Lab Report 3 – Raspberry Pi and Computer Vision

By Junaid Afzal
eeyja9@nottingham.ac.uk
Student ID: 4326724

Applied Electrical and Electronic Engineering: Construction Project
(EEEE1002 UNUK) (FYR1 18-19) (H61AEE)

Wednesday 10th April 2019
Partner(s): Emmanuel Boateng, Matthew Haywood & Oliver Jennings

Contents

1. Abstract

The aim was to traverse a course by following lines of different colours on the floor; detect symbols that are placed around the track; and take the correct action depending on the symbol detected.

The audio amplifier based on the LM386 op-amp [1] worked as intended and could output audio information from the Raspberry Pi but, the quality of the audio could have been improved by taking measures to remove noise. A bi-directional MOSFET Voltage Level Converter design [2] using the 2N7000 MOSFET [3] initially showed every address in use which suggested SDA was always being pulled low and so the problem was that SDA was being shorted to ground due to poor soldering. When adding a circuit to power the Raspberry Pi from the car, the current protection circuit was being tripped as the amount of current being drawn was exceeding the amount be supplied by the battery. The current protection was set at 0.5 A but now with the Raspberry Pi connected it needed to be increased to 3 A.

The HC-SR04 ultrasonic sensor worked as planned and gave accurate distances if the object was within the 15° range [4]. The TM1637 display worked as proposed and displayed numbers and error codes clearly.

The 3D printed camera-servo bracket worked as intended but the camera kept falling off and tape had to be used to secure it. The car Arduino code worked as expected until the I^2C buffer wasn't emptied. For the sensor code, all the commands worked as intended but when accelerating on an incline the MPU-6050 sometimes gave inaccurate results. The quality of line following was very high as the car able to follow the black line through the whole course. The car could detect symbols if the noise was kept to a minimum but on the track the noise levels were too high and the car could not even see the symbol so to fix this, filters could be implemented.

This car reached level 2 due to its automated steering and accelerating capabilities. To improve to level 3 the car would have to be able to detect the symbol and the car Arduino I^2C buffer problem would need be fixed.

2. Introduction

The aim was to traverse the track in Figure 1 by following the line on the floor. The car should detect a pink square; stop; look up at the symbol; determine which symbol it is; and thus, what it should do next. Starting from the start line in Figure 1, the car must follow a black line then see the symbol for counting shapes and turn left to count the number of shapes and notify the user. Next the car must see the symbol for following a red line and then follow the red line. If it doesn't see this symbol it will inadvertently take the long way around with the black line. Either way the next symbol is the follow black line symbol; then it is to put a ball in the goal net; then it is to measure and notify the user of the incline of the slope. Then it is to follow the green line and if the car doesn't detect it then it will take the long way around again. The symbol after that is to follow a black line. Then it is to wait until the traffic light goes green before continuing to follow black line. The final symbol is to measure the distance to a wall that is to the right of the car and notify the user.

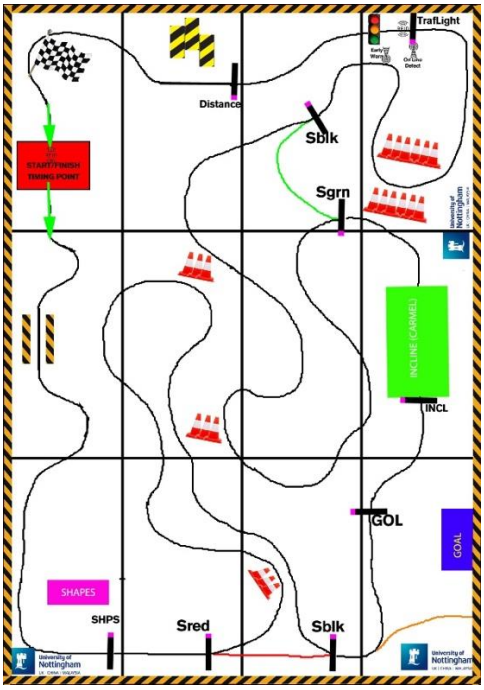


Figure 1 – Layout of final track

In Section 3 there will be a discussion on an audio amplifier circuit based on the LM386 op-amp [1]; an I^2C voltage level shifter with accompanying I^2C code; and a circuit to safely power the Raspberry Pi from the car's battery.

In Section 4 there will be a discussion on how the HC-SR04 works; accompanying code; and how the HC-SR04 was attached to the car. The car had to determine the distance it was from a wall as part of one of the challenges. The method of determining the distance had to be not-contact, reliable and easy to implement. Therefore, an ultrasonic sensor was chosen. The HC-SR04 has a measuring range of 2-200 cm and measuring angle of 15° [4], which makes it suitable for the challenge.

In Section 5 there will be a discussion on the library that was chosen for computer vision; the camera and servo system; and the ability of the computer vision system to follow a line of multiple colours and detect symbols. "Computer vision is a field of computer science that works on enabling computers to see, identify and process images in the same way that human vision does, and then provide appropriate output. Computer vision's goal is not only to see, but also process and provide useful results based on the observation." [5]

In Section 6 there will be a discussion on the use of the TM1637 display and some possible improvements. The 4-digit seven segment TM1637 LED display was chosen for the custom component proposal for its versatility of potential use – displaying distances, angles, number of shapes seen, run time error codes and general run time signifiers.

3. Raspberry Pi

The Raspberry Pi is palm-sized computer with that runs a Linux based operating system. The I/O consist of four USB ports, a HDMI port, an ethernet port, a 3.5 mm audio jack, a micro USB port for power in, a CSI camera connector, a DSI display connector, a micro SD slot and 40 GPIO pins [6].

The Arduino can update faster to real time events as it has no operating system but, the user is restricted to the Arduino IDE and its adaptation of the C language. On the Raspberry Pi the language and IDE can be changed to whatever is suitable but, it cannot update as quickly as the Arduino due to the layer of operating system. The Raspberry Pi has networking capabilities from the ethernet jack and Wi-Fi chip; a user interface via the Linux based operating system; and higher raw performance from its ARM processor [6]. For this reason, the Arduino should be used for low level tasks, such as getting information from sensors, and the Raspberry Pi for high level tasks, such as computation and networking of information.

The Raspberry Pi was setup using an already available SD card image [7] which was transferred to an SD card using Win32 Disk Imager software. This image had pre-installed all the programs that were required such as Code::Blocks [8] with the relevant libraries and VNC Viewer [9] to allow for remote use of the Raspberry Pi.

In this section there will be a discussion on an audio amplifier circuit based on the LM386 op-amp [1]; an I^2C voltage level shifter with accompanying I^2C code; and a circuit to safely power the Raspberry Pi from the car's battery.

3.1 Design

3.1.1 Audio Amplifier Design

A minimal audio amplifier circuit design [10] was chosen due to restrictions in board space. The gain of the circuit is set using pins one and eight [1]. The gain of a circuit determines the range of volume settings available, in this case it defaults to 0-20, while the volume setting is the selection of a gain value. Pin two and four were connected to ground, pin six was connected to 5 V and pin 3 was wired up to a 3.5mm audio jack which would receive analogue audio information from the Raspberry Pi's audio jack.

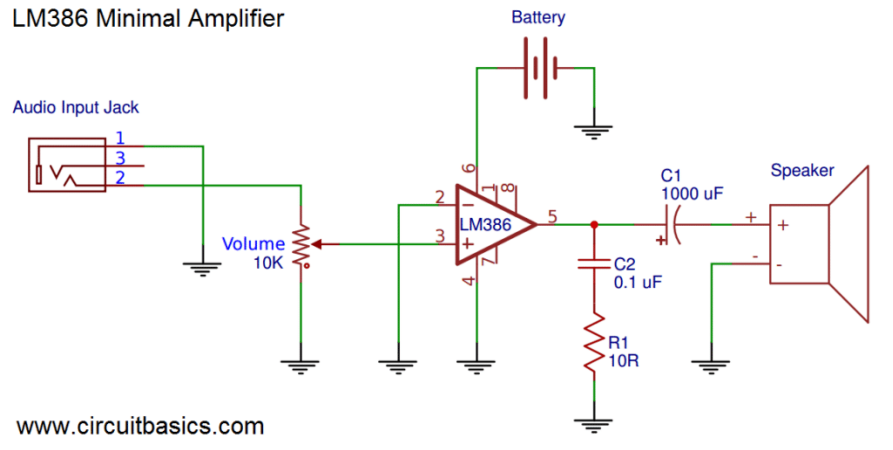


Figure 2 – Minimal audio amplifier design [10]

3.1.2 I^2C Voltage Level Shifter Design

The Raspberry Pi uses 3.3 V for I^2C communication, and has no protection circuitry, while the Arduino uses 5 V for I^2C communication. Therefore, an I^2C voltage logic level shifter is needed as protection for the Raspberry Pi.

A bi-directional MOSFET Voltage Level Converter design [2] using the 2N7000 MOSFET [3] was selected for the simple design which helps the restricted board space. This would convert a 3.3 V high of the Raspberry Pi in to a 5 V high for the Arduino and a 5 V high of the Arduino in to a 3.3 V high for the Raspberry Pi. The circuit would need to be constructed for both SDA and SCL.

“When the low side (3.3V) device transmits a '1' (3.3V), the MOSFET is tied high (off), and the high side sees 5V through the R2 pull-up resistor. When the low side transmits a '0' (0V), the MOSFET source pin is grounded and the MOSFET is switched on and the high side is pulled down to 0V.” [2]

“When the high side transmits a '0' (0V) the MOSFET substrate diode conducts, pulling the low side down to approx. 0.7V, this is also low enough to turn the MOSFET on, further pulling the low side down. When the high side transmits a '1' (5V) the MOSFET source pin is pulled up to 3.3V and the MOSFET is OFF.” [2]

The pi2c.h library was used for the in-built functions that would allow for data to be sent to the Arduino. First an object of class `Pi2c` needs to be created with its I^2C address. Then you call the member function `i2cWriteArduinoInt` to send an integer to the Arduino. If there is an error in sending an integer to the Arduino then this function will return -1 and if there is no error it will return 0, therefore it was placed in an if statement with the appropriate error message.

```
#include "pi2c.h"

Pi2c car(0x22);

if (car.i2cWriteArduinoInt(25) < 0)
	std::cout << "Error in sending integer to arduino" << std::endl;
```

Figure 3 – The main elements from the Pi2c.h library

3.1.3 Design for Circuit to power Raspberry Pi

The only way to safely power the Raspberry Pi was through the micro USB connector. It was made sure then that the Raspberry Pi was not back-powered via the 5 V header pin from the I^2C connectors as this would bypass its protection circuits. The outer pins of a female USB connector were soldered to 5 V and ground from the I^2C connector. The two inner connection were unused as they are for data transmission.

3.2 Results

3.2.1 Audio Amplifier Results

The design was first constructed and tested on breadboard to ensure correct operation and then soldered on to the breakout board of the Raspberry Pi. The LM386 was placed in to a socket so that is could be replaced in case of a fault. The existing I^2C connections provide the 5 V and ground connections.

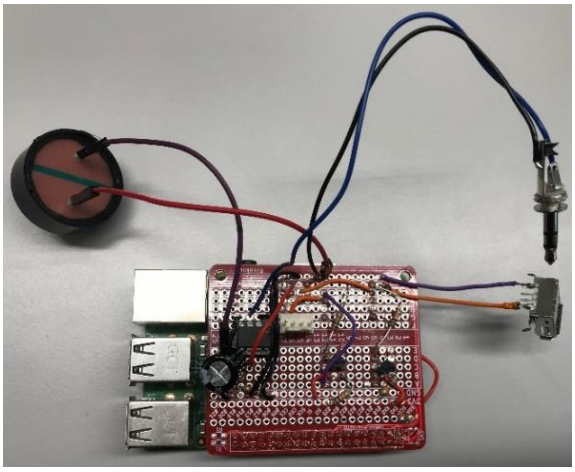


Figure 4 – Minimal audio amplifier circuit on Raspberry Pi breakout board

The circuit was tested by inputting a sinewave, triangle wave and a square wave using a signal generator. The input and output were recorded using a calibrated oscilloscope and the gain was unchanged for all tests. For all three cases the circuit outputted an in phase square wave with a higher amplitude that changed proportionally with the frequency of the input signal. These results confirm the correct operation of the minimal audio amplifier circuit [10].

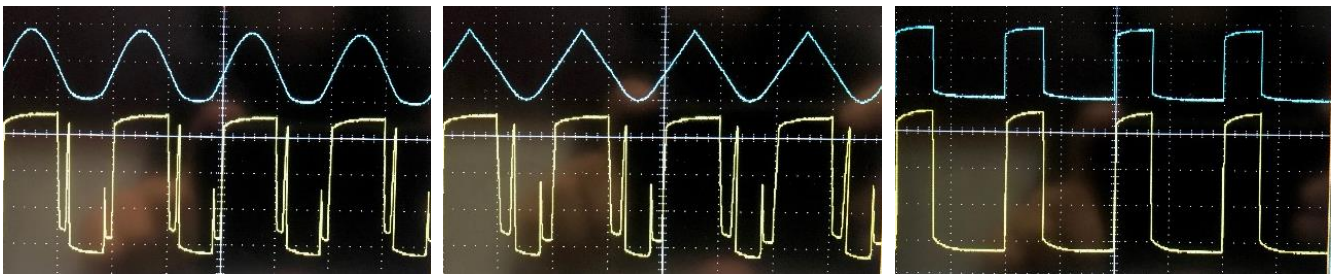


Figure 5 – Oscilloscope input signals (blue) and output signals (yellow). Note that the signals are shifted vertically apart so they can be seen clearer and does not represent their actual vertical positions

3.2.2 I²C Voltage Level Shifter

The program 'sudo i2cdetect -y 1' was used in the terminal and would display all I²C addresses that were in use but, when using it all addresses would appear to be in use. The code would compile, run and give no rum time errors but no information would be received by the Arduino.

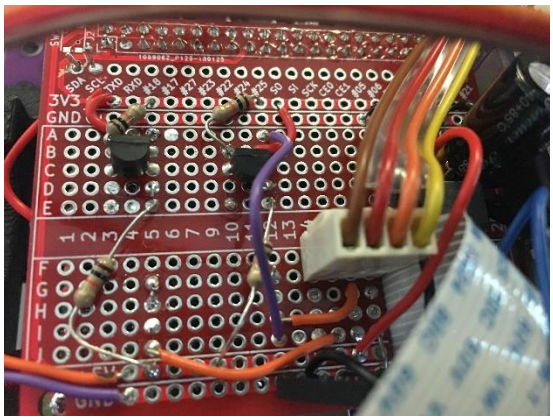


Figure 6 – I²C logic level shifter

3.2.3 Circuit to power Raspberry Pi

The current protection circuit on the mainboard of the car would trip at every attempt to turn the Raspberry Pi on.

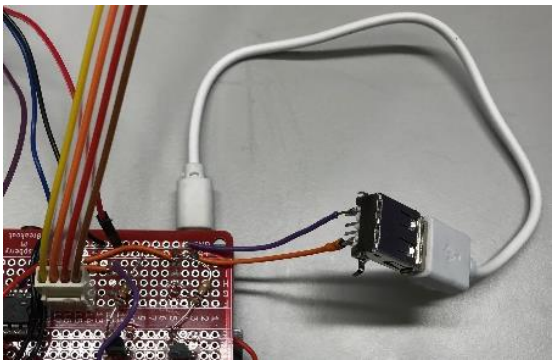


Figure 7 – Circuit to power Raspberry Pi

3.3 Discussion

3.3.1 Audio Amplifier

The audio amplifier worked as intended and could output audio information from the Raspberry Pi but, the quality of the audio could have been improved by taking measures to remove noise. Decoupling capacitors could have been used between the positive and negative power ends of the power supply, with a relatively high value capacitor (100 µF) to filter low frequency noise and a relatively low value capacitor (0.1 µF) to filter high frequency noise [10]. Addition of a capacitor between positive input signal and ground would have filtered any radio interference picked up by the audio input [10]. The separation of the audio input signal ground and other grounds would have decreased noise as other grounds tend to be quite noisy [10]. Also, the volume could have been controlled by a potentiometer by placing it in between pins one and eight [10].

These changes would have improved the audio quality but increased the complexity of soldering on the breakout board – where space is already limited – and thus the probability of errors.

3.3.2 I²C Voltage Level Shifter

The existence of a device on an address is found by checking if SDA has been pulled low – as it is usually floating high. So, if every address is in use then SDA is always being pulled low and so the problem was that SDA was being shorted to ground due to poor soldering.

After this 'sudo i2cdetect -y 1' showed the three expected addresses – the car Arduino, sensor Arduino and MPU6050 – and the I²C code was able to send data over I²C to the Arduino.

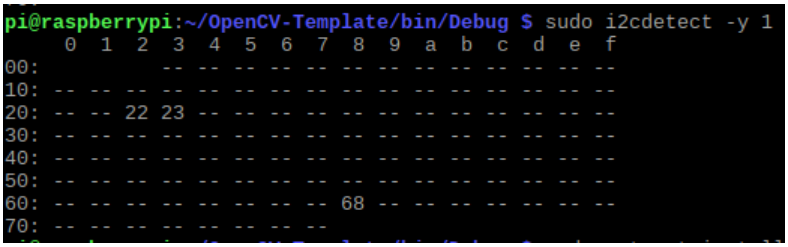


Figure 8 – 'sudo i2cdetect -y 1' showing the three expected addresses - the car Arduino, sensor Arduino and MPU6050

3.3.3 Circuit to power Raspberry Pi

The current protection circuit was being tripped as the amount of current being drawn was exceeding the amount be supplied by the battery. The current protection was set at 0.5 A but now with the Raspberry Pi connected it needed to be increased to 3 A.

$$Current\ Limit\ (A) = \frac{Set\ Point\ Voltage\ (V)}{100 \times 0.004}$$

Figure 9 – Current limit equation [7]

Using the equation in Figure 9 and with a current limit of 3 A, a set point voltage of 1.2V was chosen. This voltage would be measured between TP10 and ground and increased by adjusting the RV2 potentiometer [7]. After this change the Raspberry Pi would turn on and the current limit was not tripped.

4. Distance Measurement System

The car had to determine the distance it was from a wall as part of one of the challenges. The method of determining the distance had to be not-contact, reliable and easy to implement. Therefore, an ultrasonic sensor was chosen. The HC-SR04 has a measuring range of 2-200 cm and measuring angle of 15° [4], which makes it suitable for the challenge.

In this section there will be a discussion on how the HC-SR04 works, accompanying code and how the HC-SR04 was attached to the car.

4.1 Design

Figure 10 shows that the one half of the HC-SR04 consists of a transmitter of ultrasonic waves and the other half is a receiver of ultrasonic waves. Figure 11 shows that when the TRIG pin gets a 10 µs high pulse, the transmitter sends out eight 40 kHz ultrasonic waves. The ECHO pin is immediately set to high and the length of time that the ECHO pin is high is equal to the time the ultrasonic waves from the transmitter take to travel to the object and reflect back to the receiver. The speed of the wave is equal to the speed of sound in air, as the ultrasonic wave is a sound wave. The distance of an object can now be determined with the speed of the wave and the time taken by the wave.

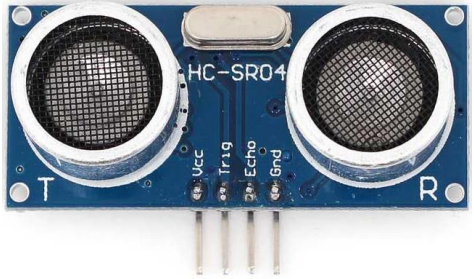


Figure 10 – Example of transmitter (left) and receiver (right) of ultrasonic waves [11]

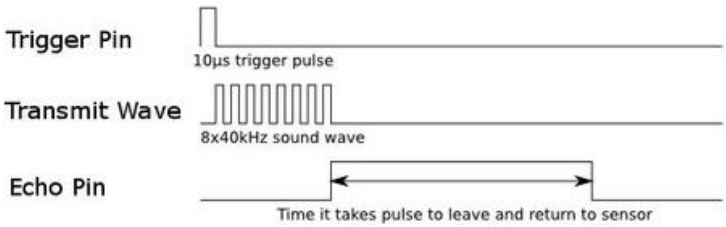


Figure 11 – HC-SR04 timing diagram [4]

The ultrasonic sensor was connected to the sensor Arduino instead of the Raspberry Pi for accuracy reasons explained in the introduction of section 3. Raspberry Pi. The HC-SR04 was placed in sockets, that were soldered to the board, so that the it can be removed, reused or moved. The Vcc, ground, TRIG and ECHO pins were connected to 5V, ground, D7 and D8 respectively on the sensor Arduino. In the final code the SR04.h library was used to simplify the code but, Figure 12 shows the low-level code that the SR04.h library uses.

In Figure 12 the ECHO pin is set as an input so that it can record the time taken for the wave to travel to the object and back and the TRIG pin is set as an output so it can be set high for 10 µs to trigger the transmitter to send out eight 40 kHz waves. The TRIG pin is set low for 2 µs to remove any noise so that the TRIG pin is set to high for exactly 10 µs. Next the duration for the waves to travel to the object and back is recorded using the `pulseIn()` function. The first variable of the function is the pin that will be timed and the second is when the timing should begin. The function in this context, second variable being `HIGH`, waits for the ECHO pin to go from LOW to HIGH and then times how long it takes for the pin to go LOW again. The function returns the time in microseconds; works on pulses from 10 µs to 3 minutes; and returns 0 if no complete pulse was detected.

To calculate the distance the duration must be divided by two, as it accounts for the return time as well, and the speed of sound must be in cm/µs, as duration is in µs and centimetres is a more useful metric. This gives the distance equation in Figure 12 and Figure 13.

```
const int EchoPin = 8;
const int TriggerPin = 7;

void setup {
  pinMode (EchoPin, INPUT);
  pinMode (TriggerPin, OUTPUT);
}

void loop {
  digitalWrite(triggerPin, LOW)
  delayMicroseconds(2);

  digitalWrite(triggerPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(triggerPin, LOW);

  float duration = pulseIn(echoPin, HIGH);
  float distance = duration*0.01715

  delay(50)
}
```

Figure 12 – Low-level code for measuring distance using a HC-SR04 sensor

$$Distance = Time \times Speed = \frac{duration}{2} \times \frac{343m/s \times 100}{1 \times 10^6} = duration \times 0.01715$$

Figure 13 – Derivation of distance equation in Figure 12

4.2 Results

The wiring and soldering were successful and the HC-SR04 was able to calculate the distance of an object in front of it, with minimum values of 2-3 cm and maximum values of 180-190 cm.

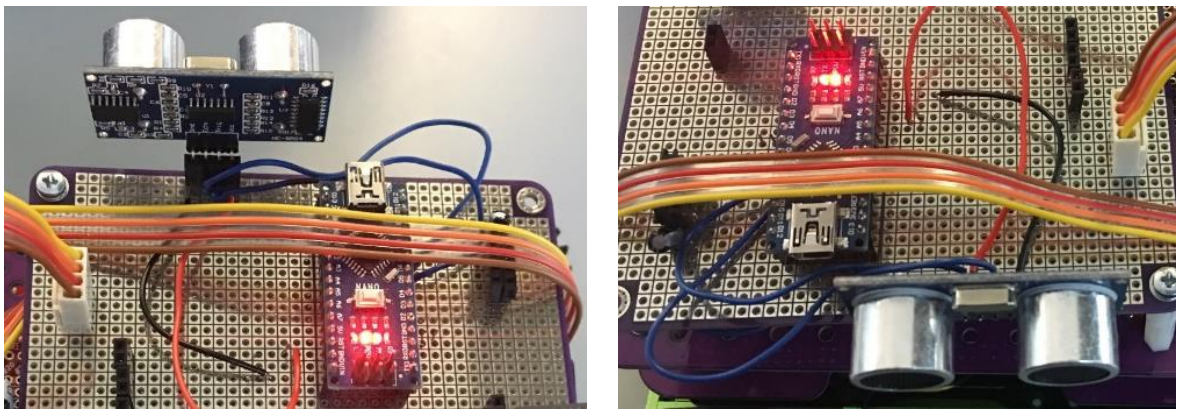


Figure 14 – Stripboard layout of HC-SR04 and sensor Arduino

4.3 Discussion

The HC-SR04 ultrasonic sensor worked as intended and gave accurate distances if the object was within the 15° range [4]. However, the mounting of the sensor was not very portable and required unsoldering and re-soldering of wires for it to be moved elsewhere on the car. Instead a ribbon cable, like the one use for I^2C communication, should have been used to connect the HC-SR04 to the sensor Arduino and 3D printed bracket should have been made so it could have been mounted anywhere on the car.

5. Computer Vision System

"Computer vision is a field of computer science that works on enabling computers to see, identify and process images in the same way that human vision does, and then provide appropriate output. Computer vision's goal is not only to see, but also process and provide useful results based on the observation." [5]

In this section there will be a discussion on the library that was chosen; the camera and servo system; and the ability of the computer vision system to follow a line of multiple colours and detect symbols.

5.1 Design

The Open Source Computer Vision library (OpenCV) version 3.4.0 [12] contains hundreds of computer vision algorithms and thus will be used for the software of the computer vision system.

The Raspberry Pi Camera Module [13] was chosen to be the camera that would enable the computer vision software to see. The camera was made specifically for the Raspberry Pi as it attaches to the CSI port on the Raspberry Pi with a 15 cm ribbon cable and produces high quality images with its Sony IMX219 sensor [14].

The SG90 servo [15] was chosen for its good compatibility with Arduino IDE and many libraries to simplify the coding process. The servo and camera would mounted in a 3D printed bracket shown in Figure 15 and this will allow the camera to be tilted down for line following mode and tilted up for symbol detection mode – see section 2. Introduction for full list of aims.

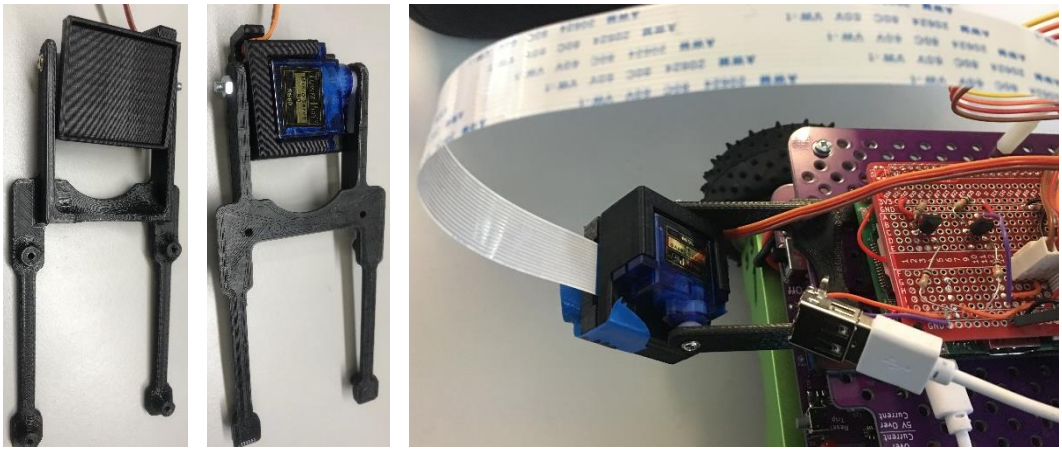


Figure 15 – 3D printed camera-servo bracket top-side (left), bottom side (middle) and mounted on car with Raspberry Pi attached (right)

Sensor Code

The sensor Arduino code, see Appendix C, consists of a switch statement on the variable `Value` within the `void loop()`. This variable contains the integer sent from the Raspberry Pi which will dictate to the sensor Arduino what commands it should execute. The variable `Value` is read in the `receiveEvent` function. `int temp` is used to clear the I^2C buffer as an integer is two bytes long while `Wire.read()` can only read one byte. Since Raspberry Pi will never send a number exceeding 255, the second byte can be discarded. `Value` is always reset to zero, which represents that the sensor Arduino is idling, as the switch command is in a loop and a decision was made that the raspberry Pi should control whether it wants the command to be looped. `Value` is initialised to zero on start up to avoid any errors.

The sensor Arduino has four commands that it can execute. The first it to move the camera up to 90° for symbol detection. The second is to move the camera down to 15° for line following mode and was chosen as it was the best comprise between accurate line following and being able to see the pink square in advance. The third is to use the HC-SR04 ultrasonic sensor to measure the distance to an object and display it on the seven segment TM1637 display. The fourth is to use the MPU-6050 [16] to measure the incline the car is on and display it on the seven segment TM1637 display.

Car Code

The purpose of the car Arduino code, see Appendix D, was to follow a line and starts with reading in `Speed` and `temp` from the `receiveEvent` function. The Raspberry Pi will send integers from 0-510. However, the value of `Speed` that is read in will always be a number from 0-255 but, the value of `temp` will be zero for 0-255 and one for 256-510. The value of `temp` being zero means the direction must be right and one means direction must be left. This method was chosen as sending chars or multiple integers always resulted in problems with I^2C.

Then within `void loop()` if statements check if `speed` is less than the threshold of 64. If it is then the car travels in a straight line at `BaseSpeed`. If the car has exceeded the threshold and thus shown enough deviation from the line, then the car will stop and rotate about its centre until it re-enters the threshold and thus returns to the line.

If the threshold value is too low then the car will continually oscillate about the line, as it becomes too reactive to the line, and struggles to meet the threshold with each turn. If the threshold value is too high then the probability that the car is unable to react to line changes quick enough increases, as the car becomes too unreactive. Thus, a threshold of 64 was the best comprise. The `BaseSpeed` should be as fast as possible but if it is too fast the car will not be able to react to line changes quick enough. So, a `BaseSpeed` of 64 was seen to match that criteria the best. If the speed of rotation is too low, the car will struggle to rotate and become stuck in a loop. If the speed of rotation is too high, the car will tend to overshoot the line. This will cause it to enter a positive feedback loop of oscillation around the line until it deviates fully from the line – at which point the car will stop. Therefore, a speed of 96 was the best compromise.

Raspberry Pi Code

The Raspberry Pi code, see Appendix B, start with the `setup()` function which configures the camera to capture images, also known as frames, with a resolution of 320x240 and sends a command to the sensor Arduino to tilt the camera down, which consists of sending a two over I^2C , so it can follow a line. The code then enters a `while(1)` loop with three core components.

The first is the capturing of the frame, which is done by the `CaptureFrameFromCamera()` function. This captures a still image of the camera and deposits it in to a matrix that holds image data.

The second is the detection of a pink square and consequently the triggering of the symbol detection routine. This is achieved by the statement `if (NumberOfPinkPixels(frame) >= 250)`. The `NumberOfPinkPixels()` function takes the frame and converts it to black and white by swapping pink pixels with white pixels and non-pink pixels with black pixels – example shown in Figure 16. This is achieved with the `GetFrameHSVPink` function which defines a range for hue, saturation and value. If a pixel in the frame is within the range, then it becomes white and if not, it becomes black. The `NumberOfPinkPixels()` function then returns the number of white pixels, and thus pink pixels, using the OpenCV standard function of `countNonZero()`.

250 pink pixels was a good threshold as if it was too low then potential noise could trigger the symbol detection routine and if it was too high then the car's distance to the square would be too close.

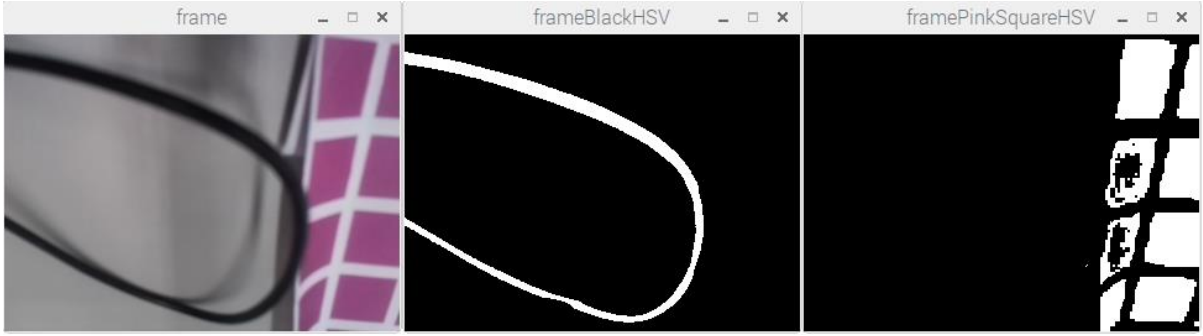


Figure 16 – Example of isolating black (left) with GetFrameHSVForBlack function and pink (right) with GetFrameHSVForPink function. Note that the same can be achieved for red and green with GetFrameHSVForGreen and GetFrameHSVForRed

If the number of pink pixels is greater than or equal to 250 then the car is stopped using the `StopCar()` function, which sends the car Arduino a speed of zero; the camera is tilted up using the `TiltCameraUp()` function, which sends the sensor Arduino a command of one; the symbol detection function is called, which returns an integer between 0-7, representing the command to be executed; and the camera is tilted back down using the `TiltCameraDown()` function, which send the sensor Arduino a command of two.

The `DetectSymbol()` function is a while loop with the termination condition being that the `MaxPercentageMatch` is greater than 67%. Within the while loop a frame is captured again using the `CaptureFrameFromCamera()` function. The frame is then passed through a `GaussianBlur` function to remove noise from the frame. The frame is then converted to black and white, with the pink pixels being isolated, using the previously explained `GetFrameHSVForPink` function. Next all possible contours within the black and white image are found with `findContours()` function. Then all contours all looped though and the one with the biggest area is found using the function `contourArea()`. This is the contour that represents the bounding box of the symbol. This contour is then simplified to a contour with four points with the `approxPolyDP()` function. The simplified contour; black and white frame; and dimensions of the frame are then passed to the function `transformPerspective()`. This function resizes the frame around the contour and then re-shapes the frame to a rectangular shape. The 8 symbol files from Figure 17 are then loaded in. A for loop cycles through each symbol file and converts it to black and white, to isolate the pink, with the `GetFrameHSVForPink` function. Then the percentage match is calculated with the `compareImages` function between the black and white symbol file and the frame that was returned by the `transformPerspective()` function. The `MaxPercentageMatch` is along with the associated symbol file, and thus the command, is recorded. The `MaxPercentageMatch` is then checked to see if it exceeds the threshold of 67%. If it does, then the function returns the command value associated with the symbol and if not, then the process is repeated again.

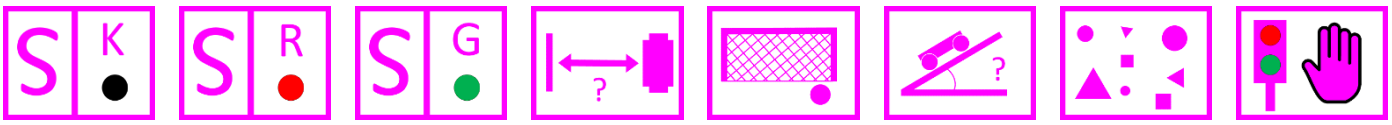


Figure 17 – The eight symbols that need to be detected from left to right – follow black line, follow red line, follow green line, measure distance, football, measure incline, count shapes, stop light

The third and final component of the `while(1)` loop in the main function is the switch statement on the value that the `DetectSymbol()` function returns. On start up the variable `Command` is initialised to zero, which means that the car will follow a black line until the first pink square is seen. For cases 0, 1 and 2 the car follows a line but for a different colour. These functions are all the same but with the difference being the colour that is isolated when creating the black and white image. This is the same process when detecting the pink square but for the respective colour of the line. Once the black and white image is produced it is sent to a function called `CalculateSpeedAndDirectionForLineFollowing`.

The for loop in Figure 18 scans from the left-hand side to the right-hand side of the frame, until it meets a white pixel, which represents the line, at which point it will exit the for loop. This gives the distance that the line is from the left-hand side of the frame. The is repeated for scanning from the right-hand side to the left-hand side, with the for loop starting from 319 and decrementing instead. Next there is a series of `if` and `else if` statements to determine where the line is with respect to the frame. If the distance from the left = distance from the right = 320 or 1 then the line could not be found, and the speed is set to zero. If the distance from the left is bigger than distance from the right, then the difference between them is found and is a number from 0-319. This number is then converted to a number between 0-255 if the car needs to turn right or 256-511 for if the car needs to turn left. The speed is then sent to the car Arduino over I^2C.

```
uchar* p = frameHSV.ptr<uchar>(1);
int DistanceFromLeft;

for(int x = 0, DistanceFromLeft = 0; x < frameHSV.cols; x++, DistanceFromLeft++)
    if(p[x] == 255)
        break;-
```

Figure 18 – Code from CalculateSpeedAndDirectionForLineFollowing for finding position of line from the left-hand side

For the distance measurement case the function `MeasureDistance()` is called which sends a three to the sensor Arduino and the `FollowBlackLine()` is called to keep the car following a black line. The same is done for the `MeasureIncline()` function except a four is sent to the sensor Arduino. The functions for `Football()`, `CountShapes()` `StopLight()` and were not completed in the allocated time – see section 5.3 Discussion for possible solutions.

5.2 Results

The camera-servo mount worked fine but the camera kept falling off the bracket due to the lack of any attachment mechanism, so tape was used to secure it.

Sensor Code

When the integers from the Raspberry Pi were sent over I^2C to the sensor Arduino, the commands were executed, and the sensor Arduino returned to idle – as planned. When accelerating on an incline the MPU-6050 sometimes gave inaccurate results.

Car Code

When sent a value from 0-510 the car reacted as intended but, sometimes the I^2C buffer was not cleared before leaving the `receiveEvent` function. This meant that subsequent `Speed` values were not read in and the car was stuck in whatever mode the last `Speed` value was. The only way to fix this was to reset the car Arduino.

Raspberry Pi Code

The car followed a line; detected the pink square from enough distance away from the symbol; tilted the camera up; had the symbol in the frame but, there was too much noise and the symbol could never be detected. However, in controlled conditions on a table, where the noise levels were lower, the camera could detect the symbol and matched it with the correct symbol files – Figure 19. This lack of detection also caused the car to get stuck in a loop from which it cannot exit. The `FollowBlackLine()`, `FollowRedLine()`, `FollowGreenLine()`, `MeasureDistance()` and `MeasureIncline()` functions worked as intended when activated manually.

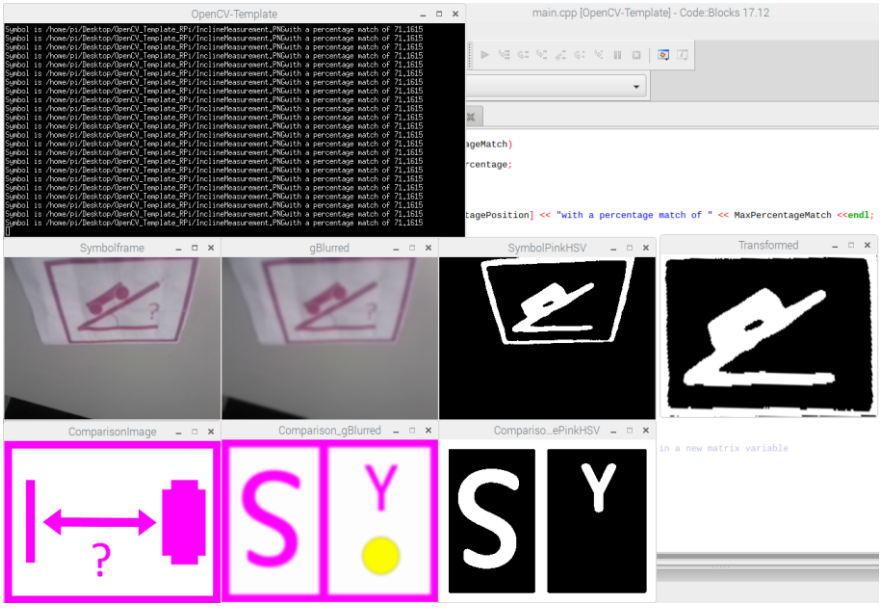


Figure 19 – DetectSymbol() function when on a table

5.3 Discussion

The 3D printed bracket worked as intended but the camera kept falling off and tape had to be used to secure it. To fix this screw holes could be incorporated in to the design and the bracket could be screwed together.

Car Code

The Car code worked as expected until the buffer was not emptied. To fix this a while loop at the end of the `receiveEvent` function could be used to ensure the buffer is empty before leaving the function.

Sensor Code

All the commands worked as intended but when accelerating on an incline the MPU-6050 sometimes gave inaccurate results. To improve the accuracy of the MPU-6050 the car should stop when on the slope. This could be coded by having the car travel for a certain distance forwards when it detects the incline symbol and stopping to record the angle.

Raspberry Pi Code

The quality of line following was very high as the car able to follow the black line through the whole course. The car could detect symbol if the noise was kept to a minimum but on the trac the noise levels were too high and the car could not even see the symbol so to fix this, filters could be implemented. Also due to time constraints some functions were not completed so here are some possible solutions.

For the `Football()` function, code would be developed to be able to identify the goal posts and then identify the ball. Next you would keep adjusting the position of the car until the ball was in front of the car and in between the goal posts. Then the car would go forwards and push the ball into the goal. Finally, the car would turn around; find the line; and continue to follow the black line.

For the `CountShapes()` function, the car would turn and keep adjusting until the shapes were in front of the camera and clear to see. Then blue pixels would be isolated as the frame of the wall of shapes is blue; call the `transformPerspective()` function to crop unwanted background information; isolate pink pixels on the cropped image; draw contours; simplify the contours to remove rough edges; then count the number of points for each contour, to give the number of corners for each shape.

For the `StopLight()` function the trafflight would be cropped with the `transformPerspective()` function so that only the traffic light is being examined. Then while it saw a certain number red pixels and while it saw less than a certain number of green pixels, then the car would remain stationary. Technically only one is neccesary but in a real life circumstance it is best to use both just in case the traffic lights are faulty.

SAE Level of autonomy

This car reached level 2 due to its automated steering and accelerating capabilities. To improve to level 3 the car would have to be able to detect the symbol and the car Arduino I^2C buffer problem would need be fixed.

6. Custom Component Proposal

The 4-digit seven segment TM1637 LED display was chosen for the custom component proposal for its versatility of potential use – displaying distances, angles, number of shapes seen, run time error codes and general run time signifiers.

In this section there will be a discussion on the use of the TM1637 display and some possible improvements.

6.1 Design

The display has four pins with CLK and DIO connected to D4 and D3. The `<TM1637Display.h>` library was used to call the `showNumberDec()` function with its only argument being the number to be displayed. The display was mounted in a socket, which was then soldered to the sensor Arduino stripboard, so that the display could be removed, reused or moved.

6.2 Results

The wiring and soldering were successful and the TM1637 display was able display all distances, angles, run time error codes and general run time signifiers.

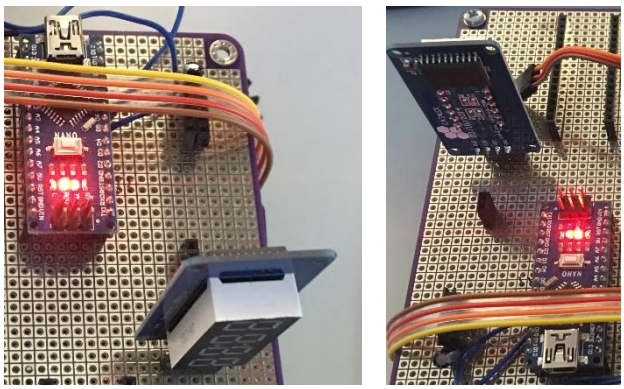


Figure 20 – Stripboard layout of the TM1637 display and sensor Arduino

6.3 Discussion

The TM1637 display worked as proposed and displayed numbers and error codes clearly. However, the mounting of the display was not optimal as it was perpendicular to the car and directed to one side of the car. This made the display difficult to see at times. To improve this a 3D printed bracket should have been made so the display could be mounted near the front of the car and connected to the sensor Arduino with a ribbon cable, like the one use for I^2C communication.

Conclusion

The audio amplifier worked as intended and could output audio information from the Raspberry Pi but, the quality of the audio could have been improved by taking measures to remove noise. Decoupling capacitors could have been used between the positive and negative power ends of the power supply, with a relatively high value capacitor (100 µF) to filter low frequency noise and a relatively low value capacitor (0.1 µF) to filter high frequency noise [10]. Addition of a capacitor between positive input signal and ground would have filtered any radio interference picked up by the audio input [10]. The separation of the audio input signal ground and other grounds would have decreased noise as other grounds tend to be quite noisy [10]. Also, the volume could have been controlled by a potentiometer by placing it in between pins one and eight [10]. These changes would have improved the audio quality but increased the complexity of soldering on the breakout board – where space is already limited – and thus the probability of errors.

Initially every address appeared to be in use which suggested SDA was always being pulled low and so the problem was that SDA was being shorted to ground due to poor soldering. After SDA was disconnected from ground 'sudo i2cdetect -y 1' showed the three expected addresses – the car Arduino, sensor Arduino and MPU6050 – and the I^2C code was able to send data over I^2C to the Arduino.

The current protection circuit was being tripped as the amount of current being drawn was exceeding the amount be supplied by the battery. The current protection was set at 0.5 A but now with the Raspberry Pi connected it needed to be increased to 3 A. Using the equation in Figure 9 and with a current limit of 3 A, a set point voltage of 1.2V was chosen. This voltage would be measured between TP10 and ground and increased by adjusting the RV2 potentiometer [7]. After this change the Raspberry Pi would turn on and the current limit was not tripped.

The HC-SR04 ultrasonic sensor worked as planned and gave accurate distances if the object was within the 15° range [4]. However, the mounting of the sensor was not very portable and required unsoldering and re-soldering of wires for it to be moved elsewhere on the car. Instead a ribbon cable, like the one use for I^2C communication, should have been used to connect the HC-SR04 to the sensor Arduino and 3D printed bracket should have been made so it could have been mounted anywhere on the car.

The 3D printed camera-servo bracket worked as intended but the camera kept falling off and tape had to be used to secure it. To fix this screw holes could be incorporated in to the design and the bracket could be screwed together.

The Car code worked as expected until the buffer was not emptied. To fix this a while loop at the end of the `receiveEvent` function could be used to ensure the buffer is empty before leaving the function.

For the sensor code, all the commands worked as intended but when accelerating on an incline the MPU-6050 sometimes gave inaccurate results. To improve the accuracy of the MPU-6050 the car should stop when on the slope. This could be coded by having the car travel for a certain distance forwards when it detects the incline symbol and stopping to record the angle.

The quality of line following was very high as the car able to follow the black line through the whole course. The car could detect symbol if the noise was kept to a minimum but on the trac the noise levels were too high and the car could not even see the symbol so to fix this, filters could be implemented. Also due to time constraints some functions were not completed so here are some possible solutions.

For the `Footbal()` function, code would be developed to be able to identify the goal posts and then identify the ball. Next you would keep adjusting the position of the car until the ball was in front of the car and in between the goal posts. Then the car would go forwards and push the ball into the goal. Finally, the car would turn around; find the line; and continue to follow the black line.

For the `CountShapes()` function, the car would turn and keep adjusting until the shapes were in front of the camera and clear to see. Then blue pixels would be isolated as the frame of the wall of shapes is blue; call the `transformPerspective()` function to crop unwanted background information; isolate pink pixels on the cropped image; draw contours; simplify the contours to remove rough edges; then count the number of points for each contour, to give the number of corners for each shape.

For the `StopLight()` function the trafflight would be cropped with the `transformPerspective()` function so that only the traffic light is being examined. Then while it saw a certain number red pixels and while it saw less than a certain number of green pixels, then the car would remain stationary. Technically only one is neccesary but in a real life circumstance it is best to use both just in case the traffic lights are faulty.

The TM1637 display worked as proposed and displayed numbers and error codes clearly. However, the mounting of the display was not optimal as it was perpendicular to the car and directed to one side of the car. This made the display difficult to see at times. To improve this a 3D printed bracket should have been made so the display could be mounted near the front of the car and connected to the sensor Arduino with a ribbon cable, like the one use for I^2C communication.

This car reached level 2 due to its automated steering and accelerating capabilities. To improve to level 3 the car would have to be able to detect the symbol and the car Arduino I^2C buffer problem would need be fixed.

References

[1] "LM386 Op-Amp Datasheet," [Online]. Available: http://www.alldatasheet.com/datasheet-pdf/pdf/8887/NSC/LM386.html.

[2] "HobbyTronics Bi-Directional MOSFET Voltage Level Converter," [Online]. Available: http://www.hobbytronics.co.uk/mosfet-voltage-level-converter.

[3] "2N7000 MOSFET Datasheet," [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/2842/MOTOROLA/2N7000.html.

[4] "HC-SR04 Datasheet," [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf.

[5] "Computer Vision Definition," [Online]. Available: https://www.techopedia.com/definition/32309/computer-vision.

[6] "Raspberry Pi Official Page," [Online]. Available: https://www.raspberrypi.org/.

[7] University of Nottingham,.

[8] "Code::Blocks Official Page," [Online]. Available: http://www.codeblocks.org/.

[9] "VNC Viewer Official Page," [Online]. Available: https://www.realvnc.com/en/connect/download/viewer/.

[10] "Minimal Audio Amplifier Design," [Online]. Available: https://www.circuitbasics.com/build-a-great-sounding-audio-amplifier-with-bass-boost-from-the-lm386/.

[11] "HC-SR04 Picture," [Online]. Available: https://www.bing.com/images/search?view=detailV2&id=30CC1865B20FC4BF5089E63B62257F69A6F05731&thid=OIP.lRWyW8ZdG1VxEvTFZU0RggHaHa&mediaurl=http%3A%2F%2Fkits4u.in%2Fwp-content%2Fuploads%2F2017%2F09%2Fhc-sr04-ultrasonic-range-finder-2.png&exph=900&expw=900&q.

[12] "Open Source Computer Vision Library," [Online]. Available: https://docs.opencv.org/3.4.0/index.html.

[13] "Raspberry Pi Camera Module," [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/.

[14] "Sony IMX219 Sensor," [Online]. Available: https://www.sony-semicon.co.jp/products_en/new_pro/april_2014/imx219_e.html.

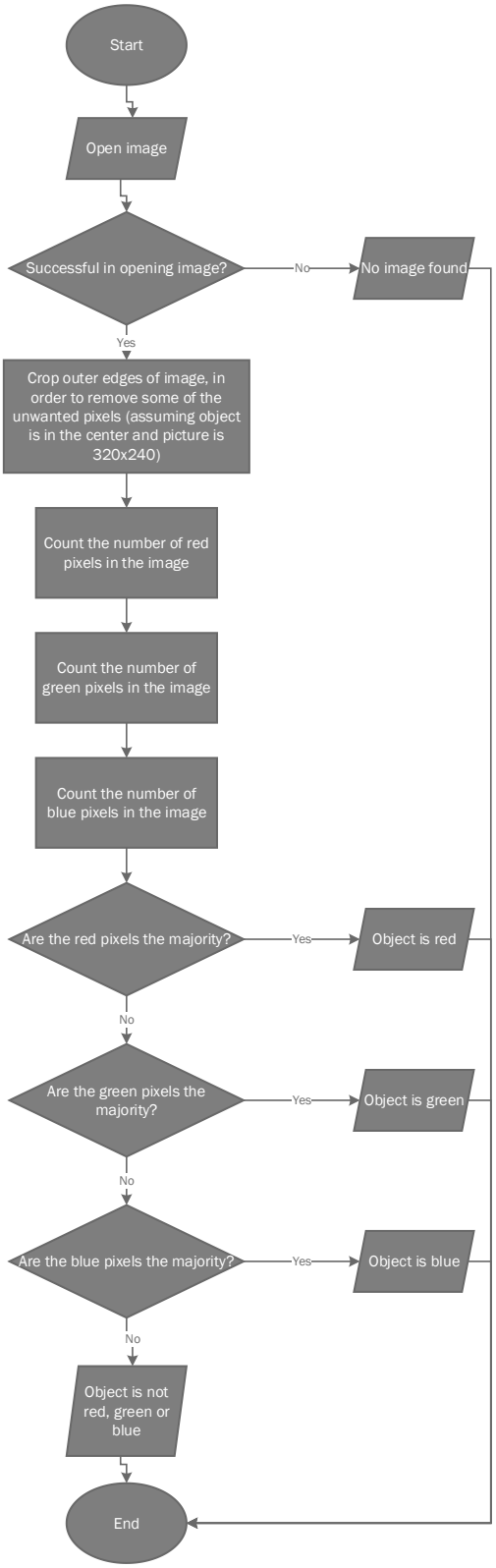
[15] "SG90 Servo Datasheet," [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf.

[16] "MPU-6050 Datasheet," [Online]. Available: https://www.alldatasheet.com/datasheet-pdf/pdf/517744/ETC1/MPU-6050.html.

Appendix A

(OpenCV Report)

This program can identify the colour of an object – assuming the image size is 320x240 and the object is close to the center. The program contains a for loop which is not shown on the flowchart. This loop cycles through the six example images that were provided –and prints to the screen the colour of the object and the number of red, green and blue pixels that the program has detected.



Appendix B

(Raspberry Pi Main Code)

```
#include <iostream>
#include "opencv_aee.hpp"
#include "main.hpp"
#include <stdlib.h>

int main(int argc, char** argv) {
    setup();    // Call a setup function to prepare IO and devices

    int Command = 0;

    while(1) {    // Main loop to perform image processing
        Mat frame = CaptureFrameFromCamera();

        /*Check if the pink square is present*/
        if (NumberOfPinkPixels(frame) >= 250) {
            StopCar();

            TiltCameraUp();

            Command = DetectSymbol();

            TiltCameraDown();
        }

        /*Work out the correct command */
        switch (Command) {
        case 0:
            std::cout << "Folllow black line command" << std::endl;
            FollowBlackLine(frame);
            break;

        case 1:
            std::cout << "Red short cut command" << std::endl;
            FollowRedLine(frame);
            break;

        case 2:
            std::cout << "Green short cut command" << std::endl;
            FollowGreenLine(frame);
            break;

        case 3:
            std::cout << "Distance measuremnt command" << std::endl;
            MeasureDistance();
            FollowBlackLine(frame);
            break;

        case 4:
            std::cout << "Football command" << std::endl;
            Football();
            FollowBlackLine(frame);
            break;

        case 5:
            std::cout << "Incline measurement command" << std::endl;
            MeasureIncline();
            FollowBlackLine(frame);
            break;
```

```
        case 6:
            std::cout << "Shape counting command" << std::endl;
            CountShapes();
            FollowBlackLine(frame);
            break;

        case 7:
            std::cout << "Stop light command" << std::endl;
            StopLight();
            FollowBlackLine(frame);
            break;

        default:
            std::cout << "Error in switch statement for command" << std::endl;
            break;
        }

        cv::imshow("frame", frame);

        int key = cv::waitKey(1);   // Wait 1ms for a keypress (required to update windows)
        key = (key==255) ? -1 : key;    // Check if the ESC key has been pressed
        if (key == 27)
            break;
    }

    closeCV();  // Disable the camera and close any windows

    return 0;
}
```

Appendix C

(Sensor Arduino Code)

```
/*Libraries*/
#include <Wire.h>
#include <Servo.h>
#include <TM1637Display.h>
#include <MPU6050.h>
#include "SR04.h"
#include "Sensor_Template.h"

/*Pin Declarations*/
const int TrigPin = 7;
const int EchoPin = 8;
const int CLK = 4; //Set the CLK pin connection to the display
const int DIO = 3; //Set the DIO pin connection to the display
const int ServoPin = 2; //Brown = GND    Red = 5V    Orange = D9

SR04 MySR04 = SR04(EchoPin, TrigPin);
Servo CameraServo;
TM1637Display DistanceDisplay (CLK, DIO);
MPU6050 MyMPU6050;

int Value = 0, PiDistance;

void setup() {
  Serial.begin (9600);
  Wire.begin(SENSOR);               /*Join I2C as the sensor*/
  Wire.onReceive(receiveEvent);

  CameraServo.attach(ServoPin);
  DistanceDisplay.setBrightness(0x0a); //set the diplay to maximum brightness
  MyMPU6050.begin(MPU6050_SCALE_2000DPS, MPU6050_RANGE_2G);
}


void loop() {
  switch(Value) {
    case 0:
      Serial.println("Idle");
      break;


    case 1:
      Serial.println("Moving camera up");
      CameraServo.write(90);
      break;


    case 2:
      Serial.println("Moving camera down");
      CameraServo.write(15);
      break;


    case 3:
      Serial.print("Measuring distance and displaying");
      PiDistance = MySR04.Distance();
      DistanceDisplay.showNumberDec(PiDistance);
      Serial.println(PiDistance);
      break;
```

```
    case 4:
      Serial.println("Measuring angle and displaying");
      Vector normAccel = MyMPU6050.readNormalizeAccel();
      int Angle = (atan2(normAccel.XAxis,sqrt(normAccel.YAxis*normAccel.YAxis+
normAccel.ZAxis*normAccel.ZAxis))*180.0)/M_PI;
      DistanceDisplay.showNumberDec(Angle);
      Serial.print("  Angle = ");
      Serial.print(Angle);
      break;

    default:
      Serial.println("Error in Value");  //turn off display
      break;
  }

  Value = 0;

  delay(10);
}

void receiveEvent(int howMany) {
  Value = Wire.read();
  int temp = Wire.read();

  delay(10);
}
```

Appendix D

(Car Arduino Code)

```
/*Libraires*/
#include <Wire.h>
#include <H61AEE_S02.h>
#include "Car_Template.h"

/*Global Variables*/
char Direction;                                 /*Line Following Mode*/
int Speed = 0, BaseSpeed = 64, Sum, Difference;   /*Line Following Mode*/

void setup() {
  car.setupVehicle();
  car.enableMotors(true);
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);
  Wire.begin(CAR);
  Wire.onReceive(receiveEvent);
  Serial.begin(9600);
}

void loop() {
  if (Speed == 0)
    car.setSpeed(ALL, 0);

  else if (Speed < 64) {
    car.setDirection(ALL, forwards);
    car.setSpeed(ALL, BaseSpeed);
  }

  else {
    switch (Direction) {

      case 'R':
        car.setDirection(RIGHT, forwards);
        car.setDirection(LEFT, backwards);
        car.setSpeed(RIGHT, 96);
        car.setSpeed(LEFT, 96);
        break;

      case 'L':
        car.setDirection(RIGHT, backwards);
        car.setDirection(LEFT, forwards);
        car.setSpeed(RIGHT, 96);
        car.setSpeed(LEFT, 96);
        break;

      default:
        Serial.println("Line Following Error");
        break;
    }
    delay (10);
  }
}


void receiveEvent(int howMany) {
  Speed = Wire.read();
  int temp = Wire.read();

  if (temp == 0)
    Direction = 'R';
```

```
  else if (temp == 1)
    Direction = 'L';

  Serial.print(" Speed = ");
  Serial.print(Speed);
  Serial.print("  temp = ");
  Serial.println(temp);

  delay(10);
}
```