ENPM673 Midterm

Brendan Neal

March 16, 2023

Contents

1 Introduction

This report outlines my process, results, and problems encountered for the ENPM673 Spring 2023 Midterm.

2 Problem 1

2.0.1 Problem 1a

The biggest questions that need to be answered for what perception sensors need to be used on an autonomous robot are: Where am I and what do my surroundings look like? In the case of designing a wall painting robot, several perception sensors can be used. Since you are indoors, a camera can be your main source of perception. Inside a home, the camera is in a controlled environment with little to no environmental concerns. The type of camera can vary, be it a 3-D camera, a 360 camera, or just a normal front-facing camera. Additionally, the number of cameras can change depending on the design of the robot. If the wall painting robot has an arm, you could have a camera on the arm and the camera on the body. In general, the camera (or cameras) can answer the two aforementioned important questions, with a lot of computer programming. It can identify where you are in the room as well as where you are relative to the wall. Additionally, it can perceive obstacles such as furniture and other people, and can change its actions from that data accordingly. These examples assume that the camera has been programmed with proper localization and calibration protocols to back out the camera pose relative to other objects. However, if all the conditions mentioned before are met, a wall painting robot can achieve its task with a camera or a set of cameras.

2.0.2 Problem 1b

In the case of an underwater robot, cameras are most likely not the best option due to the environmental factors. The water could be too murky, the pipe to be fixed could be so deep that outside light may not reach it, or the quality of camera that one needs will not survive in an underwater environment. Thus, another perception sensor is required to answer the questions where am I and what does my environment look like? Our two options include LIDAR and SONAR. Both LIDAR and SONAR have been used for non-autonomous systems involved in underwater operations such as submarines. LIDAR can give a point cloud mapping of the environment around it, and along with the SONAR measuring the distance of the pipe fixing robot from the bottom, these two sensors can localize the robot's position if programmed properly. Additionally, if the robot's sensor processing algorithms are programmed properly, the robot should be able to identify the pipe location versus the surrounding environment.

3 Problem 2

This problem had me to detect a ball being thrown using the Hough Transform technique.

3.0.1 Pipeline

To start this problem, I first loaded the video. I would then perform the following steps per frame. I would convert the image frame to gray-scale. Next, I would then blur the image using a Gaussian blur. These first two steps are integral to performing image analysis, since gray-scale cuts down the number of computations significantly (one color channel vice three) and blurring the image works to eliminate the noise. After setting the image to gray-scale and blurring, I used the function cv2.HoughCircles() on the blurred image in order to extract the ball in a frame. Since we were instructed to use the Hough Transformation technique, cv2.HoughCircles() was the most logical function to use. cv2.HoughCircles() actually combines the edge detection and Hough circle fitting into a single function. I set the edge detection technique to be Hough Gradient, my theta resolution to 1 pixel, the number of indices separating circles to 20, the gradient value for edge detection to 150, the Hough Accumulator threshold to 15, and the min and max radius distances to be 1 and 15 pixels respectively. I chose to look between 1 and 15 pixels in order to be more robust in capturing circles (the ball) in case conditions in the video changed. Once I detected the circle, I would then plot a green circle over top the video.

3.0.2 Results

Pictured below in Figure 1 is a sample frame of the detected ball with the green circle plotted over top.

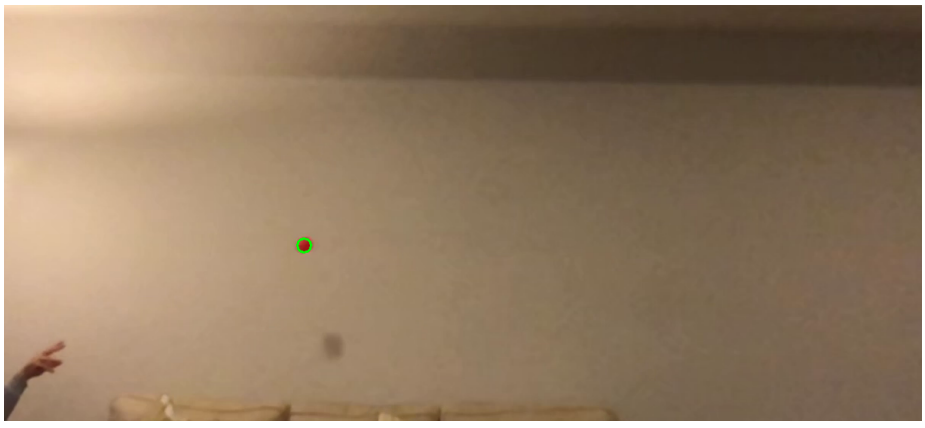

Figure 1: Detected Ball Frame

3.0.3 Problems Encountered

No major problems were encountered for this section. The only issue I ran into was tuning the cv2.HoughCircles() function to not include falsely detected circles.

4 Problem 3

Problem 3 had me transform an image of railroad tracks to be a top-down view and calculate the average distance between the tracks.

4.0.1 Pipeline

The first step in this was to load the image. After loading and examining the image file, I manually found four "points of interest." These "points of interest" were four points on the railroad tracks that would represent the corners of my warped image. Next, I set four desired warping points that would result in a flat plane, top-down view. For my points of interest, I selected ((1483,1020), (1541,1020), (950,1920), (2050,1920)). For my desired warping points on a flat plane, I selected ((0,0), (58,0), (0,900), (58,900)). These dimensions came from adjusting the points of interest. I used the "back left" corner on the train track as my origin, and mapped the other points accordingly. Essentially, I am warping the image to the size of the furthest points. Please see the "Results" section for a visual representation of what this means. I also chose four points because that is the minimum number of points required for a homography/transformation calculation.

Once I have my four pairs of points, I used the cv2.getPerspective Transformation() function with the pairs of points as inputs in order to output a transformation matrix between the original image and the outputted top-down view. Once I had the transformation matrix, I used cv2.warpPerspective() to warp the original image into the top-down view of the same size of the flat plane I defined above (58x900 pixels). I used cv2.getPerspectiveTransform() because the transformation matrix is a matrix specifically designed to map points of one image to points of another image. I used cv2.warpPerspective() to perform the image transformation operation.

Now with the warped image, I can begin the pipeline of calculating the average distance between the railroad tracks. First, set the top-down image to gray-scale and then blur the image using a Gaussian blur with a 7x7 kernel. I performed these two operations because gray-scale cuts down the number of computations significantly (one color channel vice three) and blurring the image works to eliminate the noise. Next, I used cv2.Canny() to detect the edges found in the top-down image. The goal is to fit lines to the railroad tracks using Hough Transform, and the first step to applying the Hough Transform is performing

edge detection. Next, I used the cv2.HoughLines() function to find all existing lines in the edge detected image. I set my minimum accumulator threshold to 300 votes since I want to limit the detected lines to only the long vertical railroad track lines. Since the Hough Transform function only outputs the D and Theta for each detected line, I had to initialize the lines for display by performing the same line forming calculations used in Project 2. In the process of forming the lines for display, I have the starting and ending X and Y for each line. Since these lines are vertical, I can then calculate the average distance between the tracks by taking the difference between the X values of the two railroad tack lines.

4.0.2 Results

Figure 2 shows the selected points of interest displayed over top the original image.

Figure 2: Points of Interest for Perspective Transform

Figure 3 shows a diagram of how the points of interest will be mapped to their top-down planar counterparts.

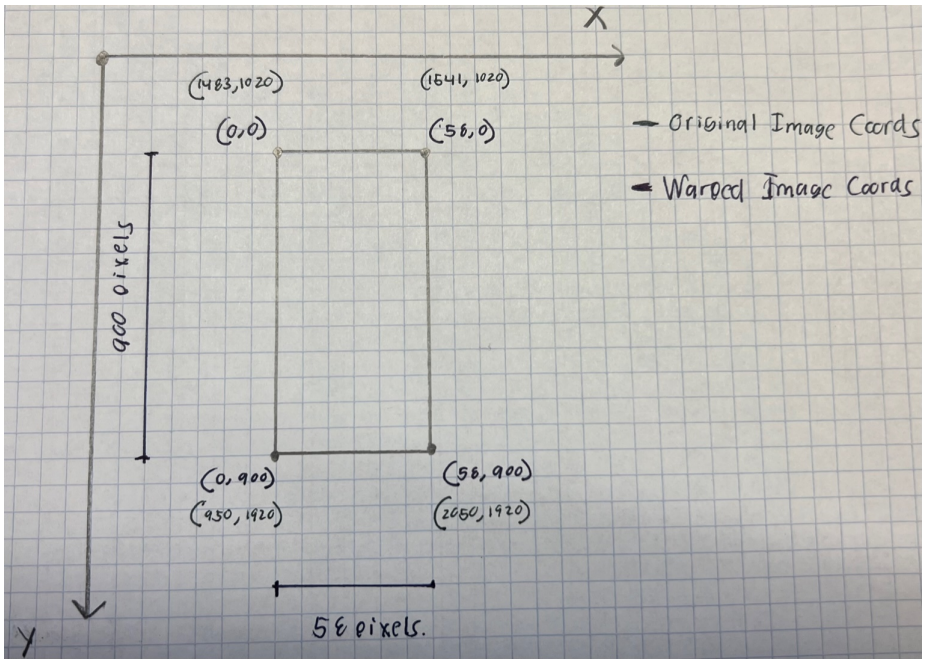

Figure 3: Theory for Mapping Points of Interest to Planar Counterparts

Figure 4 shows the Resultant top-down image after the perspective transforming and warping operations.

Figure 4: Resultant Top-Down Image

Figure 5 shows the gray-scale top-down image.

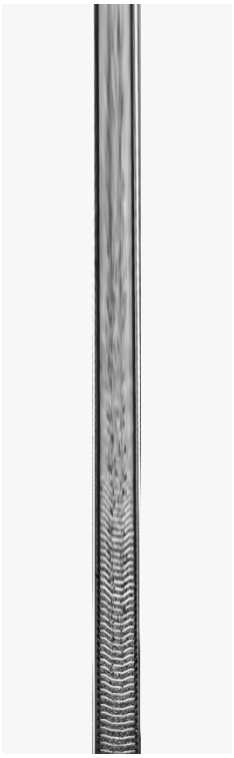

Figure 5: Gray-Scale Top-Down Image

Figure 6 shows the blurred top-down image.

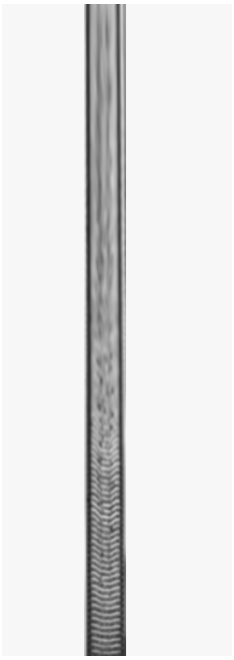

Figure 6: Blurred Top-Down Image

Figure 7 shows the edge-detected top-down image.

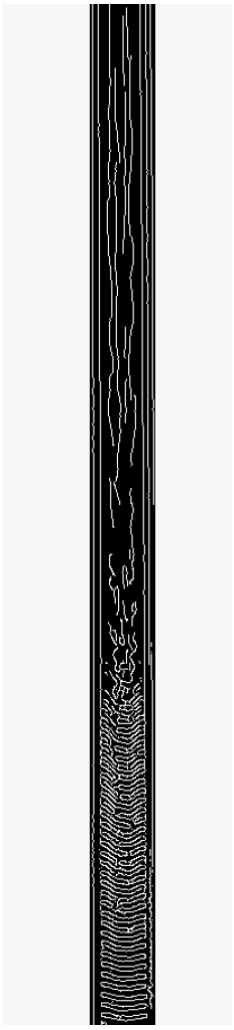

Figure 7: Edge-Detected Top-Down Image

Figure 8 shows the overlaid Hough lines on the top-down image.

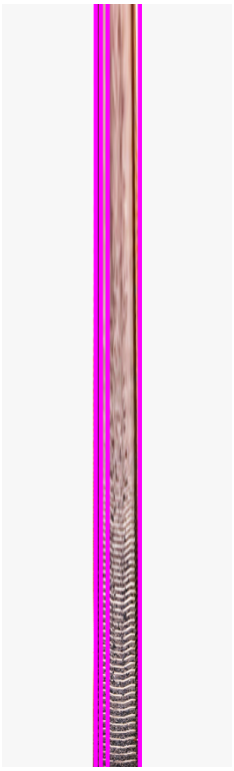

Figure 8: Hough Lines

Finally, the average distance between the railroad tracks was calculated to be 46 pixels.

4.0.3 Problems Encountered

Several problems were encountered in this section. First, the transformed image is extremely blurry, and loses a lot of the detail when traveling "up" the railroad track seen in Figures 5 and 6. This throws off the edge detection software in Figure 7 significantly and forced me to switch strategies when approaching the average distance between the two rails. Originally, I wanted to detect the horizontal lines between the two edges and calculate the distance for each row independently before averaging. However, a lot of information is lost in the upper half of the railroad track, so the next-best option was to generate lines for the metal tracks and then consider those as an average when finding the distance between the two parallel lines.

Additionally, as shown in Figure 8, the Hough transformation function is finding three lines close to each other on the left track. To remedy this, I wrote

a script that would compute the distance between each line and take the max distance (since visually, the furthest line to the left is where the track lies).

5 Problem 4

This problem has me identify distinct balloons in an image (without resolving occlusion) and label them with different colors.

5.0.1 Pipeline

The first step to solving this problem is to load the image from my workspace. Next, I would convert the image to gray-scale. After converting the image to gray-scale, I would then blur the image using a Gaussian blur with kernel size 11x11. I performed these two operations because gray-scale cuts down the number of computations significantly (one color channel vice three) and blurring the image works to eliminate the noise. Additionally, a gray-scale image is required for my next step in the pipeline: thresholding. I used the cv2.threshold() function to threshold the background away from the balloons. I used binary thresholding for this operation. I noticed that the balloons themselves were a lot darker than the background. Thus, I set my binary threshold to 95, so that pixels less than 95 in intensity would be set to 0 (black) and pixels above that thresholds would be set to 255 (white). I used thresholding because of how I wanted to identify the balloons (connected component analysis) and a required input to connected component analysis is a binary image.

After tuning and examining my threshold value, I noticed that there was a significant presence of holes in my balloons. Specifically, the baskets would get separated from the balloon itself, thus forcing the connected components algorithm to classify them as different objects. I wanted to fill those holes, so I used the cv2.morphologyEx() function to close the holes with a structuring element of ones, size 25x25. This operation accurately closed the holes in the image (reconnecting the baskets to the balloon canopy).

Now that I have a well-thresholded image, I used the cv2.connectedComponentsWithStats() function to get the number of labels, the index of each labeled object, get stats per object, and compute the centroid of the object. This function labels each "balloon blob" and computes stats for each of the objects. The important stats that I pulled from this function are the x and y location of the blob, the width and height, and the area of the blob. I would use these stats in labeling. To label these objects, I would loop through each object, use the random function to generate a random color, and draw a bounding box around each of the balloons. Each bounding box labels the balloon separately, and they are of all different colors. I chose connected component analysis for this section because of its simplicity and ease of use. There is no sense training a neural

network to identify balloons in a single image. Additionally, color thresholding would not work because the balloons are made up of many different colors. Finally, I chose to draw bounding boxes to identify the balloons rather than changing the color because cv2.connectedComponentsWithStats() will already output these values, so it just seems easiest to use them rather than manually changing the pixel colors for each connected component.

5.0.2 Results

Figure 9 shows the gray-scale image of the balloons.

Figure 9: Gray-Scale Balloon Image

Figure 10 shows the blurred gray-scale image of the balloons.

Figure 10: Blurred Gray-Scale Balloons

Figure 11 shows the thresholded balloon image with no morphology.

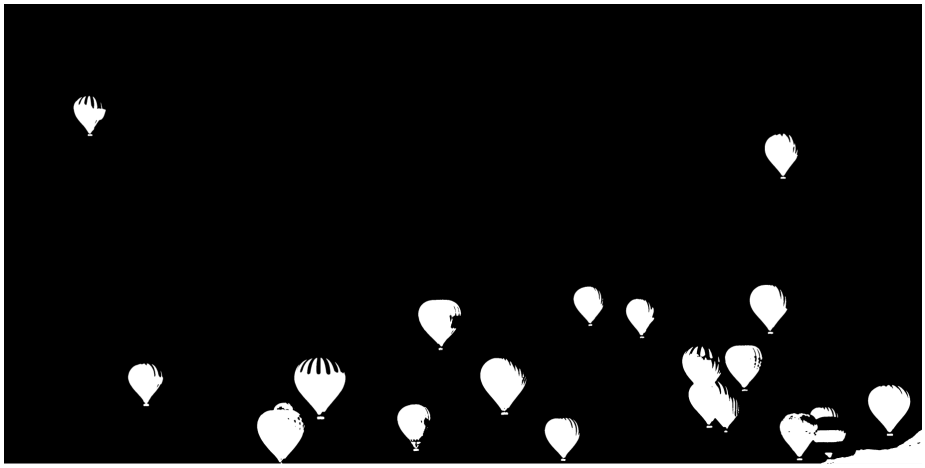

Figure 11: Original Balloon Threshold

Figure 12 shows the thresholded balloon image after morphology has been performed.

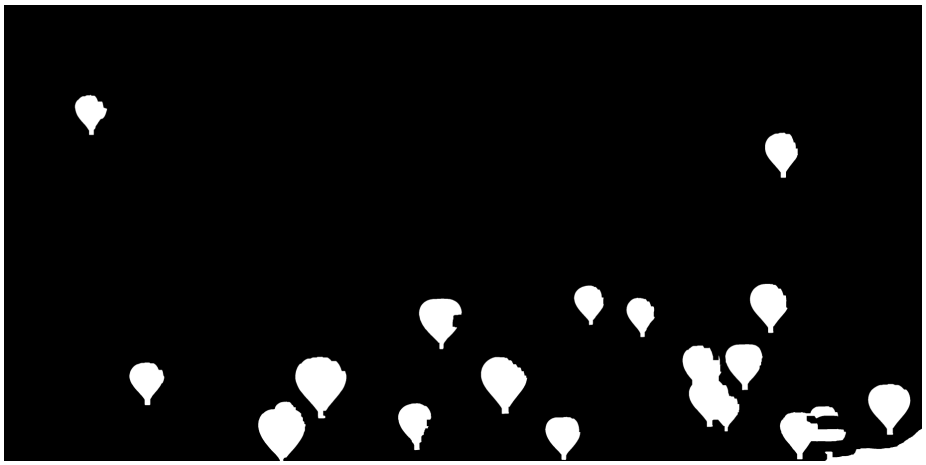

Figure 12: Closed Balloon Threshold

Figure 13 shows the bounding-box labeled balloons.

Figure 13: Identified Balloons

5.0.3 Problems Encountered

No major problems in this section, only it took a while to tune my threshold and morphology values.

6 Problem 5

Problem 5 has me applying K-Means to obtain three clusters from a set of data.

6.0.1 Pipeline

The first step in K-Means is to randomly select three starting points for our three clusters. In this example, I randomly selected 2, 11, and 15 for my starting clusters. Then I sorted each of the remaining data points into these clusters based on Euclidean distance. This yields the following three clusters:

$$C_1(2) = 2, 4, 6$$
$$C_2(11) = 11, 13$$
$$C_3(15) = 15, 22, 25$$

Once these clusters have been formed in the first iteration, I then compute the mean of each cluster. This yields:

$$Mean_{C_1} = 4$$
$$Mean_{C_2} = 12$$
$$Mean_{C_3} = 20.67$$

Once I have the means of each original cluster, I then resort the data points based on the new, adjusted clusters:

$$C_1(4) = 2, 4, 6$$
$$C_2(12) = 11, 13, 15$$
$$C_3(20.67) = 22, 25$$

Then, as a check, I recompute the mean for each cluster and attempt to re-sort. Upon doing that, there are no changes in the clusters, and thus the algorithm ends.

6.0.2 Results

Below are the sorted clusters:

$$C_1(4) = 2, 4, 6$$
$$C_2(12) = 11, 13, 15$$
$$C_3(20.67) = 22, 25$$

Please note, that the "flow" of this algorithm entirely depends on what the randomly generated initial clusters are. Essentially, the number of iterations could change based on those initial clusters.