# Introduction

This report consists of two tasks delving further into potential computer vision scenarios. The tasks all use OpenCV with C++ and use predefined datasets. To extend the analysis, additional datasets have been created where required and these are clearly identified. The two tasks are based on three key computer vision elements, disparity mapping, image filtering and object detection techniques.

Contents

[Introduction 1](#_Toc135871249)

[Task 4 – Disparity Mapping 2](#_Toc135871250)

[Task 5 – Self-Driving Car Lane 11](#_Toc135871251)

[Video Links 24](#_Toc135871252)

[References 24](#_Toc135871253)

# Task 4 – Disparity Mapping

**Context Statement: A robot is to be designed to navigate an environment, and thus must detect its surroundings. Rather than using LiDAR or ultrasonic sensors, the designer has opted to use stereo cameras. Calibrate the cameras and use disparity mapping to build a 3D representation of the room.**

#### Background Understanding

A key part of human understanding of the world relies upon our stereo vision. This gives us the ability to understand the world in 3D space and automatically complete tasks such as estimating distance. Computers typically have a monocular system which limits computer vision techniques to work in 2D space. This however can be overcome by combining multiple monocular images together and so that we can extract 3D information.

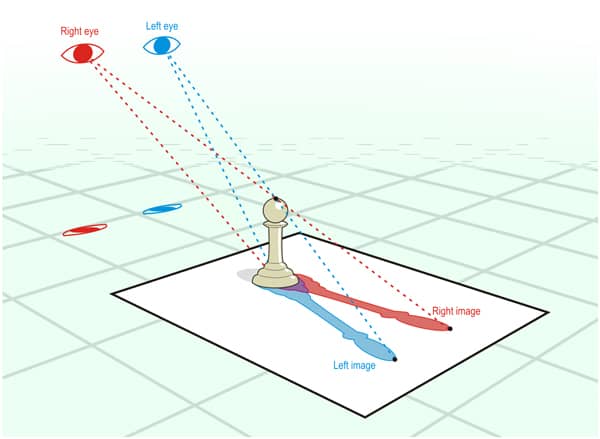


Figure 1 - Stereo vision in humans [1]

Disparity can be described as the apparent motion of objects between a pair of stereo images. This is the same effect as closing one eye and then another and some objects appearing to move. As can be seen in Figure 2, the two cameras are only a small distance apart however the object appears to have moved a lot in the resulting images. Closer objects will appear to move significantly while objects further away will move very little. These changes, disparity, between the images can be used to estimate how far away an object is.

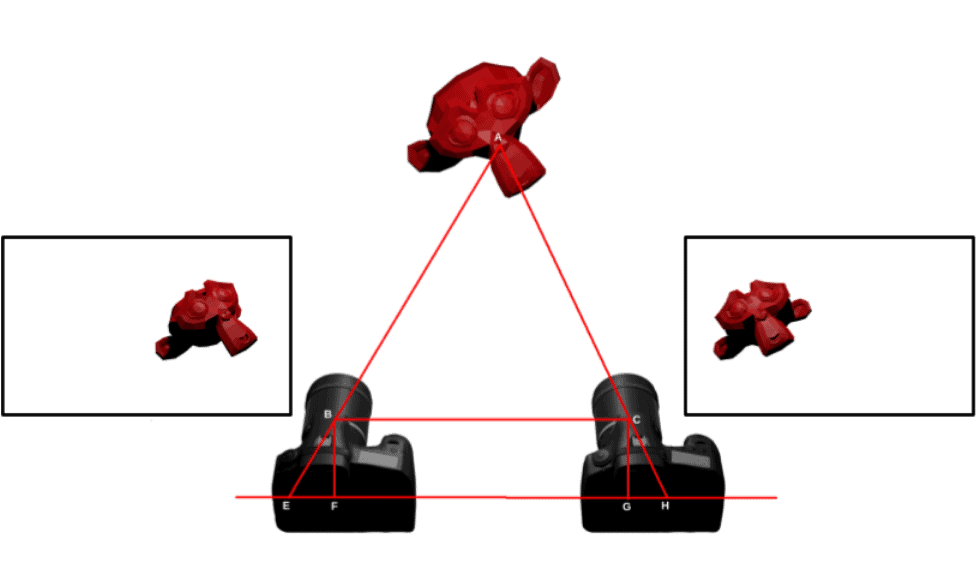


Figure 2 - Disparity "Movement" [2]

#### Method

The first step is to gather key information about the camera setup. The steps for this have been provided and is completed by running the StereoCalibration project supplied, reading the created intrinsics.txt and extrinsics.txt files and using this information within the calibration and stereoSGBM functions. This code was provided however a few variables have been renamed and it has been wrapped into a function for ease of use.

Text

Description automatically generated

Figure 3 - calibration() function creates 4 matrices

Graphical user interface, text, application

Description automatically generated

Figure 4 - stereoSGBM() function defines the block matcher which creates the disparity map

Disparity can be estimated using several algorithms, typically classified as either local or global methods. Local algorithms evaluate one pixel at a time and only consider the pixels which neighbour the target whilst global methods check across the entire image. Local methods are poor at detecting large depth changes and so usually global matching methods are utilised however these are extremely computationally expensive.

Semi-global matching is an in-between method that uses information from nearby pixels in multiple directions. Instead of using the entire image, the disparity of a pixel is calculated by considering a smaller area of the image. This block size is specified in the stereoSGBM initialisation alongside other variables. A StereoSGBM object can be used to create disparity maps from pairs of known images with the same control parameters each time.

A screenshot of a computer program

Description automatically generated with medium confidence

Figure 5 - Function to create a disparity image from two stereo images.

Following setup, a disparity map is created from each pair of known images. This is done using the SGBM’s compute() member function which inputs a pair of 8-bit stereo images and produces a 16-bit fixed-point disparity map (where each disparity value 4 fractional bits). The code for this is shown below in Figure 5 which returns the disparity image object. This image is too complex to display using imshow() and so this map is converted to an 8-bit image for display and visualisation purposes. Areas which do not produce a disparity value are shown in black. This is usually because there is such a small difference between the two images that it isn’t registered. In the example below the computer screen and the surrounding objects are ignored as they do not produce a value. This creates the black panel on the left side of the disparity image.

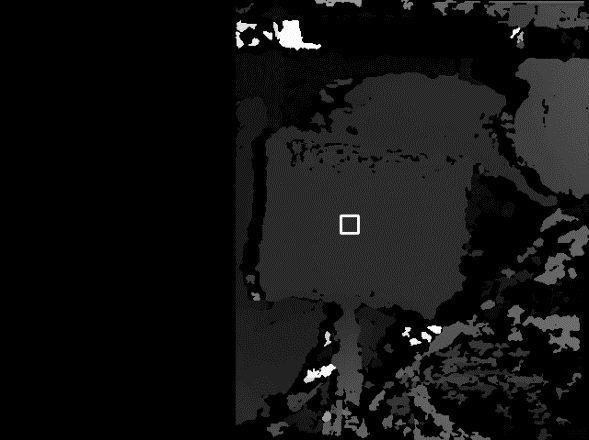
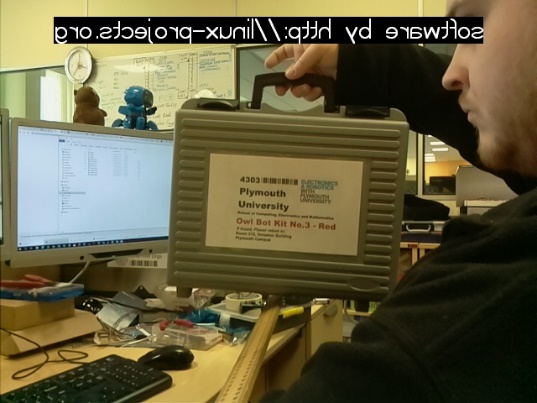


Figure 6 - A stereo pair of images (70cm) and the created disparity map

[Eq 1]

The disparity equation [Eq 1] shows that disparity and distance are inversely proportional with the proportional constant *baseline × focal length*. To calculate unknown distances, it is essential to first calculate this constant from the disparity maps for the known distance images. This value will change depending on the stereo camera setup however all images in the dataset have been taken with the same set up so it is reasonable to consider that the value of BF will be constant throughout the experiment.

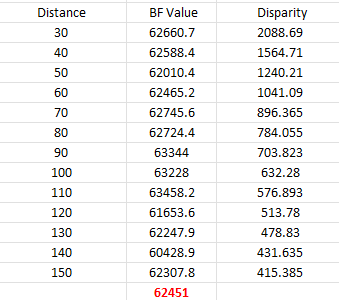


Table 1 - Calculating BF from distance and disparity values.

Figure 8 visualises the relationship between these elements to show this proportional relationship. In this solution a function, calcBF, is created to calculate this value. This function loops through each known image pair, creates a 16-bit disparity image, reads a pixel value from the target location, and calculates BF for the image pair. The calculated value is then sent to a csv file to be graphed as well as added to an array within the programme. Once all the known pairs have been analysed, the average value of the array is taken, and this is the constant that is used throughout the rest of the testing.

As well as performing the calculation, this function also sends the results to a .csv file and displays each image as it steps through them and overlays a rectangle to show the range of interest that is being used. This makes it simple to visualise any adjustments that need to be made. The code for this function shown in Figure 9.

From the .csv file we can plot the disparity values against the known distances and check that they follow the trend we expect to see.

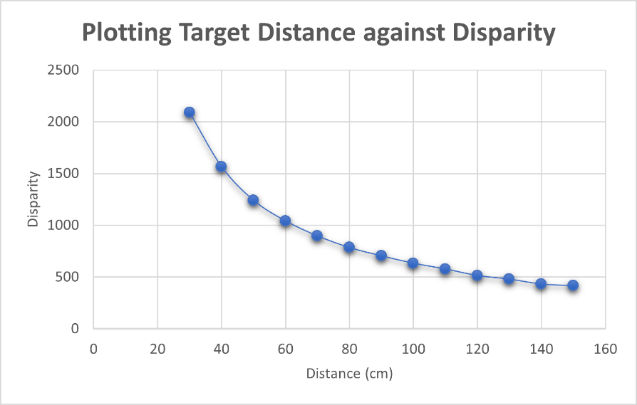


Figure 7 - Known Distances and Disparity results

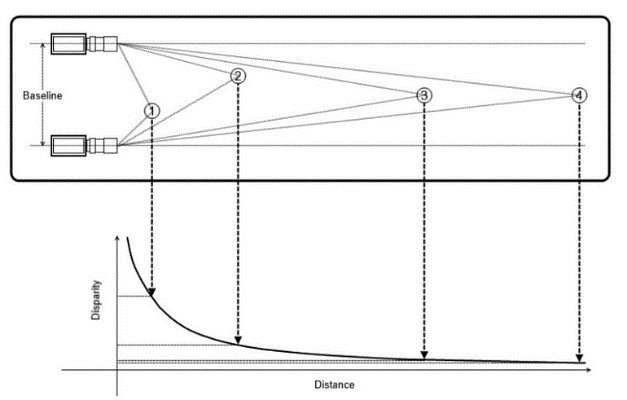


Figure 8 - Disparity and Distance [3]

Figure 8 shows the inverse relationship between disparity and distance and this pattern is mirrored in the graph created from the csv results. This shows that the main design of the solution is correct and the average BF calculation is working although it might still need improvement to be accurate.

A screenshot of a computer code

Description automatically generated with low confidence

Figure 9 - Function to calculate the average BF value.

Now that an average BF value has been created, it is possible to estimate the distance to the unknown target. The unknown targets also have a pair of stereo images for each distance and the technique to find the distance is similar to the technique to find the average BF value as it uses the disparity equation once again.

Using the disparity equation again, a distance map is created. By stepping through each pixel and reading the disparity value from the image, the distance value can be calculated using the average BF. This new value is then written to the new distance image.

This distance image will have values ranging from 0 to 255 and the lighter the area the closer it will be to the camera. Once the image is created, the distance to the target area can be determined by again taking the average value from the range of interest.

A black and white image of a square

Description automatically generated with low confidence

Figure 10 - Distance map of a known image to check accuracy.

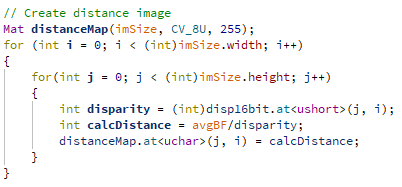


Figure 11 - Create distance map.

The code to create this distance map is shown in Figure 11. Following the creation of the distance map, the estimated distance can be calculated by once again taking the mean value from a range of interest on the distance map. The estimated distances are then printed to the terminal and sent to the csv so analysis can be performed. This data is shown below in Table 2.

Table 2 - Estimated Distance

A picture containing text, screenshot, number, font

Description automatically generated

To check the results fit within the expected inversely proportional relationship between disparity and distance they can also be plotted onto the disparity distance graph created earlier. The results can be seen in Figure 12.

A graph with blue and orange dots

Description automatically generated with low confidence

Figure 12 - Disparity distance graph with unknown image pair results

From these results we can see that our method is correct, the estimated distances perfectly fit the graph that was created earlier however it is not possible to ascertain the accuracy of the results as there is no way to compare to a ground truth. To solve this issue, the same method can be used to estimate the distance to the known targets and the discrepancy between the estimated value and the real value can be used to identify the amount of error.

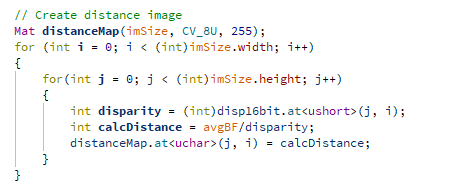
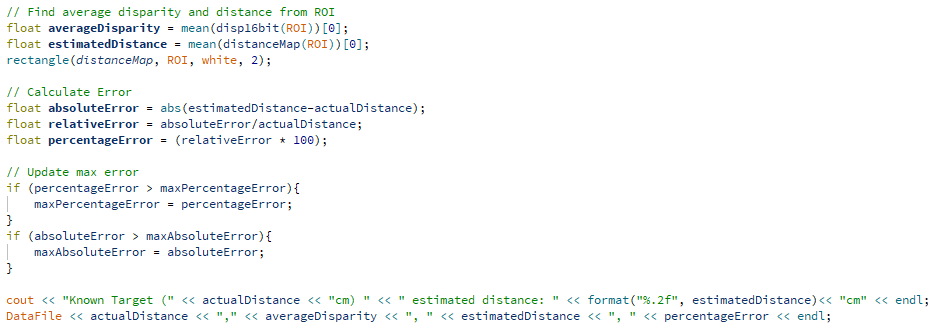


Figure 13 - estimate the distances.

#### Evaluation

To evaluate the accuracy of this solution, distance maps must be created for all 13 known distance image pairs. This involves once again making disparity maps from the pairs and using the same method as in Figure 11 once again. Once a distance map is made, absolute and percentage errors can be easily found and where necessary changes can be made and checked.

Figure 14 – Calculating error values.



In order to identify the maximum error across all the results, a global variable is created that stores the maximum percentage and absolute error values. When all the known pairs have been analysed, these values are printed to the terminal for quick access. All errors and reading values are also exported to the csv file.

The results from this are shown in Table 3 with the results with the most error highlighted in red. Percentage error can be useful to identify where issues might arise but due to the large difference in the known distances, the smaller distances error is skewed unfavourably. In fact, the worst result for percentage error is the 30cm reading at 3%, however this is still an absolute error below 1cm.

The absolute error is far more helpful to find the less successful results however even the worst result has an error of less than 5cm. To try and reduce this error, a few different techniques were attempted.

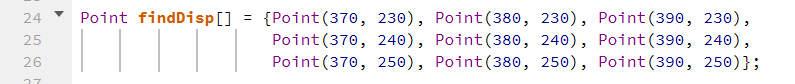
A picture containing text, screenshot, line, plot

Description automatically generatedTable 3 - Known Distance Error Calculation

A picture containing text, screenshot, number, font

Description automatically generated

The first method was to adjust the region of interest that all the averages are taken from. Originally this range was nine randomly selected points that were always found on the targets so an alternative solution was to use a Rect object instead which would mean every pixel within a region was analysed. This second method gave slightly improved results (absolute error at 14cm was 3.95cm) however specifically choosing points that gave the best results could cause larger errors if the dataset was expanded another time and so the method using a Rect was continued.





An alternative solution to picking the best results is to create a solution that requires the user to select the target region that will be averaged. This allows the user to test multiple areas or choose the size of the region that is tested, and both will change the results. Although I did not complete as much testing as I hoped with this, in the tests that I did complete the absolute error on the 140cm image pair did not improve further than 3.95cm which led to the conclusion that it is possible the image was mislabelled, or the photo was taken, and the target was not accurately positioned. As this task requires a dataset with the exact same camera setup it was not possible to extend this dataset, but this would be the most sensible next step to test and develop this solution.

A picture containing plot, text, line, screenshot

Description automatically generated

Figure 15 - Comparing the real target distance and the estimated value.

#### Conclusion

In this task, it has been demonstrated that computer vision can be used to calculate disparity within images and then estimate distance from new images using OpenCV. The results show that the chosen method was successful and accurate however there are still areas that require improvement. One improvement would be using more images in the known image dataset and including repeated distances. This would reduce the likelihood of having images that have the wrong distance labelled. Overall, the solution presented successfully and succinctly solved the task at hand.

The final estimated distances are shown in Figure 16 and I am confident that they are accurate to within 3cm of their real distance.

A picture containing text, line, diagram, plot

Description automatically generated

Figure 16 - All unknown image pairs and their estimated distance.

# Task 5 – Self-Driving Car Lane

**Context Statement: One of the most complex computer vision applications is in the growing field of autonomous driving cars. Cameras around the car capture a great deal of information, however breaking this down into useful data is a huge challenge. In this task, you are required to program a small part of this system, lane detection.**

Lane detection is a critical component of self-driving cars and demonstrates a variety of computer vision techniques which can be used in multiple contexts. For this task, as there are multiple steps to the process for each frame, a processing pipeline has been developed to accurately determine where the lane is. Three main sections can be found within the code, loading the video, performing pre-processing techniques and then identifying the lane markings and displaying the identified lane on the original frame. A video of the full solution can be found in Appendix 1, and specific sections of code have also been highlighted in the report below.



Figure 17 - Original Frame

To test and develop the processing pipeline for this task, originally it was tested on individual frames however to test on the whole video, the frames are loaded one at a time in a continuous while loop.

A picture containing text, screenshot, font, line

Description automatically generated

The first step was to apply a variety of pre-processing methods in order to identify and highlight the lane markings. As the lane markings in the provided video were white, this was the colour that was targeted initially. To do this, the image was converted to grayscale, this simplifies the image and make it easier to detect edges later. A gaussian blur is then applied to the image which removes noise from the image.



Figure 18 - Grayscale and Gaussian Blur

A picture containing text, font, screenshot, line

Description automatically generatedNext, a binary threshold is applied to the image, this creates a binary image in which each pixel only contains either the value 0 (black) or 255 (white). The grayscale image from Figure 18 is analysed and any pixels which contain values over the specified threshold are converted to white and pixels with values under the threshold are made black. The original threshold used was 230 as the lane lines are typically extremely bright however after a few tests, this was reduced to 190 in order to pick up the lines when they are in shadow.



Figure 19 - Binary Threshold

As can be seen in Figure 19, the lines have been detected successfully using this threshold technique however there are other areas that have also been detected that are not wanted. These typically come from the sky or reflections from other cars or debris along the side of the road. In order to remove these, a range of interest needs to be identified and a mask created. Throughout the process of improving the code a few different ROIs were used however the one shown in Figure 20 worked the most effectively, removing most of the noise whilst keeping the most lane information.



Figure 20 - Range of Interest

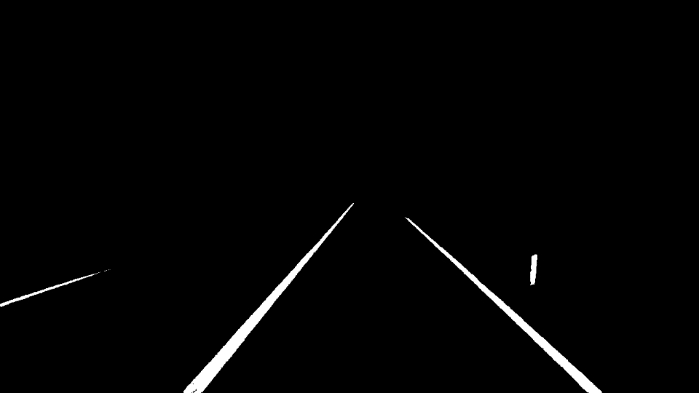


Figure 21 - Bitwise Operation

This mask is then combined with the binary image created earlier by implementing a bitwise AND operation which keeps only those pixels that have the value 255 in both images. This results in a new binary image which is less cluttered and only includes lane lines and is shown in Figure 21.

A picture containing text, font, screenshot

Description automatically generated

Figure 22 - Creating a masked ROI and bitwise AND operation.

In order to make sure the binary image created has as few flaws as possible, morphological operations [3] are applied. Morphological operations are used in image processing to remove noise and separate or connect objects together. Erosion is a method that removes pixels on object boundaries whilst dilation adds pixels to object boundaries. The number of pixels added and removed depends on the size of the structuring element. An example of both operations courtesy of OpenCV online tutorials [4] can be seen below in Figure 23 - Dilation, Original, ErosionFigure 23. The structuring element chosen for this task was chosen by trial and error to get the best results and used the MORPH\_RECT pre-set structure.



A picture containing white, black and white, design, calligraphy

Description automatically generated with medium confidence

Figure 23 - Dilation, Original, Erosion

The aim for the morphological operations was to further reduce the noisy elements in the binary image by eroding the smallest parts of the image and then dilating the image again to add thickness back to the road lines. As seen below in Figure 24, this worked and the smaller areas were successfully removed (shown in the blue squares) however the side effect of this is that any small defects in the original mask are enhanced and it can cause gaps in the road markings, as shown by the red circled areas.

A screenshot of a computer

Description automatically generated

Figure 24 - After morphological operations (left) and before (right)

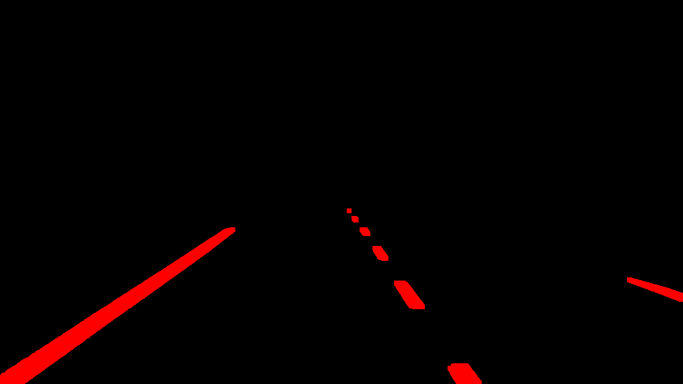
The third filtering step used the contours of the binary image to remove small areas from the image. Originally tested as an alternative to the morphological methods, the results showed that the two methods removed different issue areas and so it made the most sent to keep them both. The first step was to find all the remaining contours in the binary image. This function is available from the cv2 library and returns information about each contour that can be stepped through for analysis. The function drawContours() is also supplied by the same library and was used to show which contours had been found. Figure 25 shows the code in use and Figure 26 shows the way the smaller areas are removed.

A picture containing text, screenshot, font

Description automatically generated

Figure 25 - Applying the function findContours() to the binary image.

As can be seen above, in the final code the contours aren’t drawn in red and green but are instead only the white ones are drawn onto a Mat called disp\_contours which is once again applied to a bitwise AND operation with the comboMask image (the original threshold and ROI combination binary image) to create a final line\_mask object.

 A picture containing light, colorfulness, darkness, laser

Description automatically generated

Figure 26 - All contours displayed in red (left) and large contours in green (right).  
Showing results from video 2 as it demonstrates more clearly.

Once the final line mask has been created, the pre-processing is finished, and edge detection can take place. For this task, Canny edge detection has been used however other edge detection methods are available such as Sobel edge detection. Edge detection works by identifying the points in an image where the intensity changes sharply. As the input image is a binary image in this case, Canny works extremely well and creates clear edges around the lane lines. As the canny function automatically applies a 5x5 gaussian filter, an extra filter hasn’t been applied.

A picture containing black, darkness, black and white, screenshot

Description automatically generated

Figure 27 - Canny edge detection results

The final processing steps are based around Hough transforms. This a technique that extracts features from images and the most common use of it is for the detection of regular curves such as lines, circles and ellipses [5]. In this case, it is being used to detect the lines from the canny edge detect results.

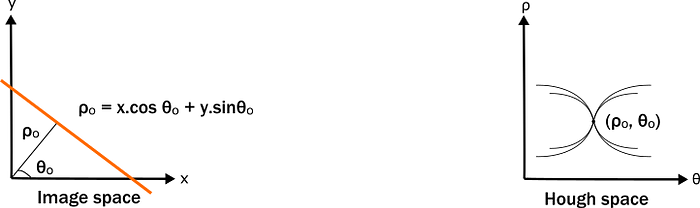
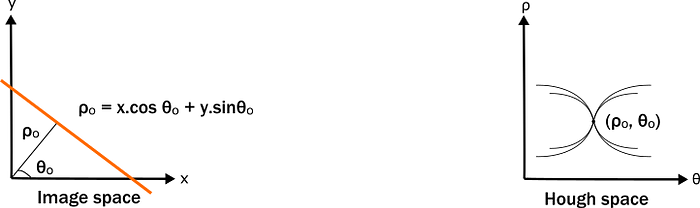


Figure 28 - An example of Image vs Parameter space. [5]

The Hough transform maps points from the image space into parameter space. The equation of a straight line in image space is in the form where is the gradient and is the y-intercept of the line. This line in parameter space will be transformed to a point of the form (M, C). If the line is represented in polar coordinates instead of cartesian coordinates, we can see that the line is represented by Theta and Rho instead. This line will now be transformed to a point in the parameter space. The results of the HoughLines() function is a container of Vectors in the form Vec2f, the first element of this represents Rho and the second element represents Theta. In order to detect only the wanted lines, there are a few variables that needed to be set using trial and error. These variables and their descriptions can be found in the comments at the top of Figure 29.

A screenshot of a computer code

Description automatically generated with low confidence

Figure 29 - Setting HoughLines() parameters

We can, of course transform these values back into cartesian form in order to re-draw the detected lines onto the video. When additional videos were analysed it became obvious that there was one major flaw in the classical Hough algorithm which is that it struggles to detect short lines. To overcome this problem, the images were analysed a second time with a method HoughLinesP() which uses the probabilistic version of Hough Transform to find lines. The probabilistic variety of the transform randomly samples the image to find lines and curves, it requires more parameters such as minimum line length and maximum gaps between lines to work but it returns finite lines including shorter lines and dashes.

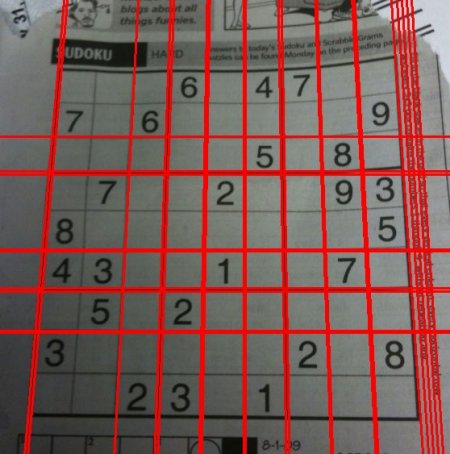
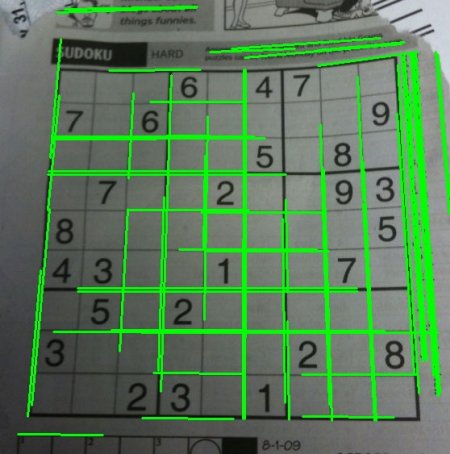


Figure 30 - Classic vs Probabilistic [6]

Once lines had been detected using both algorithms, they were filtered by their angle so that all lines that were horizontal were removed and any that were too vertical as these won’t be lane lines unless the car is changing lane.

A picture containing laser, line, light, colorfulness

Description automatically generated

Figure 31 - Filtering Hough Lines

Using the provided function LineRT(), a new function was developed called getLineCoorinates() which returned the two endpoints of a found HoughLine.

A screenshot of a computer code

Description automatically generated with medium confidence

Figure 32 - getLineCoordinates()

The main change from LineRT() was the removal of the automatic drawing function and an additional option to clip the lines. This is because lines transformed into image space are close to infinite and therefore return endpoints which cannot usually be displayed or adapted easily. To simplify the next steps the clipArea is set to clip the lines at the same height as the top of the ROI defined earlier. This is a global variable that is passed into the equation so that if an alternative clipping region is required it can be specified at time of use.



Using these endpoints, it was possible to classify the lines into groups based on their gradient. This happens for both group of lines, classical and probabilistic, and then the average gradient of each is found. For the classical lines it was simple to add up each negative and positive gradient into two variables and then find an average value.

A picture containing text, screenshot, font

Description automatically generated

Figure 33 - Finding the average gradient of left and right lines in classical Hough

Probabilistic Hough was slightly more complicated as the lines are all different lengths. The assumption is made that longer lines are more likely to be useful and so a weighting system was implemented.

HoughLinesP returns a container of lines in a different format to HoughLines, using a Vec4f instead of a Vec2f and the values returned are the end points of each line. This means the getLineCoordinates function is not required for this version. However, the gradient filtering uses the same method and the maximum and minimum values are the same. Instead of the gradient being summed in the for-loop, appropriate lines are added into two new Vec4f containers housing the left and right lines.

A picture containing text, screenshot, font

Description automatically generated

Once all the lines have been grouped by their gradient, the average is calculated in a new function called avgLine(). First the new lines are once again separated and the gradients, intercepts and distances of each line are stored in new containers to perform the averaging calculation.

A screen shot of a computer code

Description automatically generated with low confidence

A picture containing laser, light, colorfulness

Description automatically generated

Figure 34 - Classical Lines (green) and Probabilistic Lines (red) after gradient filtering

Once this is done, the distances are used to calculate the weight for each line. This is done by comparing the length of a line with the total length of all the lines together. Using dot product multiplication, the average gradient and intercept can be calculated, and end points of this average line can be found. This line is then returned as two points.

A screen shot of a computer code

Description automatically generated with low confidence

Figure 35 – calculating and using the weights

Finally, to display the original frame and the detected lane together, a mask of the internal area between each averaged line was created and then this mask was overlayed onto the original frame. This required a for-loop to complete but meant that instead of completely replacing the pixels within the region, the background is still visible which makes for a less jarring viewing experience.

A picture containing text, screenshot, font, number

Description automatically generated

Figure 36 - Overlaying the area of the lane.

Once this is all complete, the averaged probabilistic lines are the final lines that get drawn onto the original frame to show the lane detection. The colour of the region and the lines that get displayed depends on the comparison between the two averaged Hough transforms. This is mostly just checking that the gradient has successfully been calculated for all lines and that the two methods produced results that are similar to each other.

A picture containing outdoor, road, scene, way

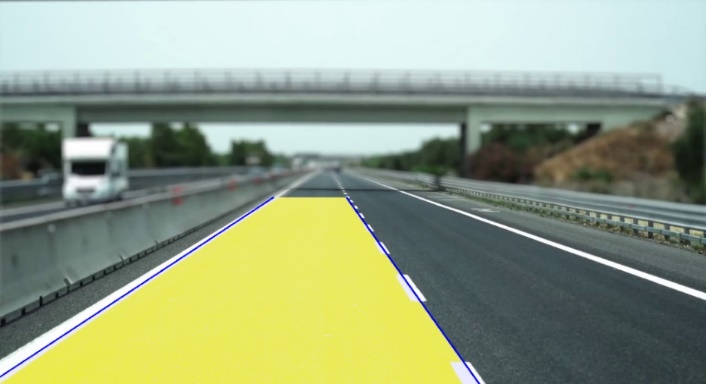
Description automatically generated

Figure 37 - Displaying the detected lane.

#### Evaluation

Initially the evaluation relied on checking if two lines had been detected or not each frame, this gave very successful results with the predefined dataset giving a 100% accuracy result. It quickly became obvious however that there were multiple false positive results when a line was detected but wasn’t a lane line.

In order to improve the evaluation three variables, good\_lines, bad\_lines and no\_lines, were created which were iteratively increased each frame depending on the results of comparison between the Hough Transform results. The main premise depended on the average gradients and making sure that the average from the classical and probabilistic transforms were within a small amount from each other. Another simple check was confirming that all the averaged lines provided actual answers. In Figure 38 for example, image B shows a red line, and the lane has not been highlighted, this visualises that only one of the lines has been identified. In image C, the lane is highlighted yellow which means both lanes have been identified but the gradients from the classical Hough and the probabilistic Hough transforms have gradients that do not match.



A

B

C

Figure 38 - Example evaluation

A picture containing text, screenshot, font, number

Description automatically generated

Figure 39 - Code for the simple evaluation methods

By printing the results of this method of evaluation to the terminal, there was an easy way to check at a glance when changing small parameters if there was an improvement in results.

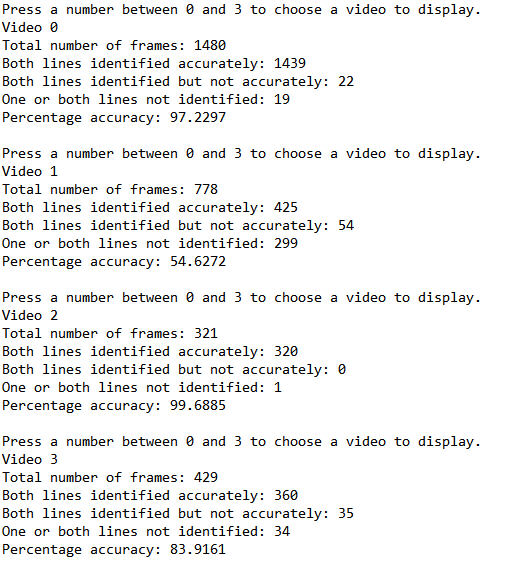


Figure 40 - Evaluation Results

In order to test the solution in more situations, three additional stock webcam videos were downloaded and tested. These introduced additional elements such as shadows, multiple lanes and dotted middle lines, and cracked asphalt which brought more challenges to the task. The introduction of these videos was also the reason for using the probabilistic Hough transform as well as the classical transform in order to detect the dotted lines more effectively. As the results to the right in Figure 40 show, the most accurate video is DashCam2 with only one frame where one or both lines are not detected.

One difficulty by adapting the solution to work for multiple videos is that the accuracy of the original video was reduced however it is still high at 97% and so this seems like a fair adaption.



Figure 41 - All four test videos

The final issue with this method of evaluation is that there are still some false results. Examples of these can be seen below with a red line indicating the correct position of the lane. A good addition to the evaluation would be to design a way to find out how incorrect the lane line is. Images A and C have been evaluated as “not accurate” but they are far more extreme than most cases of this. Image B has been identified as “accurate” because both versions of the Hough Line algorithm have identified the same gradient however the vehicle is disrupting the view and both algorithms have identified the lane incorrectly which this evaluation doesn’t consider. Image B is also a good visual representation of the fact that the Y-intercept of a straight line is also essential to compare lines so another time this would be added to the evaluation method.

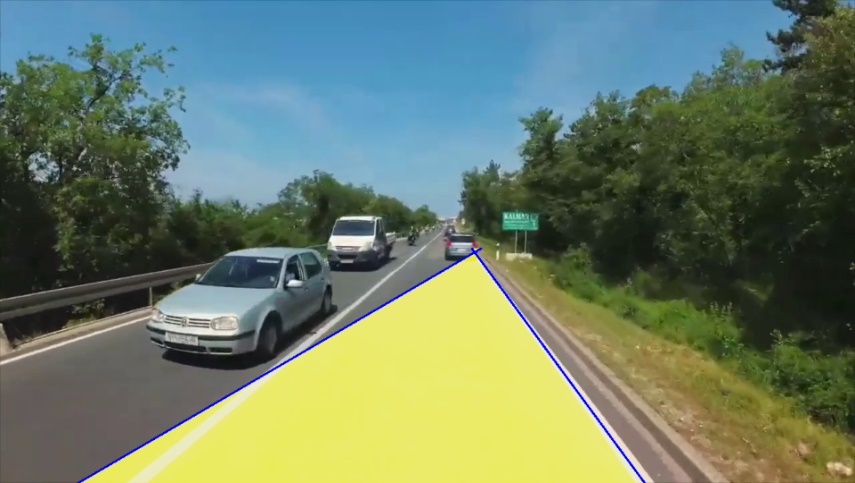


Figure 42 - Examples of incorrect evaluation

A

C

B

#### Conclusion

In this task, it has been demonstrated that computer vision can be used to detect lanes on a road using OpenCV. The code uses various image processing techniques such as Gaussian blur, Canny edge detection, Hough transform to detect lines and Morphological operations. The results show that this is a successful method for detection however there are still areas that require improvement. Some possible improvements include using the y-intercept as well as the gradient for Hough line comparison, improving the thresholding and colour processing to detect fewer extraneous areas in the initial pre-processing and improve the accuracy of the initial range of interest. Overall, the code presented in this task is successful and answers the requirements set out by the context statement and assignment tasks.

A picture containing screenshot, text

Description automatically generated

Figure 43 – Full processing pipeline for Task 5

# Video Links

Task 4 - <https://youtu.be/vMiFvvmukuY>   
Task 5 - <https://youtu.be/hxHJg9Q0BMg>

# References

|  |  |
| --- | --- |
| [1] | [Online]. Available: https://immersionvr.co.uk/blog/monoscopic-vs-stereoscopic-360-vr/. |
| [2] | [Online]. Available: https://learnopencv.com/disparity-estimation-using-deep-learning/. |
| [3] | “What is erosion and dilation in Image Processing?,” ITQAGURU.COM, [Online]. Available: https://itqaguru.com/what-is-erosion-and-dilation-in-image-processing/. |
| [4] | OpenCV, “OpenCV Tutorial, Erosion and Dilation,” [Online]. Available: https://docs.opencv.org/3.4/db/df6/tutorial\_erosion\_dilatation.html. |
| [5] | S. T. Karri, “Hough Transform,” Medium, [Online]. Available: https://medium.com/@st1739/hough-transform-287b2dac0c70. |
| [6] | OpenCV, “OpenCV Tutorials,” [Online]. Available: https://docs.opencv.org/4.x/d6/d10/tutorial\_py\_houghlines.html. |
| [7] | OpenCV, *Code Documentation,* 2015. |
| [8] | “Disparity Map in Stereo Vision,” Baeldung CS, [Online]. Available: https://www.baeldung.com/cs/disparity-map-stereo-vision#:~:text=The%20disparity%20is%20the%20apparent,away%20will%20move%20very%20little.. |
| [9] | A. Garg, “Stereo Vision: Depth Estimation between object and camera,” Medium, [Online]. Available: https://medium.com/analytics-vidhya/distance-estimation-cf2f2fd709d8. |