AINT308 - OpenCV Assignment 2 2022

Student No. 10618407
School of Engineering,
Computing and Mathematics
University of Plymouth
Plymouth, Devon

***Abstract*—Machine vision is a mature technology that is becoming more prevalent within modern engineering practises. It is being utilised more in the rapidly evolving fields of autonomy and automation. This report outlines some of the functionalities of a popular C/C++ based computer visions library *OpenCV*. The Assignment has been split into two tasks; Task 4 and Task 5. The first task, Task 4, is Disparity Mapping using a pair of stereo cameras to be able to judge distances to objects. The second task, Task 5, was Self-driving Car Lane Detection from the dashcam footage taken from a car.**

Keywords:

Computer Vision, OpenCV, Object Detection, C++, Lane Detection

I. Task 4: Disparity Mapping

A. Introduction

The first task was to perform disparity mapping to evaluate the distance of a given object in a frame to allow a robot to navigate it's environment. Disparity Mapping allows the robot to build up a 3D view of it's environment

B. Solution

The solution to this task was split up into multiple parts. Firstly, using the calibration images to calibrate the cameras. Secondly, using the calibration result to correct the stereo distance targets. Next, Combining the left and right images into a disparity map, with brightness indicating the distance from the camera. Penultimately, labelling the distance targets with their known distance to the camera. Use this to plot disparity against distance and write a formula relating distance to disparity. Finally, using the formula to calculate the distance for each of the unknown targets.

1) Camera Calibration and Stereo Image Correction

Stereo camera needs to be calibrated to one another before they can be used for stereo vision. Corrections need to be made for both physical misalignments (extrinsic error), and lens distortion(intrinsic error) [1]. This was done using the `SteroCalibration` program provided by *OpenCV* [2].

The program firstly reads in image pairs (taken from the left and right camera) of a checkerboard pattern target placed in different locations and orientations. This allows the program to create a list of intrinsics and extrinsic parameters to allow for these effects in the program. The checkerboard is of known size so the program can calibrate what it sees against what it is expecting. This known as the Bouquet stereo image calibration [3].

2) Disparity Mapping

The first stage of disparity mapping was to remap the images to correct for the positional/lens distortion, as mentioned above [4]. The two images were then combined into a 16 bit stereo block matcher [5]. The code for doing this can be found in Fig 1.

```
//Load images from file
Mat Left =imread("../Task4/Distance Targets/left" +to_string(ImageDistance)+"cm.jpg");
Mat Right=imread("../Task4/Distance Targets/right"+to_string(ImageDistance)+"cm.jpg");
cout<<"Loaded image: "<<ImageDistance<<endl;
//Distort image to correct for lens/positional distortion
remap(Left, Left, map11, map12, INTER_LINEAR);
remap(Right, Right, map21, map22, INTER_LINEAR);

//Match left and right images to create disparity image
Mat disp16bit, disp8bit;
// compute 16-bit greyscalse image with the stereo block matcher
sgbm->compute(Left, Right, disp16bit);
disp16bit.convertTo(disp8bit, CV_8U, 255/(numberOfDisparities*16.));
// Convert disparity map to an 8-bit greyscale image so it can be displayed
// (Only for imshow, do not use for disparity calculations)
```

Fig. 1: Remapping Left and Right Images into one image

Once this was done, a 16 bit disparity map of the two images has been created. To measure the disparity, a rectangular region of interest was created, as seen in fig 2. This was created to make sure that only the disparity of the measured object was used. Whilst in the actual test the disparity map used was the 16-bit version, it does not get outputted correctly, so the 8-bit version is used for demonstration purposes.

```
int x = 355;
int y = 210;
int rectangleWidth = 50;
int rectangleHeight = 50;

Rect rect(x, y, rectangleWidth, rectangleHeight);
rectangle(disp8bit, rect, Scalar(255, 0, 0));
```

Fig. 2: Code for Rectangle Drawn On Disparity Map to Indicate Search Area

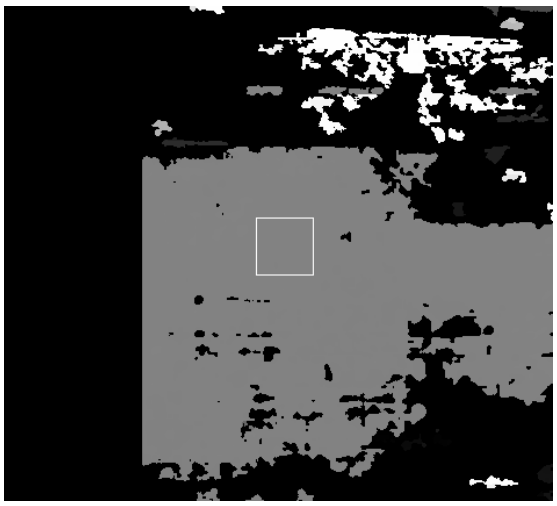


Fig. 3: Disparity Frame with Rectangle drawn on for Region of Interest

Fig 4 is the main code to check the disparity for this task. The code is searching through all of the pixels in a given area (as constructed in fig 2). If the disparity value is larger than a certain value, it is added to the list.

```
// runs through all the rows in the image
for (int i = x; i < x + rectangleWidth; i++) {
    // runs through all the columns in the image
    for (int j = y; j < y + rectangleHeight; j++) {
        // stores the RBG values in the PixelValue vector
        int PixelValue = (int)disp16bit.at<ushort>(j,i);
        if (PixelValue < 65000){
            OutputValue += PixelValue;
        }
    }
}
```

Fig. 4: Disparity Mapping Code for evaluating the disparity in a image

At the end of the run, the average value of the disparity is found, this is shown in fig 6. These values were saved to a CSV files for evaluation after execution.

3) Labelling Known Distances and Creating Disparity Function

From the values of measured distance and disparity shown in table I. The graph in fig 5 was created to map the disparity over distance.

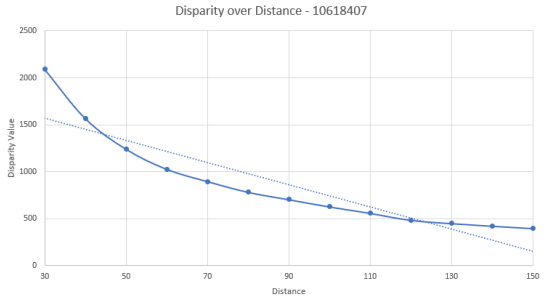


Fig. 5: Graph of Disparity over Distance

TABLE I: Distance and Disparity Table

Measured Distance	Disparity Value	Distance x Disparity
30	2089.72	62691.6
40	1566	62640
50	1240.27	62013.5
60	1026.87	61612.2
70	896.546	62758.22
80	782.35	62588
90	703.714	63334.26
100	626.804	62680.4
110	558.514	61436.54
120	483.93	58071.6
130	449.154	58390.02
140	421.226	58971.64
150	396.406	59460.9

The full size graph can be seen in Appendix C.

Fig 5 has been made using the data from table I. The value for Bf was calculated by rearranging the disparity equation 1 to equation 2. To do this, the average of the $Distance \cdot Disparity$ was taken, as shown in equation 3.

$$Disparity = \frac{B \cdot f}{Distance} \tag{1}$$

$$B \cdot f = Disparity \cdot Distance \tag{2}$$

$$(B \cdot f)_{\text{average}} = \frac{Distance \cdot Disparity}{TotalNumberofValues} \tag{3}$$

Once the value of Bf had been found it was used to calculate the the distances for the known targets, as of test of the previous measurements accuracy this can be seen in equation 2. The code for calculating the distance and outputting the result can be seen in fig 6.

```
OutputValue = (OutputValue/(rectangleHeight * rectangleWidth));

cout << "Output Value : " << OutputValue << endl;

// using calulated BF value
double BF = 61280;

// using the disparity equation
double calcDistance = BF / OutputValue;

cout << "Calculated Distance : " << calcDistance << endl;

DataFile << OutputValue << endl;
```

Fig. 6: Code to output the distance measured to the object

From the values of measured distance and disparity shown in table II, and the accuracy of the prediction was also recorded. As the distance value for the testing images was already known and the disparity value had just been calculated, it was easy to test the output of the equation for reliability and accuracy. The graph of error over distance can be seen in fig 7.

TABLE II: Distance Calculated from Disparity Mapping with Known Distances

Measured Distance	Calculated Distance	% Difference
30	29.32	2.25
40	39.13	2.17
50	49.41	1.18
60	59.68	0.54
70	68.35	2.36
80	78.35	2.10
90	87.08	3.24
100	97.77	2.23
110	109.72	0.25
120	126.63	-5.53
130	136.43	-4.95
140	145.48	-3.91
150	154.59	-3.06

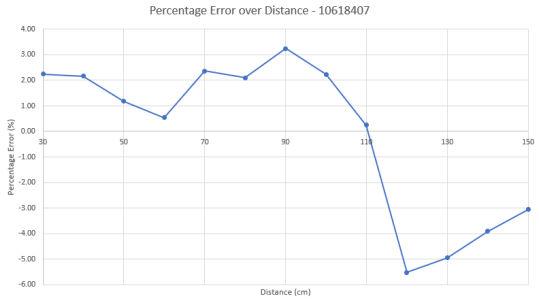


Fig. 7: Graph of Error over Distance

The full size graph can be seen in Appendix D.

4) *Using Disparity Formula to calculate distances for unknown targets*

To find the distance for the unknown targets, the disparity value was firstly calculated using the same method as described above. The disparity value, along with the value previously calculated using equation 2, can be used to calculated the distance of the object in the images.

The method for evaluating the distances of the objects in the images was very similar to the function used in, within the Disparity Mapping section. The main differences between the two codes are, the source of the files and the terminal output of the program. The files used in calculating the distance were images with an unknown distance to the object in the image. The output in the terminal was almost identical but the known distance of the object was not stated - as it was unknown.

It is unclear how close the the estimates of distance were to the actual values but given the accuracy on the known images (within 5.53%), it would be safe to assume that these values could be considered accurate within a 6% margin.

The full flowchart for the code can be seen in Appendix A.

The full breakdown of the code can be seen in Appendix B.

The video showing the known distance targets can be seen here.

The video showing the unknown distance targets can be seen here.

C. Further Improvements

Although this method was effective there are still ways in which it could be improved. The two biggest flaws with this method was how the calibration was conducted.

The distance measurement was conducted using a a box held aloft over a ruler. Whilst this did provide adequate calibration data for the disparity testing, it could have been conducted in a manor that would have made the measurements increasingly more precise.

A clearer background would have been beneficial for disparity matching. The utilised background had lots of moving objects that were not consistent between images. This required a region of interest to be used for the disparity mapping. This resulted in a smaller area for the disparity to be measured across. Ideally a plain background would have been used to allow a larger amount of the frame to be mapped. This would increase the area for the disparity mapping thus reducing the effect of outliers within the region of interest.

These were the two biggest areas of uncertainty in this method and thus would gain the best yield if improved.

D. Conclusion

Overall this task was hard to judge on the unknown data, as the results were not known and thus hard to tell the accuracy of the finalised results. Despite this, the known measurement disparity mapping was extremely successful, with a maximum error of 5.53% and an average error of only 2.9%. Although the methodology could have been improved by making the calibration data collection more precise and using a clearer background, the distances of measurement used (30cm-150cm) would negate the effects of measurement over that distance.

II. Task 5: Self-driving Car Lane Detection

A. Introduction

The second task was to track the lanes in dashcam footage from the front of a car driving on the road. This is one of the most complex computer vision applications within the field of autonomous cars. Mordern cars have a plethora of cars around their circumference, this allows them to capture large amounts of data whilst on the road. The real difficulty with this is how to break down the information into smaller useful chunks.

B. Solution

1) *Use a method to detect the lane markings*

The first part of lane detection was to convert the video frames to grey scale, this was done to reduce the informational load to compute. Although Canny Edge detection works less effectively using a grey scale image

[6], the lines for the edges of the road were still found and detected as edges. The frames are converted to grey scale using OpenCV's `cvtColor` function as seen in fig 8.

```
// Convert Frame to Grey
cvtColor(Frame,greyFrame,COLOR_BGR2GRAY);
```

Fig. 8: Code to Convert from colour to grey scale

After the frames are converted to grey scale, the frame is then blurred using OpenCV's `blur` function – insert REF – insert fig of the code. This was done to reduce the noise in the image, this makes the edge detection perform better [7].

```
// Blur Image
blur(greyFrame, detectedEdges, Size(3,3));
```

Fig. 9: Code to Blur Frame

Once the image was blurred, a rectangular mask was placed over the top half of the image using the `rectangle` function - insert REF – insert fig of the code. This was used to eliminate the top half of the image which would not be useful for finding the road markings.

```
rectangle(detectedEdges, start, end, Scalar(255,255,255), -1);
```

Fig. 10: Code to Create a rectangle over a frame

Canny Edge detection was then conducted on the masked frame using OpenCV's `Canny` method - insert OPENCV REF here – insert fig of the code. Canny Edge was chosen because it is deemed to be one of the best edge detection algorithms [8]. Canny Edge detection uses non-maximum suppression [9] and Hysteresis Processes [10]. Non-maximum Suppression and Hysteresis Process reduce the number of false edges and thus create a better starting point for further processing such as Hough Transforms.

```
// Canny Edge Detection
Canny(detectedEdges, detectedEdges,
      lowThreshold, lowThreshold*ratio,kernel_size);
```

Fig. 11: Code to Run Canny edge detection

After the Canny Edge detection, Hough Transforms are used as a method of extracting features from an image. It can be used to isolate features of regular curves such as, circles, lines, and ellipses. At it's simplest implementation, Hough Transforms can be used to detect straight lines, like the lines at the side of a road. Computational complexity deters the use of Hough transforms in a lot of situations. However,for finding straight lines, the most simplistic use case, Hough Lines were the chosen method of feature extraction [11]. Simple Hough Transforms are more resilient to noise in the frame [12]

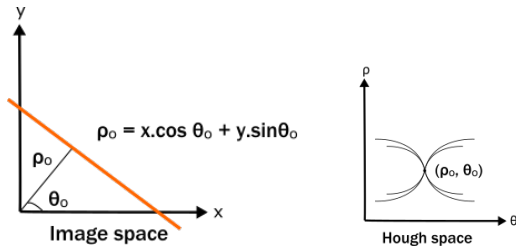
The specific implementation used chosen was `HoughLines`, which can be seen in fig 12.

```
// Find coordinates of road lines
vector<Vec2f> lines;
int rho = 1;               // resolution parameter (rho) in pixels
double theta = CV_PI/450;  // resolution parameter (theta) in radians
int thershold = 230;       // Minimum number of intersections to 'detect' a line
HoughLines(detectedEdges, lines, rho, theta, thershold, 0, 0);
```

Fig. 12: Code to Run Hough line detection

Fig 13 show the representation of Hough Lines in polar coordinate form. This is used to show the line as a singular point(ρ, θ) in the hough space.

The Hough Transform constrcuts an MxN array (where M is the distance from the origin ρ, and N is the corresponding angle from the origin θ). These values for ρ and θ can be used to find the corrinsponding x and y coordinates for each point along the lines. This is how the lines were extracted to draw on the edges of the road.



(a) Image Space (b) Hough Space

Fig. 13: Polar Coordinates for Hough Spaceh

2) Display a video with lanes detected

Once the polar coordinates were instantiated, they were evaluated and the lanes were extracted from this.

Initially a group of variables were created to calculate the lanes from the lines of the detection algorithm.

```
int q = lines.size(); // constant used for loop interation
int bottomOfLine = Frame.rows -1; // bottom y coordinate of detected road lanes
int topOfLine = Frame.rows - 300; // top y corrdinate of detected road lanes
vector< Point> cornersOfLane; // vector of four points, corners of drawn lane region
int xVals[4]; // values of the x-coordinates of the detected lines
int prevX[4]; // old coordinates of x-coordinates of the detected lines
int currX[4]; // new coordinates of x-coordinates of the detected lines
float upperbound = 1.01; // upper limit of difference in detected lines
float lowerbound = 0.99; // lower limit of difference in detected lines
```

Fig. 14: Set up Variables for Lane Detection and Plotting

After the initialisation of all the varibales shown in fig 14, the algorithm runs through various loops.

The first loop, `(i<lines.size())`, is used to iterate through all of the lines found by the Hough Lines algorithm to do lane detection on each of the lines. The second loop, `(g>Frame.rows-300)`, loops through all of the y coordinates of the lines in the images. This loop starts at the top of the page and iterates upwards to the cut off point of the denoted by `Frame.rows-300`. This was done to exclude any lines, or parts of lines,

that were above the road and thus not relevant for the lane detection. The third and final loop, `(k<q)`, is used to iterate through the lines to be used in calculating the x coordinates of the points.

```
// itterate through number of lines found
for (int i = 0;i < (int)lines.size(); i++) {
    // output the lines found using the Hough lines algorithm
    // lineRT(Frame, lines[i], Scalar(0,0,255), 2);

    // itterate through the desired y coordinates for lines
    for(int g = Frame.rows; g > Frame.rows - 300; g--){

        // itterate through the number of lines found for each given y value
        for(int k = 0; k < q; k++){
```

Fig. 15: Loops for lane detection algorithm

$$x = \left(\frac{\rho}{sin(\theta)}\right) - (y \cdot tan(\theta)) \tag{4}$$

where

x is x coordinate of the point on the line

y is y coordinate of the point on the line

ρ is the perpendicular distance from the origin

θ is the angle about the origin (in radians)

Fig 16 shows the process of finding the left hand side lane of the road. It starts by evaluating whether the angle of the line (in radians) to the origin is less than 1. If the angle is less than 1, equation 4 to calculate the x value of the given point on the line. This equation uses the output from the Hough Lines to calculate the x-value of a point given it's y-value.

After this x-value is found, the previous x-value is checked. If the new value is within a defined threshold, the value is added with 90% of the old x-value and 10% of the new x-value. If the latest x-value is not within the defined threshold, the old x-value is used. This is done to reduce the jitter within each frame and to make the road tracking smoother. This is called temporal smoothing [13]. This technique is commonly used within lane detection algorithms, it assumes that there will be no rapid movement and changing of lanes [14], things that would be considered to be unsafe driving practises on the road. The code for utilising equation 4 and the corresponding temporal smoothing can be seen in fig 16.

```
if(lines[k][1]<=1){ // if the angle of the lines (in radians) is less than 1

    // equations used to extract x coordinate for a given y value
    currX[0] = (lines[k][0]/cos(lines[k][1]) - (topOfLine*tan(lines[k][1])));
    currX[2] = (lines[k][0]/cos(lines[k][1]) - (bottomOfLine*tan(lines[k][1])));

    // if the current value is within a threshold ( to reduce jittering)
    if ((currX[0] >= lowerbound*prevX[0])&&(currX[0] <= upperbound*prevX[0])){
        xVals[0] = prevX[0] * 0.9 + currX[0] * 0.1;
    } else {
        xVals[0] = prevX[0];
    }
```

Fig. 16: checking angles of lines 1

Much like in fig 16, fig 17 aims to find the corresponding x-values given a y-value input and uses temporal smoothing to reduce the jitter between frames. The main difference between these two methods, other than where in the array the x coordinate is stored, is the angle (in radians) that is being checked. The code in fig 16 uses angles of less than 1 radian whereas fig 17 used angles greater than 2.3 radians. Other than these two differences, the methods work in exactly the same way.

```
} else if(lines[k][1]>=2.3){ // if the angle of the lines (in radians) is greater than 2.3

    // equations used to extract x coordinate for a given y value
    currX[1] = (lines[k][0]/cos(lines[k][1]) - (topOfLine*tan(lines[k][1])));
    currX[3] = (lines[k][0]/cos(lines[k][1]) - (bottomOfLine*tan(lines[k][1])));

    // if the current value is within a threshold ( to reduce jittering)
    if ((currX[1] >= lowerbound*prevX[1])&&(currX[1] <= upperbound*prevX[1])){
        xVals[1] = prevX[1] * 0.9 + currX[1] * 0.1;
    } else {
        xVals[1] = prevX[1];
    }
```

Fig. 17: checking angles of lines 2.3

Once the sides of the lane have been calculated, the middle of the lane was calculated. This was calculated simply by taking the average between the two points picked for the lanes. This was done for the top of the lanes and for the bottom point of the lines. This was a crude implementation for finding the middle of the lane. This could have been done by taking the average between every y point along line but given that the Hough Lines algorithm was outputting only straight lines, this would have given the same result, but with a more computationally complex approach. If the Hough Lines Probability method had been used, the methodolgy for finding the centre line would have been different, this is discussed further in the Further Improvements section.

```
// map the line in the middle of the road
Point topMiddleOfLane (((xVals[0] + xVals[1]) / 2), topOfLine);
Point bottomMiddleOfLane (((xVals[2] + xVals[3]) / 2), bottomOfLine);

// plot line in the middle of the road
line(Frame, topMiddleOfLane, bottomMiddleOfLane, Scalar(255,0,0), 2);
```

Fig. 18: Line in middle of road

After calculating the points for the lanes, and the centre line for the lane, The points were added to a vector to make up the four corners of the lane. Once these values have been added, a new overlay frame is created to display the lane. Once this has been done, the vecotr of corner points, are cast into Points for the polygon function along with the number of points being used. The OpenCV function `fillPoly` was used to create the polygon. This took arguments for the output frame, a pointer to the location of the points, a pointer to the number of points being outputted, the number of contours and the colour of the rectangle.

Once all of the points were calculated, the lane polygon was added on top of the main video frame using the `addWeighted` method. The code for this can be seen in figure 19.

```
// add previously calculated values
cornersOfLane.push_back(Point(xVals[0], topOfLine));
cornersOfLane.push_back(Point(xVals[1], topOfLine));
cornersOfLane.push_back(Point(xVals[3], bottomOfLine));
cornersOfLane.push_back(Point(xVals[2], bottomOfLine));

// draw the detected lane onto the frame
Mat overlayFrame;
double alpha = 0.2;
Frame.copyTo(overlayFrame);
const Point *cornerPoints = (const cv::Point*) Mat(cornersOfLane).data;
int noOfPoints = Mat(cornersOfLane).rows;
fillPoly(overlayFrame, &cornerPoints, &noOfPoints, 1, Scalar(0, 255, 0));
addWeighted(overlayFrame, alpha, Frame, 1 - alpha, 0, Frame);
```

Fig. 19: Polygon of Lanes

Once these two images have been merged together, the

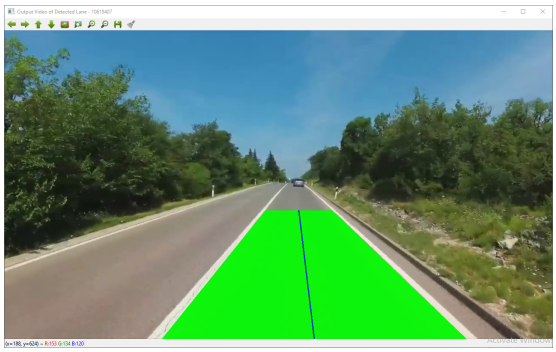


Fig. 20: Lane Identification Output

The video showing the lane tracking algorithm working can be seen here.

test vid

C. Further Improvements

One method of improving the lane detection would e to look for certain colours within the frames, and mark these as the road lines. This would be possible by utilising a similar technique used in a previous task (Task 2: Colour Tracker). This could help in eliminating false positives from the results such as a curb on the side of the road that may be close to the markings on the road. The biggest issue with using this method is variation in road marking colours. In the UK there are three standardised colours for road markings, white, yellow, and red [15]. These three colours do not take into account various other factors such as fading of the paint on the road markings and even adverse lighting. This could be overcome by using the HSV colour space looking for specific hues. This could allow just the sides of the road to be picked up and for the lines caused by the sides of the road to be ignored.

Another way of improving the lane detection would be to use differing form of Hough Lines, such as Probabilistic Hough Lines (`HoughLinesP`). Probabilistic Hough Lines finds segments in a line. Unlike a standard Hough Transform it is able to find segments of a line and be able to track non-straight lines, but it is more inaccurate than the standard Hough Transform [16]. The advantage of using standard hough lines would be increased resiliency but would require higher computational and storage requirements over the probabilistic hough transform [17].

Fig 21 and fig 22 both show a new clip being tested. It can be seen that fig 21 cannot detect the lane correctly, whereas the lane has been detected in fig 22. This could be caused by a large number of reasons. The centre line on the road has a different colour for the lane. Using the techniques previously discussed this video could have had the same results as with the previous test image.

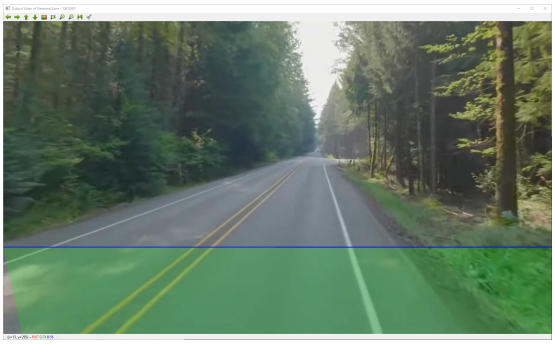


Fig. 21: Secondary Test Clip: Lane not detected

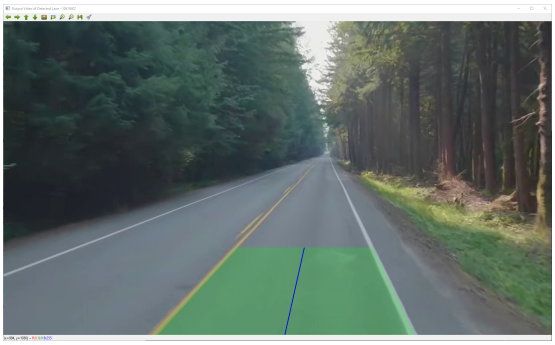


Fig. 22: Secondary Test Clip: Lane detected

D. Conclusion

The solution for 5 works fairly well within the confines of the test footage supplied. The road is detected and the detection is fairly stable, if a bit jittery in places. This could have been caused by the issues stated above or by other unforeseen issues. When shown other, un-tested footage, the lane detector struggled to detect and track the lanes for any extended periods of time. Given further development time and methods, it could be possible to make this lane tracking algorithm more generic and to work on a larger set of roads.

REFERENCES

[1] A. Y. Eser. (2020, Sep) The depth 1: Stereo calibration and rectification. [Online]. Available: https://python.plainenglish.io/the-depth-i-stereo-calibration-and-rectification-24da7b0fb1e0

[2] *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

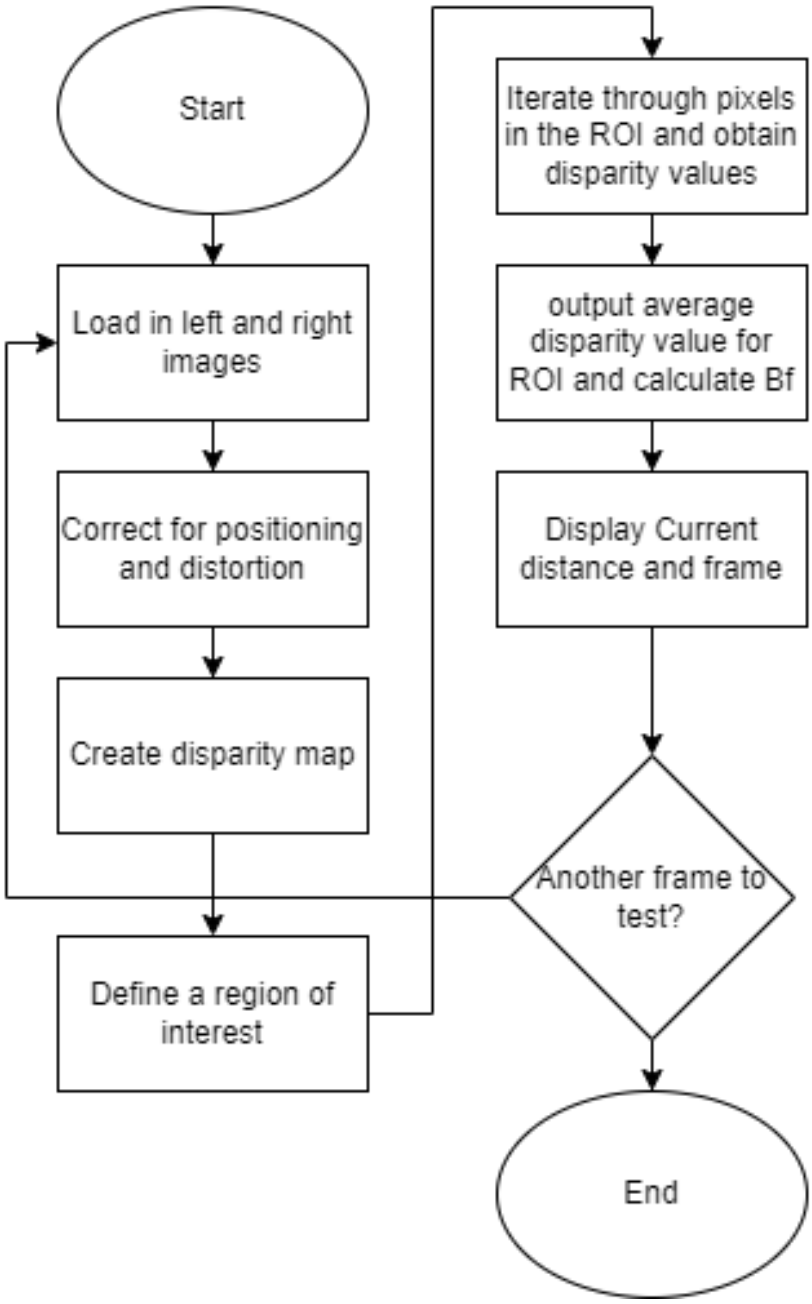
[3] J.-y. Bouguet and P. Perona, "Camera calibration from points and lines in dual-space geometry," 10 1998.

[4] OpenCV. (2022, Apr) Remapping. [Online]. Available: https://docs.opencv.org/3.4/d1/da0/tutorial_remap.html
[5] C. McCormick. (2014, Jan) Stereo vision tutorial - part 1. [Online]. Available: http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/
[6] S. Malik and T. Kumar, "Comparative analysis of edge detection between gray scale and color image," *Communications on Applied Electronics*, vol. 5, pp. 38–43, 05 2016.
[7] K. Aishwarya, A. S., and V. Singh, "A comparative study of edge detection in noisy images using bm3d filter," *International Journal of Engineering Research and*, vol. V5, 09 2016.
[8] J. F. Canny, "Canny edge detection," 2009.
[9] J. Hosang, R. Benenson, and B. Schiele, "Learning non-maximum suppression," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6469–6477.
[10] M. Sornam, M. S. Kavitha, and M. Nivetha, "Hysteresis thresholding based edge detectors for inscriptional image enhancement," in *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, 2016, pp. 1–4.
[11] S. T. Karri. (2019, Sep) Hough transform. [Online]. Available: https://medium.com/@st1739/hough-transform-287b2dac0c70
[12] T. T. Nguyen, X. D. Pham, and J. W. Jeon, "An improvement of the standard hough transform to detect line segments," in *2008 IEEE International Conference on Industrial Technology*, 2008, pp. 1–6.
[13] J. Scheffel and K. Lindvall, "Temporal smoothing - a step forward for time-spectral methods," *Computer Physics Communications*, vol. 270, p. 108173, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001046552100285X
[14] R. Bhandari, A. U. Nambi, V. N. Padmanabhan, and B. Raman, "Deeplane: camera-assisted gps for driving lane detection," in *Proceedings of the 5th Conference on Systems for Built Environments*, 2018, pp. 73–82.
[15] DVLA. (2022) Signs and signals:road markings. [Online]. Available: https://www.highwaycodeuk.co.uk/road-markings.html
[16] H. Kälviäinen, P. Hirvonen, L. Xu, and E. Oja, *Comparisons of probabilistic and non-probabilistic hough transforms*, 04 2006, pp. 350–360.
[17] T. T. Nguyen, X. D. Pham, and J. W. Jeon, "An improvement of the standard hough transform to detect line segments," in *2008 IEEE International Conference on Industrial Technology*, 2008, pp. 1–6.

The code can be found on GitHub here!

APPENDIX

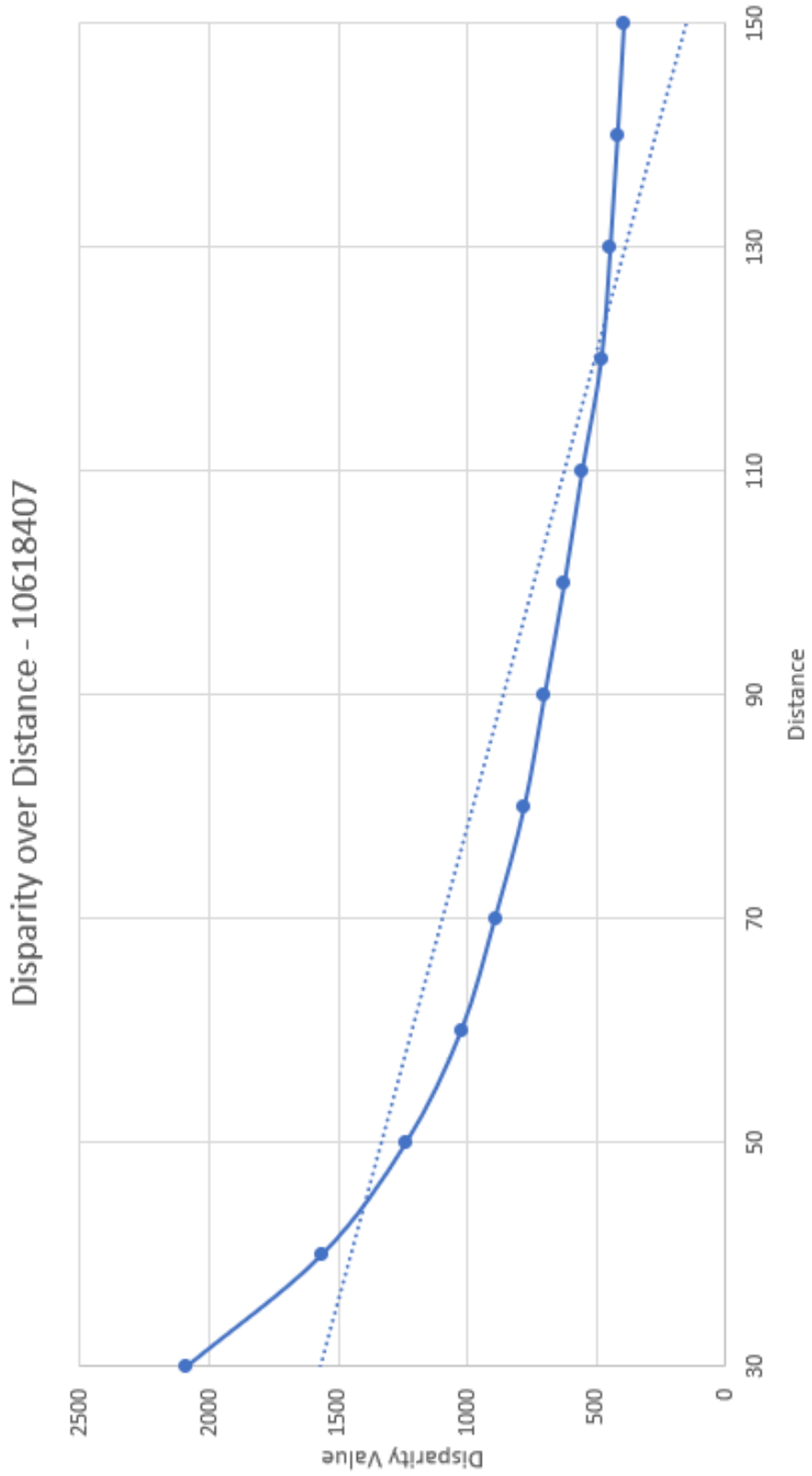
A. Flowchart for Task 4



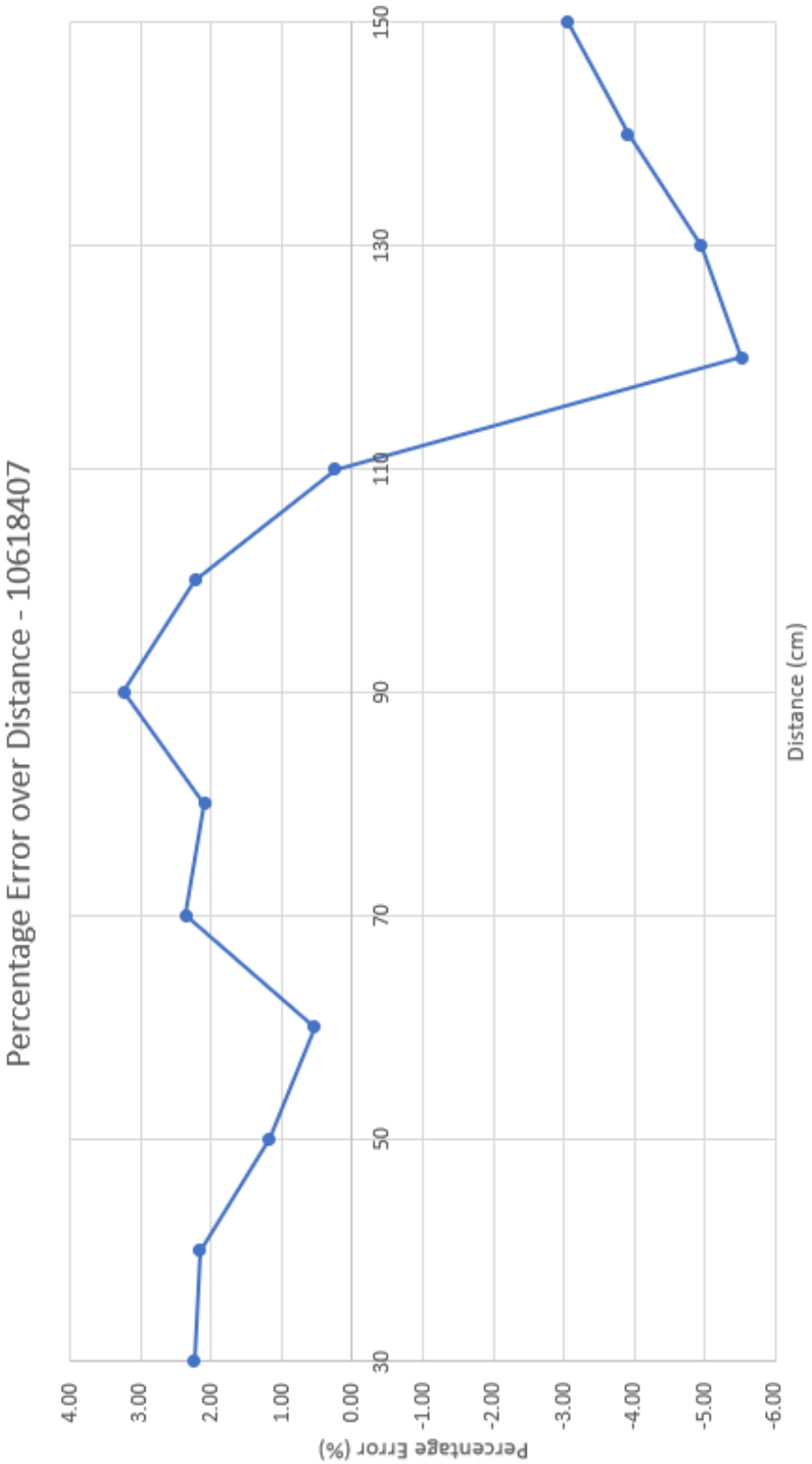
B. Full Disparity Matching Code

```
            //Load images from file
            Mat Left_Unknown =imread("../Task4/Unknown Targets/left" +to_string(ImageNum)+".jpg");
            Mat Right_Unknown =imread("../Task4/Unknown Targets/right"+to_string(ImageNum)+".jpg");
            cout<<"Loaded image: "<<ImageNum<<endl;
            //Distort image to correct for lens/positional distortion
            remap(Left_Unknown, Left_Unknown, map11, map12, INTER_LINEAR);
            remap(Right_Unknown, Right_Unknown, map21, map22, INTER_LINEAR);

            //Match left and right images to create disparity image
            Mat disp16bit, disp8bit;
            // compute 16-bit greyscalse image with the stereo block matcher
            sgbm->compute(Left_Unknown, Right_Unknown, disp16bit);
            // Convert disparity map to an 8-bit greyscale image so it can be displayed
            // (Only for imshow, do not use for disparity calculations)
            disp16bit.convertTo(disp8bit, CV_8U, 255/(numberOfDisparities*16.));

            // ==================================Your code goes here================================

            double OutputValue = 0;

            int x = 355;
            int y = 210;
            int rectangleWidth = 50;
            int rectangleHeight = 50;

            Rect rect(x, y, rectangleWidth, rectangleHeight);
            rectangle(disp8bit, rect, Scalar(255, 0, 0));

            // runs through all the rows in the image
            for (int i = x; i < x + rectangleWidth; i++) {
                // runs through all the columns in the image
                for (int j = y; j < y + rectangleHeight; j++) {
                    // stores the RBG values in the PixelValue vector
                    int PixelValue = (int)disp16bit.at<ushort>(j,i);
                    if (PixelValue < 65000){
                        OutputValue += PixelValue;
                    }
                }
            }

            OutputValue = (OutputValue/(rectangleHeight * rectangleWidth));

            cout << "Output Value : " << OutputValue << endl;

            double BF = 61280;
            double calcDistance = BF / OutputValue;

            cout << "Calculated Distance : " << calcDistance << endl;

            while(waitKey(10)!='x')
            {
                imshow("left", Left_Unknown);
                imshow("right", Right_Unknown);
                imshow("disparity", disp8bit);
            }
            ImageNum++;
            if(ImageNum>7)
            {
                break;
            }
        }
    }

    DataFile.close(); //close output file

    return 0;
}
```

C. Full Size Graph of Disparity over Distance



D. Full Size Graph of Error over Distance



E. Flowchart for Task 5

F. Full Lane Detection Code

G.