Developing Detection and Mapping of Roads within Various Forms of Media Using OpenCV

Jordan C. Lyle

University of Arkansas Fayetteville

Follow this and additional works at: https://scholarworks.uark.edu/csceuht

Part of the Graphics and Human Computer Interfaces Commons, and the Theory and Algorithms Commons

Citation


This Thesis is brought to you for free and open access by the Computer Science and Computer Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Computer Science and Computer Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu.
Developing Detection and Mapping of Roads within Various Forms of Media

Using OpenCV

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering

College of Engineering

University of Arkansas

Fayetteville, AR

by

Jordan Lyle
1. Abstract

OpenCV, and Computer Vision in general, has been a Computer Science topic that has interested me for a long time while completing my Bachelor’s degree at the University of Arkansas. As a result of this, I ended up choosing to utilize OpenCV in order to complete the task of detecting road-lines and mapping roads when given a wide variety of images. The purpose of my Honors research and this thesis is to detail the process of creating an algorithm to detect the road-lines such that the results are effective and instantaneous, as well as detail how Computer Vision can be applied to abstract ideas of detecting objects near the same level that a human would.

The detection and mapping of road-lines was executed via an algorithm that incorporated several different parameters and specifications, all of which went through many iterations before being somewhat optimized to the state it is currently in. Furthermore, the thresholding algorithm that I developed would allow and adapt for inputting many different images, as well as active video functionality. This algorithm manually reads through every pixel in the image/frames of a video and detects row patterns by preforming a multitude of custom mathematical calculations on the BGR color values of the image/frames, resulting in the filling of the bounds of the lines by row and thus detailing where the road is.

2. Introduction

There are many abstract topics in Computer Science, and therefore it was somewhat difficult trying to identify what topic I wanted to choose to work on an Honors project in. Prior to working on my Honors thesis, I became very interested in autonomous vehicles due to the
internship I had with J. B. Hunt. So, because of this internship, I decided that pursuing knowledge about Computer Vision would be useful to future career endeavors if I continued working at the company.

My Honor’s Thesis Advisor, Dr. John Gauch, significantly helped me research and decide what project I would like to pursue in relation to Computer Vision, ultimately guiding me in the direction of road line detection by analyzing dashcam footage. Overall, though, my main motivation towards pursuing this project was because of how many standard techniques of Computer Vision I would learn by doing so, since this project would require me to make and work with complex algorithms in order to apply math to art and images, in a way. Furthermore, this perspective is what drove much of my interest in the project, as being able to apply structure and do pixel analysis on images was a very new idea for me.

Without any experience in developing programs relating to Computer Vision, I decided that the first way I should tackle the project was to learn more about the Computer Vision library called OpenCV, which launched in 1999 as a part of Intel’s Research into CPU intensive applications and now has grown to over 18 million downloads worldwide [1]. Originally, I started developing the program in Python, but I eventually swapped it over to C++ after learning that this would improve the performance. Furthermore, most of what I did starting off was researching road detection work and algorithms that were already on the internet (which will be described in detail in the following section), as well as watching videos on how I could use the tools the OpenCV library contained. There were a few issues with setting up the Python environment and installing the OpenCV libraries into my IDE (Visual Studio 2019), but after those were resolved, I could start working with the library. My first step to become acquainted
with OpenCV was to start writing some basic code utilizing the libraries of OpenCV just to get a feel of how I would go about actually designing the project.

To begin the development process, I started by listening to the videos that gave some explanation of the OpenCV libraries, and then copied the tutorial code onto my own IDE so that I had the ability to change parts of the code and observe the results on my own. Many of the beginning operations I conducted were reading, writing, and showing the images in OpenCV, which are detailed by the “OpenCV Python Tutorial For Beginners” series, produced by ProgrammingKnowledge on Youtube. The OpenCV functions, imread, imwrite, and imshow in particular, ended up being very important for reading in the images/video and writing out the resulting thresholded images to my computer [2]. With this foundation, I could finally start implementing some of the ideas that I would learn from already existing projects.

3. Related Work

Road detection is a very well researched topic, so to begin I will go more in depth about some of the earlier examples of computer vision applications relating to road detection in some form. One of these examples involves a paper called “Road Traffic Sign Detection and Classification” published in 1997 by IEEE, which focused on identifying road signs by using a color thresholding and shape identification method. To start the color thresholding, the program would a piecewise function of the three RGB values, where the piecewise function uses the ratio between the intensity of the specified color and the sum of intensity of RGB to generate the threshold used against any particular image.
Then, once the image has been processed using the threshold, it calculates where the corners of the shape are based on what is left on the processed image. It achieves this by applying a mask to the image, and identifying where the edge of the filled values is to determine the corner. From there, the shape can be figured out by the angle of the corners, which will typically indicate the shape once connected. After this operation is completed, the signs are successfully shown [3].

Another research innovation, one that is more relevant to my project and is even older than the previous paper mentioned, is the self-driving car project at CMU, as detailed in the paper “Autonomous Land Vehicle Project at CMU” that was published in 1986 by ACM. CMU (Carnegie-Mellon University) developed a vehicle called the Terregator, which communicates to the host computer the visual data it is picking up with its sensors via a radio link. For the road-following functionality of the Terregator, the researchers created a road detection algorithm that uses image gradients to increase the intensity of possible road lines. Then, the possible road lines are tested by the algorithm to determine if they are properly far apart and are parallel to each other, in which case it can be assumed that the Terregator has identified the road it can travel on.

While this worked well for clear roads, rough roads and potholes created issues when the Terregator was driving at 1 kilometer per hour. Some of the future work they planned was to develop color and texture analysis that could be switched on when the Terregator starts to have some difficulties, however they eventually ended up deciding to give the Terregator the capability to navigate the campus using a pre-installed map, therefore reducing the amount of pathfinding needed. This location of the Terregator on the map was relayed back and forth to
the host computer, so the Terregator could process if it was on the road and its position on the road, allowing it to correct itself if it is deviating. This research, despite being done in 1986, was one of the first points of research on self-driving vehicles and remains similar to some of the methods that road detection is carried out today [4].

With the original focus of the project being analyzing dashcam footage, the further type of research that was conducted was to look into more recent papers that already existed and see what I could extract from them. The first paper that was encountered was “Egocentric Analysis of Dash-Cam Videos for Vehicle Forensics” published by IEEE, which was in relation to programs that have the ability to analyze dashcam footage for forensic purposes. While this was different from the main goal of my project, in that the paper was talking about a program to read motion blur to classify the vehicle, it mentions some relevant information about thresholding that is similar to the direction that my project went. This idea was of having feature vectors to determine which labels the video is defined by, with the feature vectors using blur to identify important areas in the video. My thresholding algorithm ended up preforming a similar operation, just using colors instead of blur, with the so called “feature vectors” being the comparison vector to threshold the color values and assign them with the color of black instead of a specific label [5].

A more interesting paper I found while conducting my research was “End to End Learning for Self-Driving Cars”, published by the NVIDIA Corporation in 2016. Instead of explicitly using an algorithm for detecting road outlines so that the autonomous vehicle will have guidance, a convolutional neural network is used to process road images based on pixel layouts that match the training data. After these images are processed, they are converted directly to steering
wheel directions, without having to specifically go through a process of converting them. This whole process is generated by the testing process of the neural network, in which images are sent into the program, along with the desired output. If the desired output the researcher set is not met by the program, then it will conduct back propagation weight adjustment to bring the program closer to the desired results. This adaptive process works well for many different types of roads, although a drawback of this process is the dependence on hardware the program has. If any parts of the steering wheel or sensors are modified, then the movements sent from the program to autocorrect the vehicle navigation may over or under correct the expected movement [6].

One of the other dashcam footage papers I reviewed when doing my research was “Analysis of Dashcam Video for Determination of Vehicle Speed” by SAE International. This paper was similar to the previous one, with this paper being about trying to determine the speed of a vehicle from the dashcam footage. From this paper, I learned about how they only used a small patch of the image to calibrate their speed detection software, which was similar to how I eventually excluded the top half of the image from my thresholding calculations since it is assumed that road will not occur near the sky level unless the dashcam footage is inherently flawed. Although these papers were somewhat helpful, they were not as helpful for road detection as the next resource I will be talking about, which does focus and relate to Computer Vision using OpenCV [7].

The third resource I found that was extremely useful in setting up my Python environment for OpenCV and also useful in starting the project in general was “The Ultimate Guide to Real-Time Lane Detection Using OpenCV” published by Automatic Addison. This article consisted of
how to write a lane detection algorithm in Python. After downloading and running the algorithm that was given, however, I figured out that the algorithm was fitted to only work for extremely specific circumstances where there were not significant variations in sky and the clouds that made them brighter than the road lines. This was when I realized that the lane detection algorithm would not be as helpful as I originally thought when initially reading the article. Still, regardless of the performance of the algorithm, the ideas presented in the article are what I extended upon in my own project, ones like thresholding an image and different types of edge detection (Sobel was used for edge detection for their algorithm). Having a starting point such as this resulted in the eventual success of my own project, and many of the test images I obtained were sourced from this same article (although the algorithm given by the article does not function correctly for some of the images in the article, which I kept in mind when developing my algorithm) [8].

While I may have ended up going with my own method of road detection, there are many other methods I would like to mention here that seem to produce great results as well. One of these sources, “An OpenCV Lane Detector for dummies in C++” by Jan’s Place, talks about a way that they developed a similar algorithm to mine, although using perspective transformations and a window sliding algorithm. Essentially what was done was that as opposed to my edge detection algorithm, they took a small part of the image that contained the road and warped the perspective to show it from a view similar to a bird’s eye view. Then, after thresholding the pixels in the image after using Gaussian blur (averaging the color pixel values in the image into chunks), they had their program draw a line of best fit through road dots and then visually transformed the image back to its normal form.
This method works for many cases in which the road is straight and maps a line between road line dots that are broken up in the road. However, this method does not work for curved lines, as many methods similar to this do. When the algorithm generalizes the image based on a sliding scale window and lines of best fit, it becomes significantly less accurate for sudden changes in the road such as a curve due to how predictive the algorithm is. Although my own algorithm works fairly well for curves, it is likely not as optimized or efficient as this algorithm is (even though it is as efficient as it needs to be to load video) [9].

A more official research paper, “Real-time moving vehicle detection, tracking, and counting system implemented with OpenCV”, also published by IEEE, details another algorithm that they were working on specifically for detecting vehicles moving on a road. This algorithm reads in the image data in a very similar way to how my own algorithm does, just with a different application for the data. For shadow removal, which is important to prevent the algorithm from detecting the shadows of the cars as actual cars in the video, Otsu’s method is used, which is a popular segmentation algorithm for applications such as this. From there, thresholding is used to isolate the cars in the image by observing where there are changes between each frame and drawing a box around the moving blob that is likely the car. Overall, this algorithm is slightly unrelated to the topic of my research, but it does use the OpenCV library to detect the cars, just thresholding off of movement rather than color values/edge detection [10].

4. My Approach

4.1 Processing Video Frames for Analysis
The first issue that one encounters when trying to analyze any image/video using an algorithm is having to actually access and read the pixel values of the images/frames, especially when trying to process an entire video full of frames. Luckily, this is where OpenCV excels, in that they provide a way to read frames from a video using the VideoCapture class. This class allows any video to be read into a VideoCapture object, which then allows each individual frame to be read from the object and stored in a Mat object via the Mat class. The Mat class creates Mat objects that convert frames into a matrix of pixels, which then allows for them to be entered into the thresholding algorithm for analysis. Another useful feature that the VideoCapture class offers is the ability to determine whether or not the frame was successfully gathered via a Boolean return value, which in turn details whether or not the frame was actually read in.

After this point, the while loop starts and continues to read through frames and load them into the thresholding algorithm. Then, once the VideoCapture finally runs out of frames to read, it will close the capture and thus finish reading input. Normally developing image reading would have been a very complicated process, however OpenCV helps optimize this process to allow for every frame to be processed effectively. Figure 1 displays an example of the program outputting what frames it is reading in and thresholding, mainly so I could keep track of the program progress.
Figure 1: Visualized Frame Reading Process

Originally, I tried to develop the program using the OpenCV Python package, which proved to run extremely slowly. Since at the time I did not know about the OpenCV C++ implementation, I decided to try and incorporate many different common methods of speeding up image analysis to try and combat the slow performance issues. The first two methods that I tried to incorporate are GaussianBlur and medianBlur (also called median filtering) through OpenCV. Doing these operations resulted in blurring the images, which in
turn causes a significant reduction of noise that could cause errors in the performance.

Figure 2: Example of Gaussian and Median Blur

While both of these methods help reduce the noise, they unfortunately impacted the quality of the images/frames I was working with, and since I was already analyzing every pixel value, I decided to try manually building in some noise eradication into my thresholding algorithm, if only to preserve the quality of the result.

Another method that I decided to test in order to see if it would help the performance was an image loading method called temporal coding, which is also known as predictive coding. To use temporal coding means to, instead of loading in each individual frame, to just load the difference from one frame to another to preserve memory. After testing this method, I realized that doing this had very little actual impact at all on the program, either positive or negative. When trying to figure out why this was the case, I discovered that since I was already loading in frames using OpenCV’s VideoCapture functionality, the frame loading was already fairly optimized. Despite the failures in trying to test this performance saving method, it actually brought up the idea of using a form of temporal coding specifically for mapping the road so that
only one of every five frames has to be analyzed. However, due to the later conversion to C++, this method ended up not really needing to be used in order to generate satisfactory performance.

4.2 Increasing the Speed of Image Processing

Since blurring and temporal filtering did not improve performance, I decided to implement multithreading into the program. Multithreading is the action of splitting the program into multiple threads, thus allowing multiple processes to run simultaneously. For my program, this was represented through how the images were processed, in which I would make a separate thread for each image so that they could be analyzed concurrently while still having the vector contain all of the separate images.

This worked very well whenever I was running my program with the OpenCV Python library, however it was still fairly slow and also now it was extremely taxing on my computer. The reason it was so taxing is that, due to having to analyze multiple full images at the same time, the threads took up a significant portion of the memory on my computer. To control this memory usage, I added a semaphore to limit the number of threads running at one time to ten threads. Regardless, while this way of multithreading was still the fastest way that the program could run through Python, it was still not fast enough to process video at the rate I desired, which ended with me reconsidering the efficiency of Python in building a program like this. The final solution to program speedup that I found was to simply switch the language I was using from Python to C++.
The reason that the switch from Python to C++ solved nearly all of my speedup issues and increased my program performance speed by four-hundred times the original performance speed was because of how vectors function between Python and C++. When programming in Python, one stores their values in an object called an ArrayList, which I initially thought of as simply another way to express the vector. Unfortunately, I was mistaken in this assumption, since after doing some research, I realized that Python’s ArrayList object uses a linked list instead of the typically array object, meaning that accessing a specific index in the ArrayList meant going through the linked list until the index was found. For small programs and programs that do not require a large amount of accessing particular elements in an array, Python’s ArrayList class would be sufficient, however the program that I was writing was having to take in whole images full of pixels and do comparisons. Furthermore, since my thresholding function went through each image by each individual pixel, the standard C++ vector is almost required for this type of work.

Therefore, after spending time searching for easier ways to repurpose my Python code and make it run as fast as C++ code, I ended up just having to rewrite my program into C++ with the C++ OpenCV library. As was stated before, the speedup from converting my program to C++ was over four-hundred times faster than Python, so I would certainly say that it was a good choice to do so. Although, doing so required some restructuring of the program and some rewriting in order for it to obtain the same desired result as the Python version.

4.3 The Image Thresholding Process
Now that I have covered various methods on how I optimized the program to run as quickly as possible, it is time to explain how the program that I created functions and successfully identifies pixels on the painted lines that define the lane the car is driving in. The algorithm I created to do so was what took me the most time in the development of this project, so I will try to give a large amount of detail on how the algorithm works. Furthermore, after many tests and modifications, I believe I have made this algorithm operate in the most efficient way possible for what I am trying to accomplish.

Lines on the road are white on a black background, so the classic image processing technique of thresholding is an obvious first choice. Thresholding refers to the idea of using an algorithm to separate the pixels in an image, usually allowing one to identify which pixels on an image are fit for preforming operations on. To start off, there are many different ways to execute thresholding, but I decided to make my own complex mix of global thresholding combined with a sort of edge detection with the usage of color values. The color OpenCV images that are being worked with have both BGR and alpha channels (opacity). Since the alpha channels are always equal to 1 for these images, I excluded the alpha channel from the image analysis so that I would only be analyzing the color values, making the program run faster. The main reason I decided on the method of global thresholding was so that I could quickly determine which pixels in the image were ones that I needed to be concerned with, allowing pixel comparisons to only occur on candidate pixels. Since there are roughly 921,600 pixels in some of the individual HD images that were being processed, it was very important to only run operations on pixels that fulfill the conditions of what could be considered a part of a road.
Candidate pixels, according to the initial trial and error tests that I ran to determine the most accurate values, are pixels that have the numerical value 205 or higher for each BGR value. BGR was chosen instead of the typical RGB because the OpenCV libraries use BGR as a standard, due to the developers for the library preferring it. Therefore, what a numerical color value of 205 means for BGR is that for the chosen pixel, the Blue, Green, and Red color values all have to be greater than 205, with 205, 205, 205 being a form of light grey. This prevents any pixels that do not have values greater than 205 for each BGR value from taking up unnecessary processing time when they are just going to be ignored from the algorithm anyways.

Once a candidate pixel has been identified, the continuing analysis of the pixel can progress. In order to understand how the algorithm needed to function, I first had to understand the general structure of how the roads will typically be laid out in an image. When driving on a road, there will always be some sort of shoulder on the left and right side of the road lines, as well as the actual space between the lines where the driving lanes reside. In most situations, the shoulders will always be some sort of grey color, somewhat consistent with the color of the driving lanes. What this means is that once a candidate pixel has passed the global thresholding, the colors to the left and right of the pixel can be observed to see if they match


what would be considered a color value of the shoulder or the driving lane.

![Figure 3: Example of Road with Shoulders](image)

With an image such as the one above, it is very easy to see where the shoulders and the driving lanes are. Therefore, this image was one of the images I used in the testing of my algorithm in order to determine if my program was functioning correctly. Throughout the entire process of reading the pixels, the pixels are read by row first, and then column last. This works out great for what I was trying to accomplish, because in truth only the first white pixel and the last white pixel on the same road line need to be read, the rest can just be filled if the algorithm conditions are met. Also, since the road lines run vertically in the image, reading the pixels in a horizontal fashion naturally makes it easier to grab this first and last white pixel. Overall, this way of reading pixels minimizes the accidental issue of identifying road lines as the road gets
smaller into the distance, as shown by the following image in Figure 4 that has a window that narrows as the road gets further away from the vehicle.

![Figure 4: Example of Road Lines Being Trapezoidal Based on the Perspective](image)

If a candidate pixel passes the global thresholding section and is a light grey/white pixel, then we will enter the next phase of the program that compares this pixel to pixels that are nearby. This part of the process was the most complicated part to figure out, requiring many different checks in order to determine whether the candidate pixel was truly part of a road line or not. The main check that the thresholding algorithm executes at this point is on the previous pixel in the same row as the candidate pixel. Now, in order to check the previous pixel in the vector consistently without error, the very first pixel in each row is excluded from the algorithm
so there will always be a previous pixel to check and compare the current candidate pixel with. This previous pixel is then analyzed in a similar fashion as the candidate pixel, except now it checks whether or not any of the previous pixel’s values are below the numerical color value 205 in either Blue, Green, or Red. By doing this, the algorithm can properly conclude that if the previous pixel is darker than what would be considered a normal white road pixel in any color category, then the current candidate pixel may be the start of part of the road line.

This check was the main check that occurs to determine the validity of the candidate pixels, however there is also another critical check that must take place. While the restriction of the candidate pixel to a color value of 205 in Blue, Green and Red ensures that the pixel will not deviate significantly from a color value of light grey/white, there is still some room where it can deviate. To remove this issue and further optimize the algorithm, the maximum and minimum BGR color values of the candidate pixel must be within around 40 of each other, making sure any pixel that passes this check will be a shade of grey or white, similar to what is shown in Figure 5. The maximum and minimum values of the pixel are gathered before any of the checks are conducted, in which there is a loop that goes through each value to determine what the maximum and minimum values are.
Figure 5: Example of the Grey/White/Grey Pattern That My Algorithm Tries to Detect

A later development that I worked on was to identify the patterns of Grey/Yellow/Grey that is also commonly seen in roads. This is something that I would like to mention mainly because it took some modification of the program to achieve. Simply put, I went through the program and removed some of the restrictions on the blue color value in the BGR thresholding analysis. The thresholding algorithm still correctly identifies the lines because the green and red color values are still being analyzed, with the red and green values still having to be higher than the thresholding parameter described above.

4.4 Line Filling

After these checks finish and pass, the next stage of the algorithm consists of filling the white lines with a color that is fairly easy to see and stands out compared to the surrounds, this color being the black that has the BGR values [0,0,0]. Once a candidate pixel is chosen and identified to be the start of the road, the algorithm continues going through the row, keeping count of each pixel index that it ends on every loop. Then, when the algorithm figures out that the next pixel after the current pixel will not meet the conditions for a candidate pixel, it stops counting and saves the current pixel index.

Normally, one would want to try and fill the pixels as they were looping along the row, however the algorithm I developed actually loops again backwards in order to fill them, rather than just doing both the analysis and the filling at the same time. Although doing the looping this way is inefficient in most cases, for this program it allows noise to be filtered out more easily, saving the thresholded image from having many one or two length lines that would not
accurately indicate a road while also preventing lines that are too long from being considered. Also, another check can take place, in which the algorithm checks that the pixel after the last candidate pixel is actually part of the driving lane or shoulder instead of a pixel that would not normally occur after the end of a road line (like grass, the sky, etc.). This check in particular has similar logic to the check for the candidate pixels, in that the pixel will have its maximum and minimum values taken and the range should fall within 40, meaning that the pixel will be some shade of grey.

Provided that all of these checks pass and there is truly a road line within a range of pixels on that row, the program will loop backwards and retroactively fill the pixels with the aforementioned black color value until it gets to the first candidate pixel, starting with the last candidate pixel. There are bounds in place in order to prevent accidentally running out of pixels on the left or right side of the row in the image, in which case the program just won't loop through about two pixels at the start and end of the row to ensure that analyzing the next and previous pixels will always be in bounds, similar to when doing the candidate pixel analysis. Then, finally, when all of the calculations are done on the image (which takes roughly a few hundred milliseconds to complete), the now thresholded image is shown on screen to the user until the user decides to close it.

4.5 Adaptive Thresholding

With a given image, static thresholding can work very well, especially if the algorithm is tailored towards that image. However, for my algorithm, I would like my numerical thresholding values to change to suit any image that the algorithm runs with. By adaptively changing the
algorithm for every unique image inputted into the program, the final thresholded image will be more accurate in comparison to if the image was just analyzed with the static values the algorithm already contained. So, after finishing the initial thresholding algorithm, I created a new method to determine how the values in the algorithm need to be modified in better accommodate a given image.

When starting the development of adaptive thresholding, I first had to decide what values needed to be modified, and what parameters will affect this modification. After thinking through what will compose the method, it seemed best to settle on finding the maximum and minimum color values in the image. This would achieve the ability to see if all color values are actually represented in the given image, and if not it would find the greatest and smallest color BGR values that reside within all of the pixels in the image.

Now, after the three greatest and smallest numerical color BGR values are found within the image, the method will then calculate the adjustment coefficient. The adjustment coefficient is finalized by subtracting the minimum from the maximum for each color value field and dividing it by the highest value that could occur in the field, being 255. The exact values the adjustment array holds are calculated with the formula \{(blueMax – blueMin) / 255, (redMax – redMin / 255, (greenMax – greenMin) / 255\}. This is what gives the adjustment coefficient values to apply to all of the important comparison values in the thresholding algorithm, where all three coefficients are combined into an array and returned to be used in the thresholding algorithm. Finally, the thresholding algorithm multiplies each adjustment coefficient to each corresponding field in the array that is used for comparison for thresholding the image, in that all of the pixels in the image are now compared against this new array rather than the same
default array. Although, if the maximum and minimum color field values are found within an image for any one field, then that field will have an adjustment coefficient of one, meaning that the array will still consist of the same default value for that color field. Some example adjustment coefficients generated by this algorithm are shown below in Figure 6, printed in the order of BGR.

```
0.976471
0.905882
0.980392
```

**Figure 6: Example of BGR Color Field Adjustment Coefficients.**

After doing some testing, I decided to reduce the total amount of pixels that the entire program has to sift through by only having the algorithm go through the bottom half of the image, splitting the image horizontally. This reduced a significant amount of noise from clouds and other objects, especially when considering how the road is only on the bottom half of the image with typical road dashcam footage. With this reduced space to analyze the color values, the adaptive thresholding was not thrown off by the outliers of white color values presented by objects in the sky (clouds, sun, stars, etc.) and could better map what the actual variance is in the image compared to if the outliers were left in the calculations. This also benefits thresholding in the case that it remained static, as it would not incorrectly mark parts of the sky where the clouds are grey as being bounds of a road.

### 4.6 Processing Video

All of the work that I put in thus far was fairly successful, but the main goal of this project in my mind at least was to be able to read in dashcam video footage and output a result
with the road lines identified. While I was now able to have my program correctly identify
where the road lines are in an image, my main concern was that it was only working for images
and not actually reading in video footage. Furthermore, with the library video processing
functionality that OpenCV provides, incorporating video processing into the program was
certainly possible, however most of the issues I encountered with the project that were
explained previously resulted from trying to accomplish this feature.

The reason I eventually converted my program from Python to C++ and tried to
multithread it before the conversion was to speed it up so that it could threshold images in
(almost) real time. With this optimization the program can now threshold and display images at
a rate that has the images play together as a video, which achieves the goal of the processing.
Essentially what the program does is read the video frame by frame and convert each of these
frames into an image, where each individual image is dynamically thresholded and then added
back into an array of images. Originally with multithreading the ordering of the images would be
maintained by a semaphore making sure that each image was put into the array in the same
order it they were read. While this system of ordering worked fairly well, multithreading ended
up not being necessary after converting the program from Python to C++ due to the speedup
from doing so alone. Since multithreading is no longer used, now the images are read one at a
time, with each image being put into the array before the next image starts being read.

With all of the images in the array, the last step is to display the images to the screen.
OpenCV has a method for this, called imshow, that shows the image to the screen combined
with a waitKey. Therefore, with the array of images, the imshow method, and the loop through
all of the images in the array, the images are displayed to the screen of the user in such a way
that it appears to resemble the same video they originally put into the program, just with a slightly slower speed than the original video. The OpenCV imwrite method also allows for writing of the images to the user’s computer, although this could likely fill the user’s project directory with hundreds of thresholded images if they choose a video instead of just an image.

5. Results

Below are going to be several before and after images, in that the image will be shown how it looked before the road lines were identified, and then the image will be shown again after the program identifies and maps the road lines with the thresholding algorithm. Figure 7 shows what the first image looked like before being put through the program.

![Figure 7: First Image with Road Lines Not Identified](image)
With this in mind, Figure 8 is what the image looked like after it was run through my program and through my thresholding algorithm.

![Figure 8: First Image with Road Lines Identified](image)

Now, the result of this algorithm is not absolutely perfect, notably with some empty spaces that are white that should be filled, but overall, it is fairly accurate at detecting where the road lines are, even somewhat in the distance. I would say that, as proved by the next image, that the main reason this is impressive is because the algorithm is very adaptable to different levels of color, like this image below in Figure 9.
Notice how this image is significantly different from the previous image. Some main differences consist of the bounds of the road, the different road colors, the lighting levels and the fact that there is only one road line. Despite the changing conditions, the result of the thresholding algorithm for this image is shown below in Figure 10, where the algorithm is still able to preform an accurate road detection because of the ability for it to use the adaptive thresholding I worked on.
Both of these images above have white lines, but not all lines on the road are white. In fact, the road lines are also commonly yellow, and because of this I adapted the program to accept these colored road lines as well. Figure 11 details an example of a road with yellow road lines, one that also has white lines as the bounds of the road.
Figure 11: Third Image with Road Lines Not Identified

And, with the result in Figure 12, it seems that the algorithm was able to map both the yellow lines and the white lines. Obviously, there is a small amount of noise on the left side of the image, but for the most part it is fairly accurate if the algorithm was used to determine the bounds of the road for the car to stay on.
All of the images above were thresholded using the parameters I found by using trial and error, that is the color comparison parameter of 205 * adaptive coefficients calculated by the adaptive algorithm. However, after incorporating some iterative testing into my program with different parameter values and observing the results in an excel document, I concluded that a parameter value of 205 was supported by my accuracy chart results. Figures 13, 14, and 15 show the accuracy of each parameter when compared against the ideal thresholded image I manually created for each one, which were calculated by taking the data collected from the tests of each parameter for each image and graphing the values of \( \frac{TP + TN}{TP + TN + FP + FN} \). TP represents true positive, which means the program found a road line where there should be a road line, TN represents true negative, which means the program did not find a road line where there was not one, FP represents false positive, which means the program
found a road line where there was not one, and FN represents false negative, meaning the program did not find a road line where there was one.

**Figure 13: Accuracy of First Image**

**Figure 14: Accuracy of Second Image**
Given these graphs of accuracy, I chose 205 as a safe parameter from a list of possible parameters I could choose from. The amount of non-road pixels skewed the accuracy graphs a bit (since most of the pixels are not road pixels), however from these graphs I was still able to determine that the parameter I found from manual trial and error, 205, was indeed able to properly represent the road lines on any given image without too much inaccuracy.

Although performing the algorithm on a still image proves that it is for the most part successful, I decided to extend the program to analyze video to stay true to the idea of analyzing dashcam footage to find road lines. The process of implementing the video reading functionality was already explained, but the result of the video reading will be shown here in the following figures 16, 17, 18, 19, and 20, where they represent the thresholding frames of 0, 20, 40, 60, and 80 respectively.
Figure 16: Processed Frame 0

Figure 17: Processed Frame 20
Figure 20: Processed Frame 80

As shown here, the algorithm being able to detect and display the thresholded images quick enough to create the video demonstrates that dashcam footage can indeed be siphoned through the algorithm in real time. There were some parts in the video that had more noise generated by the detection algorithm than others, however some of this was generated by the video having some motion blur. For the final result, I would say that I am satisfied with what the program produces, since if you ignore everything in the image except for the pixels with the 0 for every color field (black pixels) generated by the algorithm, then it would still be visible where the road lines reside.

Despite all of these images/videos detailing the algorithm was successful in my opinion, it is important to look at the accuracy of the algorithm from a quantitative standpoint. This is why I created another program solely for testing the result to determine how accurate each image is.
For simplicity, I will not be testing the accuracy of the video, since that was just a feature implemented for demonstration purposes alone. Instead, the two images will be focused on and analyzed against a same image that I filled manually, which excludes all of the noise that would be generated from the algorithm and only has the road lines filled with the pixels with color value fields of all 0 (black pixels).

6. Conclusions

With the major shift of the market towards self-driving cars operating on computer vision, I would certainly say that there are likely better computer vision programs on the market compared to my program. However, considering that this was my first time working with computer vision in order to write a program, I would say that I am satisfied with the ending result of the project and how it aligns with the expectation I had of the project going in. The main focus I had while working on the program was to learn about different types of utilizing computer vision, and this project gave me the opportunity to do this while being productive towards my educational goals.

Given that this was also my first time building a thesis based on research of a subject that I had interest in, I would say this also gave me my first taste of what it will be like continuing my education onto a Master’s degree. Part of the reason that I chose to stay and Honors and write an Honor’s Thesis was so that I could continue on with my Master’s degree through University of Arkansas, since I am wanting to learn more about these sort of complex topics that are only really lightly covered in my Bachelor’s level classes. In conclusion, while the writing part of the thesis was a little tedious at times, I enjoyed it and am looking forward to applying these
research skills to better understand complex topics and subjects that will be covered in the Master’s program.

7. Future Work

7.1 How This Project Could be Developed Further

While I achieved the goal of what I was aiming to accomplish for this Honors Project, there are several ways that this project could be extended. The most notable one would be actually displaying the driving lane similar to how it is shown in the related work section. Originally, I made a rough algorithm for doing so just because I got bored and wanted to work on something, but an actual algorithm created for this specific purpose could easily draw lines connecting the dots together and create a rough display of where the driving lane lies. By creating an algorithm to do just that, the program as a whole would be one step closer to actually being able to be used and tested similar to a program inside a self-driving car would be. Overall, though, even just having a program like this without the displaying of the driving lane can be used to achieve this purpose provided there is another algorithm that removes the rest of the noise.

An example of how such an algorithm could be created would be to simply draw lines connecting the very last middle road-line pixel with the very first right road-line pixel. Assuming the algorithm can locate where these pixels lie (which would require some sort of detection similar to my thresholding algorithm), then these connections can easily be drawn and completely cover the lane the vehicle is being driven in. I imagine that making such an algorithm
would not be too difficult with reference to my current thresholding algorithm, but that would still be outside of the scope of the Honors Project if I were to try and continue developing it.

![Figure 11: Example of Displaying the Driving Lane](image)

7.2 The Adapting for Untraditional Roads

One feature for the program that I had the idea of that I could not implement was to extend the program to be operational on roads which did not have the typical road lines on them. Now, since I never worked on such a feature, I cannot properly visualize how complex completing something like this would be, although I would imagine that it would require some special cases of the algorithm if a case like this occurs. One of the images shown in the results section shows this exactly, where there are only dashed white lines in the middle of the road.

Another situation where this occurs is on dirt roads like in Figure 14 and 15, which are very common in rural areas where the road has not been paved yet. With these situations my
algorithm would fail in finding any road lines despite there still being a road there. This also raises the issue of there not being defined road lanes for the car to drive on, which in all honesty could probably be resolved by having the self-driving car stick to the right of the road when no road lines are detected. Still, the current thresholding algorithm would have to be modified to allow for the driving lane to be detected from the bounds of the road.

![Figure 14: Dirt Road with Transition from Brown to Green](image)

A solution to these issues could be to simply identify where the color changes on either side of the screen, like from brown/orange to green or grey to green for example. This would be something that is added on top of the standard analysis when there is only one or no road lines detected from it. Whether it consists of modifying the current algorithm or adding more code to be run after the original one is run, code that serves this purpose for these specific scenarios would enable the program to be more flexible and adaptive to certain environments, therefore
allowing it to be more convenient and accessible for the user of the self-driving vehicle.

Furthermore, there may be situations that require a more complex solution, like the image below.

![Figure 15: Image with No Clear Color Transition from Road to Grass](image-url)
8. References


