(0.1108) - 0.9143#after
        1sec
    timeToDrive = d/distanceMotors
    #newMotorInput
    return timeToDrive


def findEdge(img, height, width, cellSize,numRow, numCol):
    occupancyGrid = np.zeros([numRow,numCol])
    img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    if(cellSize%10==0):
        originalCell = img_gray[height-cellSize:height, \
                width//2-cellSize//2:width//2+cellSize//2]
    else:
        originalCell = img_gray[height-cellSize:height, \
                width/2-cellSize/2:width/2+cellSize/2+1]

    # drive forward - tree is formed from bottom of map upwards toward edge
    robotWidth = 2   # number of boxes from mapping
    y0 = numRow -1       # starting y point
    xFinish = numCol - 1 # finish horizontal node
    xStart = xFinish - 3 # start horizontal node
    #x0 = numCol//2 - robotWidth//2 # starting x point
    row = y0
    loop = True
    while loop :
        for col in range(xStart, xFinish):
            roamCell = img_gray[row*cellSize-cellSize:row*cellSize , \
                                                    col*cellSize:col*cellSize+cellSize]
            if(row==0):
                fwdLimitCoord = (0,5)
                loop = False
                break
            occupancyGrid[row,col]
                =ssim(originalCell,roamCell,multichannel=True,win_size=5)
            if(occupancyGrid[row,col] < 0.6):
                fwdLimitCoord = (row,col) # x,y coordinate location
                loop = False
                #break
        row = row -1
    return fwdLimitCoord


# initialize distance matrix - my version of calibration matrix (incehs)
distanceMatrix = [[22, 22, 22, 20, 25, 25, 20, 22, 22, 22],
                  [14, 14, 13, 13, 13, 13, 13, 13,14, 14],
                  [9.5, 9, 9, 9, 9, 9, 9, 9, 9, 9.5],
                  [7, 7, 6.8, 6.4, 6, 6, 6.4, 6.8, 7, 7],
                  [5.5, 5.2, 5, 4.8, 4.6, 4.6, 4.8, 5, 5.2, 5.5],
                  [4.2, 4, 3.8, 3.5, 3.5, 3.5, 3.5, 3.8, 4, 4.2],
                  [3.5, 3, 2.8, 2.7, 2.7, 2.7, 2.7, 2.8, 3, 3.5],
                  [2.5, 2.3, 2, 2, 1.8, 1.8, 2, 2, 2.3, 2.5]]

arcMatrix = [[1.1, 1.05, 1.03, 1.03, 1.1, 1, 1.01 ,1.03 , 1.05 ,1.1],
           [1.15, 1.1, 1.03, 1.01, 1, 1, 1.01 ,1.03 ,1.1, 1.15],
           [1.2, 1.15, 1.1, 1.03, 1.01, 1.01, 1.03, 1.1, 1.15, 1.2],
           [1.25 ,1.25, 1.2, 1.1, 1.01, 1.01, 1.1, 1.2, 1.25, 1.25],
            [1.4, 1.25, 1.25, 1.2, 1.01, 1.01, 1.1 ,1.25 ,1.25 ,1.4],
            [1.5, 1.4, 1.25, 1.1, 1.01, 1.01, 1.1, 1.25, 1.4, 1.5],
            [1, 1, 1, 1, 1.01, 1.01, 1, 1, 1, 1],
            [1, 1, 1, 1, 1.01, 1.01, 1 ,1 ,1, 1]]

cam = VideoCapture(0)              # initialize the camera
cellSize = 60
#height, width = np.shape(img)[:2]
height = 480             # hard coding in order to reduce comp. time
width = 640              # hard coding in order to reduce comp. time
numRow = height//cellSize
numCol = width//cellSize

motorInput = 100

tryAgain = True
while(tryAgain):
    # open loop control of driving towards edge
    s, img = cam.read()                  # capture image
    edgeCoord = findEdge(img, height, width, cellSize,numRow, numCol)
    dist = distanceMatrix[edgeCoord[0]][edgeCoord[1]]
    dist = dist + 4 # distance correction
    r = arcMatrix[edgeCoord[0]][edgeCoord[1]] # ratio for arc movement
    print(dist,r)
```

```
timeRemaining = timeOfMovement(dist, motorInput)
motors(r*motorInput,motorInput)
time.sleep(timeRemaining) # distance to drive before reaching edge
motors(0,0)
response = input("Try again? ") # to repeat experiment
if(response == 'n'):
  tryAgain = False
```

Appendix B

Calibration

B.1 Distance and Arc Matrices

$$\text{distanceMatrix} = \begin{bmatrix} 22 & 22 & 22 & 20 & 25 & 25 & 20 & 22 & 22 & 22 \\ 14 & 14 & 13 & 13 & 13 & 13 & 13 & 13 & 14 & 14 \\ 9.5 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9.5 \\ 7 & 7 & 6.8 & 6.4 & 6 & 6 & 6.4 & 6.8 & 7 & 7 \\ 5.5 & 5.2 & 5 & 4.8 & 4.6 & 4.6 & 4.8 & 5 & 5.2 & 5.5 \\ 4.2 & 4 & 3.8 & 3.5 & 3.5 & 3.5 & 3.5 & 3.8 & 4 & 4.2 \\ 3.5 & 3 & 2.8 & 2.7 & 2.7 & 2.7 & 2.7 & 2.8 & 3 & 3.5 \\ 2.5 & 2.3 & 2 & 2 & 1.8 & 1.8 & 2 & 2 & 2.3 & 2.5 \end{bmatrix}$$

$$\text{arcMatrix} = \begin{bmatrix} 1.1 & 1.05 & 1.03 & 1.03 & 1.1 & 1 & 1.01 & 1.03 & 1.05 & 1.1 \\ 1.15 & 1.1 & 1.03 & 1.01 & 1 & 1 & 1.01 & 1.03 & 1.1 & 1.15 \\ 1.2 & 1.15 & 1.1 & 1.03 & 1.01 & 1.01 & 1.03 & 1.1 & 1.15 & 1.2 \\ 1.25 & 1.25 & 1.2 & 1.1 & 1.01 & 1.01 & 1.1 & 1.2 & 1.25 & 1.25 \\ 1.4 & 1.25 & 1.25 & 1.2 & 1.01 & 1.01 & 1.1 & 1.25 & 1.25 & 1.4 \\ 1.5 & 1.4 & 1.25 & 1.1 & 1.01 & 1.01 & 1.1 & 1.25 & 1.4 & 1.5 \\ 1 & 1 & 1 & 1 & 1.01 & 1.01 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1.01 & 1.01 & 1 & 1 & 1 & 1 \end{bmatrix}$$

B.2 Calibration Sheet

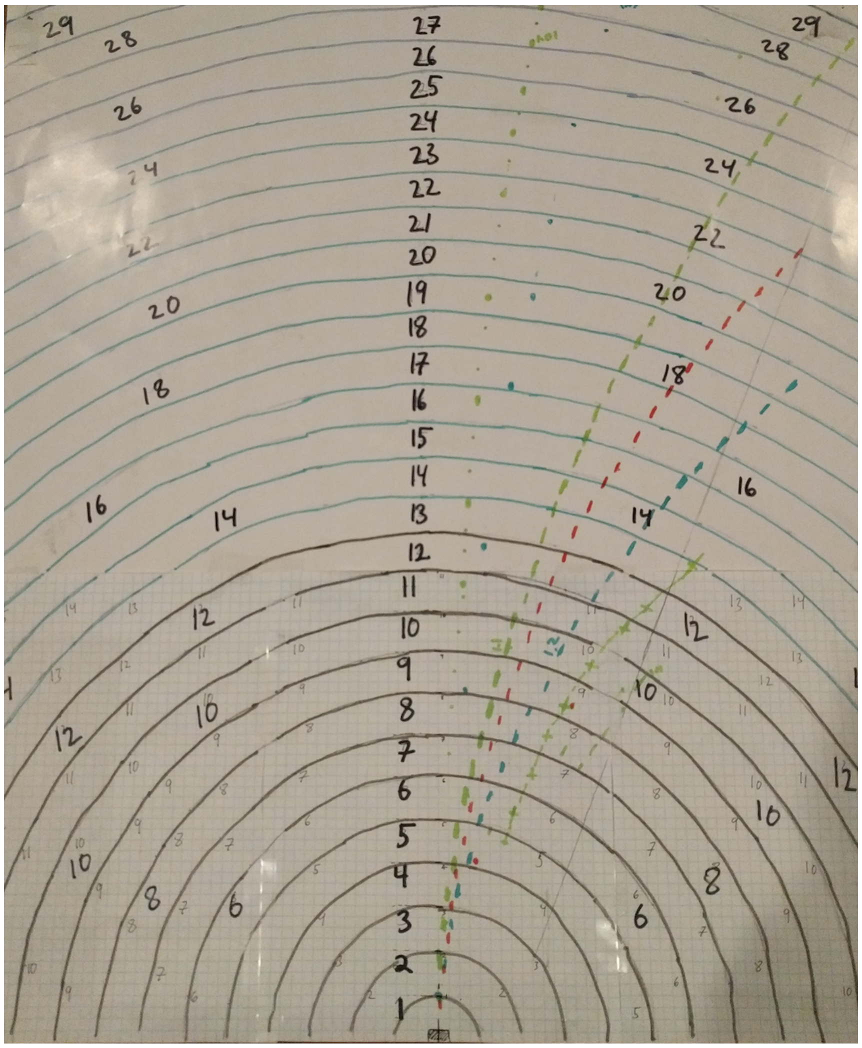

Figure B.1: Calibration sheet displaying distances from the root and the handful of arc trajectories recorded.

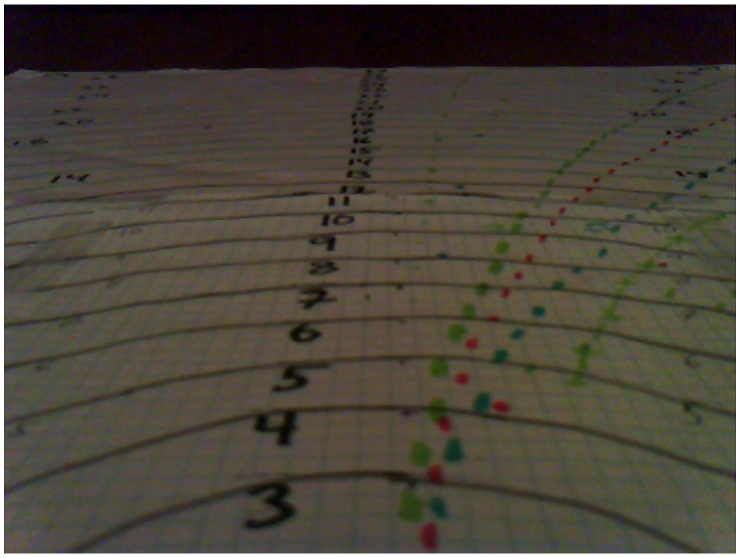
Figure B.2: Calibration sheet with arc trajectory, captured by the robot.

Appendix C

Edge Detection Sequence

C.1 K-Means Edge Detection Sequence

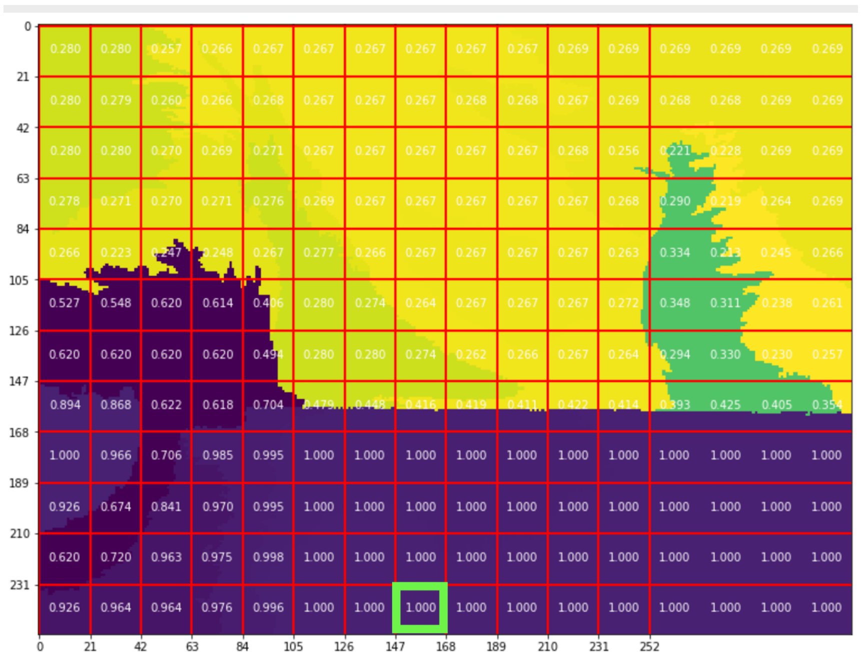

Figure C.1: K-Means Edge Detection Sequence: Initial Step.

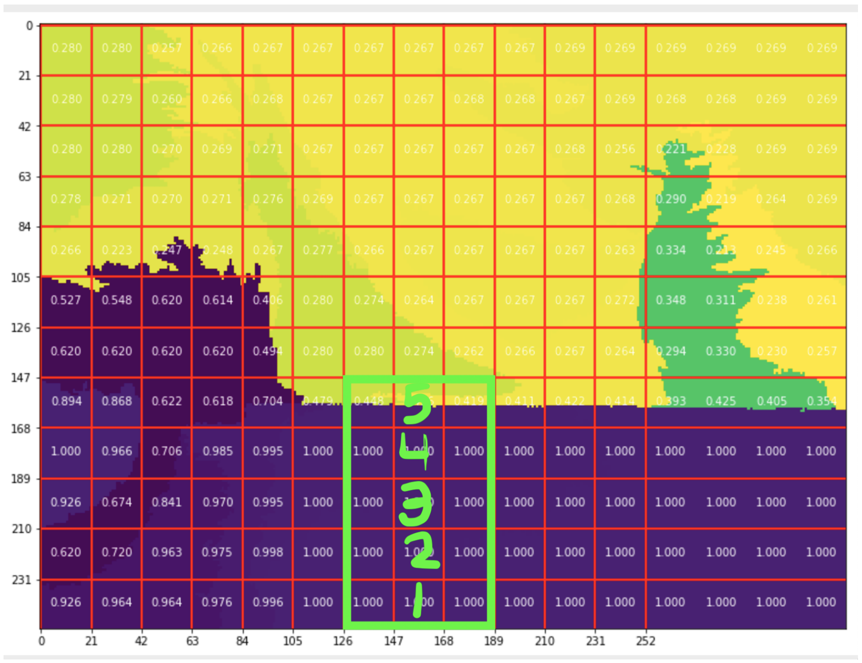

Figure C.2: K-Means Edge Detection Sequence: Final Step.

C.2 Gray Scale Edge Detection Sequence

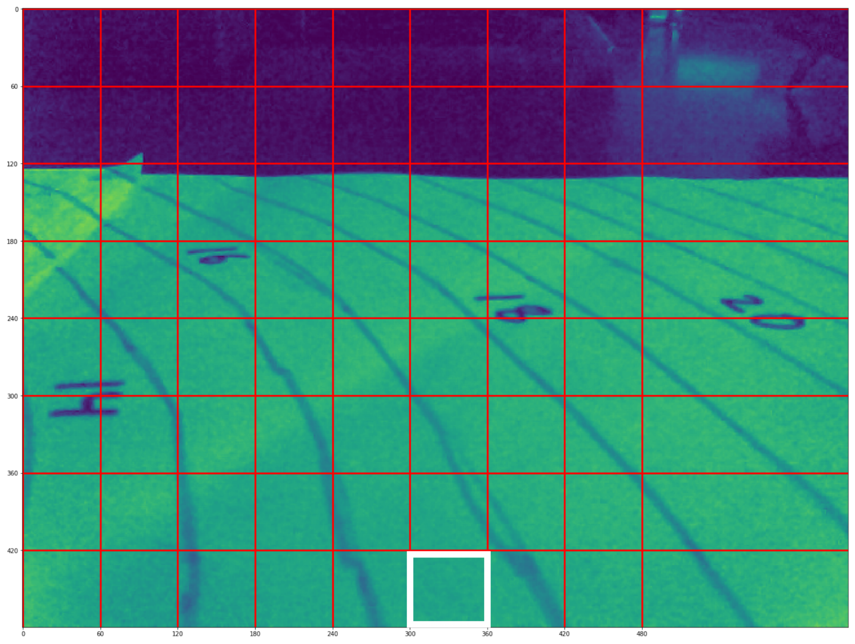

Figure C.3: Gray Scale Edge Detection Sequence: Initial Step.

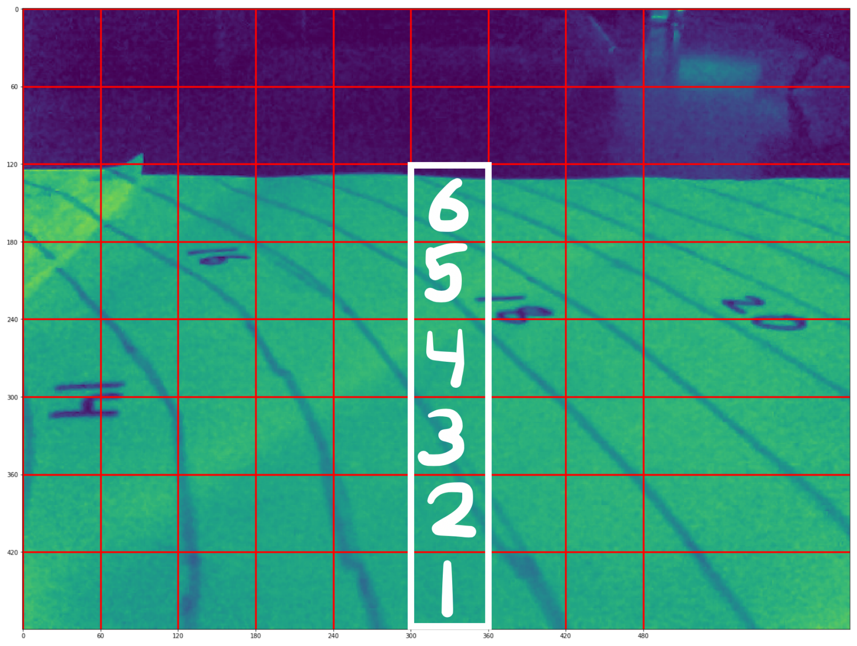

Figure C.4: Gray Scale Edge Detection Sequence: Final Step.

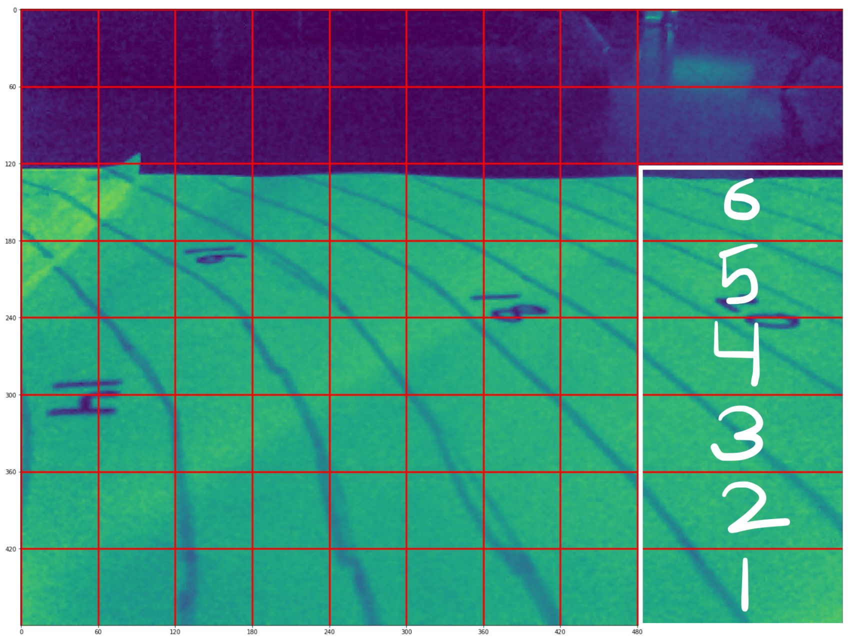

Figure C.5: K-Means Edge Detection Sequence: Final Step for Corner.

Bibliography

[BFL06] Yuri Boykov and Gareth Funka-Lea. Graph cuts and efficient n-d image segmentation. *International Journal of Computer Vision*, 70(2):109–131, Nov 2006.

[LWLS93] Liang-Wei Lee, Jing-Fa Wang, Jau-Yien Lee, and J. . Shie. Dynamic search-window adjustment and interlaced search for block-matching algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(1):85–87, Feb 1993.

[Ren93] W. D. Rencken. Concurrent localisation and map building for mobile robots using ultrasonic sensors. 3:2192–2197 vol.3, July 1993.

[RI99] James K. Rilling and Thomas R. Insel. The primate neocortex in comparative perspective using magnetic resonance imaging. *Journal of Human Evolution*, 37(2):191–223, 1999.

[Yan18] Daniel Yang. A minimalist stair climbing robot (scr) formed as a leg balancing and climbing mobile inverted pendulum. *2018 IEEE/RSJ International Conference on. IEEE*, Intelligent Robots and Systems (IROS), 2018.