A Raspberry Pi computer vision system for self-driving cars

Zach Isherwood[1] and Emanuele Lindo Secco[1[0000-0002-3269-6749]]

[1] School of Mathematics, Computer Science & Engineering, Liverpool Hope University, L16 9JD, UK
18004704@hope.ac.uk, seccoe@hope.ac.uk

Abstract. This paper presents a prototype of a self-driving vehicle that can detect the lane that it is currently in and can aim to maintain a central position within that lane; this is to be done without the use of special sensors or devices and utilizing only a low-cost camera and processing unit. The proposed system uses a hand-built detection system to observe the lane markings using computer vision, then using these given lines, calculate the trajectory to the center of the lane. After locating the center of the lane, the system provides the steering heading that the vehicle needs to maintain to continuously self-correct itself; this process is real-time performed with a sampling frequency of 20 Hz. Due to the increased number of calculations, the heading is smoothed to remove any anomalies in observations made by the system. Since this system is a prototype, the required processing power used in an actual vehicle for this application would be much higher since the budget of the components would be more significant; a higher processing speed would lead to an overall increased frame rate of the system. In addition, a higher frame rate would be required for higher speeds of the vehicle to allow for an accurate and smooth calculation of heading. The prototype is fully operational within an urban environment where road markings are fully and clearly defined along with well-lit and smooth road surfaces.

Keywords: Self-Driving System, Raspberry Pi, Low-Cost System.

1 Introduction

1.1 Outline of the field of work

Driving a vehicle is a complex process due to the number of parameters that need to be evaluated and reacted to accordingly each second. Some examples of this may include but are not limited to (i) the vehicle's speed compared to that of the vehicle ahead, (ii) offset to the center of the road, (iii) lighting conditions, (iv) stopping distances, (v) road conditions or road occupancy, and so on.

The ongoing increase of licensed vehicles entering the road per year as well as the increased population and town sizes results in busier roads and therefore more room for human error, partially due to a lack of observational awareness, device distraction,

lack of concentration, or other road users poor driving standards; these facts result in an increased requirement to remain aware of other road users, their actions or indeed inactions. Hence, car manufacturers are seeking out assistive features for some of these processes which can seek to account for any lapses that drivers may incur due to this increased congestion.

Tesla is arguably the leading car manufacturer working towards autonomous vehicles; in the UK alone, there is recorded to be 90,900 Tesla vehicles on the road (as of November 2020); this was a 336% increase from 2019, where 27,000 vehicles were recorded [1]. Current Tesla *autopilot* system can control the car with limited assistance from the driver. The car can maintain central road positioning, avoid collisions with obstacles in its path, adhere to traffic laws such as road markings and signs. The system uses 8 surrounding cameras to provide full 360° of vision with a potential distance of up to 250 m; there are also 12 ultrasonic sensors around the vehicle and a forward-facing radar for low-visibility conditions such as fog or heavy rain [2].

Although Tesla models have the capability of completing an entire journey with little to no intervention from the driver, the autopilot feature is still marketed as an assistive feature to aid with *the most burdensome parts of driving* and that the car requires active supervision while the autopilot feature is enabled.

1.2 Ethics of the system

False or poor marketing, or more particularly the poor user understanding of the implemented safety systems, can lead to an increased belief of complete full autonomous travel. These beliefs can, and have, been known to be the root cause of avoidable accidents due to the driver not being fully attentive or engaged, or, in a recent extreme circumstance, not even in the driver's seat [3].

The sensationalistic media coverage of these incidents can often misrepresent the assistive features that these vehicles present, most commonly from overestimating their functions and ability to navigate without human aid by seeking to lay blame for these incidents on the vehicle or devices themselves rather than the lack of required interactions or social responsibility that the owner/occupier of the vehicle should have had at the time of the incident, i.e. not being in the driving seat at the time of a crash. The fact that as humans we are already willing to adapt and or surrender our controls within vehicles based on the existence of such devices, and capabilities, is an ongoing conundrum. We like our control, we like to be able to drive where, when and how we like, but conversely, we're happy to relinquish this control when it suits us. Based on the content of sci-fi films we can see such elements becoming a reality and, ultimately, we are willing to accept these changes.

The prototype system detailed within this paper is not designed to drive the whole journey of the vehicle but instead to reduce fatigue on long and relatively unchanging parts of the journey, such as a motorway or long consistent road. When combined with other safety devices on the vehicle it can allow light touch driving rather than full hand on steering and control. As with other manufacturers' systems, the driver

should always be conscious and aware of their surroundings even when the system has control.

The advances of such systems in vehicles leads to the ethical dilemmas of moral decision-making being handed to autonomous systems or Artificial Intelligence rather than remaining in the hands of the fully functioning and morally capable individual. These dilemmas are at the heart of today's developments and will continue to be so as both current and future consumers will not seek to take the lesser of two safe options where choices are available.

In the end, such systems should be used correctly and fairly - users should always follow the guidelines set by the manufacturer without fail. This is a point that should be reiterated to vehicle users at the point of sale and in all literature thereafter. Such reiterations would seek to ensure that the user does not become too reliant on the system and start to make poor decisions due to lack of attention as a result.

This paper is not seeking to address these here but rather to identify that one step forward leads inexorably to the next but in doing so we must also be aware that the path will contain a large number of obstacles and resistance to change. It does seek to demonstrate the effective use of computer vision for practical everyday use to help with a repetitive task, leading into the work towards self-driving vehicles, capable of adhering to local speeds limits, maintaining road position - even in changing road conditions - and avoiding collisions with fixed or moveable obstacles such as road furniture, pedestrians, cyclists or similar.

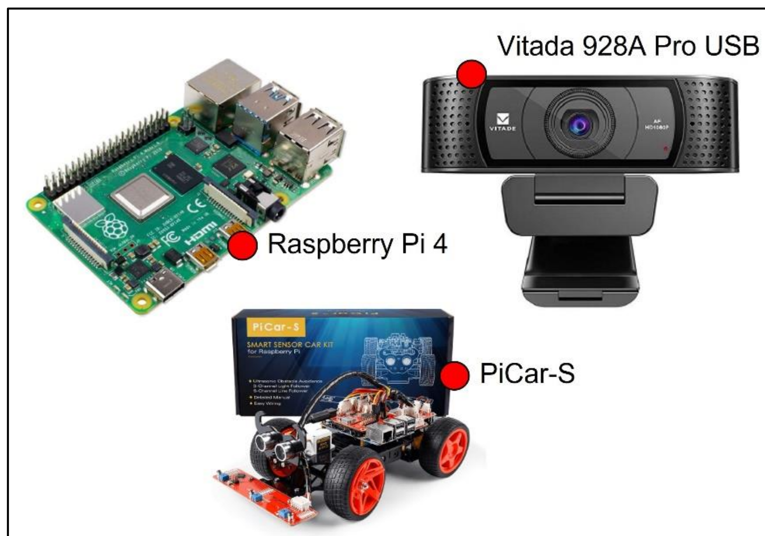

Fig. 1. The Raspberry Pi 4 board, the Vitada webcam and the PiCar-S

2 Materials & Equipment

2.1 Hardware

Embedded system - The main controller of the system is a Raspberry Pi model 4 which is a small, lightweight and inexpensive computer. This embedded system will handle all of the computation, from controlling the car itself to detecting and tracking the lane. The installed operating system is Raspbian, which is a free Linux version built solely for the Raspberry Pi line (Figure 1, top left panel - [4]).

Camera - The visual input of the system is implemented by using a Vitade 928A Pro USB Computer WebCam; this device was selected over a standard Raspberry Pi camera due to its more rigid structure and its automatic light correction functionality (Figure 1, top right panel). This camera has a 80° wide-angle lens and allows video acquisition with a resolution of 1080 pm and a frame rate of 30 fps.

Sunfounder PiCar-S - The car used for the prototype model of the system is a PiCar-S from Sunfounder (Figure 1, bottom panel). This PiCar is designed to run from a Raspberry Pi using a PiCar library which is provided online. The car comes with correct drivers and wires to run the motors safely as well as accurately control the front servo for the steering heading. Three different front modules are also included; these are an *ultrasonic sensor* and 2 types of *infrared sensors* for line following capabilities; however, these additional modules are not used in our design.

Power supply - The PiCar system requires two 18650 Lithium-ion Rechargeable Batteries to operate with sufficient power. These batteries are often used for high-drain applications such as this project, these batteries plug into the hat attached to the Pi to power the whole system including the Pi and the USB camera. Each battery supplies 3.7 V and 4800 mAh.

2.2 Software

Python - The programming language used to control the system is Python. Python is a high level and highly versatile programming language that allows for the use of various types of libraries and different programming paradigms. The language's design tries to help produce logical and easily readable code; this therefore not only helps in the development of a system but also in the debugging and maintenance phases of the system. The version used in the system is Python 3.8, this is the most up to date version compatible with the use of the OpenCV library.

OpenCV - OpenCV is an open-source computer vision library that provides many programming functions for the use of real-time computer vision and claims to have more than 2500 optimized algorithms to aid in this process. Typical uses of the library include object and face recognition, object tracking and simple to complex image manipulation [5]. In this work, OpenCV is used to capture a frame from the camera, pass that frame through the image manipulation pipeline to prepare it for line detection [6]. This pipeline is as follows: (i) Load in the frame, (ii) convert the frame from BGR to RGB, (iii) grayscale the frame, (iv) blur the frame and (v) apply edge

detection. In addition, functions from the library such as (vi) hough lines and (vii) canny edge detection are also used to find the road's edges.

VNC Connect - Virtual Network Computing (VNC) Connect is a remote desktop software that enables the user to control another device/computer over an internet connection as long as that device also has VNC Connect installed and allows remote access. The device that is being connected to has VNC server installed, and the device used to control from has VNC Viewer installed. Sends inputs such as mouse movements and keystrokes over the internet to the other device. The display is also sent over the internet so the user can see in near real-time what is happening on the device. Used to control the pi without need to have a keyboard and mouse along with a monitor constantly plugged in. Allows the project to be constantly tested and improved upon without interference.

Pycharm IDE - Before the system was deployed to the Raspberry Pi, it was programmed using the Pycharm IDE. Pycharm was developed specifically for the use of the Python Programming Language and aims to improve the user's experience in terms of productivity and useability. Functions, classes, loops and conditional statements can all be collapsed to allow a user to traverse a larger program with less hassle.

Fig. 2. Implementation of the 'Loading an Image Frame' and of the 'Gaussian Blur and Canny Edge Detection' (left and right panels, respectively)

3 Methods & Results

3.1 Basic Requirements

Within a lane assist system, two fundamental processes need to occur. The first is *perceiving and tracking the lane lines*; the second uses these given lines to calculate

the *wheel heading to steer the vehicle* in the correct direction to remain within the lane. Finally, the system needs to work on a live video feed from the USB webcam. However, throughout the development, the system can be tested on an image frame since live video is just an iterative loop of loading in image frames.

3.2 Implementation – image processing

Loading an Image Frame - The first step needed for the image processing is to load the image frame into a variable; in this case, the sample image is *test.jpg*, the frame is loaded and stored in the variable *frame* using the *cv2.imread()* function (Figure 2). The frame can then be converted into a single-color channel using the *cv2.cvtColor()* function; here, the loaded frame is passed into the function along with the code for the color conversion. The color formatting for OpenCV is Blue, Green, Red (BGR) instead of the standard RGB; therefore, the color conversion required is from BGR to Grayscale using *cv2.COLOR_BGR2GRAY*. The two frames are then displayed on the screen using *cv2.imshow()*.

Gaussian Blur and Canny Edge Detection - Once the image frame has been loaded and converted into a grayscale format, the next step is to detect any edges that the image can find. Before this can be done, the frame needs to be blurred slightly to remove noisy parts of the image that may interfere with finding the lines (Figure 2). The frame is blurred using the following function:

$$cv2.GaussianBlur(src, ksize, \sigma_x, \sigma_y, borderType)$$

where

src - Image source / Frame to be used

ksize - Kernel size, the standard is a 5x5 matrix

σ_x - Standard deviation of the kernel in the horizontal axis

σ_y - Standard deviation of the kernel in the vertical axis

borderType - Applies a border to the boundaries of the image while the kernel is applied

Gaussian Blur - The Gaussian blur function checks each pixel in the image and compares it to the pixels in the surrounding predefined box (kernel). A weight is then applied to each pixel within this kernel; more weight is added to the pixel in the center of the kernel compared to the pixels further away from the center [7]. All of the pixels within the kernel are then added together and an average is taken, the central pixel is then replaced with this average value. This is an iterative process that occurs for every pixel in the image.

$$K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The kernel value can be any number, however, it is strongly recommended to use an odd-square kernel such as (5,5) or (7,7) to allow for a central pixel in the kernel. The σ parameter controls the variation around the mean value of the kernel, larger values of σ allow for more variety around the mean, whereas smaller values allow less variety; if the value is zero, then the kernel is applied to every pixel in the image. *If only* σ_x *is specified,* σ_y *is taken as equal to* σ_x*. If both are given as zeros, they are calculated from the kernel size* [5]. Once the frame has been blurred, it is then stored in a new variable called *blurred_frame* that can be used to detect the edges of the lines within the frame.

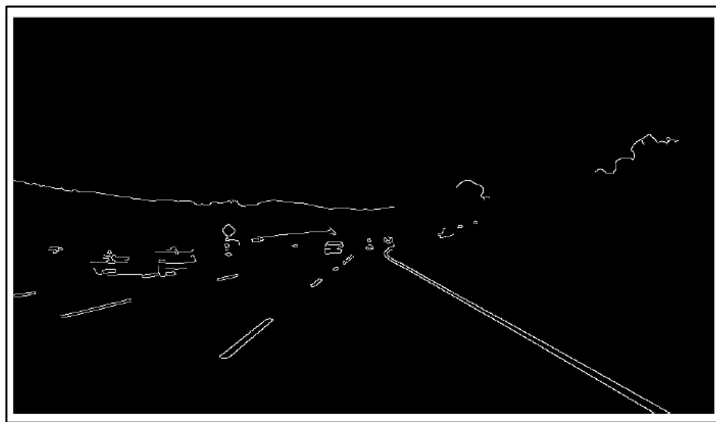

Fig. 3. Completed Edge Detected Frame

Canny Edge Detection - Once the frame has been blurred, the next step is to detect the edges of any lines using *cv2.Canny()*. The summary of the multiple stages of the command is as follows [8]:

1. Find intensity Gradient of the frame - identify parts of the frame with the most substantial intensity gradients (using Sobel kernel)
2. Non-maximum Suppression - Thin out edges to removes pixels that might not be part of the edge
3. Hysteresis thresholding
   - Accepting pixels as edges if the intensity gradient value exceeds an upper threshold (*maxVal*).
   - Rejecting pixels as edges if the intensity gradient value is below a lower threshold (*minVal*).
   - If a pixel is between the two thresholds, accept it only if it is adjacent to a pixel that is above the upper threshold.

The Canny function takes the following arguments:

cv2.Canny(src, threshold1, threshold2)

where

Src - This is the frame to input into the Canny function

Threshold1 - This the minimum value (*minVal*)

Threshold2 - This is the maximum value (*maxVal*)

These two defined thresholds are used in the hysteresis process. Figure 3 shows the result of this image processing.

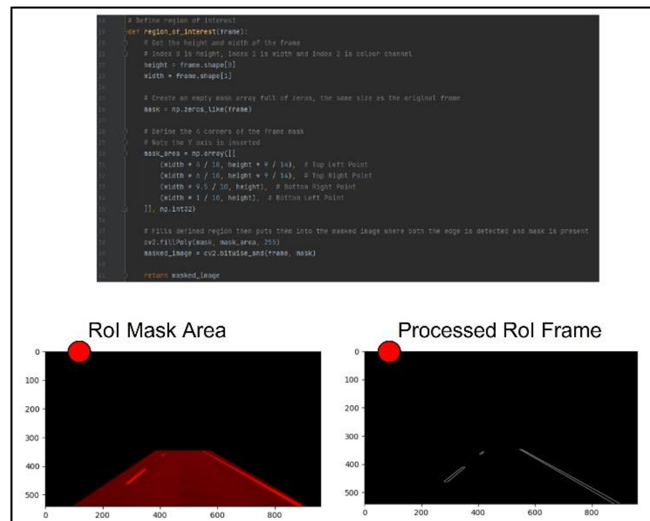

Fig. 4. Region of Interest (RoI)

Region of Interest (ROI) - The Canny frame is a step closer to detecting lines in the frame; however, the frame is full of lines that the system would find redundant and noisy. The best way to remove this excess noise is to mask out a region of interest; only lines within this region will be detected [9].

Defining an effective region requires the camera to be calibrated correctly since the region of interest is not dynamic; the region is defined before the program runs. Incorrectly calibrating the camera or defining the region of interest to be too small can result in the system being unable to find the lane lines. On the other hand, making the region too large can cause a noisy estimation of the heading; therefore, this region needs to be fine-tuned to each setup it is used in. We use a function which declares a polygon named *mask_area* using a NumPy array; anything outside this polygon is disregarded as noise since it is not within the vehicle's path and could cause complications such as detecting other lanes lines, which would in turn, return an incorrect steering heading. Therefore, it holds:

- *mask = np.zero_like(frame)* sets a NumPy array full of zeros, with the same dimensions as the original frame.

- *cv2.fillPoly(mask, mask_area, 255)* This command fills the empty mask with 1's instead of the region defined by the mask area, the rest of the array is left as zero's.

Figure 4 shows the result of this further step of the image processing. The next step is to store only the masked area into the masked image frame, this is done using *masked_image = cv2.bitwise_and (frame, mask)*. The bitwise uses the AND operator to detect where edges are detected in both the frame and the mask.

Fig. 5. Adding lines to the original frame

3.3 Implementation – line definitions

Hough Lines Transform - The Hough line transform is an extraction method used to detect straight lines in an image frame, which works perfectly with the previously created isolated region from the Canny frame. OpenCV has two methods of using the hough lines transform, the first is the standard *cv2.HoughLines*, the second is the *cv2.HoughLinesP()* which is the Probabilistic Hough Lines Transform. The proposed system uses the *cv2.HoughLinesP()* as it finds the extremities of each line which gives a much more accurate reading to be used to calculate the heading required to correct the current heading of the vehicle. The values *minLineLength* and *maxLineGap* need to be adjusted depending on the camera resolution, distance to the road and the size of the region defined. If the values are too large, no lines will be found, if the values are too small too many lines will be detected as well as many false. This function does not display the lines on the image, it instead finds the coordinates of any line that is found and stores it in a NumPy array called *lines*.

Draw Lines - The lines found in the previous function are passed through, along with the empty frame to allow for the lane lines to be calculated and drawn onto the empty frame. This returned frame can then later be overlapped onto the original image [10].

Drawing Lane Lines – Then the first step is to allocate each line found in the array to either the right or left lanes. This can be done using a nested for loop that separates the x_1, y_1 and the x_2, y_2 points in each line from the array; these points are then passed through the *np.polyfit()* function to get the gradient and y-intercept of the line. If the gradient of the line is negative, the line is from the left lane; otherwise, the line is from the right lane. After all the lines in the frame have been allocated to a lane, the next step is to check there are both lane lines found, as, for the current test frame, both lanes are present. To do this, the system checks that there are coordinates stored in the *left_gradient* and *right_gradient* arrays. The maximum and minimum values of the *Y-axis* are already known and are defined by the region of interest. All that is required to find the points of each line is to find the x_1 and x_2 values, this can be done by rearranging the $y = mx+c$ equation into $x = (y-c)/m$. Once these points have been calculated, they can then be drawn onto the image using the *cv2.line()* function which takes the arguments: (x_1, y_1), (x_2, y_2), *color, thickness*.

Add lines to the original frame - Once the lines have been drawn onto the empty frame, they can be overlapped onto the original frame using *cv2.addWeighted()*. The weighted frame overlays the two frames with the opacity of the original frame reduced slightly in order for the yellow lane line drawn on the image to show through the white of the actual lane line (Figure 5).

Heading Line - Adding a trajectory line is a fairly simple process once the lane lines have been detected, the lower point of the line will always come from the center of the frame, this is assuming the camera is calibrated to the center of the vehicle. The upper point of the line takes the average between the right and left lane lines as this is the center of the lane (regardless of where the vehicle is positioned within the lane). This line has no other purpose than a visual representation of the desired vehicle path for the user [11].

Calculating Heading - The heading function reads in four parameters, the first being the *line_frame*; this is the frame that the heading text will be displayed on. The *upper_left_width* and *upper_right_width* are the x_2 values for the right and left lane. Finally, *min_height* is the highest y-coordinate value recorded for the lines. The front wheels of the PiCar are controlled by a servo with a range of motion from 0° to 180°, therefore when the line is straight, the servo control should receive a 90°, a left turn should be between 0° and 89°, whereas a right turn should be between 91° and 180°. The steering angle is calculated using trigonometry with the offset of both the *x-axis* and *y-axis*.

Detecting Only One Lane - Until this point, the system has only been able to calculate headings and line trajectories if both lanes were detected. If only one lane was found, the system would stop giving a steering heading and break the lane tracking, which is not optimal and should be amended [12, 13]. The approach is taken to this with the

theory that if only one lane is detected, the vehicle is on a curve/turn as shown in the images above and should send a sharp steering heading to the wheels to straighten the vehicle along the current path. Once straightened, the system should then successfully detect both lanes again (Figure 6).

Fig. 6. Left turn (i.e. right lane found) and right turn (i.e. left lane found) on the left and right panels, respectively.

3.4 Implementation – steering & self-driving

Using Webcam as Video Feed - In OpenCV using a live video feed is the same as reading a new image frame from the camera, therefore, to read the frame each time is it available a while loop is used since there is no time limit to this loops operation, however, with this method, the loop cannot be stopped without closing the entire script which is why the *cv2.waitKey* command is included. This checks every loop if the user has pressed the *q* key, if they have, the loop breaks. The frame can be processed between each cycle by calling the *process_frame* function for the current image; this begins the image processing pipeline outlined throughout the previous parts of this documentation [14].

Controlling the PiCar - The PiCar system runs off the Raspberry Pi using two driver modules and a Pi Hat to provide the system with enough power to run using the two 18650 rechargeable 3.7 V Lithium-ion batteries. The files were transferred from the computer they were created on, to the pi using VNC viewer. The Raspberry Pi comes with VNC viewer automatically installed which allows the remote connection process without the prior use of a screen and keyboard. After installing all the correct packages required for the system to work, running the code on the Pi gave the following results as reported in Figure 7.

Now the code successfully runs on the Raspberry Pi, all that is left to do is output the calculated heading to the servo motor and move the PiCar forward. During the PiCar setup, the *picar* library is installed onto the Raspberry Pi to allow easy control of the vehicle. The front wheels are declared as *fw* using the *front_wheels* class from the library. The wheels are then turned from 45° to 135° one step at a time and then back to 45°, through testing this function it was found that the servo would not update

its position if the previous command did not have a delay of at least 0.05 s. Since the front wheels are able to turn to the selected heading, once the heading calculation is completed in the code, the system will turn the servo to that heading, the time delay is then added into the while loop to ensure the servo position changes each time the system updates.

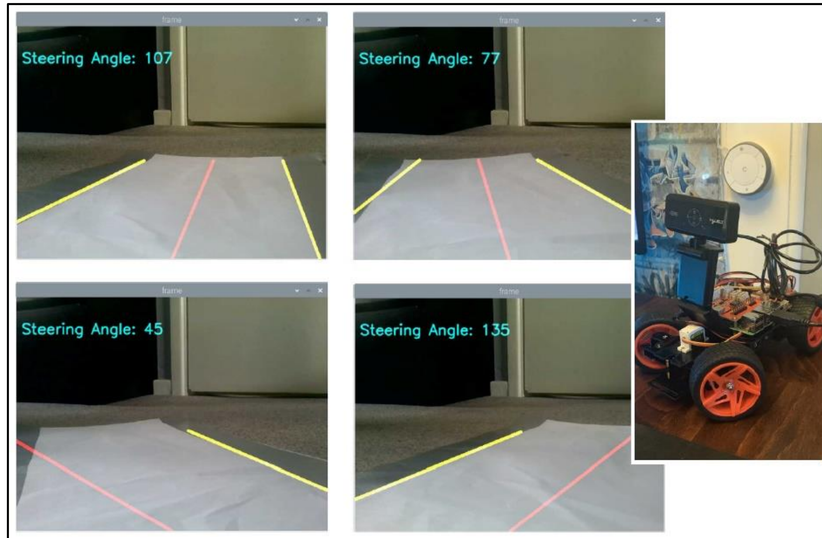

Fig. 7. Determination of the steering angles according to 4 different scenarios as captured by the camera and processed with the Raspberry Pi board

Steering Smoothing - The final addition needed to allow the system to work correctly is to smooth the heading value using previously calculated values. Without heading smoothing the steering can be very violent as any anomalies can create drastic changes in the value calculated. For example, the vehicle can be driving in a straight line and suddenly only detect one line for a few frames, it would cause the heading value to change from 90° to 135° (or 45°) without warning, which in a real-world vehicle would be a safety hazard. To combat this issue, the system only allows the angle to deviate from its current heading by a maximum of 5° if both lanes are detected and only 1° if a single lane is detected. When the system detects both lanes, it is reasonably confident the new calculated angle is correct; therefore, the deviation from the current value can be higher, whereas if only one lane is detected, the steering angle will change to the extremity point value based on which lane is found. To prevent this from being a drastic change to the current heading, the deviation is set to 1° per cycle to prevent overshooting the lane once the opposite line has been detected. If the new angle does not deviate more than the maximum assigned deviation, that heading is output to the servo without being smoothened. The smoothing method works by adding the required angle change to the current heading. The required angle

change is calculated by subtracting the current heading value from the new heading value: *change = desired position - current position*. The direction of the required change should return a value of 1 or -1, which is handled by:

Direction of change = (actual_angle_change/abs(actual_angle_change))

where the *abs()* function turns the number positive regardless of its previous state. This direction value is then multiplied by the maximum deviation and added to the current heading, this final value is the new heading that the vehicle will turn to in order to stay on track.

4 Conclusion & Discussion

This work only preliminary analyses computer vision application in an industrial environment and real-world scenarios [15-17]; many improvements could be made to the system to make it more reliable, since current system is capable of tracking straight lines, but curved lines are not accounted. At present (i) the system detects both lane lines and calculates an offset heading to the right or left and (ii) detects a single lane line and turns the right or left extremity point. With the smoothing of the heading calculations, this is not a significant issue; however, it is still not practical given that the tracking of lane lines on a curve can be inconsistent. The vehicle can still successfully remain within its lane given this flaw, but currently, this is limited to reasonably slow speeds since the frame rate is locked at 20 given the 0.05 s delay per cycle. However, this is not a significant concern as with a higher processing speed and improved equipment, it is expected that this limitation would be solved.

A major drawback of the current system is that it would not work in an environment that has a lack of well-defined road markings/lane edges. Future iterations of the system could use a deep learning machine model to handle the steering inputs for lane control instead of the current hand-coded model; this, in turn, allows for the implementation of object detection for road signs and obstacle avoidance. Detecting road signs could allow the vehicle to comply with speed limits and road traffic law automatically.

Another benefit of a machine model would be the handling of curves as these can be taught during the training stage of the system [18, 19]. The use of a machine model would also allow for the system to be used on urban roads and in environments where lane lines are less visible, however it would require a large dataset to be able to train such a model to successfully identify all these opportunities, but in such a rapidly expanding industry, these issues could be quickly solved.

Acknowledgment

This work was presented in dissertation form in fulfilment of the requirements for the BEng in Robotics for the student Zach Isherwood at the School of Mathematics, Computer Science & Engineering, Liverpool Hope University.

References

1. Statista. 2021. Number of cars in the UK 2000-2016 | Statista. [online] Available at: https://www.statista.com/statistics/299972/average-age-of-cars-on-the-road-in-the-united-kingdom/ [Accessed 23 April 2021].
2. Tesla.com. 2021. Autopilot. [online] Available at: https://www.tesla.com/en_GB/autopilot [Accessed 27 April 2021].
3. The MagPi magazine. 2021. Raspberry Pi 4 specs and benchmarks — The MagPi magazine. [online] Available at: https://magpi.raspberrypi.org/articles/raspberry-pi-4-specs-benchmarks [Accessed 24 April 2021].
4. The Verge. 2021. Two people killed in fiery Tesla crash with no one driving. [online] Available at: https://www.theverge.com/2021/4/18/22390612/two-people-killed-fiery-tesla-crash-no-driver [Accessed 1 May 2021].
5. Opencv24-python-tutorials.readthedocs.io. 2021. Smoothing Images — OpenCV-Python Tutorials beta documentation. [online] Available at: https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_filtering/py_filtering.html [Accessed 25 March 2021].
6. OpenCV. 2021. Canny Edge Detection in OpenCV. [online] Available at: https://docs.opencv.org/master/da/d22/tutorial_py_canny.html [Accessed 25 March 2021].
7. Datacarpentry.org. 2021. Blurring images – Image Processing with Python. [online] Available at: https://datacarpentry.org/image-processing/06-blurring/ [Accessed 26 March 2021].
8. Sahir, S., 2021. Canny Edge Detection Step by Step in Python — Computer Vision. [online] Medium. Available at: https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123 [Accessed 20 April 2021].
9. Wang, Z., 2021. Self Driving RC Car. [online] Zheng Wang. Available at: https://zhengludwig.wordpress.com/projects/self-driving-rc-car/ [Accessed 5 January 2021].
10. Medium. 2021. Tutorial: Build a lane detector. [online] Available at: https://towardsdatascience.com/tutorial-build-a-lane-detector-679fd8953132 [Accessed 11 February 2021].
11. Arduino Project Hub. 2021. Lane Following Robot using OpenCV. [online] Available at: https://create.arduino.cc/projecthub/Aasai/lane-following-robot-using-opencv-da3d45 [Accessed 15 February 2021].
12. Hassan, M., 2021. self-driving-car-using-raspberry-pi. [online] Available at: https://www.murtazahassan.com/courses/self-driving-car-using-raspberry-pi/ [Accessed 15 February 2021].
13. Desegur, L., 2021. A Lane Detection Approach for Self-Driving Vehicles. [online] Medium. Available at: https://medium.com/@ldesegur/a-lane-detection-approach-for-self-driving-vehicles-c5ae1679f7ee [Accessed 22 March 2021].

14. Tian, D., 2021. DeepPiCar — Part 4: Autonomous Lane Navigation via OpenCV. [online] Medium. Available at: https://towardsdatascience.com/deeppicar-part-4-lane-following-via-opencv-737dd9e47c96 [Accessed 24 March 2021].
15. Assets.publishing.service.gov.uk. 2021. Reported road casualties in Great Britain: 2019 annual report. [online] Available at: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/922717/reported-road-casualties-annual-report-2019.pdf [Accessed 18 May 2021].
16. Buckley N, Sherrett L, Secco EL, A CNN sign language recognition system with single & double-handed gestures, IEEE Signature Conference on Computers, Software, and Applications, 2021, accepted
17. K. Myers, E.L. Secco, A Low-Cost Embedded Computer Vision System for the Classification of Recyclable Objects, Congress on Intelligent Systems (CIS - 2020), Intelligent Learning for Computer Vision, Lecture Notes on Data Engineering and Communications Technologies 61
18. D. McHugh, N. Buckley, E.L. Secco, A low-cost visual sensor for gesture recognition via AI CNNS, Intelligent Systems Conference (IntelliSys) 2020, Amsterdam, The Netherlands
19. A.T. Maereg, Y. Lou, E.L. Secco, R. King, Hand Gesture Recognition Based on Near-Infrared Sensing Wristband, Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2020), 110-117, 2020