DRIVER TRAFFIC VIOLATION DETECTION AND DRIVER RISK CALCULATION THROUGH REAL-TIME IMAGE PROCESSING

by

Fritz Sutherland

Submitted in partial fulfillment of the requirements for the degree
Master of Engineering (Electronic Engineering)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology

UNIVERSITY OF PRETORIA

August 2017

SUMMARY

DRIVER TRAFFIC VIOLATION DETECTION AND DRIVER RISK CALCULATION THROUGH REAL-TIME IMAGE PROCESSING

by

Fritz Sutherland

Supervisor(s):	Dr. H.C. Myburgh
Department:	Electrical, Electronic and Computer Engineering
University:	University of Pretoria
Degree:	Master of Engineering (Electronic Engineering)
Keywords:	Driver rating, Image processing, Camera, Hough circle transform, Blob detector, Road traffic sign detection, Neural network

ABSTRACT

Road safety is a serious problem in many countries and affects the lives of many people. Improving road safety starts with the drivers, and the best way to make them change their habits is to offer incentives for better, safer driving styles.

This project aims to make that possible by offering a means to calculate a quantified indicator of how safe a driver's habits are. This is done by developing an on-board, visual road-sign recognition system that can be coupled with a vehicle tracking system to determine how often a driver violates the rules of the road. The system detects stop signs, red traffic lights and speed limit signs, and outputs this data in a format that can be read by a vehicle tracking system, where it can be combined with speed information and sent to a central database where the driver safety rating can be calculated.

Input to the system comes from a simple, standard dashboard mounted camera within the vehicle, which generates a continuous stream of images of the scene directly in front of the vehicle. The images

are subjected to a number of cascaded detection sub-systems to determine if any of the target objects (road signs) appear within that video frame. The detection system software had to be optimized for minimum false positive detections, since those will unfairly punish the driver, and it also had to be optimized for speed to run on small hardware that can be installed in the vehicle.

The first stage of the cascaded system consists of an image detector that detects circles within the image, since traffic lights and speed signs are circular and a stop sign can be approximated by a circle when the image is blurred or the resolution is lowered.

The second stage is a neural network that is trained to recognize the target road sign in order to determine which road sign was found, or to eliminate other circular objects found in the image frame. The output of the neural network is then sent through an iterative filter with a majority voted output to eliminate detection 'jitter' and the occasional incorrect classifier output.

Object tracking is applied to the 'good' detection outputs and used as an additional input for the detection phase on the next frame. In this way the continuity and robustness of the image detector are improved, since the object tracker indicates to it where the target object is most likely to appear in the next frame, based on the track it has been following through previous frames.

In the final stage the detection system output is written to the chosen pins of the hardware output port, from where the detection output can be indicated to the user and also used as an input to the vehicle tracking system.

To find the best detection approach, some methods found in literature were studied and the most likely candidates compared. The scale invariant feature transform (SIFT) and speeded up robust features (SURF) algorithms are too slow compared to the cascaded approach to be used for real-time detection on an in-vehicle hardware platform. In the cascaded approach used, different detection stage algorithms are tested and compared. The Hough circle transform is measured against blob detection on stop signs and speed limit signs. On traffic light state detection two approaches are tested and compared, one based on colour information and the other on direct neural network classification.

To run the software in the user's vehicle, an appropriate hardware platform is chosen. A number of promising hardware platforms were studied and their specifications compared before the best candidate

was selected and purchased for the project. The developed software was tested on the selected hardware in a vehicle during real public road driving for extended periods and under various conditions.

LIST OF ABBREVIATIONS

ADC	Analog to Digital Converter
CHG	Circular Hough Transform
CSG	Constructive Solid Geometry
EKF	Extended Kalman Filter
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GIL	Global Interpreter Lock
GPU	Graphical Processing Unit
HOG	Histograms of Oriented Gradient
JDL	Joint Directors of Laboratories
JPG/JPEG	Image representation format (Joint Photographic Experts Group)
MPEG	Video representation format (Moving pictures Experts Group)
MTBF	Mean Time Between Failure
NOOBS	New Out Of Box Software
RBG	Red, Blue and Green (colour representation format)
RGB	Red, Green and Blue (colour representation format)
RGBA	Red, Green, Blue and Alpha (colour representation format)
SIFT	Scale Invariant Featue Transformation
SIR	Sample Importance Resample
SSE	Sum of the Squared Error
SURF	Speeded Up Robust Features
SVM	Support Vector Machine
U-SURF	Upright Speeded Up Robust Features

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION **1**
1.1 PROBLEM STATEMENT 1
1.1.1 Context of problem 1
1.1.2 Research motivation 2
1.2 RESEARCH QUESTIONS 2
1.3 HYPOTHESIS AND APPROACH 3
1.4 RESEARCH GOALS 3
1.5 RESEARCH CONTRIBUTION 4
1.6 OVERVIEW OF STUDY 4

CHAPTER 2 LITERATURE STUDY **5**
2.1 OBJECT DETECTION AND RECOGNITION 5
2.1.1 Aspects 6
2.1.2 Object detection and recognition systems 9
2.1.3 Feature extraction techniques 11
2.1.4 Object representation 16
2.1.5 Object detection methods and algorithms 18
2.1.6 Implementation components 23
2.2 OBJECT TRACKING 25
2.2.1 The setup 25
2.2.2 Object tracking: Types of objects tracked 27
2.2.3 Object tracking: Types of tracking filters 28
2.2.4 Object tracking: Types of data association methods 29
2.2.5 A more practical look at object tracking in videos using a Kalman filter . . . 29
2.2.6 A more practical look at object tracking in videos using a particle filter . . . 33

2.3 SENSOR FUSION . . . 35
2.3.1 Benefits and drawbacks . . . 35
2.3.2 Methods and Algorithms . . . 38
2.3.3 Fusion model architecture . . . 47
2.4 EXISTING ROAD TRAFFIC SIGN DETECTION SYSTEMS . . . 54
2.5 CONCLUDING REMARKS . . . 54

CHAPTER 3 METHODS AND CONTRIBUTION . . . 55
3.1 ALGORITHM STRUCTURE OPTIONS . . . 55
3.2 DESIGN GOALS . . . 56
3.2.1 Execution speed . . . 57
3.2.2 Low rate of false positives . . . 57
3.2.3 System cost . . . 57
3.2.4 (Embedded) Hardware platform . . . 57
3.3 SYSTEM IMPLEMENTATION OVERVIEW . . . 58
3.4 SUBSYSTEM DETAIL . . . 58
3.4.1 Dashboard camera . . . 58
3.4.2 Object detection . . . 60
3.4.3 Object tracking . . . 61
3.4.4 Object recognition . . . 62
3.4.5 Outputs . . . 63
3.4.6 Sensor fusion . . . 63
3.4.7 Software . . . 65
3.4.8 (Embedded) Hardware platform . . . 65
3.5 IMPLEMENTATION . . . 69
3.5.1 Target object: Stop signs . . . 69
3.5.2 Target object: Traffic lights . . . 74
3.5.3 Optimisation of the Hough circle transform parameters . . . 78
3.5.4 Data-set for neural network training . . . 78
3.5.5 Training the neural network . . . 82
3.5.6 System outputs . . . 83
3.5.7 Combining the sub-systems to build a complete real-time system . . . 86
3.6 ACHIEVING A REAL-TIME FRAME RATE . . . 86

3.6.1 Capturing method . . . 88
3.6.2 Codec . . . 91
3.6.3 Splitting into threads . . . 92
3.6.4 Creating a custom video stream to grab video frames . . . 94
3.6.5 Using a different driver and a compiled language . . . 95
3.7 COMPARING DETECTOR METHODS FOR STOP SIGNS . . . 96
3.7.1 SURF vs Hough circle transform . . . 96
3.7.2 Hough circle transform vs Blob detection . . . 97
3.8 (RE-)COMPARING DETECTOR METHODS ON SPEED LIMIT SIGNS . . . 99
3.8.1 Data collection method . . . 100
3.8.2 Neural network training . . . 100
3.9 COMPARATIVE TRAFFIC LIGHT STATE DETECTION . . . 101
3.9.1 Traffic light state by colour information . . . 102
3.9.2 Traffic light state by neural network . . . 105
3.10 CONCLUDING REMARKS . . . 107

CHAPTER 4 RESULTS AND DISCUSSION . . . 109
4.1 INITIAL CANNY-HOUGH-NEURAL NETWORK STOP SIGNS DETECTION . . . 109
4.2 INITIAL CANNY-HOUGH-NEURAL NETWORK TRAFFIC LIGHTS DETECTION 111
4.3 RESULTS OF COMPARING DETECTOR METHODS FOR STOP SIGNS . . . 113
4.3.1 Algorithm accuracy . . . 113
4.3.2 Algorithm speed . . . 114
4.4 RESULTS OF (RE-)COMPARING DETECTOR METHODS FOR SPEED SIGNS . . . 114
4.4.1 Data collection results . . . 114
4.4.2 Neural network training . . . 115
4.4.3 Performance test . . . 116
4.5 COMPARATIVE TRAFFIC LIGHT STATE DETECTION RESULTS . . . 119
4.5.1 Processing time . . . 119
4.5.2 Detection and classification accuracy . . . 120
4.5.3 Results analysis . . . 122
4.5.4 Performance results from other methods in literature . . . 123
4.6 FRAME-RATE IMPROVEMENT RESULTS . . . 124
4.7 PROJECT GOALS FULFILLMENT . . . 125

4.8 CONCLUDING REMARKS 125

CHAPTER 5 CONCLUSION 126

REFERENCES 129

CHAPTER 1 INTRODUCTION

1.1 PROBLEM STATEMENT

1.1.1 Context of problem

One of the leading causes of death in the world today is road accidents. Many of these accidents are caused by reckless and negligent drivers. In order to improve road safety, some initiatives are emerging to encourage good driver behavior and safe driving habits. For example, some insurance houses have started offering incentives for good drivers, like lowered premiums or cash-back bonuses. However, this approach is based on rewarding only good driving behavior, which makes it fundamentally dependent on some form of rating or score to be determined for each driver.

Safe driving habits can be measured in a number of ways. Most insurance houses already make use of a person's background, age, social standing and accident history, which is only a generalised measure of risk. This information can be combined with a driver safety rating to give a better indication of driver risk, and also gives the driver the opportunity to improve his/her rating by adopting safer driving habits. However, at the moment the only driver behavior monitoring being used involves recording the acceleration, braking and cornering of the vehicle and analysing trends on this data to get a more quantified measurement of how a vehicle is driven. This only indicates the aggressiveness of the driver and thus only part of the whole risk profile. There must be more ways in which safe driving habits can be measured, in order to get a more complete indication of driving habits.

One proposed way to determine how safe a driver's habits are, is to measure adherence to the rules of the road, and especially safety critical rules such as stops and traffic lights [1] [2], most of these rules are indicated by road traffic signs next to the road. In order to detect when a driver violates the

rules indicated by these signs, a road traffic sign detection sensor is required, and thus the focus of this project will be to develop and optimise such a sensor (specifically for stop signs, traffic lights and speed signs). This sensor will enable the visual detection of these signs to be combined with driving parameters such as speed and location in order to yield an indication of how well the driver complies with some of these indicated rules, however, the details of calculating the actual driver rating is beyond the scope of this project.

1.1.2 Research motivation

Many vehicles on South Africa's roads already have a vehicle tracking system installed since most insurance companies now require it. This allows for the addition of a road traffic sign detection system to expand upon the capability of these tracking devices, which can be combined with existing techniques to calculate a more accurate driver safety rating. Insurance companies can then use this driver safety rating to motivate drivers to adopt safer driver habits, which in turn will lead to fewer accidents and safer roads overall.

This can be achieved by developing a device that collects visual data about the road ahead of the vehicle using a dashboard mounted camera, and using this data on an on-board embedded processing system where real-time image processing is combined with machine learning and data fusion techniques to detect various road traffic signs under various conditions [3].

1.2 RESEARCH QUESTIONS

The research questions are:

1. Can certain safety critical road traffic signs be detected from a vehicle dashboard mounted camera?
2. Can the software for such a detection system be developed using existing image processing techniques, while allowing optimisation for execution speed and also customisation to detect specific road signs?
3. Which image processing method is the most suitable for detecting road signs such as stop signs, and speed limit signs?

4. If the image processing method can be adapted to detect traffic lights, is there also a way (or ways) to detect the current state of a traffic light (which light is now on).
5. Is there standard off-the-shelf hardware available that is small enough to be fitted in a user's vehicle, and yet powerful enough to run such detection software in real-time?

1.3 HYPOTHESIS AND APPROACH

The project makes the hypothesis that by finding and using, adapting or combining existing image processing techniques, and that by a combination of software optimisation and choosing the right hardware, a road traffic sign detection system can be implemented to run in a user's vehicle to process images from a dashboard mounted camera in real time. The detection outcome must be output in a way that can be used by a vehicle tracking system where it can be combined with speed and location data to be sent to a central server.

To achieve this, the following approach was followed:

1. Study literature to find the image processing methods and algorithms that are most likely to work well with all of this project's goals.
2. Perform preliminary comparative tests to find which of these methods are most suitable.
3. Compare standard off-the-shelf hardware to find the best platform and camera for the project.
4. Optimise the software to detect stop signs, traffic lights (including current state) and speed limit signs with minimum false positives and maximum execution speed.

1.4 RESEARCH GOALS

The project aims to prove that an image processing based, in-vehicle road traffic sign detection system can be developed to enable the calculation of a driver safety rating. The goal is to detect stop signs, traffic lights (including current state) and speed signs with minimum false positives in a system that can be used as a sensor by other devices (such as a vehicle tracking system). It also aims to optimise the whole system so that it can run in real-time on an off-the-shelf computer system of reasonable cost and size.

1.5 RESEARCH CONTRIBUTION

The project will add to the knowledge on the use of image processing techniques for road traffic sign detection and the selection and optimisation of such techniques when implemented in software to be executed on constrained hardware.

1.6 OVERVIEW OF STUDY

The findings of a literature study on real-time image processing, data fusion and road sign detection are summarised in Chapter 2. Chapter 3 explains the methods used and the contributions made by the project, the results are discussed in Chapter 4.

CHAPTER 2 LITERATURE STUDY

In this chapter the findings of a literature study on the image processing topics of object detection and tracking, sensor fusion and road traffic sign detection are discussed.

2.1 OBJECT DETECTION AND RECOGNITION

Object detection consists of first finding features in the target image and in the scene image, and then common feature points must be matched (associated) between the two images.

The aim is to find an object detection technique that is both fast and has a very low rate of false positives (instances where the system indicates that it found the target object where it was in fact not the target object). The technique will be applied to find and recognise road-signs from a dashboard mounted camera feed in a vehicle.

It has to be fast since the video feed from the camera must be analysed at a sufficiently high rate so that road signs are not missed when the vehicle is traveling at a high speed. A vehicle traveling at 120 km/h covers a distance of around 33 metres every second. This means that if a road sign can only be recognised when it is less than 33 metres away and the process runs at a rate of less than 1 frame per second, then it is very possible that some road signs can be missed.

It is also required to have a low rate of false positives. Although it may have many application, the aim here is specifically to monitor driver behavior in order to assign drivers a rating. This rating may be used by insurance companies, potential second-hand car buyers and other institutions and it should not unfairly penalise a driver because of false road sign recognition outputs.

2.1.1 Aspects

The task of object detection from images with the aims mentioned above, involves the following considerations [4]:

2.1.1.1 Camera

Cameras have various settings and parameters that determine how it captures and converts the incoming light into image data. Although most of these parameters can be automatically and dynamically adjusted by the camera in most modern cameras, it still affects the way in which the scene is converted to pixel data, which in turn complicates the object recognition goal.

The camera's focus is also important. Although some object recognition techniques specifically blurs an image, it may become a great obstacle when the resulting image is already blurred due to the camera being unfocused.

Camera lenses may also cause blur. Most dashboard cameras makes use of a wide-angle (also called "fish-eye") lens. This has the advantage of capturing some parts of the scene next to the vehicle (and not just the small section right in front of it), but it significantly reduces the size of objects that are far away, similar to the effect of zooming out, and it also distorts the scene, especially near the edges. The effect on road sign detection is that when a sign is still far ahead it has little distortion but is too small for detection. As the vehicle approaches, the sign appears larger but more distorted, making it again difficult for detection.

This distortion near the edges is made even worse when the vehicle approaches the sign at high speed, since the optical flow near the frame boundary causes it to blur even more due to the vehicle's speed.

2.1.1.2 Scene

The camera's location and angle should also be taken into account since a filmed or captured scene is always taken from a certain viewpoint. The viewpoint determines the location, proportion and rotation

of objects in the scene. From one viewpoint, some object may be completely occluded by others, while from a different viewpoint other objects may be hidden. The angle also distorts the image that an object projects towards the camera. For instance a flat, round road sign has a nice round shape when viewed from an angle perpendicular to the face, but when changing to a greater viewing angle the shape becomes an ellipse.

Lighting, or the illumination of the scene, is another very important aspect and can cause certain parts of an image to be under- or over-exposed, eliminating some of the visual information from those parts of the image due to it being outside the range of the camera. Illumination also changes the colour information, since the same colour is represented differently when it is brightly illuminated or filmed in low light. In some extreme cases very low light even takes away the colour leaving only shades of grey, while looking directly into a very bright light like the sun, saturates the camera to the point where no image is recognisable. Thus the variation caused by illumination also complicates the object recognition goal.

2.1.1.3 Conversion and processing rate

A decent minimum object detection processing rate has to be achieved for the proposed application. Many object detection tasks requires it to be real-time in order to be useful. The actual rate is determined by the application, for instance an anti-missile system on an aircraft will demand a much faster rate than a system that counts the number of ripe fruits on a tree.

In this application, if the processing is not completed by the time the next frame should be displayed (as determined by the video's frame-rate) then the video will be slowed down, or in a real-time application some frames will be dropped (i.e. the ones that cannot be processed in time will be skipped).

The achievable rate is determined by the abilities of the hardware, the efficiency of the software and the method or algorithm used. The specific parameters of the system, like for instance image resolution, and also the conversion rate at which the source (camera) can capture and transmit the images influence the rate that can be achieved.

Recording frame-rates differ between videos taken with different equipment, and generally a higher

frame-rate leads to a better looking video. Of course as the frame-rate increases, the time between frames decreases, leaving less time to complete the object detection processing. Fortunately there is a work-around for this problem, when an object moves relatively slowly across the scene it will appear on several frames, each time at a slightly different position. This means that some frames can be skipped, since it is not necessary to detect the object in each one, as long as enough frames are processed to detect the object at least once. The object should be detected at least a few times in order to track it, when performing object tracking.

In the road sign application there is no need to display a high frame-rate video to a user, so the frames that are not processed can simply be dropped.

Another way to overcome the processing time problem is to work with lowered resolution. By moving each frame down the image resolution pyramid (re-sampling to reduce the resolution), the processing can be achieved on fewer pixels and so will take less time to complete.

The processing time required can also be minimised by optimising the code that does the processing for maximum speed.

Finally, if the detection performance is still too slow it may be necessary to run the code on better hardware.

2.1.1.4 Target variations

Another aspect complicating the object recognition task is that the target objects are not exactly the same. In most cases a class or group of objects should be detected and not just one specific instance of that object. Taking the road sign example, to detect a stop sign the system has to take into account that stop signs come in different sizes, the mounting pole heights sometimes differs, it may not face the camera directly and it may be bent, scratched, rusted or faded among many other variations.

2.1.2 Object detection and recognition systems

A typical object recognition and detection system comprises various aspects. Some systems are built very specific to their application, while others are more generic and can handle various tasks, scenes and objects. A generalised system is shown in Fig. 2.1, with the elements found in most object detection and recognition system. Each aspect thereof will be discussed briefly:

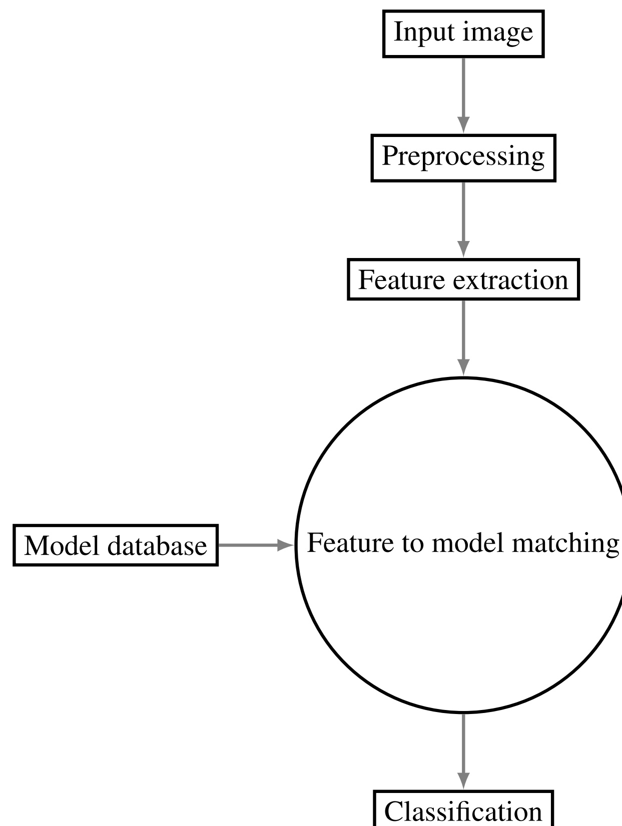

Figure 2.1. Generalised object detection system diagram

2.1.2.1 Input image

The input is usually a simple image taken by a camera or similar scanning device. The object detection algorithms work on the raw uncompressed image, where each pixel is represented in an array of either shades-of-grey (grey-scale) or three values representing the red, green and blue (RGB) colour mixtures of a colour image. The image dimensions, ratio and resolution are often fixed to match the algorithm or technique used. Successive images are analysed at a fixed rate for video analysis and this rate is

usually measured in frames per second. Some video based techniques require a certain depth of historic images to be kept for comparison.

2.1.2.2 Preprocessing

Sometimes some form of preprocessing is required to prepare the raw image data for the feature extraction method that is used. This may take the form of conversion, re-sizing, re-sampling or filtering or even a combination of these, depending on the feature extraction method used in the next step.

2.1.2.3 Feature extraction

This is the point where certain key elements of an image is looked for, which can be used to recognise the target object by comparing the key elements found in the image with the ones stored in the model database. The type of feature that is looked for and how it is found varies dramatically between the different feature extraction techniques. This will be discussed in more detail in the next section.

Many techniques require this step to be iterated over every pixel, or over many scrolling image windows of various scales and positions to determine whether that image portion contains any of the features looked for [5].

2.1.2.4 Model database

The model database is simply a record of the key features that is known for the target object. During the search for an object in the image, the key elements found in the image will be compared to the features listed in the database. The entries will take a form determined by the feature extraction technique that is used.

The database may contain more than one set of features for an object, for instance different perspectives of the same object if the technique used does not cater for that intrinsically.

It may also contain entries for various objects, if more than one target object should be detected.

2.1.2.5 Feature to model matching

This is where the features found in the image are compared to the reference features from the model database. Some techniques output a simple boolean variable to indicate a match or mismatch, while others may give a degree of match or a confidence factor.

Various features may be compared in this manner and the output of each is then passed to the next stage to make a decision based on these features matching [6].

2.1.2.6 Classification

At the classification stage, the results (outputs) of the feature matching stage, are evaluated to make a decision on whether the target object was found or not.

The extraction process may look at how well features A, B and C in each pixel or pixel window of the image matched the features A, B and C from the target object's entry in the model database. The result or results are combined here to make a classification decision on whether this image portion contains only background, or else which target object it contains.

The combination process may take one of many forms similar to sensor fusion, which depends to some degree upon the feature extraction technique that was used. For instance, say features A and B showed a match, while feature C gave a mismatch, using a majority vote the output will then be classified as a match, since both A and B indicated a match and only C did not. Similarly, when comparing two images a correlation coefficient (CC) can be derived from the matching features to indicate how similar the features of the two images are [7].

2.1.3 Feature extraction techniques

There are several techniques for extracting object features from images. The feature that is extracted, for instance colour, outline/contour, texture, etc. depends on the technique or algorithm used. A number of these will be discussed below.

2.1.3.1 Hough transform

The generalised Hough transform is a generalised application of the Hough transform to detect instances of certain shapes in images as shown by Ballard [8]. Ballard states that many objects can be recognised by the shape of their outlines or edge lines, which is also referred to as the "primal sketch".

This technique starts off by applying an edge operator on the original grey-scale image. The edge operator represents changes in the grey-scale to indicate the location and direction of edges in the image.

The generalised Hough algorithm makes use of this edge information to map the edge point's orientation to a reference point on the shape to be detected and then produces a rating of how well it matches the shape. The key to the algorithm being in the use of the directional (orientation) information, which is represented in the R-table. The R-table is constructed by tabulating R, which is the difference between the chosen reference point and any boundary point, as a function of the gradient direction ϕ. So the R-table's rows are indexed by ϕ and each row contains the spatial relation of any boundary point(s) to the reference point at that specific gradient direction. Using the R-table an accumulator array A is then updated by incrementing all the points $x+r$ for each of the edge pixels x.

2.1.3.2 Scale Invariant Feature Transformation (SIFT)

David Lowe's scale invariant feature transform [9] can be used to detect 3D objects in an image. The method is not sensitive to scaling, rotation and illumination changes. Detection is performed by the following staged filtering approach.

The first filtering stage makes use of a local maxima and minima from the output of a difference of Gaussian smoothing function to determine the key locations. This is calculated over an image pyramid at different levels of re-sampling. The 2D Gaussian's convolution can be applied by 2 passes of a 1D Gaussian function over the horizontal and vertical directions with a σ of $\sqrt{2}$.

After applying the difference of Gaussian function, the maxima and minima are determined by comparing each pixel to its eight neighbors. If it is indeed a maximum or minimum point then the

nearest point one level below and one level above in the image scaling pyramid are also compared to its eight neighbors. In this manner the local maxima and minima are located independent of scale.

In the second stage, a feature vector is then calculated for each of these key locations to describe the image in the vicinity of the key point. The resulting feature vectors are called SIFT keys and consists of the image gradient magnitude and orientation at the key point in the image. To make it less sensitive to illumination changes the gradient magnitudes are subjected to thresholding at 10% of the maximum gradient value, while the orientation is determined by the peak in a histogram of gradient orientations around the key point.

In the third stage, using a nearest neighbor approach, the SIFT keys are then used with a Hough transform hash table to match possible target objects. A bin size of 30 degrees for orientation, a factor of two for scale, and 0.25 times the maximum model dimension for location are used to allow for detection even with some geometric distortion.

If at least three keys agree on the presence of the object, each such possible match is further verified by finding a low-residual least-squares solution for the model parameters relating to the object in the image. Each match must agree within 15 degrees orientation, $\sqrt{2}$ change in scale, and location within 0.2 times maximum model size.

2.1.3.3 Speeded Up Robust Features (SURF)

SURF [10] is an object detection method based on SIFT, but with some key differences for improved processing speed.

SURF makes use of a different method to locate the key points, placing them at the maxima of the determinant of the Hessian matrix, while relying on integral images to reduce the computation time. The resulting detector is called a Fast-Hessian detector. SURF also builds the image scale pyramid by applying up-scaled square filters, instead of repeatedly down-scaling the image for each layer.

Upright SURF (U-SURF) is even faster but can only be used when it is known that the scene will be captured upright (i.e. camera rotation only allowed around the vertical axis). The orientation has to be

determined when not using upright SURF. This is achieved by fixing the orientation of a key point based on the dominant orientation from a circular region around the interest point.

To describe the area around key points in SURF, a square area centered on the key point and oriented along the dominant orientation (previously calculated) is used. The area is further divided into smaller sub-areas and a distribution of Haar-wavelet responses in the x and y directions is calculated over these and then summed.

During the matching stage, SURF makes use of the sign of the Laplacian to distinguish a bright object on a dark background from a dark object on a bright background, this further speeds up the process since only features with the same sign are considered.

2.1.3.4 Edge detection

There are various techniques to detect edges in images. Edge detection is a useful feature extraction method since it reduces the image to a "primal sketch" containing much less data than the original by leaving only the outlines of the objects in the image. A couple of methods will be discussed in more detail:

1. Canny:

   Introduced by John Canny [11] [12], the Canny edge detection method strives to detect lines or edges in images with low error rate, to mark each line or edge only once and to mark it as close as possible to the real line or edge in the image. The Canny edge detector is applied through the following steps:

   (a) Apply a Gaussian smoothing filter to the image.
   (b) Use a two dimensional first derivative operator to determine intensity gradients in the image.
   (c) A non-maximal suppression algorithm is applied within an upper and lower limit along the ridges in the intensity gradient image in order to "thin" the lines or edges, effectively getting rid of secondary weak edges along the already detected edges.

2. Marr–Hildreth:

   The Marr-Hildreth edge detector is a Laplacian of Gaussian method [13]. To apply it:

   (a) Smooth the original image by convolving it with a two dimensional Gaussian.
   (b) Take the second order (directional) derivative using the Laplacian (Mexican hat) operator. The edges, or intensity steps, are then represented by zero crossings in the result.

3. Deriche:

   Rachid Deriche [14] further improved on the Canny edge detector by using a two dimensional infinite extent filter which is recursively applied moving in opposite directions over the image. The method is sometimes referred to as the Canny-Deriche edge detector.

4. Roberts cross:

   An edge detection method by Lawrence Roberts [15] proposed in 1963. The method is fast and simple and makes use of the approximated gradient, which is computed by convolution of the image with the kernels $\begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix}$ and $\begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$. The method is sensitive to noise.

5. Sobel:

   The Sobel edge detector [16] uses an approach very similar to the Roberts cross method, except that 3 x 3 kernels are used with the x and y directions being $\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$ and $\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$. The wider kernels gives it better noise immunity at the expense of slightly more computation. A thinning algorithm may need to be applied to the resulting lines.

6. Prewitt:

   Another method using the convolution of a 3x3 kernel with the original image to determine image intensity gradients. The Prewitt operator kernels being [17]: $\begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}$ and

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix}.$$

2.1.4 Object representation

For an object to be found from an image and subsequently described, it has to be represented by some mathematical model. There are various ways of representing objects, a few of these are discussed briefly:

2.1.4.1 Object outline

The basic outline or boundary of an object often contains enough information necessary for humans to recognise the object [8]. In fact, the neuroscientist David Marr called it the primal sketch [13] and showed that its recognition is one of the early steps of human and mammalian visual systems. The outline or primal sketch is essentially the output when edge detection is applied to an original/raw image.

2.1.4.2 Graphical projection

An object may be represented by using images taken from multiple viewpoints. For example, taking the front-, side- and top-view of an object similar to the technique used in drafting. Storing all 3 views of an object may enable the object recognition algorithm to recognise the object irrespective of which side of the object is facing the observer (camera).

2.1.4.3 Constructive Solid Geometry (CSG)

CSG is a method whereby a complex shape is represented by combining simple geometric shapes. These simple shapes, called primitives, can be any shape that is simple to represent mathematically, like cubes, cylinders, spheres, cones, etc. The primitives can be combined by boolean operations, like addition (union), subtraction (difference) or taking their intersection areas to make up a complex shape

as illustrated in Fig. 2.2. Many CAD programs make use of this method and it can be applied as a feature for object detection, but since arbitrarily curved objects cannot be represented using just a few chosen primitives, CSG approaches are not very useful in object recognition on real world scenes [4].

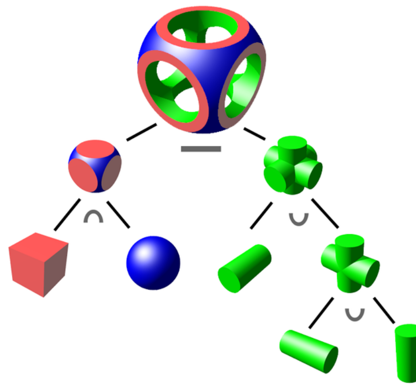

Figure 2.2. CSG composition

2.1.4.4 Voxel volumetric representation

A voxelized object is a 3D discrete representation of a continuous object on a regular grid of voxels [18]. Basically a 3D space is divided into a grid of boxes or cells and an object is then represented by the volume of these cells that it occupies.

2.1.4.5 Generalised cones/cylinders

Thomas Binford presented a way to represent objects by a set of generalised cylinders or cones. First, complex objects are segmented into sub-parts which are formed by simpler objects. These sub-parts are then described by a set of cones which consists of a space-curve called the axis and along this axis a couple of cross-sections normal to the axis [19]. An example is shown in Fig. 2.3.

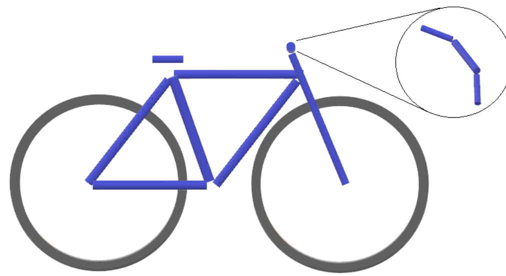

Figure 2.3. A bicycle in generalised cones

2.1.4.6 Simple models

To simply indicate the position of an object in an image, often a very simple marker is used to represent the object. These can be a mean point or centre-of-mass point, a circle or ellipse enclosing most of the object, a set of key points on the object or even an outline or silhouette.

2.1.5 Object detection methods and algorithms

An investigation into some of the popular methods and algorithms for object detection from images follows.

2.1.5.1 Viola-Jones

Viola and Jones [6] developed an image processing algorithm for object detection that makes use of three key concepts:

1. Integral image:

   The first concept is to do detection using features rather than the normal image where each pixel represents a brightness or colour value. The features used by the Viola Jones method are based on Haar basis functions and it is called the "integral image". The point ("pixel") value of an integral image is the sum (or integral) of the rectangle formed by all the pixels to the left and above the referred point. These rectangle features can be determined quickly from the original

image i by calculating the integral image ii at the point (X,Y) using the summing formula shown in Equation 2.1.

$$ii(X,Y) = \sum_{x,y}^{X,Y} i(x,y) \tag{2.1}$$

2. Feature selection by learning algorithm:

   The second is to use a learning algorithm to select a small number of visual features. The learning algorithm is based on adaBoost. AdaBoost combines a couple of weak classifiers to form a stronger classifier [6]. In other words a few simple classifiers with a low classification success rates are trained and used together by each making a weighted contribution which is added together and compared to a threshold value to make a strong classifier system in the form of a perceptron. In the end each weak classifier becomes one of the rectangle features from the integral image technique discussed above.

3. Cascading classifiers:

   Perhaps the most important concept is to cascade a number of simple classifiers. Each classifier layer decides whether the relevant portion of the image contains an object or only background. If this portion is classified as background no further processing is spent on it and the algorithm moves on to the next portion. On the other hand, when classified as an object it is passed on to the next classifier layer and the process repeats. This method wastes less processing time on "uninteresting" parts of the image, and the more complex classifiers only need to look at parts of the image that were not yet filtered out by the (earlier) simpler classifiers.

2.1.5.2 Rowley-Baluja-Kanade

The Rowley-Baluja-Kanade method [20] is an object detection method develop to detect human faces in digital images, but it can also be applied to detect other objects. The method consists of a small scanning window that outputs a portion of the image which is evaluated by a neural networks whose outputs are combined to classify whether the image portion contains a face or not.

The main challenge in this method is training the neural network, especially in obtaining the "negative" images, since an image that does not contain a face may contain just about anything else.

The method has the drawback that the neural network must evaluate the image over many positions and at many scaling levels, which is processing intensive.

The performance of this method reached around 77.9% to 90.3% correctly detected faces in a set of 130 images and it was able to scan through a 320 x 240 pixel image in 2-4 seconds when running on a 200 MHz processor.

There are three main sub-systems; the neural network, merging/eliminating overlapping detections, and the arbitration of the outputs from multiple neural networks.

1. The neural network(s):

   Rowley, Baluja and Kanade [20] provided their neural networks with a 20 x 20 pixel image and it returned a value in the range -1 to 1 indicating how strongly it detected the presence of a face in that image.

   Since faces may appear anywhere in the image and at any scale, the 20 x 20 pixel image is taken at all locations of the original image, and at various 'zoom' ratios which are obtained by sub-sampling the original image. In their implementation the image was down-sampled by a factor of 1.2 to form a pyramid of image scales. The neural network hidden layers are also grouped to further evaluate pixel subregions of 10 x 10 and 5 x 5 pixel blocks as well as 10 x 5 pixel stripes. The network input has a connection to each pixel (retinal) and there is a single output.

   Some preprocessing was applied to the original images to ensure normalised intensity (lighting).

2. Merging/eliminating overlapping detections:

   A side effect of the scanning and scaling method used to obtain the 20 x 20 sub images from the original image is that a single face appears in several of these sub images and are therefore detected multiple times around the area where the face appears in the original image. This phenomenon can be used to eliminate false detections, since a non-face object will not be detected in all the surrounding sub images. This is achieved by counting the number of sub images around a certain position in which a face was detected. If the count exceeds a certain

threshold the area does contain a face (and not just false detections) and the position with the highest count is marked as the location of the face in the original image.

3. Neural networks output arbitration:

   In order to further refine the system's output multiple neural networks are used. Although topologically similar these networks have different random initial weight values and they are trained on slightly varying training sets. Their outputs need to be arbitrated to get a single system output for each sub image. The arbitration is done by using a "AND" combination. In other words for a certain scale and position in the original image all the networks have to agree that it contains a face before it is marked as such. In this way false positives can be further eliminated since all the networks seldom agree on a false detection.

2.1.5.3 Colour processing and edge detection

Hussin [21] used an object detection method with colour processing and edge detection to locate and count mangoes in a mango tree. The method turns all colours that are not close to the colour of the target object (in this case a mango) to black, leaving only the parts of the image where the object's colour occurs. This image is then subjected to edge detection by Circular Hough Transform (CHT) to further isolate the target objects from background information.

The steps taken were:

1. Resizing:

   The images were resized to a predetermined size, in this case 320 x 240.
2. Colour adjustment:

   Colours are adjusted to normalise the lighting (brightness).
3. Colour discrimination:

The colours of each pixel are compared to the default colour for the target object. If the pixel's colours do not match that of the target object the pixel colour is changed to black to effectively mark it as a background region of the image.

4. Circular Hough Transform:

   A CHT is applied to the image to detect objects of a circular shape. The Hough transform is a feature extraction technique used to detect lines or boundaries in an image to identify shapes by means of a voting process (as already discussed). As preprocessing for the CHT the image is converted to grey-scale, filtered and smoothed to get clearly defined object edges.

This method is very dependent on consistent lighting, since the lighting conditions when the image is taken has a great effect on the image's colour representations.

2.1.5.4 HOG+SVM

Dalal and Triggs [22] studied features sets to detect humans (pedestrians) from an image by using a combination of grids of Histograms of Oriented Gradient (HOG) descriptors with SIFT and Support Vector Machines (SVM). Their method is implemented by the following steps:

1. Gamma and colour normalisation:

   Although this step only had a modest effect on the algorithm's performance, they did find that RGB or LAB colour representation yielded slightly better results than simple grey-scale values, as did square root gamma compression.

2. Gradient computation:

   Gradients were computed with Gaussian smoothing followed by a discrete derivative mask. They experimented with several masks and found a 1-D mask with a zero mean to work best.

3. Spatial and orientation binning by voting:

   This step divides the image into blocks called cells and in each cell for every pixel in that cell a vote for an edge orientation histogram channel is calculated and accumulated. The edge

orientation bins can be evenly divided over either 180 degrees for unsigned gradients or 360 degrees for signed gradients. The vote itself is based on the magnitude of the gradient or some function of the magnitude of the gradient. Performance was very dependent on how finely this orientation encoding was done.

4. Contrast normalisation over spatial blocks:

   In this step contrast normalisation is applied to minimise the effect of lighting on the image. For this, cells are grouped into blocks and each block is then contrast normalised. Best performance was achieved by letting these blocks overlap extensively, so that each cell is covered in more than one block.

5. HOG over detection window: The HOGs can be applied in blocks or circles and they are computed in dense grids at a single fixed scale. The size of the blocks and their relation to the target object's proportions effects performance.

6. Classification by SVM:

   The detection window used should be large enough to allow for a border region around the target object. A linear SVM is trained and used as a classifier.

2.1.6 Implementation components

To implement a real-time image processing system there will be software to realise the algorithms used and hardware to execute it on, this section will give an overview of the practical aspects thereof.

2.1.6.1 Hardware

The question of which hardware to implement image processing tasks on remains a difficult one. There are not only the cost versus performance trade-off to consider, but also size, power consumption and suitability to typical image processing tasks.

Since most image processing algorithms are quite computationally intense, often involving large arrays of data, there have been various attempts to lighten the CPU load by moving some of the image

processing computations elsewhere, be that a GPU (Graphical Processing Unit) [23], FPGA (Field Programmable Gate Array) [24] or simply a parallel [25] or secondary processor.

Lindoso and Entrena [24] published some good results using hardware acceleration on FPGAs for image processing algorithms like filtering, correlation and convolution.

However, for embedded system specialised hardware still presents the best balance of cost, size, and power consumption, although there are other drawbacks like longer development times, limited scalability and limited application.

Although large-scale image processing systems are still used for massive imaging applications, the trend continues toward miniaturising and blending of general purpose small computers with specialised image processing hardware [5].

2.1.6.2 Software

Some functions and libraries are available for the implementation of common/general image processing tasks, some of these include parts that are relevant to object detection and object recognition.

- *OpenCV:* OpenCV ("http://opencv.org/") is a cross-platform, free and open-source library of functions, mainly aimed at computer vision, real-time image processing and machine learning. The libraries have interfaces for languages like C, C++, Python and Matlab/Octave. Functions can be called to implement various things like drawing, filtering, applying transformations, applying feature/object/edge detection, stitching, denoising, and many more.
- *Occlusion regions finder:* Humayun *et al.* [26] proposed a way to find occlusion boundaries in videos. As an object moves in the video relative to the background or other objects, certain pixels on the movement boundary are occluded and dis-occluded between the video frames. The software at "http://visual.cs.ucl.ac.uk/pubs/learningOcclusion/" implements their technique.
- *Scilab:* Although not specifically made for image processing, Scilab is an open-source software package for numerical computation specifically aimed at engineering and scientific application. It can be downloaded from "https://www.scilab.org/download/5.5.2" and used for free.

- *Matlab and Octave's Image Processing libraries:* There is an image processing toolbox for Matlab and an image package for Octave which contains a wealth of image processing functions. An overview of the functions available can be seen at "http://octave.sourceforge.net/image/overview.html".

2.2 OBJECT TRACKING

This section investigates video object tracking techniques in more detail, with some focus on the Kalman filter [27] and the particle filter [28].

Object tracking in video, also simply called "video tracking", attempts to locate and follow a specific object or objects in a series of image scenes (a video) as the object moves relative to the background imagery and relative to the observer (the camera) by estimating the object's trajectory from past frames and using that to predict what the object's position will be in the following frames.

Two stages are required for reliable object tracking in a video. The first being an object detection algorithm to find the object(s) in the current frame, and the second being a tracking filter to predict its next position. The filter helps to keep a smooth detection position even when the detection algorithm sometimes misses the object due to any of the number of the challenges associated with detecting an object from a real world scene.

2.2.1 The setup

Object tracking can be applied in various scene scenarios which will depend on the setup used, the application and the goal of the video tracking in question. Different algorithms will perform to varying degrees under the different scenarios as some are more suited to certain scenarios than others.

The particular case implemented here will be using the following setup and scenarios:

1. The aim is to detect and track certain standard road signs and traffic lights.
2. The input will be the video feed from a dash-cam mounted in a vehicle.

3. The processing must be performed on an embedded platform (the hardware will be installed in the vehicle).
4. The detection must take place while driving (the vehicle may be in motion).

Here are some examples of the different combinations for scenarios and setups [4]:

2.2.1.1 Fixed camera, moving object

Typical for surveillance scenes, the camera is fixed and the background static. An object (a vehicle, person, animal, etc.) is tracked as it moves over the background.

Suliman [29] showed that a person can be tracked in a surveillance video from a fixed camera by combining optical flow analysis with a Kalman filter. The steps involved converting video frames to greyscale, applying the optical flow analysis (segmentation, median filtering, morphological clustering and blob analysis) and finally passing the output to a Kalman filter to track the object as it moves across the scene.

2.2.1.2 Moving camera, object fixed in the scene

This setup is usually the case when the observer (camera) is fixed to a moving platform like a vehicle or a person, and the object to be detected or tracked is part of the background or scenery. The road sign recognition application studied in this assignment falls into this category, since the dash-cam will be mounted in a moving vehicle and the road signs to be detected and tracked are fixed next to the road.

2.2.1.3 Moving camera, moving object

There are two cases here that fall under this setup; One being an observer that is following a moving object (maintaining a constant distance from the object), making it appear that only the background is moving, and the other being a moving observer that is fixed to a moving platform but tracking an object that is itself moving relative to the background. This last case is particularly difficult [4] and may be

complicated further when it involves other objects also moving relative to the background [30]. From this Wojek [30] made some important notes to keep in mind when doing object tracking, particularly: "robust tracking performance is currently best achieved with a tracking-by-detection framework". For this reason, among others, techniques like motion correspondence and optical flow will not be explored further for the road sign tracking application.

An example for the first case (observer following a moving object at a fixed distance) is the camera man seated on a bike that films cyclists while they are riding. An example of the second case (moving observer tracking a moving object that is also moving relative to the observer) can be the pedestrian warning systems found in some high-end vehicles which uses a vehicle mounted camera to detect and track pedestrians, which are then highlighted relative to the background and played back to the driver on a screen in the vehicle.

2.2.2 Object tracking: Types of objects tracked

2.2.2.1 Outline/silhouette tracking

Outline tracking and silhouette tracking are very similar and tracking is done by estimating the region where the object should be in the next frame, then using shape matching [32] to locate the object in that region in that frame.

2.2.2.2 Point tracking

The objects that are being tracked are represented by points on the image and the association between points and objects are maintained based on the previous object position and motion.

An independent mechanism is used to detect the object in each frame. The object is then represented by a point in the image, and the trajectory of that point along with the object detection mechanism is used to detect it again in the next frame.

2.2.3 Object tracking: Types of tracking filters

2.2.3.1 Kernel based tracking

In kernel based tracking an object is tracked by computing the motion using a kernel for parametric transformation. It may take forms such as translation, rotation, or affine. The method proposed by [31] uses feature-histogram based target representations, which are regularised by a kernel. A metric derived from the Bhattacharyya coefficient is usually used as a similarity measure.

2.2.3.2 Kalman filter tracking

Kalman filters are recursive Bayesian filters. The Kalman filter is an iterative state estimator that can be used to predict the next location of an object in an image based on detection candidate points and the previous position and motion (trajectory) of the object. The standard Kalman filter is only suitable for linear systems.

The Kalman filter has been extensively used in the vision community for tracking [32] and will be discussed in more detail in the next section.

2.2.3.3 Particle filter tracking

Particle filters are also recursive Bayesian filters. The particle filter is another iterative predictor that can be used to predict object locations in an image. The method makes use of hypothesised particles which are filtered using the Sample-Importance-Resample (SIR) principle. It starts with an initial sample, then the detection candidate points are used to weigh each sample according to its likelihood given the likely position of the object based on its previous position and motion (trajectory). A new sample set is then taken from the weighted (updated) samples and the process repeats. Such sequential Monte Carlo methods are well suited to highly non-linear models.

Particle filters recently became popular in computer vision where they are especially used for object detection and tracking [32] and thus will also be discussed in more detail in the next section.

2.2.4 Object tracking: Types of data association methods

2.2.4.1 Joint Probability Data Association Filter

JPDAF is a multi-object data association and tracking method. It makes use of a track, which is a set of points along the path followed by an object. The goal is to assign all measurements to existing tracks which will associate those data points to an object.

The main draw-back of this method is that it assumes the number of objects (paths) to remain constant.

2.2.4.2 Multiple Hypothesis Tracking

MHT maintains several hypotheses for each object across several frames of the video. Each hypothesis is a number of possible partial tracks and for each of these a prediction is made for its position in the next frame. Using a distance measure the actual positions are compared with the predicted positions and the final track of each object is the most likely set of hypotheses over time. The partial tracks are then associated using the distance measure, which creates a new set of hypotheses and the process repeats.

The MHT method is robust in the sense that it can account for objects leaving the image area and new ones entering. It can also handle occlusions by continuing a track even if some of the measurements from an object are missing [32].

2.2.5 A more practical look at object tracking in videos using a Kalman filter

To track an object in a video an object detector is required to detect the object in at least some of the frames [32]. As for the other frames, the answer is to track the object as it moves across the video frame using one of the tracking methods listed above. Whenever the detector output is too poor to say with certainty that the object was detected, it often still manages to match a number

of feature keys onto the object. The placement of these features can be used as a "noisy" or poor measurement/observation-input for a state prediction filter such as a Kalman filer.

2.2.5.1 Kalman filter theory

The two Kalman filter equations are [28] [33]

$$\hat{x} = Ax_{k-1} + Bu_k + w_k, \tag{2.2}$$

and

$$\hat{z} = hx_k + \upsilon_k, \tag{2.3}$$

where:

- k denotes the current iteration number or discrete time index
- $\hat{x}$ is the state prediction
- A is the state-transition matrix
- x_{k-1} is the state at the previous discrete time instance
- u_k is the action or control vector (only when some control is applied)
- B is the control matrix
- w_k is the zero mean Gaussian process noise with covariance Q
- $\hat{z}$ is the sensor prediction
- h is the observation matrix which converts the predicted state to the predicted measurement
- υ is the zero mean Gaussian observation noise with covariance R.

The Kalman filter can be implemented by following these steps, derived from Equation 2.2 and Equation 2.3 [33]:

Prediction step (time update)

1. Calculate the predicted state

$$x_k = Ax_{k-1} + Bu_k. \tag{2.4}$$

2. Predict covariance

$$P_k = AP_{(k-1)}A^T + Q, \tag{2.5}$$

where Q is the estimated process error covariance.

Correction step (measurement update)

1. Compute the Kalman gain K with

$$K = \frac{P_k H^T}{HP_k H^T + R}, \tag{2.6}$$

where R is the measurement noise covariance.

2. Update the estimate using the measurement with

$$x_k = \hat{x}_k + K(z_k - H\hat{x}_k), \tag{2.7}$$

so that the difference (error) between the measurement z_k and the predicted measurement Hx_k is multiplied by the Kalman gain.

3. Update the error covariance

$$P_k = (I - KH)\hat{P}_k, \tag{2.8}$$

where I is the identity matrix.

These steps should be repeated for each (processed) frame of the video.

2.2.5.2 Implementing a Kalman filter

The object detector used in conjunction with the object tracker (Kalman filter) will give a position of the target object on the frames where the object was positively recognised. This position can then be translated to give a single point in the centre of the target object.

This ’centre’ point will now be used as an observation value to track the detected object in the video. In each frame the Kalman filter makes use of a noisy observation and a predicted state based on the system model to calculate the probability distribution of the next state vector, which can be used to predict the next (filtered) position of the object being tracked.

The Kalman filter combines the available measurement (observation) data (even when uncertain) with a model of the system's behavior to predict the target object's next state distribution. Because the setup in the proposed project will be made up of a moving observer (camera) and an object stationary relative to the background, the model for the movement of the object's state through the frame is made up of simple Newtonian kinematic (movement) equations.

Assuming a nearly constant relative object speed between frames, the equations for position from constant speed in Equation 2.9 and Equation 2.10

$$x(t) = x_0 + V_{x0}t \tag{2.9}$$

$$y(t) = y_0 + V_{y0}t, \tag{2.10}$$

and for change in speed in Equation 2.11 and Equation 2.12

$$V_x(t) = V_{x0} \tag{2.11}$$

$$V_y(t) = V_{y0}, \tag{2.12}$$

can be applied, where x is the horizontal component and y the vertical component. Since there is a constant time between frames the time reference can be changed to a discrete frame index n and the time itself replaced by the number of frames, which will be 1 for successive frames. Changing Equations 2.9-2.12 thus leads to Equations 2.13-2.16 where $t = 1$.

$$x_n = x_{(n-1)} + V_{x(n-1)}.t \tag{2.13}$$

$$y_n = y_{(n-1)} + V_{y(n-1)}.t \tag{2.14}$$

$$V_{xn} = V_{x(n-1)} \tag{2.15}$$

$$V_{yn} = V_{y(n-1)} \tag{2.16}$$

The state representation in the Kalman filter will require the x component position and velocity and also the y component's position and velocity. Since the Kalman filter will be implemented using matrices, the state vector will be represented by the column vector

$$X = \begin{bmatrix} x \\ V_x \\ y \\ V_y \end{bmatrix}. \tag{2.17}$$

Since the image is represented as a matrix of pixels, the x and y components now refers to the column and row positions of the pixel values. Knowing the Kalman filter's state vector X, Equation 2.13 and Equation 2.15 can now be implemented into the state transition matrix A, so that

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.18}$$

Since no control input is being applied, the matrix B will be zero, eliminating the control term u. As mentioned earlier, the observation value will be the object detection algorithm's average x and y position. Knowing that and the state vector X, the observation matrix H can now be determined since the cross-product of H with X should convert from the state vector format to the observation format, which is just an x-y position. Thus H should 'extract' only the position values from the state vector which leads to H being

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{2.19}$$

To start off with, the covariance matrices Q and R can set to unity identity matrices so that $Q = I_4$ and $R = I_2$.

If the object detector used can give an output that does give some indication of how well the object was detected, then this information can be used to adapt the Kalman filter so that it relies more on the observation input when the object detector returns a high detection certainty, and to rely more on the state estimation when the detector was not so certain about its detection output. This is called dynamic covariance and was used by Xu in [34]. To do this the observation covariance matrix R is adapted on each frame.

2.2.6 A more practical look at object tracking in videos using a particle filter

In the same way that the Kalman filter was used to track the detected object as it moves over the video frame, a particle filter can also be used. Particle filters are also sometimes called the condensation algorithm or Bayesian bootstrap filter and was first presented by Gordon [35]. The main advantage of the particle filter is that it is not dependent on the problem's state change being linear as is the case of a normal Kalman filter. The drawback though is that the particle filter tends to take more processing time, especially when the required number of particles is very large.

In the proposed project, where it will be used as a road sign recognition application, the movement of the object through the frame can be described by Newtonian kinematic equations, which are linear equations. As already discussed, the processing time required when doing object detection and tracking in a video is critical when the method needs to be applied in real time. For these reasons the particle filter is less well suited to the road sign recognition application than a Kalman filter.

2.2.6.1 Implementing a particle filter

For an observation the same X-Y point is used as for the Kalman filter, which is the geometric centre-point of the target object as output by the object detection algorithm.

To implement a particle filter, the following steps should be iteratively applied to each frame:

1. (Re)Sample N particles called x^i where $i = 1, \ldots, N$ from the prior density function (with respect to the weights from the previous iteration).
2. Propagate particles in time using the system's state model
$$x_k = Ax_{(k-1)} + \Gamma q_k, \tag{2.20}$$
where q_k is the zero mean Gaussian system noise with covariance Q.
3. Take a noisy measurement with
$$z_k = \begin{bmatrix} x_{observation} \\ y_{observation} \end{bmatrix}. \tag{2.21}$$
4. Update the weight of each particle with the measurement z_k so that the weight is the probability of the value of the particle given the observation that was just made, or
$$w^i = P(z|x^i). \tag{2.22}$$
5. Normalise the weights by dividing each one by the total sum of all the weights
$$w^i = \frac{w^i}{\sum_{i=1}^{N} w^i}. \tag{2.23}$$

2.3 SENSOR FUSION

Data/sensor fusion can take place at different levels of the system architecture and also at different stages of data processing. The fusion process can itself be performed using different techniques and with various architectures.

The term data- and sensor-fusion covers many fusion terms and fields, like amongst others; decision fusion, data combination, data aggregation, multi-sensor data fusion, and sensor fusion [36].

This section will take a closer look at the benefits offered by sensor fusion, the methods and algorithms for sensor fusion, and the various architectures for sensor fusion.

2.3.1 Benefits and drawbacks

2.3.1.1 Benefits

The main benefits of sensor fusion are [37]:

1. Robustness:

   Since the system merges data from various sensors it is not totally dependent on one sensor alone, which makes it more tolerant to perturbations.

   Say for instance there are two cameras monitoring some target when a person passes in front of one of them, or one breaks down. The system will continue to function (to some degree) with the data from the other camera. If the first camera was the only one, the system will be unable to monitor the target during that time that the camera's view is blocked, or until the broken camera is fixed.
2. Reliability:

   Sensor fusion can ensure that a system achieves better up-time.

Take the camera example mentioned above. The system's down-time is dependent on the mean time between failure (MTBF) of the camera when only one camera is used, but by using data fusion with the two cameras, the system will now be down only when both cameras fail at the same time.

3. Accuracy:

   By combining data from different sensors, inaccuracies that are specific to a certain sensor or sensor type makes a smaller statistical contribution to the result, thereby resulting in a more accurate end result.

   Take for example a system measuring temperature with 3 (cheap) sensors. One of them has an unknown offset. The system can use data fusion, say by taking the statistical average, to get a more accurate reading than one sensor alone, since the fusion process reduces effects such as sensor offset or other inaccuracies.

4. Fault tolerance:

   By using a smart system to combine different sensors or data sources, some of the redundancy between the sensors can be used to verify the sensors or sources themselves. In this way the system can detect if a sensor fails or gives consistently incorrect readings. By combining this with machine learning techniques, the system can then learn to ignore or scale the faulty sensor's contribution.

   Take a tank in a chemical plant with a critical fluid level which is measured by ultrasonic echo and with a float switch. The data from the two sensors can be fused (say by a voting system) to control a pump which changes the level in the tank. If the tank is filled and the ultrasonic sensor reports a full tank, but the float switch got stuck and reports that the tank is not full yet, the pump will be stopped by the sensor fusion output. The system can also issue an alarm when the sensors do not concur (votes differed) to state a possible sensor malfunction.

5. Noise reduction:

   Data fusion can be used to reduce the noise from a sensor or sensors, or to improve the signal-to-noise ratio.

For example, the reading obtained from an analogue to digital converter (ADC) on a micro-controller. The reading (even when measuring a constant voltage) will fluctuate somewhat due to noise. By oversampling and using data fusion to merge several readings from the same ADC, a resultant measurement with some degree of noise immunity can be obtained.

6. Dimensionality:

   Fusing data from different sensors can lead to a whole new dimension from the data, giving information that was not available from any sensor alone, but only from the combination thereof.

   For example, two cameras looking at an object some distance away from slightly different perspectives, gives two views of the same object. But apart from that, the two data sources can be fused to give information about the distance to the object by using the stereoscopic effect, similar to how a human gains depth perception by looking at an object with both eyes.

7. Coverage:

   By fusing data from various sensors, or multiple data from one sensor the span or coverage can be increased. The fusion process can find areas where sensor data overlaps and combine it to create information about the span of both sensors or data sets.

   For example, the typical reverse camera in a car has a certain field of view that only covers a certain area behind the vehicle. By installing a couple of cameras around the rear of the vehicle and using data fusion to combine ("stitch") the images a wide continuous panoramic image can be created that covers the entire area behind and to the the sides of the vehicle.

2.3.1.2 Drawbacks

There are also negative aspects to multiple sensors and sensor fusion [37]:

1. Processing cost:

   Inevitably the data fusion process requires some processing and the cost thereof depends upon the number of sources, the type and quantity of data, the architectural level and the fusion

algorithm used. Axelsson [38] did a comparative test of different fusion algorithms on an industrial robot which showed how different algorithms can deliver similar results, but at greatly different expense as to modeling time and processing cost.

In some cases the processing cost will be huge, while in others it may be negligibly small, but it remains a factor that must be considered and weighed against the benefits offered by the fusion process.

2. Timing:

   Data from various sources are not always available immediately or at the same frequency. This may lead to some timing difficulties during the fusion process. Failure to properly align the data to be fused may lead to incorrect data output from the fusion process. Choosing the correct fusion architecture can simplify this type of problem in some cases.

3. Calibration:

   Certain sensors need to be calibrated. The calibration procedure may be influenced by the fusion process and should be kept in mind during the design phase, especially when choosing the fusion architecture.

4. Complexity:

   Using more sensors, or more data from a sensor and fusing the data adds complexity to the system. The added complexity, like the processing cost, should be weighed against the benefits gained by using the data fusion.

2.3.2 Methods and Algorithms

Many different sensor data types can be used for sensor fusion, and there are many different ways and algorithms to accomplish this. In general, data can be fused by following the generalisation rule. The data must be generalised by transforming it to a common vocabulary. This may involve converting the data to the same unit of measurement or to the same coordinate system or whichever conversion is necessary to make the data qualitatively similar. A measure of confidence or reliability may be added to

apply preference for better data when the fusion takes place [39]. These methods are inherent in most of the popular data fusion methods and algorithms which will now be discussed in some detail.

The main algorithms used for sensor- and data-fusion are [36] [37] [40]–[42]:

1. Kalman filters (including distributed-Kalman filters)
2. Extended Kalman filters
3. Particle filters/Sequential Monte Carlo (including distributed particle filters)
4. Naive Bayes
5. Dempster-Shafer (also called "Evidential reasoning" or "Theory of evidence")
6. Fuzzy logic
7. Interval methods
8. Voting (boolean)
9. Rule-based
10. Data association techniques (like K-means and probabilistic data association)
11. Semantic methods
12. Probabilistic grids

Many of these methods, like the Kalman filter, sequential Monte Carlo and naive Bayes are Bayesian probabilistic methods.

Recall Bayes' rule in Equation 2.24 [43].

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{2.24}$$

Since sensor fusion aims to make decisions based on data from sensors, Bayes' rule lends itself to sensor fusion since it provides a way to determine probability given an observation. The probabilities are used in making the decisions, while the given observations are the data from the sensors. For this reason it is logical that Bayesian probabilistic methods seem to dominate the sensor data fusion field.

Often the different fusion algorithms produce very similar results, but they vary greatly in modelling time and computational cost, as shown by Axelsson's [38] comparative test of different fusion

algorithms on an industrial robot. Clearly the suitability of the different algorithms depend a lot on the specific problem and its architecture.

Each of the listed algorithms will now be discussed in more detail:

2.3.2.1 Kalman filters

Since the Kalman filter was already covered in the section on object tracking it will not be repeated here.

2.3.2.2 Extended Kalman filters

The Kalman filter only works for linear state problems with Gaussian noise [27]. In order to get around the linearity constraint of Kalman filters, one of the common techniques used is the extended Kalman filter (EKF).

The EKF works by simply using a local linear approximation for the non-linear aspects of the problem that the Kalman filter will be applied to.

The state-transition model f and observation matrix h cannot be used directly, instead a partial derivative (Jacobian matrix) is used [33]. In the EKF case the Kalman filter equations change to:

$$\hat{x}_k = f(x_{(k-1)}, u_{(k-1)}) + w_k \tag{2.25}$$

$$\hat{z}_k = h(x_k) + v_k, \tag{2.26}$$

where f and h are non-linear differentiable functions, and w and v represents the process and measurement noise.

The EKF now approximates the state and measurement by linearising Equation 2.25 and Equation 2.26 as [33]:

$$x_k = \hat{x}_k + A(x_{(k-1)} - \hat{x}_{(k-1)}) + Ww_{k-1} \tag{2.27}$$

$$z_k = \hat{z}_k + H(x_k - \hat{x}_k) + Vv_k, \tag{2.28}$$

where

- x_k and z_k are the true state and measurement vectors.
- $\hat{x}_k$ and $\hat{z}_k$ are the approximate state and measurement vectors.
- A is the Jacobian partial derivatives of f with respect to x or $\frac{\partial f}{\partial x}(\hat{x}_{(k-1)}, u_{(k-1)})$.
- W is the Jacobian partial derivatives of f with respect to w or $\frac{\partial f}{\partial w}(\hat{x}_{(k-1)}, u_{(k-1)})$.
- H is the Jacobian partial derivatives of h with respect to x or $\frac{\partial h}{\partial x}(\hat{x}_k)$.
- V is the Jacobian partial derivatives of h with respect to v or $\frac{\partial h}{\partial v}(\hat{x}_k)$.

2.3.2.3 Particle filters

Since the particle filter was already covered in the section on object tracking it will not be repeated here.

2.3.2.4 Naive Bayes

The Naive Bayes method is a statistical method based on Bayes' theorem (stated in Equation 2.24) while making the (naive) assumption that there is total independence between the input features used, even when that does not hold.

To implement it as a classifier, Bayes' rule is applied with a decision rule so that the probability of the sample E belonging to the i'th class C_i is given by

$$P(C_i|E) = \frac{P(E|C_i)P(C_i)}{P(E)}. \tag{2.29}$$

Since the naive Bayesian classifier assumes independent features, $P(E|C_i)$ can then be expressed as the product of the probabilities indicated by each of the J features F_j, that this example belongs to the given class C_i, or

$$P(C_i|E) = P(C_i)\prod_{j=1}^{J} P(F_j = C_i|C_i). \tag{2.30}$$

2.3.2.5 Dempster-Shafer

The Dempster-Shafer method is reported to be a generalised Bayesian method introduced by Dempster [44] and re-interpreted by Shafer [45]. Their theory, often called evidential reasoning or belief theory presents a method for working with uncertainty by using beliefs, which are essentially a way of expressing a degree of confidence or certainty.

The method differs from probability theory in the sense that it can express a belief-mass to all subsets of events [40], by making use of a concept called a frame of discernment. For example, lets say there is a door around the corner that is either open or closed. In probability theory if $P(open) = 0.3$, then since probabilities should add to 1, it will be that $P(closed) = 0.7$ which will result in the set shown in Equation 2.31.

$$\left\{ \begin{matrix} P(open) & = 0.3 \\ P(closed) & = 0.7 \end{matrix} \right\} \tag{2.31}$$

In evidential reasoning an exhaustive set of all states of the system is considered, such as the set shown in Equation 2.32.

$$\left\{ \begin{matrix} open & = 0.3 \\ closed & = 0.5 \\ either\ open/closed & = 0.2 \\ neither\ open/closed & = 0 \end{matrix} \right\} \tag{2.32}$$

Such a set encompasses all the single states the system may be in and is therefore a frame of discernment [36], with each element representing a hypothesis about the state of the door, and the values assigned are called the mass functions (which must add to 1). In effect this is stating that the evidence shows a 30% chance of the door being open and a 50% chance of it being closed. The remaining 20% in this case is the chance of the door being either open or closed and indicates the degree of (un)certainty or the lack of knowledge about the state of the door. The method also accounts for ways of obtaining related measures called support and plausibility which, in effect, provide upper and lower probability bounds in agreement with Dempster's original formulation of this method [40].

2.3.2.6 Fuzzy logic

Fuzzy logic is an extension of boolean logic [46] and is often used when dealing with uncertainty in high-level data fusion or decision making [40]. Fuzzy logic, also known as infinite-valued logic, deals

with logic in a mathematical way where variables take on a degree of truth between 0 and 1, rather than being either 0 or 1 as in (crisp) boolean logic. This is especially useful when one wants to indicate or measure something that is not exact or cannot be assigned a specific measured value.

For instance, indicating whether it is warm outside or if a vehicle is traveling fast. The main difference lies in the 'fuzziness', where the different classes that a set is divided into can overlap. Looking at the example of how warm it is outside, a boolean set can be created by 2 classes, say below 15 degrees Celsius it is classified as cold, and above 15 degrees Celsius it is classified as hot. In a fuzzy set the degree of warmness is indicated. Say there are 3 fuzzy classes, cold, warm and hot. if it is -2 degrees Celsius the 'cold' class will be strongly indicated, say 0.9, while the 'warm' class will only indicate slightly, say 0.2, and so forth. This shows that there are no clear boundaries in a fuzzy set, the classes overlap somewhat and each indicates how well the current condition fits that class.

The mathematics of fuzzy logic is based on t-tautology (triangular tautology) which is defined by continuous triangular norms [47]. Several mathematical rules and operations can be applied to fuzzy sets, like commutativity, associativity, distributivity, De Morgan's law, etc. [40].

Fuzzy logic can be expressed (and implemented) as a set of 'if', 'then' and 'else' rules that are close to normal human language [46]. For example, "if it is very hot inside, then turn on the air conditioning". Again note the degree of applicability for fuzzy logic, which is here expressed as "very hot".

2.3.2.7 Interval methods

Parametric curve fitting is used to describe the object of the sensor data, but the range and accuracy required is defined by interval methods [42]. Interval methods deals with uncertainty by placing an upper and lower limit (boundary) on the range of the value on which the uncertainty is applicable. This method is used more for detection than for data fusion [40], and so it will not be discussed in detail.

2.3.2.8 Voting (boolean)

Voting is a method whereby each data source or sensor gets a boolean 'vote' on a specific state. Different rules can be assigned to the voting outcome, but usually the majority rule is used whereby if most votes are '1' then the outcome is '1' and if most votes are '0' then the outcome is also '0'. Votes can also be weighted, to give preference to a more accurate sensor or when one data source is more reliable than the others.

Other voting methods exists, like using voting to get a confidence indication, the more sensors or data sources agree (by voting) on a certain state the higher the confidence level for that state.

A slight alternative to majority voting is threshold voting, whereby a certain state is output if more than a certain number of inputs voted for that state.

Although voting is a very simple form of fusion it usually requires some form of (classification) processing on the raw sensors or raw data in order to get a boolean value from it. The boolean 'outputs' can then be fused by voting in order to produce a fusion output. [48]

2.3.2.9 Rule-based

A general description of rule-based systems or knowledge-based systems is that a rule is evaluated, and when the rule evaluates to 'true', a corresponding action is performed [49]. So these types of systems consist of a body of built-in knowledge about the task that it is applied to and a rule basis whereby this knowledge is used by the system to make decisions and take actions, often referred to as a knowledge base and an inference mechanism [49]. The method is very similar to expert systems, and it relies on the knowledge of human experts to capture knowledge about the problem/task in the knowledge base for it to use.

This method is very simple to translate into computer code or similar symbol-based or lookup systems. For example the rules can be tested by 'if-then-else' or 'case' statements in a C program, or matched to entries in a database containing the rules and actions. The action to take may also be to evaluate further rules, leading to a nested system.

In sensor- and data-fusion the rule-based method should work well when certain combinations of the input data leads to explicit outputs. Unfortunately not all real world problems have a definite answer or output that can be directly derived from the combination of outputs. Although fairly complex rule-based systems have been developed very successfully, like the MYCIN medical system in [49], this remains the main limitation of the method.

2.3.2.10 Data association techniques

Data association techniques describe a broad field of methods to associate certain data into groups or clusters. It can be the association of data from similar variables or sensors, or the association of data from different variables or sensors [50]. Data association could appear at any or all of the fusion levels, but typically fall into level 1 of the JDL model (object refinement) [36]. Some of the popular data association techniques will now be discussed:

- Nearest neighbors is a simple technique that groups data by measuring the distance between the data points. If the distance is less than the specified threshold, the data belong to the same group or cluster.
- K-means is similar to the nearest neighbors technique and makes use of an iterative approach. To implement it, a number of cluster centroids are randomly chosen, then the following two steps are repeatedly applied until a stopping condition is met: First, assign each data point to belong to the cluster that has the nearest centroid. Then, re-define the cluster's centroid to be the centre-of-mass of all the data points that belong to that cluster.
- Graphical models are a way to include prior knowledge about the problem in a easy to grasp visual manner which can be used to reason about the probabilities and dependencies. Although the visual representation is not strictly necessary, it provides a great way to illustrate complex relationships between a problem's variables. Graphical models can be directed models for Bayesian networks, or undirected for Markov models.

  Graphical models can be used for data association since it provides a way of representing the structure of dependencies (and independencies) between the variables. Chan [51] presented a graphical model method for data association that solves an inference problem using the max-

product algorithm. Their method uses a distributed iterative implementation of this algorithm based on parallel message passing.

- Maximum likelihood probabilistic data association is also known as the modified filter of all neighbours [36]. Data points or measurements from sensors are assigned an association probability for each hypothesis that is used to associate the data. For example, when targets are tracked by a camera, each measurement is assigned a probability that it belongs to a given specific target. The probabilities are then used to associate the measurements and can also be used to estimate the target state [36].

2.3.2.11 Semantic methods

Semantics is a description of associating words or symbols to a specific meaning which they represent (what they stand for). Semantic methods of data fusion is a high-level method that makes use of words, phrases or symbols (in a formal language) to represent some high-level features, properties or states which have been integrated or derived from the low-level sensor data [36]. A very large and complex network of sensors can be built when a common, generic language set is used since all the data can then be used by fusion on a high level using the common semantics.

These semantics can then be sent to a central fusion processing node or used for a lookup in a knowledge base or data base or even on-line, or used for further data association or data fusion. A central fusion node can collect the semantic input from the processed sensor data and make decisions, send notifications or extract other meaningful information from it [52]. Since only the extracted semantic information is sent, it results in a large saving on the required communications bandwidth.

2.3.2.12 Probabilistic grids

Probabilistic grids is a method of data fusion mostly used for tracking, navigation and mapping. The area to be mapped is divided into a grid of cells, and the cells are then assigned probabilities of having a certain property. The property in question can be something like "occupied". As the sensor data is collected, the observations are used to apply Bayes' rule to update the probabilities of each cell [40]. In this way a 'map' is built, showing how the property that is measured is spread out over the area that is represented.

2.3.3 Fusion model architecture

The fusion model architecture describes how the different sensors and sub-systems of a sensor data fusion system are connected or how they relate to each other with regard to certain criteria. The criteria may include things like physical connections, where the processing takes place or at which level the data are combined.

Sensor fusion architectures are usually classified according to the following criteria [37] [40]:

1. Communication and interaction
2. Processing distribution
3. Hierarchical relations

A detailed discussion of each will follow.

2.3.3.1 Communication and interaction

In a sensor fusion system, the sensors need to be connected together in some manner. The way in which they are connected and how they communicate with each other make up the communication and interaction architecture of the fusion system in question.

Looking at the communication setup in a fusion system, messages are sent between different nodes via the physical (or logical) connections between these nodes. A node can be a processing unit or a sensor (or both), but more on this in the "processing distribution" section. The rules relating to the flow and addressing of messages are determined by the communication architecture.

If nodes are only allowed to send messages to its nearest neighbours or to some neighbouring nodes, the communications setup is said to be localised. In this configuration the system is more scalable and reconfigurable [40] than a global system where any of the nodes may communicate with each other.

2.3.3.2 Classification by data source relations

Classification can also be performed according to the data's source relation into [36]:

- Complementary data:

  Different perspectives of the same entity being measured, increasing the coverage.
- Cooperative data:

  Measurements from different sensor types on the same entity, increasing the scope of the information.
- Redundant data:

  Different sensors' measurements of the same entity, increasing the confidence.

2.3.3.3 Processing distribution

The fusion architecture can also be classified according to where the main fusion processing takes place.

The model is classified as a centralised fusion architecture when all the sensors' data are sent to a central location in a fairly raw format [50] to be processed and fused at a single point. In contrast, when each sensor or a node that is local to each sensor does the processing for that sensor and then proceeds to inform the other nodes only of the features or outcome (high-level, pre-processed information [50]) obtained by that sensor, then the model is of a decentralised architecture. This is also called the per-node processing architecture. It is also possible to use a combination of centralised and decentralised architectures, in which case the model will be called a hybrid architecture.

A comparison follows:

1. Centralised:

In a centralised system all the fusion processing is processed at a single central node. To do this a lot of low-level data have to be sent to the centralised processing point. This demands both high bandwidth data buses between the sensors and the central processing node and high speed data processing at the central processing node [37].

On the other hand, the centralised architecture makes no assumptions or conclusions about the data from the sensors, and so no sensor data dilution takes place before the fusion point. There is also the advantage of having a global overview over all the sensor data at a central point [50].

A conceptual diagram of the centralised architecture is shown in Fig. 2.4.

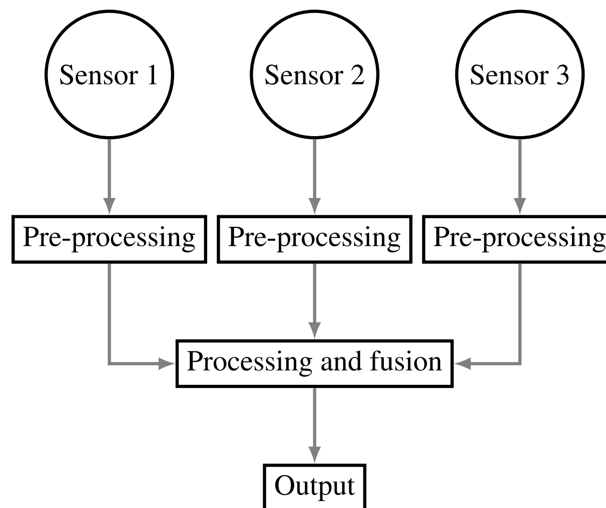

Figure 2.4. Centralised fusion architecture

2. Decentralised:

   In a decentralised system there is no single central processing point where everything comes together. For this reason there is not such a high performance demand on a single processor and on the data buses, but this comes at the cost of added complexity to the system [50], since there are now fusion processes taking place at each node/sensor. Each node has to do the fusion, extract high level information and share it with its neighboring nodes. At the same time such information must be received from the neighboring nodes and together with the sensor's own data used for the fusion process at the node in question.

The decentralised architecture lends itself more towards scalability and is easier to reconfigure [40]. In general it is also more fault tolerant since a failure on one node does not necessarily mean a total system outage.

A conceptual diagram of the decentralised architecture is shown in Fig. 2.5.

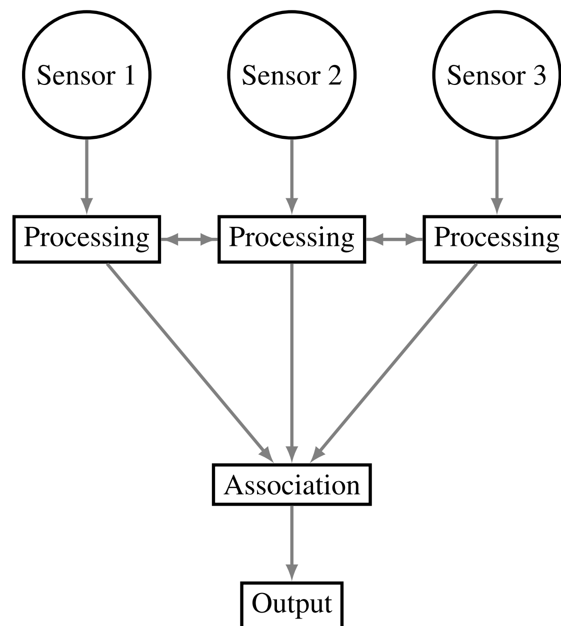

Figure 2.5. Decentralised fusion architecture

3. Hybrid:

Sensor data fusion systems are not always entirely centralised or decentralised, but can also be a combination of both architectures.

A hybrid system may combine fusion of low-level (raw) data and high-level (decision) data, or it can use the nodes to do some initial processing on the raw data, which eases the communications load because less data is then sent to the central processing point where the data are further processed and fused.

In some instances a hybrid configuration can give the advantages of both centralised and decentralised architectures, but once again at the cost of system complexity.

2.3.3.4 Hierarchical relations

There are many hierarchical relation models that are used to design and classify sensor data fusion systems. An overview of some follows:

1. The JDL model:

   In 1992 the JDL (Joint Directors of Laboratories) of the USA's department of defence published a hierarchical data fusion model with a military focus [37] [50] [53] [54] :

   The JDL model defines the levels shown in Table 2.1.

Table 2.1. The JDL data fusion model levels

Level number	Function
0	Source preprocessing
1	Object refinement
2	Situation assessment
3	Impact assessment
4	Process refinement

   - *Level 0:* The initial signal level processing is performed on the data to reduce the volume and keep the useful information for the higher levels [36]. In the military sense the goal is to locate objects and predict their attributes (speed, position, etc.) [50].
   - *Level 1:* Here the information from level 0 is further refined. This level attempts to consolidate and complete the information about each separate object located in level 0 to predict a continuous state estimation [37].
   - *Level 2:* Assesses a situation by considering groupings, clusters and patterns between the different objects and how they relate to each other.
   - *Level 3:* The situation identified in level 2 is analysed to predict and identity possible outcomes and the risk thereof [55]. In the military sense this level will look for threats and intents and asses the situation and the severity of its consequences [37].

- *Level 4:* This level is more of a managerial task that takes a supervisory look at the other level and at the system as a whole, therefore it is not directly part of the core functionality of the system. The level attempts to adapt and improve resource management and to prioritise tasks within the system [36].

2. Data fusion levels:

   Data fusion can also be classified by the abstraction level, or at which level of the data hierarchy the fusion takes place. The following levels are commonly used [36] [41]:

   - *Data/Signal-level* fusion is the combination or fusion of unprocessed sensor data, like speed sensor pulses or images from a camera.
   - *Characteristics/Feature-level* fusion is the combination of features extracted from the different sensors' data that represent object properties such as size and position of an object detected in an image.
   - *Symbol/Decision-level* fusion combines detections (or detection probabilities) as derived from different sensors. For instance, combining a vehicle's speed with the fact that a stop sign was detected in a dash-cam image to trigger a traffic violation alert.

3. System input/output classification:

   Following on from the data fusion levels, the relations between the data level and the input and output of the system (or a sub-system) can also be used for classification in the Dasarthy model [36] [37] [56]:

   - *Data in, Data out* - Data-level fusion
   - *Data in, Features out* - Feature selection and extraction
   - *Features in, Features out* - Feature-level fusion
   - *Features in, Decision out* - Pattern recognition
   - *Decision in, Decision out* - Decision-level fusion

4. OODA:

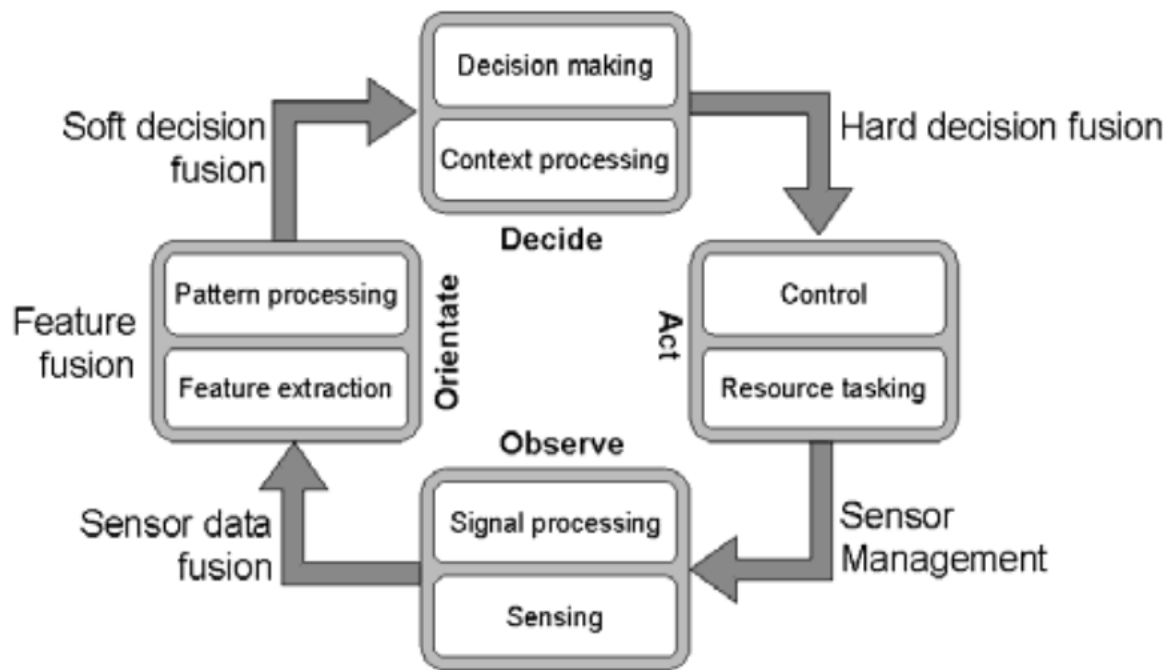

Figure 2.6. The omnibus model for data fusion (from [57] ©IEEE 2000)

Another military fusion classification system is according to the high-level iterative strategic stages of decision making. This classification method is similar to the JDL model and is called the OODA loop by Boyd in 1987, for each of the stages [37]:

- *Observe* - Read sensors data to take in the situation.
- *Orientate* - Analyse the sensor data to asses the situational relation between yourself and the enemy (or objects).
- *Decision* - Use the orientation information to take a strategic decision.
- *Action* - Implement the strategic decision that was taken.

5. The Omnibus Model:

Bedworth and O'Brian [57] proposed the omnibus model in an attempt to combine the associated advantages of the many popular level-based data fusion models such as the waterfall model, JDL model, Dasarthy model, etc.

The model itself is based on an iterative loop which is similar to the OODA loop model, but draws in aspects of many other data fusion models. The model is shown in Fig. 2.6.

2.4 EXISTING ROAD TRAFFIC SIGN DETECTION SYSTEMS

There are already several implementations of systems that makes use of road traffic sign and traffic light recognition for various applications like driver assistance [58] and autonomous vehicles [59].

Most existing systems make use of a combination of methods. First, during the detection phase either a colour-based or shape-based detection method is used. The areas of interest found are then further scrutinised by other algorithms, typically a neural network or support vector machine (SVM) in the recognition phase, in order to find the target object. Some also includes an object tracking mechanism. A few examples follow:

1. Hough circle transform for circular signs or Hough line transform for triangular ones, followed by a neural network for classification [58] [60].
2. Finding symmetrical shapes around a centre point by using the fast radial symmetry transform with gradient information for road-signs [3] and for traffic lights when combined with an SVM [1].
3. Colour and saturation based detection, combined with an SVM for recognition [2].

2.5 CONCLUDING REMARKS

In this chapter the findings of a literature study on the relevant image processing topics were presented and discussed. This gives an indication of the techniques used in existing systems and how they work and perform, an overview of the methods and algorithms that are available and what they do and also a practical look into how these techniques can be implemented or applied. In the next chapter this knowledge will be applied to design and build a system that can fulfill the project designs goals.

CHAPTER 3 METHODS AND CONTRIBUTION

This chapter provides a detailed account of the initial methods used to detect stop signs and traffic lights, followed by a detailed discussion of the investigation into, and comparison of alternatives. In this chapter it will shown how the following contributions were made as part of this project:

1. It proves that a real-time traffic sign detection system can run on off-the-shelf embedded hardware at a sufficient rate to detect the traffic signs from a vehicle traveling at 120 km/h.
2. SURF, blob detection and Hough circle transforms are explored as an initial object detector (on stop signs).
3. In this project blob detection was re-evaluated against Hough circle transforms when used as an initial object detector on speed limit signs.
4. Two different approaches to traffic-light state detection were implemented and compared.

3.1 ALGORITHM STRUCTURE OPTIONS

Most real-time object detection system algorithms fall into one of two categories, as seen from existing systems found in literature. These categories are:

1. Single: A single method or algorithm is used to do both the object detection and object recognition steps. These algorithms use either a single classifier with a great number of features (eg. SURF) to find the target object, or many repetitions of similar simple classifiers cascaded to find the target object (eg. Viola-Jones).
2. Combinations: A couple of known algorithms are combined in various steps (eg. Canny edge detection, followed by blob detection as a detector and SVM for discrimination).

This project was realised by combining and comparing a number of known algorithms as subsystems.

Since the specifications require the system to detect both stop signs and traffic lights, the object detection needs to be combined in some way. This can either be achieved by running two different detection methods in parallel on each frame of the video, or by using/training a single method to detect both objects. The choice was dictated by the specific object detection method chosen.

For example, the Viola-Jones method seems to be popular because of its low processing requirements but the amount of training data required and the training time involved rendered it unsuitable for this project. An existing database with sufficient entries of stop sign and traffic light images could not be found. Some initial tests indicated that a combination of Hough circle transforms and a neural network executed quite fast and it is suitable for both stop sign and traffic light detection. For these reasons this method was further developed.

3.2 DESIGN GOALS

The goal of this project is to develop a real-time sensor system that makes use of the video stream received from a dashboard mounted camera inside a vehicle. The design goals can be derived from the project goals. The system must be designed to detect stop signs, traffic lights (including current state) and speed signs under normal daylight driving conditions. It must be optimised for algorithm execution speed and detection accuracy, while the hardware must be small, powered from 12 V and not too expensive. More specifically:

1. Process more than 2 frames per second.
2. Detect at least 70% of the targets correctly.
3. Hardware must be smaller than 500 cm^2 (100 mm x 100 mm x 50 mm).
4. Hardware cost below R2000 per complete unit.

In order to reach these design goals to develop a system that addresses the original problem statement, the design and implementation of the system had to be optimised to best meet the following requirements:

3.2.1 Execution speed

The system had to be optimised for execution speed to run in real-time. This means that enough frames must be captured, analysed and the necessary action taken per second so that the system does not fall behind. Even with the vehicle traveling at 120 km/h, frames should be processed fast enough so that a road sign will not be missed. A vehicle traveling at 120 km/h covers a distance of about 33 metres every second. Tests with the chosen method and camera showed that a road sign can be recognised when it is between approximately 1 and 15 metres away from the vehicle, which means that the system must process more than 2 frames per second to ensure that no road signs will be missed.

3.2.2 Low rate of false positives

The system had to be optimised for a low rate of false positives; since the goal here is specifically to monitor driver behavior in order to assign drivers a rating. This rating will be used by insurance companies, potential second-hand car buyers and other institutions and it should not unfairly penalise a driver due to false road sign recognition outputs. i.e. the benefit of doubt should be given to the driver.

3.2.3 System cost

The system, and especially the hardware, must be developed for minimum cost. To make a real difference in road safety such a system must be adopted by the majority of road users and a high cost will prohibit a large market uptake.

3.2.4 (Embedded) Hardware platform

Since the system must be installed in a user's vehicle, the hardware must be physically sufficiently small, yet still powerful enough for the computational demands set by the image processing software that it will execute. It must also be commercially available (off-the-shelf) at a minimum cost.

3.3 SYSTEM IMPLEMENTATION OVERVIEW

From the literature studied in Chapter 2 the design goal can be met by cascading simple existing image processing algorithms to implement an object detector stage, an object recognition stage, a detection filter and an output indicator. The system can be made up by developing and combining several sub-systems, and these would be comparatively tested to find the ones best suited to the application. An overview of the system can be gained by the systematic flow of information as illustrated in the system block diagram in Fig. 3.1. The purpose and function of each module will be discussed in detail.

3.4 SUBSYSTEM DETAIL

3.4.1 Dashboard camera

Since a visual approach will be used to detect road signs, images of the scene directly in front of the vehicle must be obtained. To apply the detection process in real-time, a continuous stream of images (frames) must be acquired at a sufficiently high rate. The images must be delivered in a format which can be processed by the successive image processing modules.

The camera itself must be sufficiently small and robust to be mounted on a vehicle's dashboard, or to be attached to the inside of the windscreen (typically behind the rear-view mirror from the driver's perspective). These are the best mounting positions in order to have a clear forward-facing view while still being inside the vehicle to protect the camera against the elements.

Various commercial cameras were compared in order to find one with a compatible hardware interface, sufficiently high resolution and frame-rate and an acceptable viewing angle. The size, enclosure and price also had to be taken into account.

A colour camera is only necessary if blob detection will be used, however, it is a fairly simple and quick data manipulation to change a colour image to a grey-scale one, so this is not a critical feature but may be included either way.

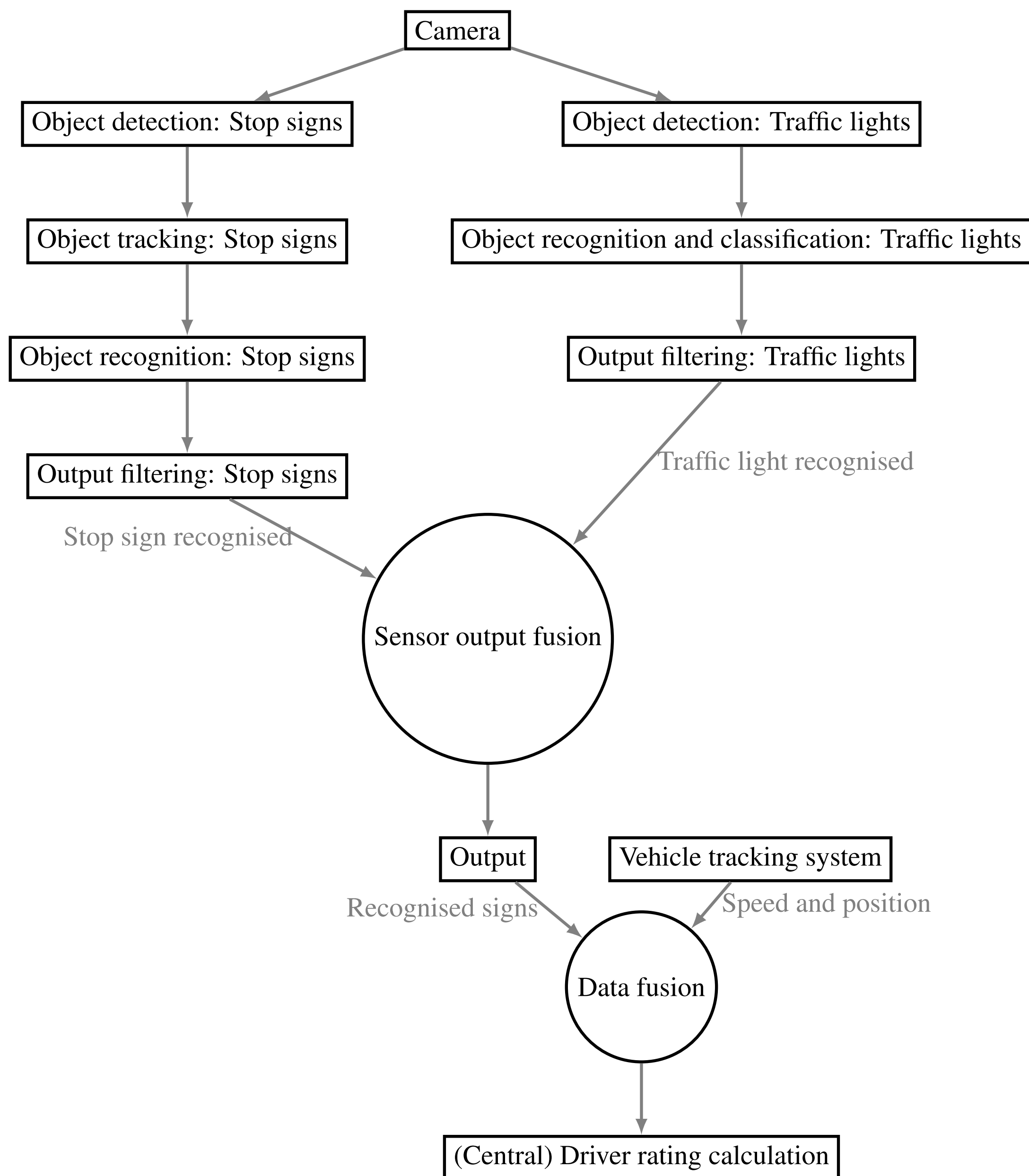

Figure 3.1. System block diagram

The picamera proved to be a good choice. It is easy to use, gives great image quality and speed and it is available from many suppliers at a very reasonable price. The picamera has the following specifications:

- A small size of around 25mm x 25mm.
- 5 megapixel resolution (2592 x 1944 pixels).
- Video recording at 1080p30, 720p60 and 640x480p60/90.
- Field of view of 65 degrees (horizontal) with the standard lens.
- Focal ratio (F-Stop) of 2.9.
- Focus is fixed at 1 m to infinity.

3.4.2 Object detection

The object detection module is used to locate possible instances of the target objects within each camera frame. The target objects are stop signs and traffic lights from the scene outside the vehicle.

In the literature study it was found that there are numerous algorithms that can be used for object detection in an image. In order to achieve the goal of a real-time detection system the speed, accuracy and processing requirements were the critical factors that determined which algorithm was chosen. Apart from these, the algorithm had to be suited to detection of the specific target objects through their visual features.

The main visual characteristics of the front face of a stop sign are:

- Octagonal shape
- Bright red face
- White lettering
- White border
- (Usually) Mounted on a vertical pole on the left side of the road
- There is usually only one stop sign on the approach side of the intersection.

The main visual characteristics of the front view of a traffic light are:

- Circular lights, with a fixed standard size (radius)
- Lights are bundled into clusters
- Lights in a cluster are mounted against a black background with a white border
- Within a cluster the lights are stacked vertically
- Background has vertical sides and a semicircular top and bottom
- Backgrounds come in standard sizes
- Some variance in light colour exits (Standard vs LED lights, age, etc.)
- An intersection usually has many clusters, some mounted to the left and some overhead. Standard configuration is one pole on the approach side of the intersection and another on the opposite side.

Clearly there are great differences between a stop sign and a traffic light, so two separate detection paths had to be implemented (as indicated in Fig. 3.1).

The number of false detections had to be kept to a minimum, since each detection results in an image sent on to the object recognition stage, which takes up more processing time.

Since the octagonal shape of a stop sign closely resembles a circle (especially from far away, or when blurred or viewed at low resolution), both the Hough circle transform and blob detection algorithms were suitable as object detectors in this project. The detection of stop signs and traffic lights will be discussed separately in more detail further on.

3.4.3 Object tracking

Once the target object has been located in a few successive frames, object tracking can predict where the object will appear in the next frame. This may increase the success rate of the object detection algorithm, since it gives a region where there is a higher probability of finding the target object in the next frame. Thus this implementation does not really require object tracking, especially if the object detection stage can detect the target object in each frame. However, when the object detector misses the target in one frame the position predicted by object tracker can be used to take a sample image as

if the object detector located the target at that position, thus sending more images to the recognition stage that are likely to contain the target object. Some of the most widely used object tracking methods are particle filters and Kalman filters [32]. Particle filters have the advantage that it can be applied to linear and non-linear applications, while the Kalman filter allows only linear problems, unless the extended Kalman filter is used. On the other hand the processing requirements to implement a Kalman filter is much less than for a particle filter, especially when using a large number of particles. Since the movement of the target object across the frame can be described by simple Newtonian movement equations and the speed of the object will be relatively constant between frames, the linear restriction of the Kalman filter is irrelevant in this application. For this reason the Kalman filter was chosen as an object tracking method.

3.4.4 Object recognition

Once the detection algorithm locates a possible instance of the target object, the object recognition algorithm must decide whether or not it really is the target object that was found. It must also classify certain sub-types of the target object, for example marking a traffic light as being in the red, amber or green state.

Some algorithms combine the detection and recognition steps, such as the Viola-Jones method [6]. The object recognition can be feature based, such as Scale Invariant Feature Transformation (SIFT) and Speeded Up Robust Features (SURF) or based on machine learning techniques such as neural networks or support vector machines (SVM).

In this project one feature based method was compared to one machine learning method in order to find which one yields the best overall performance (highest success rate and shortest processing time). SURF will be the feature based method used since it is designed for speed. For a machine learning based alternative a neural network will be implemented, since they are widely used in machine vision applications [20] [58] [60].

SURF needs to be provided with a sample image of the target object in order to look for it in each frame. For this purpose, a typical photo of the target object was taken with the camera in the same location and with the same setup as it was later used to implement the tests.

For the neural network a training set of target images must be provided. An image set was collected by driving with the system and taking sample images. Although this was time consuming it generated images with real-life lighting and background changes and from a typical mounting position inside a vehicle. The images were manually classified in order to be used for training the network.

3.4.5 Outputs

The road sign detection module can be viewed as a separate system since it will run on its own dedicated hardware. This system is then a sensor and the output of the sensor had to be in a format that can be used by the tacking system to derive a traffic sign violation system. There are various protocols for communication between embedded computer systems, but since the information from the road sign detection system is a simple boolean indicator (detected or not detected), a simple digital output line was chosen (one for each target). Using this output type has the added benefit that the output state can be indicated to the user by adding an LED for visual indication or a beeper for audible indication.

The output of the traffic sign violation system (the road sign detector combined with the speed and distance information) is of the same format, however, in order to use this information to adapt a driver rating, the vehicle tracking system must send each detection (rising edge of the output) to a central base station where the number of violations can be added up and statistically processed to derive the desired driver rating. Thus the output must also be encoded in a message in the format used by the tracking system.

3.4.6 Sensor fusion

The output of the road sign detection module is combined with speed information in order to decide if the driver has violated the detected road sign or not.

The sensor fusion step used for the output of the stop sign detection path and the red traffic light detection path was accomplished by a simple 'or' function. When either of these makes a positive detection, the road sign detection output is set.

Another sensor fusion step is applied to combine this road sign detection output with speed information from the tracking unit. The road sign detection output is set when either a stop sign or a red traffic light were detected, which means that the driver should stop. To check whether or not the driver has indeed stopped, the vehicle speed should drop to zero (or at least near zero) within a few seconds of the detection output going high. Note that there will be a time (distance) offset between the road sign detector output and the vehicle speed reaching zero. Referring to Fig. 3.2, since a stop sign normally passes out of the camera's field of view θ just before the vehicle reaches the stop line, it is shown that the system has to take into account that the vehicle will likely only stop after the detection output has been set and cleared again due to the distance d. Two approaches can be followed to solve this offset problem. The simple option is to check for an 'and' combination of near-zero speed and a delayed version of the road sign detection output, thus allowing for the vehicle to stop just after the stop sign has moved beyond the edge of the frame. The second option is to estimate the distance d, and then combine speed and time to derive the distance that may be traveled after road sign detection output is cleared before the vehicle speed must drop to near-zero.

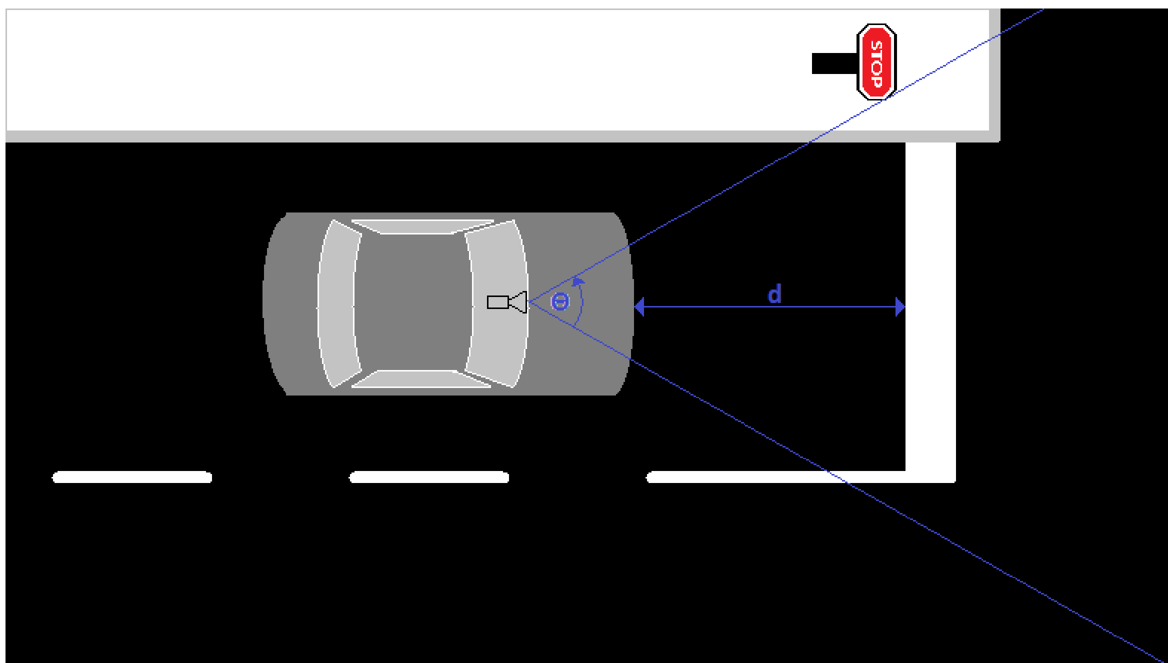

Figure 3.2. Distance to stop sign after the sign leaves the camera field-of-view

3.4.7 Software

As shown in Chapter 2, existing software libraries are available where various image processing methods and algorithms have already been implemented, such as OpenCV, Scilab and the Matlab/Octave image processing libraries. The OpenCV library was used extensively to save development time.

3.4.8 (Embedded) Hardware platform

The stop sign and traffic light state detection algorithms must run as a stand-alone system on an embedded computer platform so that it can be used as a "sensor" in a vehicle. This involves finding the most suitable hardware platform for the task and making the necessary changes to the code for it to run on the chosen platform. The chosen hardware should fulfill the processing power, availability, cost, size and power requirements for the intended end product.

Hardware had to be compared to choose one of the several single-board computers that are commercially available. Since the code is very processing intensive, only a few of the latest high-end boards were considered. All of them already satisfy the basic requirements for an in-vehicle sensor system, such as:

- Sufficiently small size to install in a hidden location or to place on the vehicle's dashboard.
- Required power must be low enough to not require special wiring changes to the vehicle, such as the addition of direct wiring from the vehicle's battery or alternator.
- The board's power supply must be direct current and of a voltage that can either be directly accessed from the vehicle's accessory power outlet, or derived from such an outlet by standard commercially available power adapters.
- The hardware may not generate excessive electromagnetic interference or noise on the vehicle's power lines.
- The hardware must operate normally across the temperature range usually encountered within the passenger compartment of a passenger vehicle.
- The system must be able to boot up within a reasonable time, or alternatively it must support a low power mode from which it can recover within a reasonably short time. This is to ensure

that it will not drain the vehicle's battery when stationary, but must still be up and running fast enough to not miss much of the start of a trip.

To do an objective comparison, the specifications of the hardware that was considered [61]–[64] are aligned in Table 3.1.

Besides performance, the availability and price should also be taken into account since the project aims to prove that the system can be implemented on off-the-shelf hardware at a reasonable cost. So a supplier for each board type was found and the prices obtained, this is shown in Table 3.2.

Table 3.1 shows that these boards should all be very close in performance, since their specifications do not differ much, but considering that the Udoo costs about double the average of the other boards, this option was thrown out. The pcDuino 3 runs on a dual-core processor at only 1 GHz and still costs more than the other two options, so it too was removed.

This left the Raspberry Pi 3 and Odroid C1+ which both use quad-core processors running at clock speeds above 1 GHz and they both cost less than R1000, so they were selected for the "short list".

To decide between these two, other factors were also taken into account. Among these were ease-of-use, support, availability of accessories and peripherals. Unfortunately ease-of-use is hard to determine for someone who has not used these boards before, so to compare this in another way a couple of internet searches for problems, solutions and queries regarding each board type were made. The result of this search led to the next element, which is support, and in this regard the Raspberry Pi is far ahead. This is not official support from a manufacturer or distributor, but rather the informal answers, articles and tutorials written by other developers and published on the internet in forums, blogs, etc.

The last element to consider is accessories and peripherals. Here the popularity of the Raspberry Pi counted in its favour again, with many different enclosures, functions, applications and examples being available for it. The most relevant to this project being the picamera, which is a five mega pixel and 30 frames per second capable camera specifically made to work with the Raspberry Pi through a dedicated film cable camera interface port on the Pi board.

Table 3.1. Specifications on popular single-board computer hardware

Component	**Raspberry Pi 3**	**Udoo**	**pcDuino 3**	**Odroid C1+**
CPU	ARM Cortex-A53	Freescale i.MX 6 ARM Cortex-A9 CPU	Allwinner A20 ARM Cortex A7	Amlogic ARM® Cortex®-A5(ARMv7)
CPU cores	4	2/4	2	4
Clock	1.2GHz	1GHz	1GHz	1.5Ghz
GPU	Broadcom Video-Core IV	Integrated graphics (OpenGL® ES2.0 3D and OpenVG)	OpenGL ES2.0, OpenVG 1.1, Mali 400 Dual Core	Mali-450 MP2 OpenGL ES 2.0/1.1
RAM	1GB LPDDR2 (900 MHz)	DDR3 1GB	DRAM 1GB	1Gbyte DDR3 SDRAM
Cache	32kB Level 1 + 512kB Level 2	N/A	N/A	N/A
Storage	MicroSD	4GB Flash, SATA	microSD card (max 32GB), SATA Host	Flash Storage slot, UHS-1 SDR50 MicroSD
Network	10/100 Ethernet + 2.4GHz 802.11n wireless	Ethernet RJ45 (10/100/1000 MBit) + WiFi	N/A	Gigabit Ethernet
Display	HDMI + DSI	HDMI + LVDS	HDMI, LVDS LCD	N/A
Camera interface	Camera Serial Interface (CSI)	Camera connection	N/A	N/A
Other interface	4× USB 2.0, 40-pin GPIO header	4x USB, 76-pin GPIO, Analog I/O	Arduino headers	40pin GPIOs + 7pin I2S + USB 2.0 x 4
Supported OS	Raspbian (and others)	N/A	Lbuntu 12.04 / Android	Ubuntu or Android OS

Table 3.2. Availability of popular single-board computer hardware (on 2016/04/11)

Board	Available from	Price
Raspberry Pi 3	Communica	R934
Udoo	http://www.digikey.com	$142 (Approx. R2085)
pcDuino 3	Communica	R1105
Odroid C1+	http://www.cyberconnect.co.za	R790

This camera, costing R548 and having a selection of housings available for it, along with the other elements already mentioned above, tipped the scales in favour of implementing the system on a Raspberry Pi 3 board.

Now that it has been decided to use a Raspberry Pi 3 hardware platform, the hardware had to be obtained, set up, and the detection system made to run on it.

The Raspberry Pi is a single board computer, but several peripherals are still required to use it like a PC. Although not all of these are required when using it as an embedded system, they are still necessary to get the system running and to implement this particular application on it. These include the Raspberry Pi 3 model-B single board computer itself (with suitable enclosure), a micro SD-card (Verbatim 16 Gb, class 10), a USB keyboard and mouse, USB to Micro-USB cable connected to a 5 volt 1 ampere power supply, and HDMI cable (connected to a suitable screen) and the Pi-camera (5 mega-pixel stills or 1080p30 video or 720p60 video) with suitable enclosure. A 40-pin header connector with ribbon-cable was added for system output (and input when collecting videos).

After studying various Raspberry Pi setup techniques, it seemed that the simplest way would be to use the "New Out Of Box Software" (NOOBS) package, which was written specifically for this purpose. Using NOOBS, Rasbian was loaded onto the SDCard, then everything was connected together and the system started up for the first time. All the other necessary software and libraries (such as OpenCV) were downloaded to the Pi over WiFi. Finally an application can be made to run automatically after boot up by adding it to the /etc/rc.local file.

3.5 IMPLEMENTATION

3.5.1 Target object: Stop signs

There are a number of possible approaches for the initial detection of a stop sign in the frame image, which will execute fast enough for an embedded processor to do in real-time, among these fast radial symmetry transform [3] and Hough circle transform [8] look promising. Since stop signs are always predominantly red and of a regular shape, blob detection can also be used [2] [21]. The problem with using colour is that the exact pixel colour value can vary over a great range of values. This exact value is a function of many things, such as the camera itself, camera settings, the illumination of the object, weather conditions, the actual colour of the object (paint fades), etc. As such, it is necessary to work with either a range of values or with the colour values relative to each other, which becomes computationally intensive and often takes too long to process. For this reason the initial focus for an stop sign detector was on the Hough circle transform, although blob detection later turned out to give better results.

At the early stages of the project, video data was recorded using a Vacron CDR-E07 dashard camera with a 720p (1280 x 720 resolution) output and a 120 degree wide-angle lens. The device also has a rear-facing camera, audio, speed and acceleration sensors, but only the forward facing camera's video information was used since that is common on all dashboard cameras. This camera was used for initial tests only, until the picamera was chosen as the best camera for the project. Several trips were recorded to include different lighting, road and weather conditions. Each trip was broken up into several smaller video segments. A simple program was written in Python using openCV to break the video up into smaller sections so that only the portions that contained stop signs or traffic lights could be kept.

Since many road signs have a round shape, the Hough transform should work great in detecting them in an image. However, this application is only concerned with stop signs and traffic lights. Stop signs are octagons, which may prove to be a problem for the circle transform, so that left two avenues to explore:

- Use the Hough line detector function to find lines in the image. The eight lines making up the outside of the stop sign can be found (grouped) since they should all be multiples of 45-degrees

and when lines are drawn perpendicular to each and through its centre, these lines will all pass through a single point in the octagon's centre.

- Down-sample the image (make it smaller) and apply blurring to make the octagon appear like a circle. The Hough circle transform can then be applied to find the stop sign as a circle in the image.

Both of these methods were implemented as a trial.

The first method did not work very well, since the line detector did not always detect all 8 lines forming the octagon or only parts of those lines. Further, the angle between the camera and the stop sign also distorted the octagon to a point where the lines no longer formed a good octagon. A sample of a video frame containing a stop sign after Canny edge detection is shown in Fig. 3.3, and the same image is shown in Fig. 3.4 where the red lines indicate lines resulting from the Hough line detection feature. It can be seen that the white edge of the stop sign results in a double octagon and that some segments of the octagon are not complete. Shape distortion due to the viewing angle is also apparent. All of these flaws combine to make this a rather complex approach to stop sign detection.

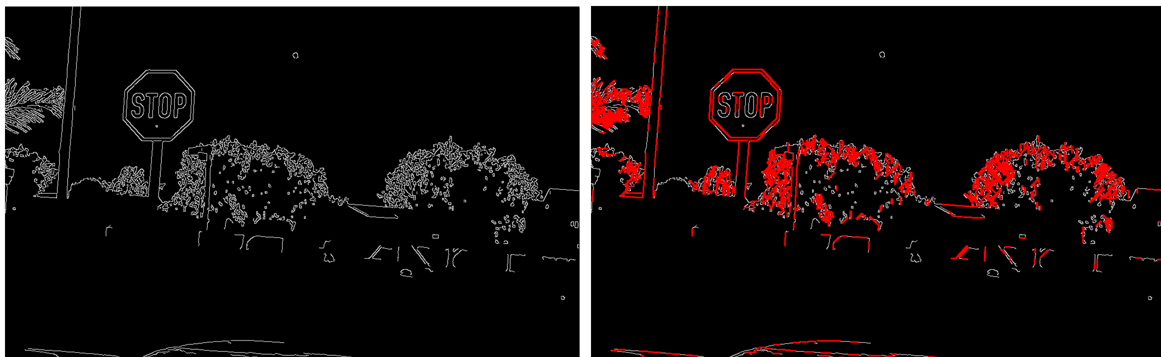

Figure 3.3. Stop sign image after applying Canny edge detection

Figure 3.4. Hough line detection applied to a stop sign image

The second method worked better. By moving the original 1280 x 720 image two steps down the resolution pyramid and applying Gaussian blurring the Hough circle transform was able to detect the stop sign as a circle when it is about 10 metres away until it is so close that the angular distortion makes it appear so elliptical that the transform no longer recognizes it as a circle (when the vehicle is estimated to be about two metres away). The circle transform also yields less resultant circles than the

number of lines from the line transform, which leaves less time to be spent sifting through the results to find the actual stop sign. The result of applying the Hough circle transform to the same image (as the Hough line detector) is shown in Fig. 3.5, note the smaller image (lower resolution) due to the sub-sampling.

Figure 3.5. Hough circle transform applied to a stop sign image

In order to use the Hough transform on an image, it has to be converted to gray-scale and reduced to its basic lines by applying an edge detection algorithm such as the Canny edge detector. The openCV function ’cv2.cvtColor’ was used to make the colour video frame a gray-scale image. As for the edge detection, this is automatically applied when calling the ’cv2.HoughCircles’ function, where ’param1’ specifies the line detection threshold. Applying Canny edge detection separately does not seem to affect the results of the circle transform.

Once the Hough transform has detected the circles in the image, a small square sample is taken from the gray-scale image centred at the detected circle’s centre point and with each side being 2R long. The sample (or box) is then resized to be 32 x 32 pixels in order to always have a fixed resolution picture which can be used as input to the neural network classifier. One such sample image is shown in Fig. 3.6, but take note that for better visibility the image size has been doubled here. Once the classification

Figure 3.6. Sample box image taken around a detected circle

stage has positively identified a stop sign from one of the detected circles, that object is tracked by applying a Kalman filter to its position in the original image. Once the Kalman filter is active the

output position of the Kalman filter (its estimate of where the tracked object should be in this frame) is used to take another image sample (box) which is also fed through the neural network as if the Kalman filter's position estimate was another detected circle. The only difference is that the radius of the last positively identified stop sign circle is used.

Next the sampled image of 32 x 32 pixels is put through a neural network to classify it as either a stop sign or not a stop sign. To do this the image matrix is first "rolled out" into a single vector (array) with 1024 elements. The neural network used is of a simple three layer architecture of 1024-1024-1, since it takes the 1024 image pixels as input and only require a single output neuron to indicate the classification result. A truncated version of the neural network architecture is illustrated in Fig. 3.7, showing the (I) input-, (H) hidden- and (O) output-layer. The commonly used sigmoid function

$$\Phi = \frac{1}{1+e^{-z}}, \tag{3.1}$$

was chosen as the neuron activation function.

Once the initial Canny-Hough-Neural network system was implemented the captured video segments were applied to the system. To verify the system's output, circles or tracked positions that were positively classified were drawn in colour onto the original gray-scale video frame for visual confirmation. Detected stop signs were circled in green, while detections from the Kalman filter output were circled in blue. A few sample frames from the resulting videos are shown here in Figures 3.8 to 3.11.

So the steps that made up this initial stop sign detection system were:

1. Read the video frame and convert it to gray-scale.
2. Resize it to the standard resolution of 1280 x 720 if the camera provided a different resolution.
3. Sub-sample twice to reduce the image size (resolution) to 320 x 180 for better performance.
4. Apply Canny line detection (this step is included in openCV's Hough Circle function).
5. Apply the Hough circle transform.
6. Take a sample image block around each detected circle, and also at the Kalman filter's predicted position.
7. Resize the sampled block to 32 x 32 resolution.
8. Feed the sample block to the neural network to classify it as stop-sign/non-stop-sign.
9. For successive positives, apply their positions to the Kalman filter to track the detected object.

Figure 3.7. Stop sign classification feed-forward neural network

Figure 3.8. Detected stop signs

Figure 3.9. Detected stop signs

Figure 3.10. Detected stop signs

Figure 3.11. Detected stop signs

10. Display detected stop signs by drawing a coloured circle onto the original gray-scale image.

The results of this initial stop sign detection system are discussed in Section 4.1.

3.5.2 Target object: Traffic lights

Since there are several lights on each traffic light and they are all of a circular shape, the Hough circle transform proved to be ideal for detecting the lights on a traffic light. As with the stop signs the video frames have to be converted to gray-scale and reduced to its basic lines by applying an edge detection algorithm such as the Canny edge detector. The openCV function 'cv2.cvtColor' was used to make the colour video frame a gray-scale image and as mentioned before the 'cv2.HoughCircles' function already incorporates edge detection. The big difference is that no sub-sampling takes place this time. This is due to the traffic light circles being perfect circles (no approximation required as with stop signs) and also due to the lights being relatively small, especially the ones that are far away (on the opposite side of a big intersection). The smaller circles also resulted in significantly different optimum parameters for the transform function. Since the Hough circle transform detects a single light which is only a part of the traffic light cluster, it is at this stage unknown which one of the 2 or 3 lights in the cluster was detected. To overcome this, a vertically elongated rectangular sample is taken from the frame to ensure that the sample will include all lights of the cluster regardless of which one was originally detected. A sample of such an elongated box is shown in Fig. 3.12, note that the image size has been doubled here for better visibility.

Once the rectangular sample is taken it is passed to the next module (object recognition step). The

Figure 3.12. Elongated sample box image showing the secondary circles detected within the box itself

sampled image is again "rolled out" into a vector to be used as input to the classification neural network. The neural network will classify it as either a standard traffic light cluster, or a turn-signal traffic light cluster. The neural network used is once again a simple 3 layer architecture, this time of 512-512-2, since it takes the (16 x 32 =) 512 image pixels as input and requires 2 output neurons to indicate either a standard or a turn light cluster. A truncated version of the neural network architecture is illustrated in Fig. 3.13, showing the (I) input-, (H) hidden- and (O) output-layer, and again the commonly used sigmoid function

$$\Phi = \frac{1}{1+e^{-z}}, \tag{3.2}$$

was chosen as the neuron activation function.

Once the initial Canny-Hough-Neural network system was implemented the captured video segments were applied to the system. To verify the system's output, the positions of standard traffic light clusters are indicated by drawing a red rectangle while turn-signal traffic light clusters are framed by a turquoise rectangle. These are drawn on the original gray-scale video frame for visual confirmation. Object tracking (such as a Kalman filter) has not been added yet due to there often being multiple detected traffic lights in one frame. An example of the result is shown in a couple of sample frames shown in Figures 3.14 to 3.17.

So the steps that made up this initial traffic light detection system were:

Traffic light detection routine:

1. Read the video frame and convert it to gray-scale.
2. Resize it to the standard resolution of 1280 x 720 if the camera provided a different resolution.
3. Apply Canny line detection (this step is included in openCV's Hough Circle function).

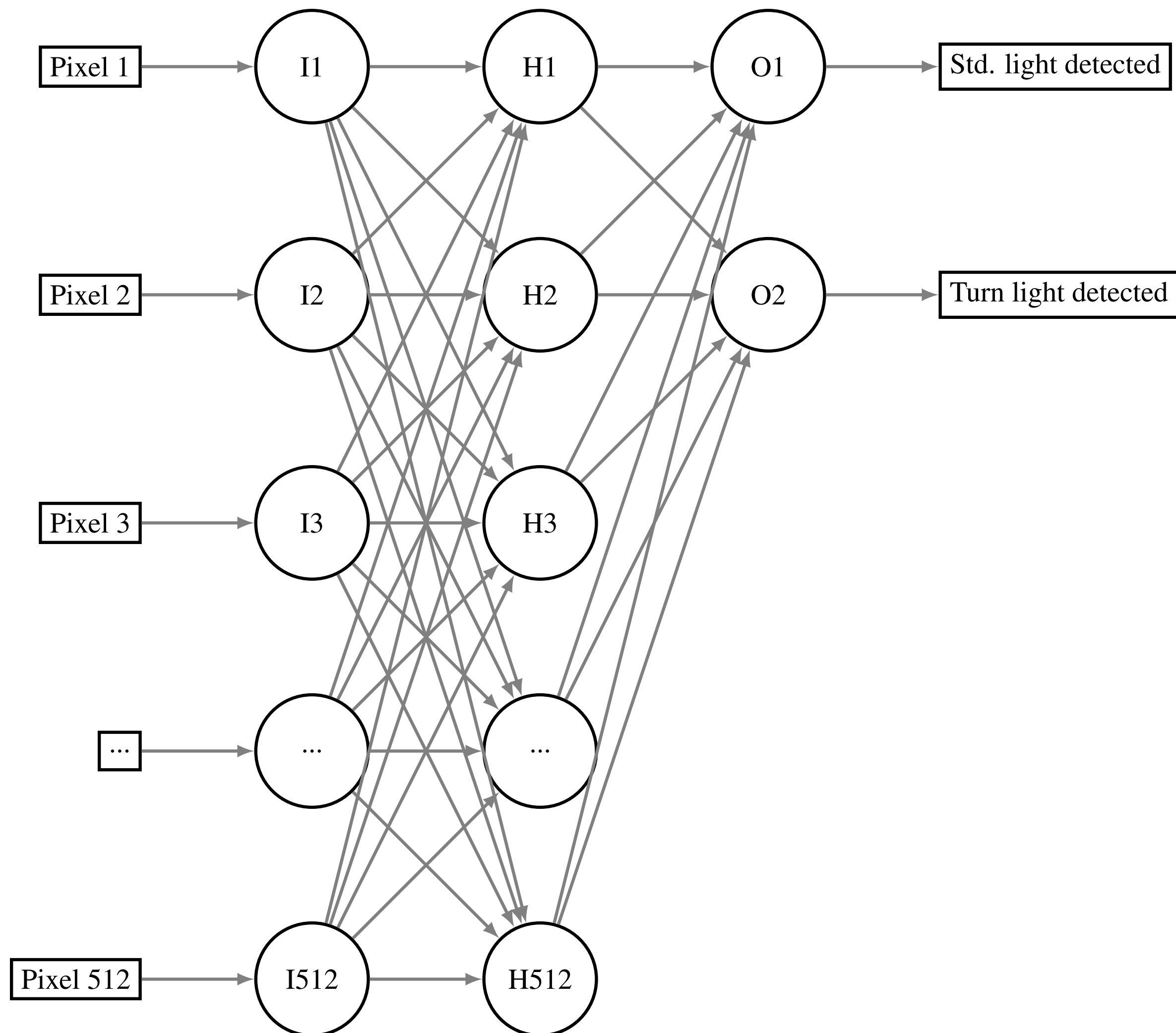

Figure 3.13. Traffic light classification feed-forward neural network

Figure 3.14. Detected traffic lights

Figure 3.15. Detected traffic lights

Figure 3.16. Detected traffic lights

Figure 3.17. Detected traffic lights

4. Apply the Hough circle transform.
5. Take a sample image block around each detected circle, extend the block vertically to allow for any of the 2 or 3 lights in a traffic light cluster detected.
6. Apply the Hough circle transform again, this time to the sampled box to determine where in the box the lights are located.
7. Based on the location of the centres detected by the circle transform, crop the sampled box around the traffic light cluster.
8. Resize the sampled block to 16 x 32 resolution.
9. Feed the sample block to the neural network to classify it as a traffic light turn-signal cluster, traffic light standard cluster or not a traffic light.
10. Display detected traffic light clusters by drawing a coloured rectangle onto the original gray-scale image, using turquoise for a turn-signal cluster or red for a standard cluster.

The results of this initial traffic light detection system are discussed in Section 4.2.

This method worked well to detect a traffic light cluster, but to determine when the driver violated the traffic light rule the current state of the traffic light had to be determined. Two different approaches were implemented and their results compared to find which one is best suited to this application. The details of this comparative analysis can be found in Section 3.9.

3.5.3 Optimisation of the Hough circle transform parameters

The performance of most image processing algorithms are very dependent on the settings or parameters passed to them. The Hough circle algorithm and blob detection algorithms are no exceptions, so to optimise the parameters of these algorithms for the task of detecting stop signs (and traffic lights and speed limit signs), the method illustrated in Fig. 3.18 were followed. Typical parameters for the Hough circle transform were DP of 1, minDist between 1 and 3, param1 between 300 and 600, param2 between 8 and 20, minRadius between 2 and 12 and maxRadius between 4 and 24. For example, the best parameters for detecting speed signs were Dp = 1, minDist = 1, param1 = 400, param2 = 20, minRadius = 12 and maxRadius = 19.

3.5.4 Data-set for neural network training

For a neural network to perform a specific function like image classification or recognition, it must be trained to do so. Back-propagation was chosen as a learning method since it is simple, effective and relatively fast. A data-set of sample images is required to train the neural network, and since an existing set could not be found it had to be collected.

Initially 31 video snippets with stop signs and 24 video snippets containing traffic lights were recorded and processed to be used for the first training set. More training data were collected for the comparative test done later on, but the same basic steps were followed.

To construct a set of good training images and their corresponding classification information, 2 simple programs were written in Python (and later ported to C++) that executes the first couple of steps of the stop sign and traffic light detection methods. At the stage where a sample box has been taken and resized, the box is then either displayed to the user who can use keyboard input to manually assign each image into the class that it belongs to, or simply saved to disk. All the images saved in this manner, except for a few very bad samples that were removed manually, were selected to be used as the data-set. Usually 75% of the images were kept as a training set while the rest was moved to a separate folder as a validation-set. For stop signs the valid inputs are shown in Table 3.3, and for traffic lights in Table 3.4.

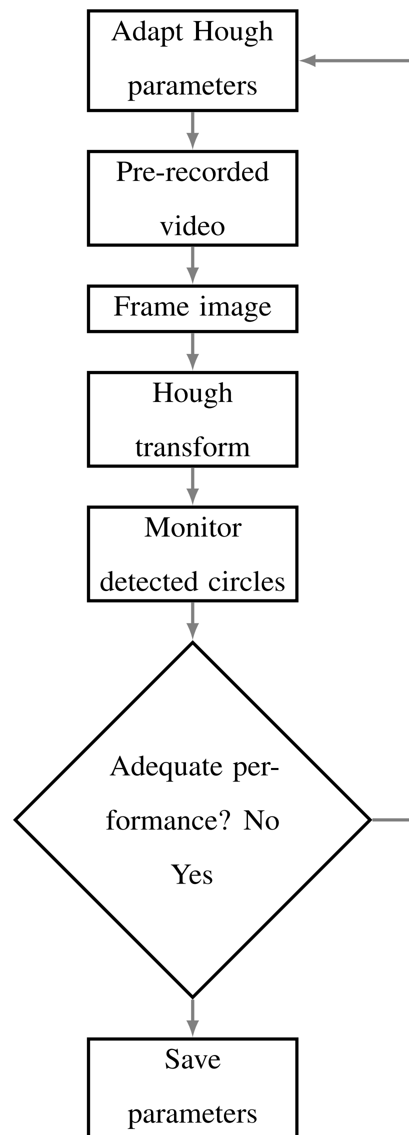

Figure 3.18. Program flow for optimising the Hough circle transform parameters

Table 3.3. Stop sign manual classification inputs

Valid input character	Class
’y’	Yes it is a stop sign
’n’	No it is not a stop sign

Table 3.4. Traffic light manual classification inputs

Valid input character	Class
’0’	Not a traffic light
’1’	Standard traffic light cluster
’2’	Turn-signal traffic light cluster

To simplify referencing this information during the training process, the classified images were initially all stacked together horizontally (like a long array of images in a single matrix) and written to a file, although later on each image were simply saved separately. To annotate the data, the class input from the user is written to a text file where each character represents the class for the corresponding stacked image in sequence. These programs were applied to all the training video files that were collected and sorted as training samples for stop signs and traffic lights. A stacked image set and the corresponding classification text file were stored for each video. An example of a stacked images set for one of these videos is shown in Fig. 3.19 and the corresponding class text is shown in Fig. 3.20.

Figure 3.19. A horizontally stacked set of stop sign training images

```
nnnnnnnnnnnnnnnnnnnnnnnnnynyynnnnynnnynyn
```

Figure 3.20. Stop sign training classes text

The diagrams in Fig. 3.21 show the program flow for the two different methods of collecting training data. In the first (initial) method a pre-recorded video is used and the images are manually classified and then saved on a per-frame basis, while the second (later) method saves the training images directly from the live camera feed and classification is later done by the user in a separate program. A sample of the annotated data (for stop signs) in the output file of the classification program using the second method is shown in Fig. 3.22, note that the image file name is recorded against the class that it was assigned to by the user.

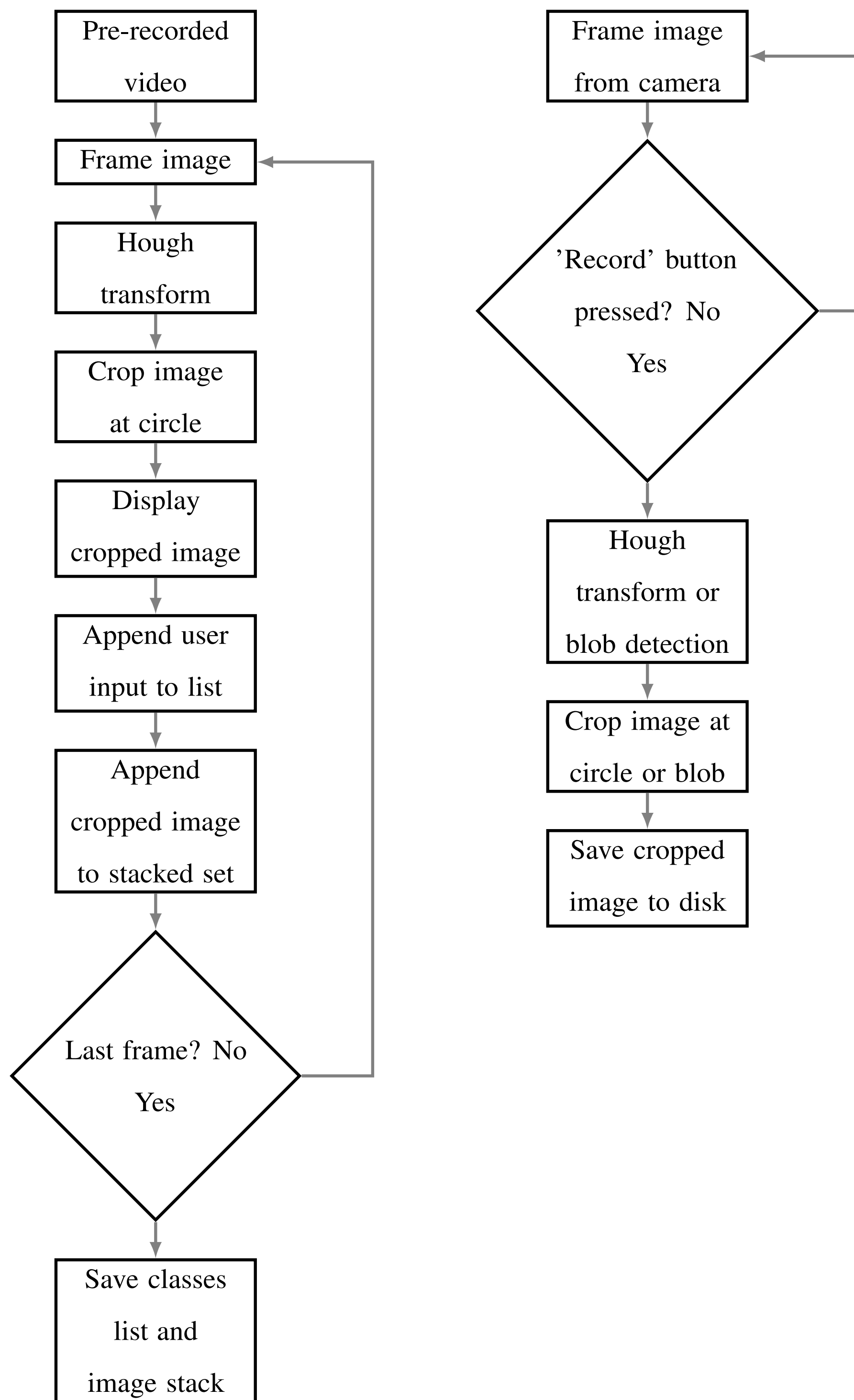

Figure 3.21. Program flow for two methods of collecting image data sets

```
01483993454_677160.jpg,n
01483993673_697037.jpg,n
01483888858_242637.jpg,n
01484400027_429907.jpg,n
01483993454_753770.jpg,n
01484399888_278745.jpg,y
01484399888_732008.jpg,y
01484246457_646926.jpg,y
01484399888_496027.jpg,n
01483993702_335642.jpg,n
01484399939_521136.jpg,n
01484246533_339461.jpg,n
01483615113_255140.jpg,y
01484246531_661986.jpg,y
01484399889_047181.jpg,y
01484399958_723189.jpg,n
```

Figure 3.22. Section of the data classification program's output file

3.5.5 Training the neural network

Now that a training set is available in easy to use format, the networks can be trained. A training program was written for each network type, which iterates many times through the number of videos that were used to collect training data for each network type. After initializing the weights matrices with random values between -1 and 1, the program performed the following steps on each video number :

1. Open and load the horizontally stacked image set corresponding to the video number in this iteration.
2. Open for reading the text file with the classification information that also corresponds to the video number in question.
3. Iterating through each image in the horizontally stacked set:

    (a) Isolate and load into a matrix the next image in the set.

    (b) "Roll out" this matrix into a vector.

    (c) Add this vector as another line in this video's training images matrix.

    (d) Read the next character from the classification text file and convert it to a class number.

    (e) Add the class number to an array of classes for this video's training images.

4. Using the entire images matrix for this video (where each row represents one training image), perform a normal forward pass through the network to obtain the outputs.
5. Calculate the error term by getting the difference between the calculated output (from the forward pass) and the given classes array.
6. Using the error term, perform back propagation through the network, with the gradient of the sigmoid function $\Phi(z)(1-\Phi(z))$.
7. Calculate the sum of the squared error (SSE) and add it to a grand SSE for all the videos

After each pass through all the video numbers, the grand SSE is displayed to indicate the training progress.

3.5.6 System outputs

To make the system useful for detecting when a user did not stop at a stop sign or a red traffic light, the system's output must be in a format that will make it easy to combine with the vehicle's tracking system so that the stop sign/red traffic light detection can be used as a 'sensor' input to be combined with the speed (and perhaps location) of the vehicle.

Herein lies a few challenges:

- **The traffic light right-turn problem:** The detection algorithms run on a dashboard camera that is installed in the vehicle to always point in the same direction (forward). This is great when a vehicle approaches a traffic light since the camera will face the traffic light which can then be detected. When the vehicle enters the intersection the first row of traffic lights, which should be in the green state now, disappears behind the vehicle and out of the video frame, but fortunately the second row of lights (on the opposite side of the intersection) are now close enough to detect the traffic lights and their current state. The problem comes in when the user turns the vehicle into the intersecting street, since the camera now faces the traffic lights for that street which should be in the red state.

  The system will set the detection output when it detects the red traffic light. The tracking system will combine the indicated red light state with the vehicle's speed, which will not have gone down

to zero since the driver entered on a green traffic light, and consequently trigger an incorrect traffic light violation. This happens mostly on a right turn, since South Africans drive on the left hand side of the road, which leaves a greater distance to face the intersecting street's traffic light and detect the wrong state.

- **Intermittent detection:** Another problem is that the detection sub-system does not detect the target object in every frame. This and other factors sometimes causes the neural network to reject or incorrectly classify the target object, causing an irregular "jittery" detection output because of these missed detections.

The problems mentioned above can be overcome to some degree by the way in which the detection algorithm's outputs are translated to the system's outputs. During the conversion of the stop sign detection output and the traffic light state detection output to a set of system outputs, they should be filtered over several frames in order to give a steadier output. At first a timer based filter was used to do this. Each time one of the algorithms returns a detection state other than 'none', a timer corresponding to the returned state is started (or re-started if it is already running) for a couple of hundred milliseconds. As long as the timer is running the system output for that state is active.

This scheme worked great for smoothing the intermittent output, but it actually made the false detection problem worse, since a single false detection is now shown on the system output for as long as the timer runs instead of only on 1 frame.

A different approach was then followed that would solve both problems at once. This was achieved by writing the detected states into a first-in-first-out (FIFO) buffer so that a history of the states of the last few frames are kept. On each frame the detected state is added and the oldest entry is discarded to keep the buffer at a constant length. After the latest entry has been written in, the contents of the buffer is checked and the state with the most instances (votes) is set as the system output. In this way the state detected by the last few frames is combined to make a decision on what the system output should be.

On the stop sign algorithm the number of "stop sign found" outputs are counted and if they exceed a certain threshold the system output is set to indicate a stop sign.

On the traffic light algorithm the number of *green* states and *red* states are counted and compared. If

the number of *red* states in the buffer exceeds the number of *green* states and also exceeds a certain threshold then the system output is set to indicate that a red traffic light is detected. The same rule is applied for the *green* state. In this way the system output is not just a function of the current state detected but also of the majority state detected in a number of historical frames. Furthermore a minimum number of detections are required so that spurious false detections are ignored when there are no traffic lights in the current section of video. Turn-signal detection was not added to this mechanism (but can be).

By only setting the output when a minimum number of detections are present in the FIFO buffer, the right-turn problem mentioned can be solved to some degree since the intersecting street traffic lights are only in the camera view for a very short period of time. If the driver pulls away slowly it may however still show up in enough frames to trigger the output. A proposed complete solution for the right-turn problem is to further filter the data using accelerometer or location (GPS) data. In this case a detected red light can be ignored when it immediately follows a green light while the additional sensors indicate that the vehicle is turning. Since this is actually an implementation exception (not a requirement of the road sign detection sensor), it should not be implemented within the sensor system itself, but rather in the system where the road sign detection sensor is used, which places it beyond the scope of this project.

A sample of the FIFO buffer during a video containing a green traffic light is shown in Fig. 3.23. In the example figure the FIFO buffer contains a “g” where a *green* traffic light state was detected, while a single “r” shows where the traffic light was incorrectly classified in the *red* state. The “n” entries shows where there were frames where the algorithm could not detect the state of the traffic light. In the figure it can be seen that the FIFO buffer was set to keep 30 entries which means that the detection history is kept for 30 frames (or just over 1 second) of video. The FIFO buffer contents of the very next

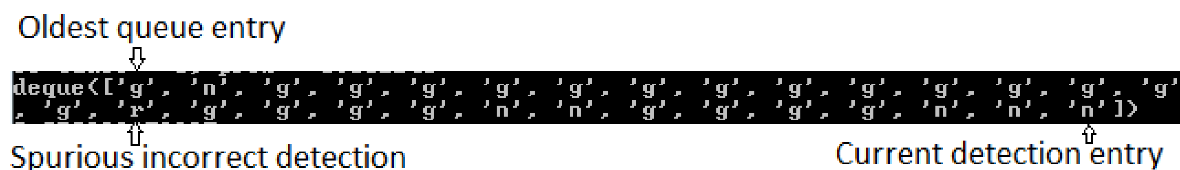

Figure 3.23. Printout of the detection state FIFO buffer

frame is shown in Fig. 3.24. It can now be compared to Fig. 3.23 to see how all the values have shifted to the left and the new (current) frame’s detected state have been added on the right hand side.

```
deque(['n', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g'
, 'r', 'g', 'g', 'g', 'g', 'n', 'n', 'g', 'g', 'g', 'g', 'n', 'n', 'n', 'g'])
```

Previous frame's entry

Current frame's new entry

Figure 3.24. Printout of the FIFO buffer on the next frame

Now the output of this iterative (FIFO buffer) filter can be written out as the traffic sign detection system output, which can be indicated to the user and also used by the vehicle tracking system.

3.5.7 Combining the sub-systems to build a complete real-time system

Once the neural network was trained, the real-time video frames from the camera can be passed through an initial detector (Hough circle transform or blob detector), and the detector's output can be used to sample a small section of the video frame which is passed to the neural network for classification. The output of the neural network indicates whether or not the target object was detected in the image. This output is further filtered and buffered and finally written to a hardware output pin. The process is illustrated in Fig. 3.25.

3.6 ACHIEVING A REAL-TIME FRAME RATE

In order to realise a real-time system the processing rate is of great importance. This is an even greater challenge when working on embedded or portable hardware that usually don't have the brute power of a desktop PC or server. For an embedded system to work in real-time a balanced approach should be followed for optimal speed. This means not only getting the best or fastest hardware, but also to optimise the code and algorithms used for fastest execution speed. The code can also be modified to make optimal use of the available hardware, such as spreading the processing tasks across multiple threads in order to make use of all the processor cores (if there are multiple cores available). This section describes the steps taken to achieve a satisfactory detection frame-rate on the chosen embedded system.

Initially frames were obtained from the picamera by calling the "camera.capture_continuous()" [65] [66] python driver function, since this seemed to be the simplest method of obtaining image frames

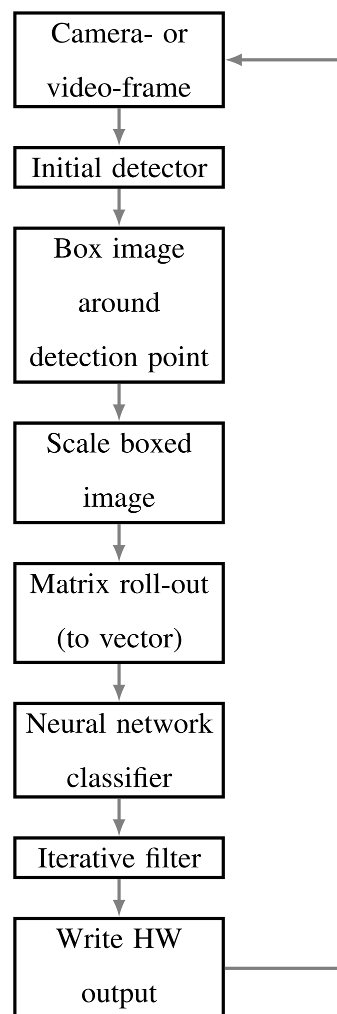

Figure 3.25. Program flow illustrating how the different sub-systems are inter-connected

from the camera. This function also allows the user to dictate when the next frame should be delivered, since it simply gives one frame every time the function is called (typically in a 'while' programming loop).

This worked great except that it takes significant amount of time after the function is called until the next frame is delivered. This time can be accurately measured using the frame-rate LED output. By switching the output on while doing the algorithm's processing and off while obtaining a frame, the durations can be seen using an oscilloscope connected to the output. Fig. 3.26 shows the output to this frame-rate LED, where it can be seen that both the frame processing time and the time spent obtaining the next frame are around 250ms each. Thus the resultant effective frame rate for the whole system is only two frames per second...which, although close, is not quite sufficient for a true real-time system.

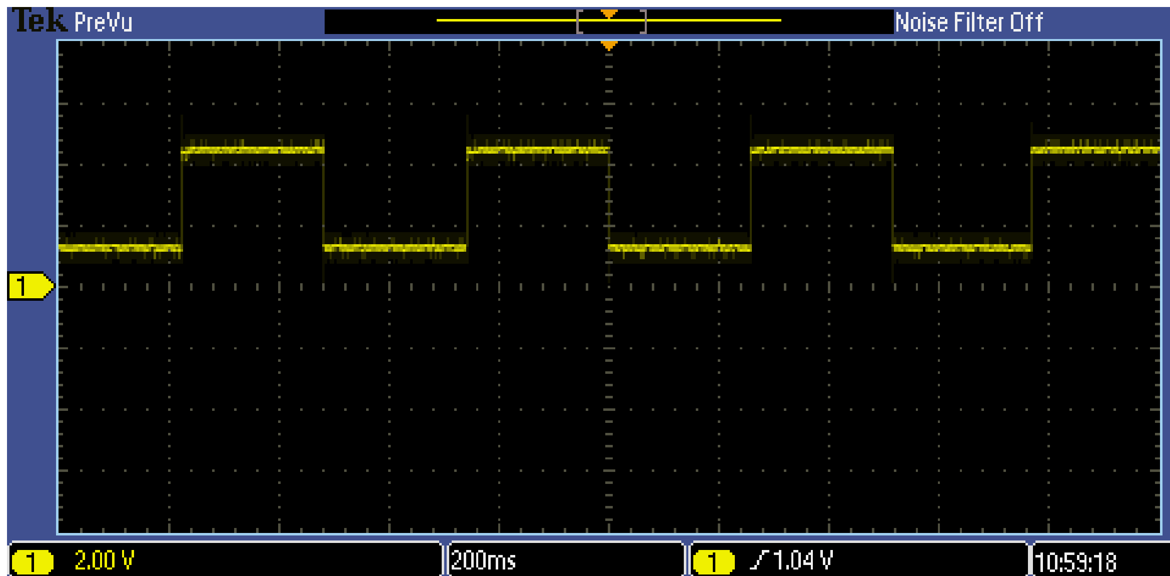

Figure 3.26. The complete system's frame-time output captured on an oscilloscope

To optimise the code for execution speed, these different approaches were combined:

- Find the fastest capturing method of the picamera driver that can be used for real-time applications.
- Find the fastest codec format which can be used with the method in step (1).
- Split up the code to run independent processes/function in different threads so that the processing load can be distributed across more than one CPU core.

3.6.1 Capturing method

Instead of using the "camera.capture_continuous()" function to obtain an image once the previous one's processing has completed, another method can be used. The other method uses the camera more like a video recorder, and is initiated by calling the picamera driver function "camera.start_recording()" [65] [66] with a custom stream to make the frame data available for processing. Before porting the entire algorithm to work with this function, both of them were first implemented using the RAW format in a bare script that only reads from the camera and sets the frame-time output LED so that the time it takes

to obtain a frame with each method can be measured. The results are shown in Fig. 3.27 and Fig. 3.28. In each of these the low sections of the signal indicate the time spent waiting for the next frame.

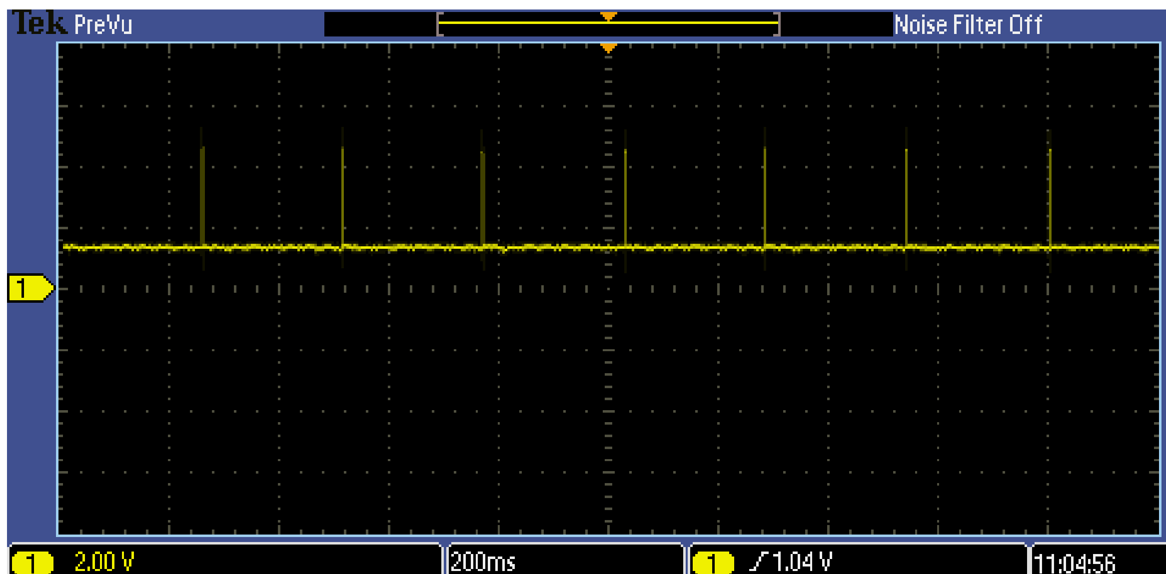

Figure 3.27. Frame-time output for the "capture_continuous" method without any processing

The results show an improvement from 250 ms to 210 ms. Although better, this is still not ideal since 210 ms still translates to only 4.8 frames per second (without any processing done on the frame), which is far from the 30 frames per second advertised for the camera. Besides that, using the "camera.start_recording()" has another disadvantage, which is the problem that the custom stream function where the driver writes the frame data to, must complete and return before the next frame is due. The algorithm in this application has a variable processing time that depends on the number of circles found by the Hough circle transform. Therefore it is difficult to determine a good fixed time which will allow for the longest frame processing time possible. Selecting a time that is on the safe side (longer) means most of the time when few circles are detected the system will not be running at optimum speed, since it has to allow for the possibly longer processing time when more circles were detected, while choosing a shorter time runs the risk of 'stalling' the process by processing too long to return before the next frame is due. So although slightly faster, this approach is not ideal, and still not as fast as the camera should be able to go.

Another option is the "camera.capture_sequence()" method. In [67] the author claims to have reached 30 frames per second but at a 1024 x 768 resolution and using the JPEG format, but to make it run at the claimed speed a queueing system was used that runs in separate threads. This was attempted but not even when using the exact code sample in [67] did this setup reach such a high frame rate. The

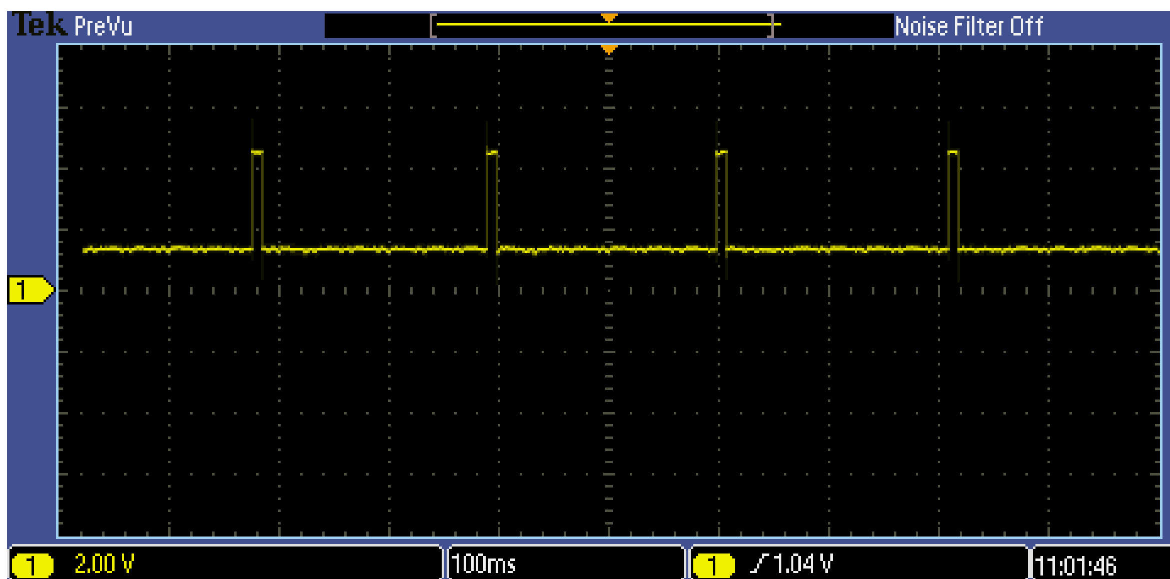

Figure 3.28. Frame-time output for the "start_recording" method without any processing

different threads does not execute at regular intervals and thus causes erratic processing of frames, which is not ideal in this sort of application. On the other hand, the author states on this method: "If you are intending to perform processing on the frames after capture, you may be better off just capturing video and decoding frames from the resulting file rather than dealing with individual JPEG captures." The problem with this suggested approach is that it is intended for recording a video file to the SDCard and then processing the frames by playing back the video file. This is not a real-time method so it cannot be used in this solution.

Apart from the driver method that is used, the way in which the format is read from the camera also plays a role. As shown by the Table 3.5 that was adapted from [68], the capturing method should be matched to the correct resolution for optimum speed. Choosing a resolution that needs conversion adds a lot of processing time to each frame that is captured.

The closest suitable resolution to the 1280 x 720 that was used before is 1296 x 730, so that is the one that will be used. Switching from the standard 1280 x 720 to the exact 1296 x 730 specified in the camera hardware table significantly increased to the obtainable frame rate.

So although the "camera.capture_continuous()" method is the simplest to work with, it looks like it is also the slowest method. "camera.capture_sequence()" is slightly faster but does not suit this

Table 3.5. Picamera's optimum modes

#	Resolution	Framerates	Video/Image	FoV
1	1920x1080	1-30fps	V	Partial
2	2592x1944	1-15fps	V I	Full
3	2592x1944	0.1666-1fps	V I	Full
4	1296x972	1-42fps	V	Full
5	1296x730	1-49fps	V	Full
6	640x480	42.1-60fps	V	Full
7	640x480	60.1-90fps	V	Full

application well. The "camera.start_recording()" is the fastest, but cannot be allowed to "stall".

3.6.2 Codec

The codec that is used also influences the speed at which frames can be captured. Raw formats like BGR uses less time to be encoded and decoded than compressed formats like JPG, but the resulting raw frame data occupies much more memory space which takes time to move around and manipulate.

Only image-type formats were compared since they are easier to work with on a frame-by-frame basis than the video-type formats. Since RGB and BGR are essentially the same, only BGR was tested since that is the colour order used by openCV. The averaged frame times of JPG and BGR formats calculated over 50 consecutive frames were:

$$JPEG = 0.150\ seconds/frame \tag{3.3}$$

$$Raw(BGR) = 0.302\ seconds/frame \tag{3.4}$$

The JPG format turned out to be twice as fast as the raw format!

From this it looks like the slow frame rates are due to slow data transfers between the camera and the Raspberry Pi, and not due to the camera itself. If this is the case, then using the "camera.start_recording()" method with a high compression ratio format such as H.264 (MPEG-4) should give the best results. To confirm this the "start_recording" method was set up with the 1296 x 730 resolution and camera frame rate set to 49 (as per item 5 in Table 3.5) with the H.264 method. With a

custom streaming function that only prints the time taken for each frame the average frame interval was measured at 24.3 ms per frame over a five second period. This translates to just over 40 frames per second, which is more in line with the advertised rate for the camera.

This would have been the ideal solution because of the high frame rate. Unfortunately it is no trivial task to translate the H.264 encoded video data to an openCV image frame, so that the algorithm could use it to detect stop signs and traffic lights. Thus this avenue was marked as a last resort and not further developed for now.

3.6.3 Splitting into threads

The threading program used to demonstrate the “rapid capture and processin” example at [67] were mentioned above. Although it did not perform as claimed it did however spark an idea. Raspbian has a handy processor load indicator, which only ever went up to 25% on all the applications coded so far. This is clearly an indication that only one of the four cores of the Raspberry Pi’s CPU is used to execute the application’s Python script. Perhaps the code can be adapted so that the processing load will be distributed across more CPU cores, thus making optimal use of the available hardware on the Raspberry Pi platform for faster code execution.

There are different ways to apply multi-threading in Python. At first the ’thread’ module was used, but even though this breaks the program up into different threads, they are still executed in a sequential manner. This is due to Python making use of the Global Interpreter Lock (GIL) which restricts execution to a single core. Fortunately this limitation can be overcome to some degree by making use of the later ’multiprocessing’ module. This module also breaks the program up into threads, but they are now free to be executed in parallel and thus can be done on different cores simultaneously. Off course certain programming restrictions still apply, like for instance thread locking or using semaphores when accessing shared resources. The “multiprocessing” module also offers useful methods such as *queues*, *pipes* and *locks* to buffer and transfer data between threads (or processes) and to manage and synchronize threads.

The current implementation of the stop sign and traffic light state detector code has a few separable processes which can each be run on different cores. Instead of sequentially acquiring a frame, then

applying it to the traffic light state detection algorithm, then applying it to the stop sign detection algorithm and finally updating the outputs according to the results, some of these processes can be run in parallel.

The implementation of this process is illustrated in Fig. 3.29. The parent process kicks off a new process (thread) that captures frames from the camera as rapidly as possible, writing them into a queue until the queue has 2 frames awaiting processing. The parent process (thread) reads and removes an image from the queue and passes the image to processes 2 and 3. Process 2 applies the frame to the traffic light state detection algorithm, while process 3 applies it to the stop sign detection algorithm. The parent process waits for these to return their results and then analyze these and update the output port pins accordingly.

Surprisingly the 4-threaded program (as in Fig. 3.29) did not perform much better than the original simple single thread program. The output of the frame-timing pin is shown on an oscilloscope in Fig. 3.30. The poor performance increase is probably due to the large amounts of image data that now needs to be piped between the different processes canceling out the gained processing power from the extra threads/cores. To test this theory the 4-threaded system was reduced to 2 threads so that the image data is only passed between processes once for each frame. In this case process 1 is still used for the image capturing, but now the parent process is used to read the image from the queue and also to execute the stop sign and traffic light state detection functions. The process is illustrated in Fig. 3.31. This did indeed increase the performance a little, as can be seen from the frame-time output pin which is shown on an oscilloscope in Fig. 3.32.

The system still took 400 ms to capture and process a frame, which means only 2.5 frames per second, which is getting closer, but not yet good enough for a real-time solution. So some more alternatives were sought to increase the frame rate, it was decided to try the following:

1. Creating a custom video stream to grab video frames.
2. Using a different picamera driver.
3. Rewriting the code in a compiled language.

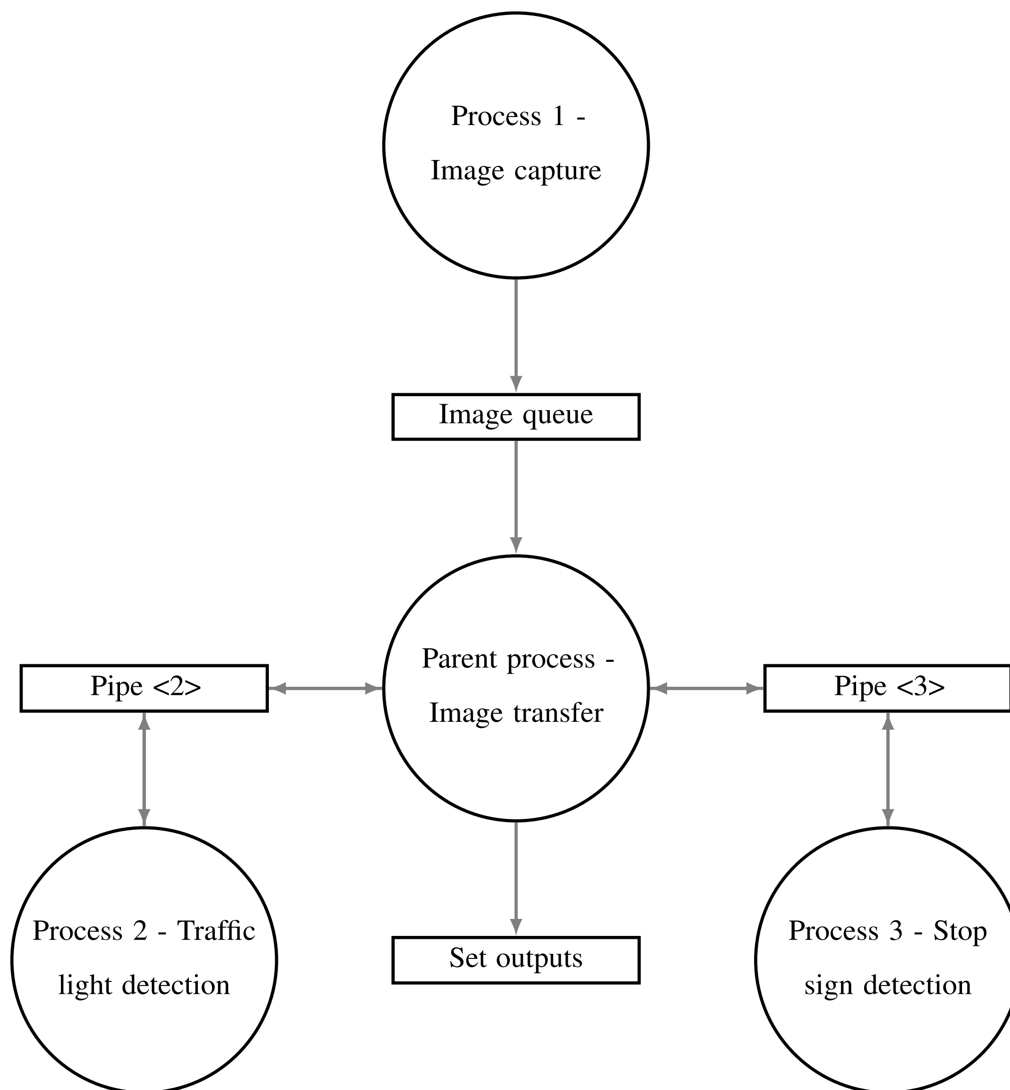

Figure 3.29. Multi-core (4) process division flow-diagram

3.6.4 Creating a custom video stream to grab video frames

Since the picamera can comfortably record video at 25 frames per second, an attempt was made to redirect the video recording stream that was writing to the file-system to write to a custom stream instead. The custom stream is simply a function that emulates a stream-write function, and as the frames arrive they can then be processed instead of writing them to file.

The custom stream was created and the video directed to the stream, but no way could be found to decode the data-stream into separate frames for processing, and so the data-stream could not be converted to usable information and this avenue was abandoned.

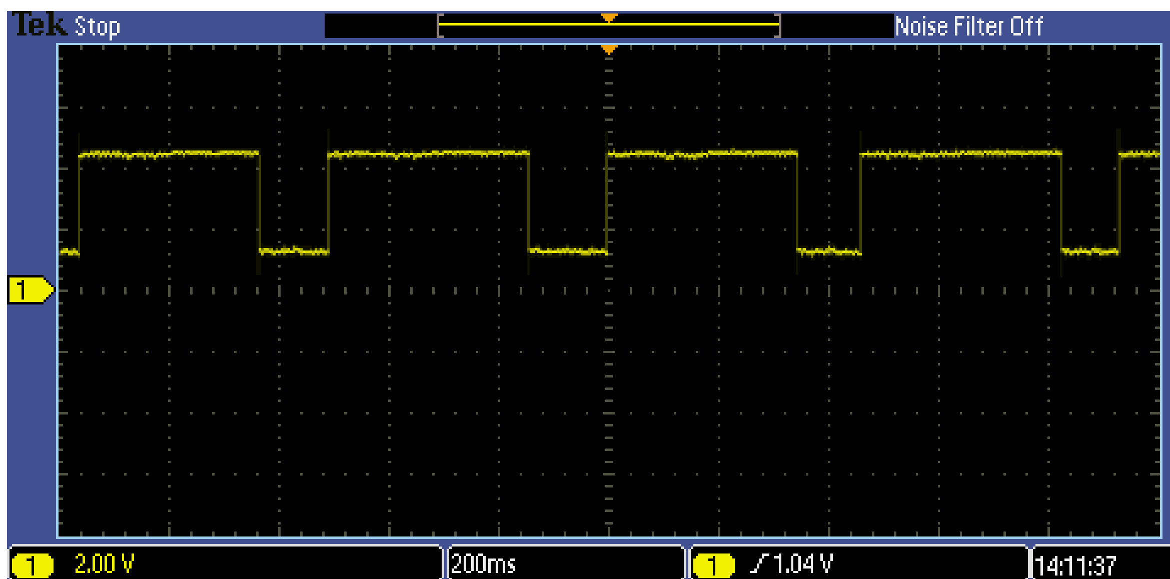

Figure 3.30. Frame timing on the 4-thread program

3.6.5 Using a different driver and a compiled language

Since moving to a different language will require the use of a different driver, it made sense to combine these two approaches.

Python is an interpreted language and thus it tends to execute somewhat slower than most compiled languages. The code already relies heavily on the OpenCV library, and since OpenCV is compatible with Python and C++ the whole program was simply ported to C++, which is a compiled language.

A C++ driver for the picamera was downloaded from ḧttp://robotblogging.blogspot.co.za/2013/10/an-efficient-and-simple-c-api-for.htmländ implemented. Timing tests showed a great improvement in the retrieved frame-rate, the results are discussed in Section 4.6.

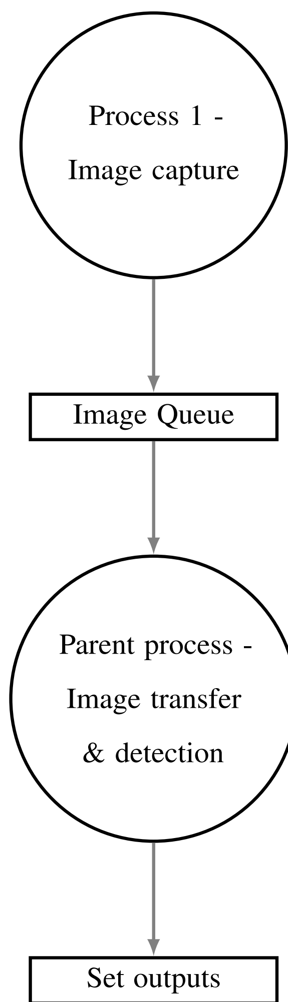

Figure 3.31. 2-core process division flow-diagram

3.7 COMPARING DETECTOR METHODS FOR STOP SIGNS

3.7.1 SURF vs Hough circle transform

The SURF-FLANN function in the openCV image processing library was used together with the image of a stop sign that is shown in Fig. 3.33.

The function was applied to the frames of a pre-recorded video and executed on a laptop computer (Intel I5 2.5 GHz). Results are discussed in Section 4.3.

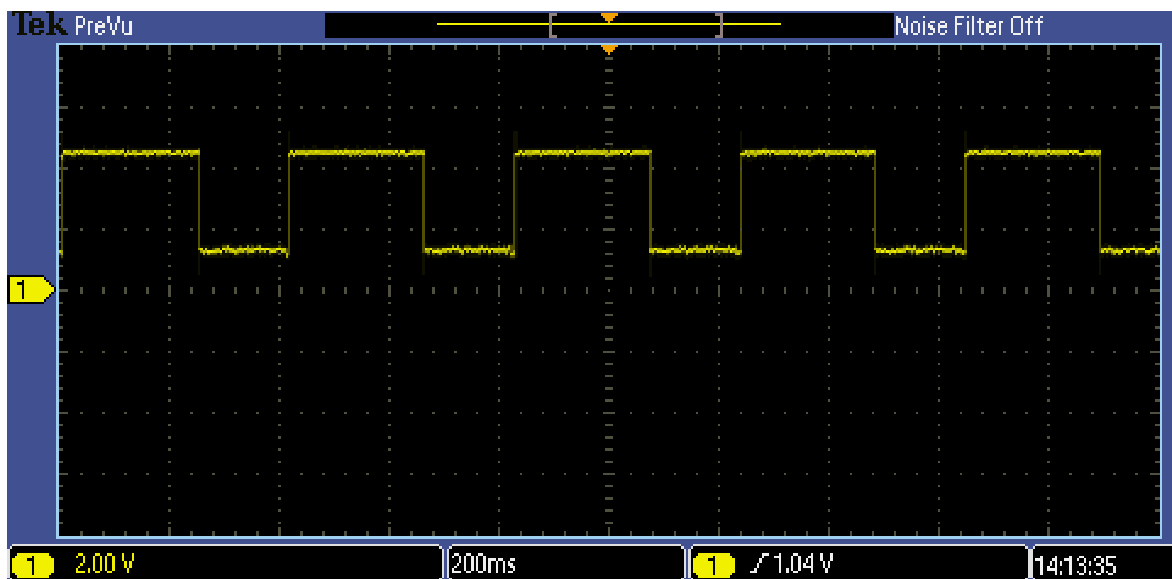

Figure 3.32. Frame timing on the 2-thread program

Figure 3.33. Reference image for a stop sign used with the SURF algorithm

3.7.2 Hough circle transform vs Blob detection

During the stop sign data collection process, it was noted that the Hough circle transform was not very consistent in detecting the stop signs as circles in the captured video frames. Furthermore, between successive frames of a video the circle center and radius were not very consistent, with the center position often jumping around near the center of the stop sign (or other circle in the image) and the circle radius also varying widely. Sometimes on a single frame the transform will detect several circles of different radius around the same stop sign. To counter this effect an image slightly larger than the detector radius was captured from the video frame while building a data set. Examples of the images captured from video frames by the Hough circle transform are shown in Fig. 3.34.

Figure 3.34. An indication of Hough circle transform output variation in successive frames

This great variation led to poor quality images in the dataset, and it was later identified as the cause for poor stop sign recognition (the system generated many false positives). Another indicator was that despite a fairly large dataset (1100 training set images and 432 test set images), and even letting the training run for up to 15000 epochs, the test set accuracy only reached 86.5%. It is possible that the Hough circle transform was hampered by the lowered image resolution that was necessary to make the octagonal stop signs appear circular, but since this step is necessary when detecting stop signs it had to be applied when evaluating the detector's performance on stop signs.

To verify that it was the Hough transform's dataset image quality that caused the poor performance, another detection algorithm had to be implemented for a comparative test. So blob detection was implemented, to see how the resulting images would compare with the ones detected by the Hough circle transform and to see if the neural network could be trained to better accuracy then. Unfortunately both the Hough circle transform and the blob detector algorithm rely heavily upon the parameters that it is used with, which means that the outcome of a comparative test will be strongly biased towards the one with the best parameter set for the task it is given. Still, it should give some indication of whether or not the system's high number of false positives is due to the noted problems with the Hough circle transform.

So a blob detector was implemented to detect stop signs in video frames. The blob detector results were a lot more consistent, with the detected area's center point and radius remaining much more consistent across successive frames of the video. As a result the images always contained a whole stop sign and little else, unlike the Hough circle transform images that often cut off parts of the sign or had background included in significantly large sections of the captured images. Examples of the images captured by blob detector are shown in Fig. 3.35 and can be compared with the Hough circle transform images of Fig. 3.34.

The results of the stop sign detection performance comparison are discussed in Section 4.3.

Figure 3.35. An indication of blob detector output variance in successive frames

In a project such as this one where the program will be executed by specific hardware and where it is critical for the system to keep up with the real-time frame feed from the camera, the accuracy of the algorithms can not be compared without also considering the processing time implications.

Three test videos were recorded, each one containing a stop sign and all being 148 frames long. Two programs were written that prints the time duration necessary to work through (process) all the frames of a video. These two programs were identical in everything except for the detection algorithm used, with one applying the Hough circle transform on each frame and the other a blob detector.

The results of the processing time comparison are in Section 4.3.

3.8 (RE-)COMPARING DETECTOR METHODS ON SPEED LIMIT SIGNS

As an initial detector method, the Hough circle transform performed surprisingly poor on stop signs compared to blob detection (see Section 4.3.1). This is probably due to the fact that the shape of a stop signs is actually an octagon and not a true circle, although an octagon can be approximated by a circle especially at very low image resolution, or high levels of blurring. It appears that this method of approximation hampered the performance of the Hough circle transform. To test this hypothesis, the two detector methods were compared in the same manner, but this time using a target object that is a true circle.

Fortunately there are many other traffic signs that have a circular shape, and so speed-limit signs were chosen, as they can be used in a manner very similar to the rest of this project to determine when a driver is not abiding by the rules of the road and then adjust his/her driver rating accordingly. Speed signs are also similar to stop signs in size and typical location (distance from the observer/camera, height above the ground, typical backgrounds, etc.).

Since both algorithms are applied to exactly the same image frames, the results will give an indication of which algorithm works best by checking:

1. The statistics of true speed signs detected versus other objects detected.
2. Using the resulting images as a dataset to train a neural network to detect speed signs, and to then compare the performance (training outcome, speed and accuracy) of the two 'systems' on pre-recorded videos containing speed signs.

If the Hough circle transform now performs much better against the blob detector as compared to the results from the stop signs run, then the hypothesis is true that it was the octagonal shape of the stop signs that caused the poor performance of the Hough circle transform as an initial detector on stop signs. However, if the blob detector still performs much better than the Hough circle transform, then the hypothesis is not true, and it may prove that the Hough circle transform is just not as suitable as the blob detector as an initial detector for circular (or approximately circular) road signs when used together with a neural network.

3.8.1 Data collection method

To begin the comparison, the parameters of both algorithms were first tuned for the best balance between capturing and rejection of circles within the image so as to capture as many speed signs as possible while minimizing the capturing of any other (non speed sign) objects. This was implemented by starting with a set of best-guess parameters and then adjusting it by manual trail-and-error. This involved applying each algorithm to a couple of videos that contains typical dashboard camera footage that includes speed limit road signs. The frames are played back as a video while each positive output from the detector is marked with a circle drawn onto the video frame.

3.8.2 Neural network training

Now that a collection of images captured by each detector method has been obtained, the neural network that forms the image classification stage can be trained using this dataset.

In order to compare the quality of the data collected from each detector method by the training outcome, the exact same training algorithm had to be applied in both cases. This not only meant that the same learning algorithm should be used, but also the same learning rate factor and the same number of iterations (epochs).

The neural network training method that was used is similar to the one described in Section 3.5.5.

Each data set was divided so that 75% of the images make up the training set and the remaining 25% is kept as a test set.

The results of these tests are discussed in Section 4.4.

3.9 COMPARATIVE TRAFFIC LIGHT STATE DETECTION

In the "Methods" chapter it was shown how a traffic light cluster can be detected in real-time from a video frame.

Only detecting the presence of a traffic light is not much use in this application, so here the traffic lights are further analyzed to find the current state of the traffic light. In order to rate a driver for not stopping at a stop sign or a red traffic light, only the red (stop) state of the traffic light needs to be detected. However, since traffic light detection algorithms in literature most aimed at detecting at least red/green or all red/yellow/green states, this algorithm will aim to detect both red and green states.

Two methods were identified that can be applied harmoniously to the cluster detection method already used:

- **Colour based state recognition**: The state of the traffic light is determined by identifying the colour of the circle with the highest red or green colour component.
- **Neural network based state recognition**: The neural network that classified the box image as either a standard or turn-signal traffic light cluster is expanded to differentiate between a turn-signal cluster, a standard cluster with top light (red) illuminated or a standard cluster with the bottom (green) light illuminated.

3.9.1 Traffic light state by colour information

The state of the traffic light is determined by finding circles within the box image (as used when detecting traffic light clusters) and identifying the colour of the circle with the clearest red or green dominance (and a minimum illumination value). A step-by-step explanation of the implementation of this method will follow.

Recall the method used to detect traffic light clusters:

1. Read the video frame and convert it to gray-scale.
2. Resize it to the standard resolution of 1280 x 720 if the camera provided a different resolution.
3. Apply Canny line detection (this step is included in openCV's Hough Circle function).
4. Apply the Hough circle transform.
5. Take a sample image block around each detected circle, extend the block vertically to allow for any of the 2 or 3 lights in a traffic light cluster detected.
6. Apply the Hough circle transform again, this time to the sampled box to determine where in the box the lights are located.
7. Based on the location of the centres detected by the circle transform, crop the sampled box around the traffic light cluster.
8. Resize the sampled block to 16 x 32 resolution.
9. Feed the sample block to the neural network to classify it as a traffic light turn-signal cluster, traffic light standard cluster or not a traffic light.
10. Display detected traffic light clusters by drawing a coloured rectangle onto the original gray-scale image, using turquoise for a turn-signal cluster or red for a standard cluster.

The colour based traffic light state recognition ties in at two points in these steps. First, where the Hough circle transform is applied a second time to find the circles within the elongated box in order to crop the box around the detected lights, the list of circles (which should consist mostly of the lights of the traffic light cluster) is saved for later use. A few examples of these circles are shown in Fig. 3.36 (images were scaled by 300% here for better visibility).

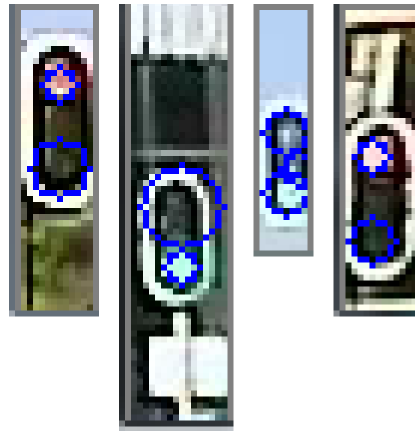

Figure 3.36. Circles (lights) detected in a few cluster image boxes

Second, once the neural network has classified the image in the box as a standard traffic light cluster, the circles are recalled in order to identify the current state. Iterating over each circle found in the box, these steps are applied:

1. Using the circle's centre point within the box, and the box's position within the original frame, the circle's (light's) position in the original frame is calculated.
2. From the original colour video frame, sample the colour values of the pixel in the centre of the circle as well as the 24 pixels surrounding it in a square area covering 2 pixels to the left, right, above and below the centre pixel. The process is illustrated as follows: Fig. 3.37 shows the circles (lights) that were detected in a cluster image box, while Fig. 3.38 indicates which pixels from the detected circle were sampled. The sampled pixels were changed to black and the image scaled by 300% for better visibility.
3. Calculate the mean value of each colour component (blue, green and red) of the 9 pixels that were sampled. The list for the traffic light sampled in Fig. 3.38 is shown in Fig. 3.39, the element order is blue, green, red. From the sample it can be seen that the top row is for the red light that was on, since the values are relatively high it means the light was on (high illumination), while the bottom row was for a light that was not on and thus it has low illumination values. Since the top row's red component is much higher than the green component it can be derived that the traffic light in the example's state was 'red'.
4. Check the values of the red and green components of the calculated mean colour values in the 25 sampled pixels. If the red component is more than 30% greater than the green component and the red component is more than a certain illumination threshold (100 worked well), then

the traffic light state is output as ’red’, while if the green component is 30% more than the red component and above the threshold the state is output as ’green’. The threshold ensures that that the pixels are of a minimum brightness to avoid setting the traffic light state based on the colour from a circle on the background of the image.

Figure 3.37. Circles (lights) detected in a cluster image box where colours will be sampled

Figure 3.38. Pixels sampled for their colour components are shown in black

```
[201  81 105]
[39 51  46]
```

Figure 3.39. Mean value of the colour components for two sampled circles

The blue component was intentionally ignored, since the green light of some traffic lights has a high percentage of blue in it, while others do not. This may be due to new traffic lights using LED lights while the older types use incandescent bulbs with a coloured filter.

All intersections have multiple traffic light clusters, usually 1 or 2 clusters on the approach side of the intersection, 2 on the opposite side, and one overhead on the opposite side. This means that this detection method will generate many light state outputs for each frame when a traffic light group is within detection range. These are combined into a single output by a voting system. Each detected traffic light’s state is appended to a fixed-length list and at the end the state with the most instances (votes) is output as the detected traffic light state for that frame.

3.9.2 Traffic light state by neural network

Since the traffic light clusters are recognised by using a neural network to classify standard and turn-signal clusters, it should be possible to train the neural network to also distinguish between the states of the traffic lights. During the initial detection implementation this was attempted without success, but the failure was probably caused by the following factors:

1. An attempt was made to recognise too many different states. These included the standard red, yellow/amber and green as well as turn signal green, yellow/amber and off (6 states). While this may work with a very large training set that has many different instances and a lot of variety for each, the training set that it was trained on was not big enough to get satisfactory outputs.
2. The training of the neural network was limited due to a very small training set. The initial training consisted of only 214 images. The set was limited in both size and variety due to the time it takes to collect, download, sort, segment, and manually classify video footage of traffic lights.

Another attempt was made to use the neural network as a traffic light state classifier, while addressing the above mentioned problems. Additional data was collected so that a slightly larger training set could be used (705 images) and the outputs were limited to recognizing red and green in a standard traffic light cluster (2 states). Although this will not give an output for all the possible traffic light states it should be sufficient to illustrate the concept given the limited training set size.

Using a 16 x 32 pixel image box as the input to the neural network as before, there needs to be 512 input nodes to the network. As mentioned, the outputs will be limited to indicate only the red and green states of the standard cluster while the turn-signal will be kept to only recognize the cluster (and not the turn-signal state). This leaves 3 states for the neural network's output layer, so the architecture will be 512-512-3. A truncated version of the neural network architecture is illustrated in Fig. 3.40, showing the (I) input-, (H) hidden- and (O) output-layer. As before the commonly used sigmoid function

$$\Phi = \frac{1}{1+e^{-z}}, \tag{3.5}$$

was chosen as the neuron activation function.

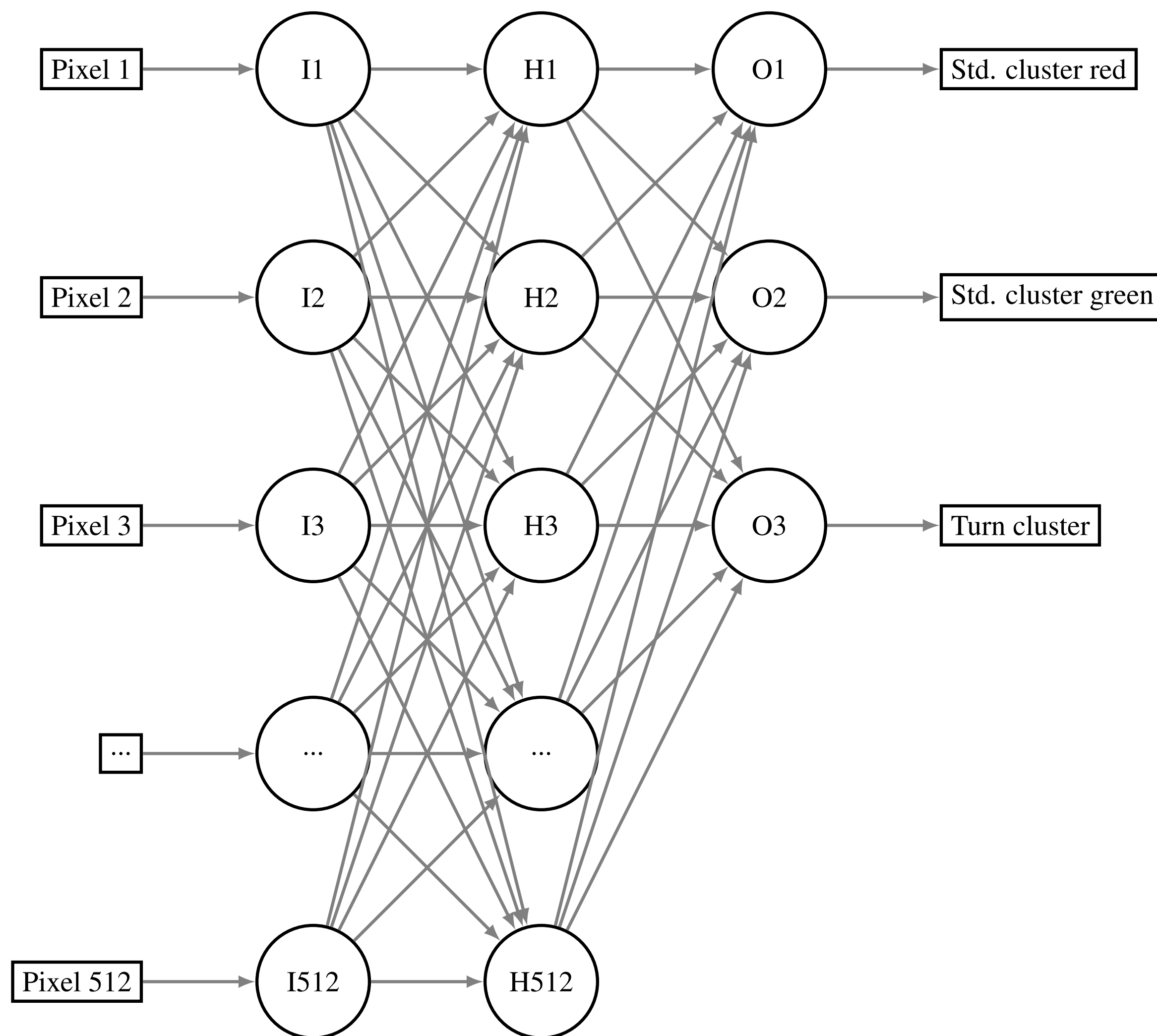

Figure 3.40. Traffic light state classification feed-forward neural network

The neural network was trained using the back propagation method. The training requires a training set, verification set and test set of images in the same format that will be given to the neural network during the classification process (forward pass), but on the training and verification sets the images must be accompanied by a correct class indication set for each image.

In order to construct such a set of good training images and their corresponding classification information, the programs that were used for the simple traffic light detection were adapted to accept the altered output classes that are required to also distinguish the state of the standard light cluster. The altered valid classes input for the training set construction program are shown in Table 3.6.

In order to simplify referencing this information during the training process, the classified images are

Table 3.6. Traffic light manual classification inputs including light state

Valid input	Class	Network output translation
’r’	Standard traffic light cluster in the red state	[1 0 0]
’g’	Standard traffic light cluster in the green state	[0 1 0]
’t’	Turn-signal traffic light cluster (any state)	[0 0 1]
’n’	Default (not one of the above options)	[0 0 0]

all stacked together horizontally (like a long array of images in a single matrix) and written to a file. The class input from the user is written to a text file where each character represents the class for the corresponding stacked image in sequence. An example of a stacked images set for one of these videos is shown in Fig. 3.41 and the corresponding class text is shown in Fig. 3.42.

Figure 3.41. A horizontally stacked set of traffic light state training images

nnnnnnnnnnngngngnnnngngnnnnnnnnn

Figure 3.42. Traffic light state training classes in a text file

As mentioned with the colour based algorithm, all intersections have multiple traffic light clusters, which means that this detection method will also generate many light state outputs for each frame when a traffic light group is within detection range. These detected states are combined into a single output by a voting system. Each detected traffic light’s state is appended to a list. Once all the detected circles in the frame have been classified the state with the most instances (votes) in the list is output as the detected traffic light state for that frame.

3.10 CONCLUDING REMARKS

In this chapter it was shown how a road sign detection system was designed and implemented by cascading existing image processing algorithms and methods. Alternatives were also investigated and

compared to determine which is most suitable to the task of detecting stop signs, speed limit signs and determining the state of detected traffic lights. In the next chapter the results of tests and measurements will be presented and discussed.

CHAPTER 4 RESULTS AND DISCUSSION

In this chapter the results of the tests and measurements from the previous chapter are presented and discussed.

4.1 INITIAL CANNY-HOUGH-NEURAL NETWORK STOP SIGNS DETECTION

These are the results of the initial method in Section 3.5.1.

The algorithm was run on all 32 videos that had been recorded at the time which contained stop signs. On these videos the processing time was measured for each frame. The times indicate the duration spent on processing the frame on a laptop. It does not include the time required to retrieve the frame nor the time to display the frame on the screen, since this will not be done when the system is deployed on an embedded in-vehicle platform. A few statistics of the processing times are shown in Table 4.1. A

Table 4.1. Stop sign processing time per frame

Total frames	5441
Mean time per frame	8.262 ms
Minimum time per frame	5.00ms
Maximum time per frame	54.00ms

mean time of 8.262 ms equates to a speed of 121 frames/second, which is much more than the standard video rate of 25 frames/second. The longest time spent on a frame was 54 ms, which equates to 18 frames/second.

In a couple of videos the stop sign was not recognised at all. There appears to be 3 main reasons for this.

1. **Bad lighting:** The dashboard camera automatically adjusts its saturation levels based on the average lighting in the entire frame. This may lead to overexposure or even underexposure of the stop sign if it is in much brighter light or in much darker shadow than the rest of the frame, as can be seen in Fig. 4.1, or even when the sign is in partial shadow such as in Fig. 4.2. These problems occur mostly early morning and late afternoon when long shadows fall across the road. The image's brightness can be normalised, but this takes more processing time and it will not help much if the affected area of the image has already been driven into saturation.

Figure 4.1. Stop sign with overexposure

Figure 4.2. Stop sign in partial shadow

2. **Extreme angle:** If the stop sign is planted in such a way that the vehicle approaches the sign at a bad angle, the stop sign's circular outline is distorted to appear more elliptical, such as in Fig. 4.3. In this case the shape is not recognised as a circle by the Hough circular transform and thus the stop sign is never detected. Perspective transformation can be applied to compensate for this, but again it will take more processing time.
3. **Blurring:** If the stop sign is approached at very high speed, or even when the vehicle's windscreen is very dirty, such as in Fig. 4.4, the video is sometimes blurred to the point where the stop sign is not recognised. This cause is less severe than the others though, and only affects the detection in extreme cases.

Figure 4.3. Stop sign viewed from a bad angle

Figure 4.4. Stop sign viewed through a very dirty wind-screen

Some false classifications were made by the neural network, which indicates that the neural network was not trained well enough. Extending the training set and training time in successive tests did lead to better results.

4.2 INITIAL CANNY-HOUGH-NEURAL NETWORK TRAFFIC LIGHTS DETECTION

These are the results of the initial traffic-light method discussed in Section 3.5.2.

The algorithm was run on all 24 videos that contained traffic lights. On these videos the processing time was measured for each frame, as processed on a laptop. The measured time did not include the time to display the frame on the screen, since this will not be done when the system is deployed on an embedded in-vehicle platform. Statistics about the processing times are shown in Table 4.2. A

Table 4.2. Traffic light processing time per frame

Total frames	6581
Mean time per frame	29.642 ms
Minimum time per frame	24 ms
Maximum time per frame	48 ms

mean time of 29.6 ms equates to a speed of 33 frames/second, which is still more than the standard

video rate of 25 frames/second. The longest time spent on a frame was 48 ms, which equates to 20 frames/second.

A first attempt at detecting traffic lights was made with the neural network not only classifying a standard or turn-signal cluster, but also the current state of the standard cluster (ie. red light, yellow light or green light illuminated). This did not work well at all, the problem probably lay in the number of training samples per class being too small with that many classes, and with the limited number of training images available. Reducing the classification task to only classify standard clusters and turn-signal clusters drastically improved the recognition performance of the neural network.

The Hough circle detection method worked very well on traffic lights. It mostly missed traffic lights that were too far away from the camera, which may sometimes be across a very large intersection. This is caused by a combination of the lights being much smaller than a stop sign, and the effect of the dashboard camera's wide angle lens.

Another challenge with traffic lights is that due to its brightness and the height above the road, the light's reflection from the vehicle's bonnet can sometimes be detected as a traffic light. In this case the light order is reversed (mirror effect) and this may later cause problems when the traffic light state will also be detected. See Fig. 4.5 for an example.

Figure 4.5. Traffic light reflection on the vehicle's bonnet

4.3 RESULTS OF COMPARING DETECTOR METHODS FOR STOP SIGNS

These are the results of the method in Section 3.7.

Initial tests (performed on a laptop before the embedded hardware platform was chosen) on the SURF algorithm indicated a mean execution time of 0.368 seconds per frame, which translates to 2.7 frames per second. This is much slower than the Hough circle transform and neural network combination when tested on the same machine, so it is unlikely this method will meet the speed requirements of the project when it is executed on the intended embedded hardware. The algorithm also missed the target object in many frames from the video, and often registered false detections on other objects in the background of the image. Fig. 4.6 shows how the SURF algorithm matches key feature points between the reference image and the target in a video frame.

Figure 4.6. Key feature points used by the SURF algorithm

SIFT was also briefly tested on one of the videos, but although it proved a lot more accurate than SURF, it was even slower to calculate.

4.3.1 Algorithm accuracy

The results showed great variance between the different videos with the settings as tested, and it was clear that the background objects had significant influence on the number of other (or “false”) circles detected. In general the blob detector yielded a better ratio of real stop signs to other objects (non stop signs), the comparative results on video number 1 only, are shown in Table 4.3.

Table 4.3. Comparative detection accuracy results on two detection algorithms for video number 1

	Stop signs detected	Total circles detected	Accuracy ratio
Blob detector	38	53	71.1%
Hough circle transform	29	38	76.3%

4.3.2 Algorithm speed

The results of the comparative algorithm execution speed are shown in Table 4.4.

Table 4.4. Comparative execution speed results on two detection algorithms

	Video 1, time (s)	Video 2, time (s)	Video 3, time (s)
Blob detector	22	19	19
Hough circle transform	12	10	11

Clearly the Blob detector took significantly longer to execute than the Hough circle transform. Once implemented on the embedded hardware it would have been a disqualifying factor, but fortunately the better speed performance gained from code optimization and the switch to C++ which was later made, allowed for either algorithm to be implemented while staying within the required execution time per video frame.

4.4 RESULTS OF (RE-)COMPARING DETECTOR METHODS FOR SPEED SIGNS

These are the results of the method in Section 3.8.

4.4.1 Data collection results

Running both the Hough circle transform and the blob detection algorithm on exactly the same frames from the camera, a review of the resulting (captured and saved) images will indicate how the algorithms compared at capturing the circular speed limit signs. The total captured images for each category were tallied by manually classifying all the images from both algorithms, the results are given in Table 4.5.

Table 4.5. Numbers of speed limit signs captured by concurrently running algorithms

Speed limit category	Hough circle transform	Blob detection
40	26	4
60	78	123
80	110	75
100	33	49
120	2	2
Non speed-sign	46	159
Total images	295	412

The Hough circle transform seems to have performed better on truly circular target objects such as speed signs compared to approximated circles such as stop signs (octagons). Despite the apparent improvement, the algorithms performed nearly equally well, although the blob detector did detect significantly more non-speed sign objects. One has to keep in mind that the performance of both algorithms depends on how well their setup parameters are optimised for the target object to be detected. Still, in this case both were optimised by the same process and more or less to the same degree.

From the collected images it would appear that variation in the target's surroundings and environmental conditions such as lighting, contrast and background affected the different algorithms in different ways. Such that one algorithm did well in one set of image circumstances while the other did better in another set. The actual target image also affects how well an algorithm performs, clearly the algorithm should be matched to the application...there is no one-algorithm-fits-all solution.

This leads to the conclusion that for best system wide detection performance it will be best to run both of these algorithms in parallel. This will only be possible if he processor can keep up with the increased number of images that will be forwarded to the neural network for classification.

4.4.2 Neural network training

The exact same neural network training algorithm was applied to the data sets collected by the two initial detection methods. Once again back propagation learning was used. 5000 iterations were run on

each set and the learning rate factor was kept at -0.0001.

The training outcomes are shown in Table 4.6, and the training graphs of the Hough circle transform compared to the blob detector are shown in Fig. 4.7 and Fig. 4.8 respectively. Note that the graphs show test set accuracy (vertical axis) plotted against training epoch (horizontal axis).

Table 4.6. Training outcome of the two speed sign datasets

Data set	Hough circle transform	Blob detection
Detection accuracy on test set	61%	96%

Although the Hough circle transform had a slightly lower sum of squared error (SSE) after 5000 iterations, the test set accuracy on the blob detector dataset was much higher. This shows that the neural network was much better at detecting the speed sign types in the (previously unseen) images collected by the blob detector than the ones collected by the Hough circle transform.

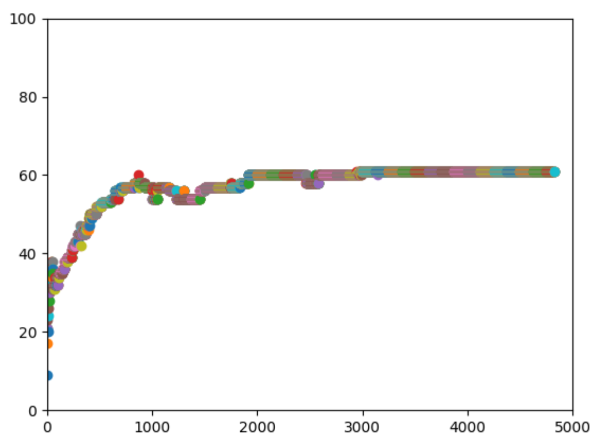

Figure 4.7. Hough transform training

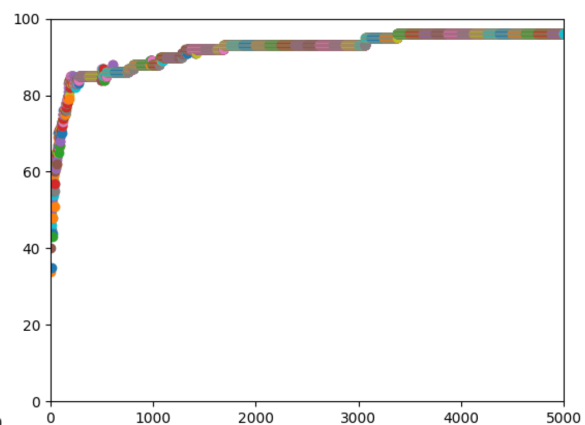

Figure 4.8. Blob detection training

4.4.3 Performance test

To test the system performance on recognizing and classifying speed signs, a number of videos containing speed signs were recorded using the actual hardware while driving on South-African roads. A set of speed sign video samples were taken for each of the 40 km/h, 60 km/h, 80 km/h and 100 km/h speed zones. Signs for 120 km/h were deliberately left out since not enough 120 km/h speed sign samples were collected for training the neural network. Table 4.7 shows the detection outcome after

running both the blob detector based, and Hough circle transform based systems on these pre-recorded videos. In the table the speed sign categories were assigned the following letters:

A = 40 km/h

B = 60 km/h

C = 80 km/h

D = 100 km/h

N = No sign detected or unknown category

The accompanying digit indicates the number of times that category was detected in the video. The correct detections made by each algorithm is shown in bold. The sum of the number of correctly and incorrectly identified cases over all the videos indicates that the blob detection based system got 38 correct and 20 incorrect while the Hough circle based system got 32 correct and 62 incorrect.

Thus the blob detection based system performed better. The precision can be calculated with Equation 4.1 and the results are shown in Table 4.8.

Clearly the Hough circle transform based system detected more objects but most of these were wrong. The blob detector based system had identified more signs correctly and it got significantly fewer incorrect detections. Thus it is clear that the blob detector based system performed much better than the Hough circle based one, especially in this application where it is critical to minimize the number of false positives.

This proves the hypothesis of Section 3.8 false. It was not the octagonal shape of the stop signs that resulted in the poor performance of the Hough circle transform compared to the blob detector, but rather the algorithm itself being inferior in accuracy (although computationally much faster) to the blob detector when combined with a neural network as used in this specific application and dataset.

Table 4.7. Speed sign video detection results (A = 40 Km/h, B = 60 Km/h, C = 80 Km/h, D = 100 Km/h, N = Not detected or unknown)

Actual speed zone	**video number**	Blob system output	Hough system output
40 (A)	1	4B	**5A** 12B 2D
40 (A)	2	3B	5B 2D
60 (B)	1	**3B**	**4B**
60 (B)	2	N	1A **1B**
60 (B)	3	**3B**	**4B** 1D
60 (B)	4	N	**1B**
60 (B)	5	**4B** 1C	**5B** 3D
60 (B)	6	**2B**	1A 1C
60 (B)	7	**2B**	N
60 (B)	8	N	N
60 (B)	9	1A **1B**	1A
80 (C)	1	N	N
80 (C)	2	1B **1C**	N
80 (C)	3	1B **2C**	1A 8B
80 (C)	4	2B	N
80 (C)	5	2B **1C**	1B 1D
80 (C)	6	4B	4B **6C**
80 (C)	7	**3C**	N
80 (C)	8	**1C**	2B
100 (D)	1	**4D**	N
100 (D)	2	N	2A
100 (D)	3	N	N
100 (D)	4	**1D**	2B **1D**
100 (D)	5	**6D**	1A 12B **5D**
100 (D)	6	**4D**	N
100 (D)	7	1B	3A

Table 4.8. Precision values comparison between algorithms on speed limit signs

	Blob detection	**Hough circle transform**
Precision	$\frac{38}{38+20} = 65.5\%$	$\frac{32}{32+62} = 34.0\%$

4.5 COMPARATIVE TRAFFIC LIGHT STATE DETECTION RESULTS

4.5.1 Processing time

The processing times required for the *combined* stop sign detection and neural network based traffic light state detection are shown by the statistics in Table 4.9.

Table 4.9. Colour base traffic light state detection processing time per frame

Video number	Mean processing time	Maximum processing time
1	32.123 ms	71 ms
2	34.724 ms	52 ms
3	33.398 ms	54 ms
4	30.892 ms	46 ms
5	37.682 ms	45 ms
6	39.934 ms	52 ms
7	38.535 ms	62 ms
8	34.245 ms	46 ms
9	29.824 ms	42 ms
10	32.775 ms	45 ms

The processing times required for the *combined* stop sign detection and neural network based traffic light state detection are shown by the statistics in Table 4.10.

The processing was done on a desktop computer with a 2.5 GHz Intel I5 CPU and 4 GB of RAM.

Table 4.10. Neural network based traffic light state detection processing time per frame

Video number	Mean processing time	Maximum processing time
1	34.273 ms	89 ms
2	37.264 ms	48 ms
3	34.694 ms	49 ms
4	32.083 ms	46 ms
5	41.493 ms	54 ms
6	44.331 ms	59 ms
7	42.440 ms	65 ms
8	36.611 ms	48 ms
9	31.296 ms	43 ms
10	35.025 ms	46 ms

4.5.2 Detection and classification accuracy

The algorithms were run with the test-set videos and the detection results are shown in Table 4.11. In the Table the detections are categorized as follows:

- When the target object appears in the video and it is correctly shown on the system outputs it is marked as ’C’ (correct).
- When the system output indicates a stop sign or traffic light when there was none in the video it is marked as ’I’ (incorrect). This is also used when an arbitrary object is identified as the target (even if the target object appears in the same frame). For traffic lights, the ’I’ is also used to indicate that the state of the traffic light was incorrectly detected (ie. Output indicated green when the light was red).
- When the target object appeared in the video but it was not shown on the system output once over the entire video, it is marked as ’M’ (Missed or not identified). The same applies to a traffic light that was detected but the state could not be determined (at all).

Table 4.11. Detection results on the test-set (TLSD = Traffic Light State Detection, C = Correct, I = Incorrect, M = Missed)

Video number	Colour TLSD	Neural Network TLSD	stop sign detection
1			C
2			M
3	I	CI	I
4	C	CI	I
5	C	C	
6			C
7			C
8	I		C
9			CI
10			IM
11	C	C	
12			IM
13	M	M	I
14			C
15			IM
16			C
17	C	C	I
18	M	CI	
19	C	C	I
20	C	C	I
21	C	C	I

4.5.3 Results analysis

Although the mean processing time for the combined algorithms on many of the videos came in below 40 ms, the maximum time spent on a frame over the whole video proved to be too long to use on a 25 frames/second recorded video. Surprisingly the colour-based traffic light detection scheme was much faster than the neural network scheme.

As for the detection accuracy, the results can be further analysed by calculating the precision and recall as follows:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{4.1}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{4.2}$$

From Table 4.11 true positives are the correctly identified target objects represented by ’C’ and false positives by ’I’. False negatives are when the object was present but not identified (missed), which is represented by ’M’.

So counting from Table 4.11 the following confusion matrices can be drawn up: Colour traffic light state detection in Table 4.12, neural network traffic light state detection in Table 4.13 and stop sign detection in Table 4.14.

Table 4.12. Colour TLSD detection results confusion matrix

	Detected	**Not detected**
Present	7	2
Absent	2	10

Table 4.13. Neural network TLSD detection results confusion matrix

	Detected	**Not detected**
Present	8	1
Absent	3	11

Table 4.14. Stop sign detection results confusion matrix

	Detected	**Not detected**
Present	7	4
Absent	11	3

Applying the results to Equation 4.1 and Equation 4.2 gives the precision and recall shown in Table 4.15.

Table 4.15. Calculated precision and recall results

	Precision	**Recall**
Colour TLSD	$\frac{7}{7+2} = 77.8\%$	$\frac{7}{7+2} = 77.8\%$
Neural network TLSD	$\frac{8}{8+3} = 72.7\%$	$\frac{8}{8+1} = 88.9\%$
Stop sign detection	$\frac{7}{7+11} = 38.9\%$	$\frac{7}{7+4} = 63.6\%$

The traffic light state detection algorithms performed quite well, with only a very small difference in accuracy between the two different algorithms used. The colour-based method had slightly better precision while the neural network showed better recall. Since the colour-based algorithm took less time to process it would make sense to use it instead of the neural network based one. There is also the possibility of using a sensor-fusion method to combine the output of the two algorithms for a better end result, although this approach will probably lead to a great increase in processing time per frame. Some time spent on tweaking the thresholds and parameters involved may also lead to slightly better accuracy.

4.5.4 Performance results from other methods in literature

The method implemented by Siogkas *et al.* [1] used colour, symmetry and spatiotemporal information on a 640 x 480 pixel video to detect traffic lights and their states. Their method was verified on other video feeds as well. They achieved a precision of 61.2% and a recall of 93.7% . Their algorithms had a processing time of between 0.1 ms and 0.5 ms per frame on computer with a Core 2 Quad CPU at 2.83 GHz, and 4 GB of memory.

For general road-sign recognition Maldonado-Bascón [2] used a frame resolution of 720 × 576 pixels and they achieved a mean processing time of 1770 ms per frame on a 2.2 GHz Pentium 4-M.

Another general road-sign recognition system by Loy [3] used a 320 x 240 pixel image and their method's detection and classification was able to be run at 20 Hz (50 ms). They achieved correct detection at 96.6% but it was measured on still images.

Nashashibi and De Charette's method [69] for traffic light state recognition could be applied in real-time at 26 frames per second (38.5 ms) on a 2.9 GHz single core desktop computer and achieved a mean precision of 88.3% and mean recall of 48.7%.

4.6 FRAME-RATE IMPROVEMENT RESULTS

While using Python code and the picamera Python library, the frame-rate was around 2.5 frames per second given that the algorithm did not detect a large number of circles (which resulted in an even slower frame-rate).

With the code ported to C++ and using the new C++ picamera library, the frame-rate was initially tested without any frame processing to get an idea of how fast the driver can grab and convert frames from the camera. When only calling camera->ReadFrame and converting the frame format from RGBA (Red-Green-Blue-Alpha) to BGR (Blue-Green-Red) as this is the format used by the OpenCV functions, it took 34 seconds for 1000 frames at a resolution of 1280 x 960 pixels, which comes to a frame-rate of 29 frames per second. This is equivalent to the 30 frames per second specified for the camera, considering that is takes additional time to convert the colour format on each frame image.

After adding in the stop sign detection algorithm, the frame rate was still above 10 frames per second. This is well within the requirements for a real-time detection system. Thus the change to a new driver and porting the code to the C++ compiled language resulted in a speed gain large enough to meet the design goal.

4.7 PROJECT GOALS FULFILLMENT

Referring to the project goals in Section 1.4, the goal to develop a sensor system to detect stop signs, traffic lights (including current state) and speed signs in real-time on an off-the-shelf computer system of reasonable cost and size has been fulfilled.

Looking at the specific metrics of Section 3.2:

1. Processing speed: After applying the frame-rate improvement steps in section 4.6 the frame-rate was well above 2 frames per second.
2. Detection precision: At 77.8% and 72.7%, both traffic light state detection methods met the 70% accuracy goal. The blob detector based speed sign detection reached 58.3%, which is below the goal but still good for the number of classes detected with the size of the dataset that was used (the accuracy can be increased by collecting a larger dataset).
3. Hardware size: At 86 mm x 58 mm x 19 mm the Raspberry Pi 3 is much smaller than 1000 cm^2.
4. Hardware cost: The cost for a Raspberry Pi 3 model B, picamera, an enclosure, SD-Card and power supply came to R1942 which is just below the specified R2000 per complete unit.

4.8 CONCLUDING REMARKS

This chapter presented the results of tests and measurements that refers back to the previous chapter, and the findings are discussed to make the implications clear. The main findings are that blob detection works better than the Hough circle transform (35% better accuracy on speed signs test set) when used together with a neural network classifier to detect stop signs and speed limit signs, and that there is little performance difference between the colour based approach and the direct neural network approach to detection of traffic light states. The next chapter will give a detailed conclusion.

CHAPTER 5 CONCLUSION

This project proposed that road safety could be improved by encouraging better and safer driving habits if a driver can be assigned a rating according to his/her adherence to the rules of the road. To measure this a device is necessary to visually detect road traffic signs from within the vehicle in real-time, in order to determine when the driver did not stop at a stop sign or red traffic light. This involved developing computer vision detection software with sufficient accuracy to prevent unfairly penalizing the driver, and the software had to be optimised for speed in order to run on an embedded hardware platform that can be installed within the vehicle. It also involved finding suitable hardware and testing the software in real-world conditions.

On the hardware side several single-board computer platforms were compared and the Raspberry Pi 3 was chosen as the most suitable for the task. It is relatively small, has sufficient processing power, is compatible with many software platforms and it can be powered from the vehicle's 12 V system. The Raspberry Pi 3 has the option of a small, reasonable priced and high-performance add-on camera called the picamera which is easy to use with the Raspberry Pi and suits the project design goals well.

For the software there are many techniques and methods that can be used to detect, recognise and track road traffic signs such as stop signs, speed limit signs and red traffic lights. The chosen method had to be robust to work reliably in varying lighting-, weather- and road-conditions that will be encountered in a real-world application, it had to have very good false positive rejection, and it also had to be able to execute fast in order to run on the chosen hardware while still analysing enough frames to detect the target objects reliably in and in real-time. A cascaded approach was followed, starting with a detection algorithm applied to each camera frame, followed by a neural network classifier and finally an iterative filter before the output. For the detector stage the Hough circle transform was compared to both blob detection and the SURF algorithm. SURF was far too slow for real-time detection, and although the

Hough circle transform was the fastest to calculate it was found that the blob detector yielded the best combination of accuracy and speed.

An important aspect pointed out by the development of the detection software for this project is that the system should be designed towards satisfying the design goal of the project and the detection algorithms should be chosen and planned around the target objects. Simply picking an algorithm that worked well in another project does not mean that it will be optimum for this one, and the implementation thereof should be adapted for the task. For instance, detection data alone is sufficient for stop signs, but with traffic lights the current state of the traffic light must also be determined before a useful decision can be made from the data. In this project two different methods of detecting the traffic light's state were implemented and compared in terms of processing cost and detection accuracy. One method added the use of colour while the other extended the neural network used for classification to also discriminate between the green and red states on a standard traffic light cluster. Surprisingly the colour based algorithm computed faster than the neural network algorithm, but in the end there were very little difference between the detection accuracy and the speed of the two methods.

Real driving tests proved that the system performed well enough to satisfy the design goals. The frame processing rate was just over 10 frames/second which is well above the required 2 frames per second and the detection and recognition accuracy were higher than 70% for stop-signs (76.3%) and traffic lights (77.8%) which included the traffic light state. However, speed sign detection and recognition only reached 65.5% precision. There were a few instances where the system completely failed to detect a sign. This was caused by variables such as lighting (over- or under-exposure), extreme viewing angles, a dirty vehicle windscreen and other factors. The system did not work in heavy fog or during night driving, except in areas with very good lighting.

After completing some initial test on a laptop, the software was ported to the python scripting language to be executed on the embedded hardware platform. Execution speed with python on the Pi was insufficient for true real-time operation. Several steps were taken to speed up the system, including the use of different capturing methods on the camera driver and splitting the code into separate thread to be executed in parallel on the 4 cores of the processor. Although the speed improved it was still not quite adequate, so in the end the entire program had to be ported to a compiled language and changed to use a different camera driver before it achieved the necessary execution speed to truly work in real-time. This proved that there are many factors that can influence software execution speed and that optimising

code for speed can make a significant difference in the time it takes to run a certain algorithm on a certain piece of hardware.

Owing to the many factors that may change the visual data model that represents a real-world object, it was found that object detection algorithms will miss a target object in some of the frames of a video sequence. Object tracking can be used to track target objects as they move over a video frame, and so it can predict the location of the detected object in frames where the detection algorithm failed to detect the target object for whatever reason, leading to a more continuous output across frames (by 'detecting' the object at the position where the tracking algorithm predicts it to be in a missed frame). Object tracking can also smooth out the track of a detected object across successive video frames if the detection algorithm's output gives somewhat erratic or noisy output location information (detected position is jumping around on successive frames).

The results of the detection and recognition stages had to be filtered using historical data from a first-in-first-out (FIFO) buffer in order to reduce detection output jitter and to filter out the occasional incorrect detection, before writing it to an output. A majority vote combined with a minimum threshold was applied to the FIFO buffer to determine which states should be indicated on the system's outputs. Once filtered the detector output states were written to digital output port pins so that it can be used by other electronic devices such as a vehicle tracking system as a "sensor" input.

REFERENCES

[1] G. Siogkas, E. Skodras, and E. Dermatas, "Traffic lights detection in adverse conditions using color, symmetry and spatiotemporal information." in *Proc. VISAPP*, 2012, pp. 620–627.

[2] S. Maldonado-Bascon, S. Lafuente-Arroyo, P. Gil-Jimenez, H. Gomez-Moreno, and F. López-Ferreras, "Road-sign detection and recognition based on support vector machines," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 2, pp. 264–278, 2007.

[3] G. Loy and N. Barnes, "Fast shape-based road sign detection for a driver assistance system," in *Proc. IEEE RSJ International Conference on Intelligent Robots and Systems*, vol. 1. IEEE, 2004, pp. 70–75.

[4] R. Jain, R. Kasturi, and B. G. Schunck, *Machine Vision*. McGraw-Hill, 1995.

[5] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Prentice Hall, 2001.

[6] P. Viola and M. Jones, "Robust real-time object detection," *International Journal of Computer Vision*, vol. 4, 2001.

[7] Z. Wang, D. Ziou, C. Armenakis, D. Li, and Q. Li, "A comparative analysis of image fusion methods," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, no. 6, pp. 1391–1402, 2005.

[8] D. H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111–122, 1981.

[9] D. G. Lowe, “Object recognition from local scale-invariant features,” in *Proc. ICCV*, vol. 2. IEEE, 1999, pp. 1150–1157.

[10] H. Bay, T. Tuytelaars, and L. Van Gool, “SURF: Speeded up robust features,” in *Proc. European Conference on Computer Vision*. Springer, 2006, pp. 404–417.

[11] J. F. Canny, “Finding edges and lines in images,” DTIC Document, Tech. Rep., 1983.

[12] J. Canny, “A computational approach to edge detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 6, pp. 679–698, 1986.

[13] D. Marr and E. Hildreth, “Theory of edge detection,” *Proc. of the Royal Society of London B: Biological Sciences*, vol. 207, no. 1167, pp. 187–217, 1980.

[14] R. Deriche, “Using Canny’s criteria to derive a recursively implemented optimal edge detector,” *International Journal of Computer Vision*, vol. 1, no. 2, pp. 167–187, 1987.

[15] L. G. Roberts, “Machine perception of three-dimensional soups,” Ph.D. dissertation, Massachusetts Institute of Technology, 1963.

[16] O. Seger, “Generalized and separable Sobel operators,” *Machine Vision for Three-Dimensional Scenes*, p. 347, 2012.

[17] J. M. Prewitt, “Object enhancement and extraction,” *Picture Processing and Psychopictorics*, vol. 10, no. 1, pp. 15–19, 1970.

[18] D. Cohen-Or and A. Kaufman, “Fundamentals of surface voxelization,” *Graphical Models and Image Processing*, vol. 57, no. 6, pp. 453–461, 1995.

[19] R. Nevatia and T. O. Binford, “Description and recognition of curved objects,” *Artificial Intelligence*, vol. 8, no. 1, pp. 77–98, 1977.

[20] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 23–38, 1998.

[21] R. Hussin, M. R. Juhari, N. W. Kang, R. Ismail, and A. Kamarudin, "Digital image processing techniques for object detection from complex background image," *Procedia Engineering*, vol. 41, pp. 340–344, 2012.

[22] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. CVPR*, vol. 1. IEEE, 2005, pp. 886–893.

[23] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: A GPU-accelerated framework for image processing and computer vision," in *Proc International Symposium on Visual Computing*. Springer, 2008, pp. 430–439.

[24] A. Lindoso and L. Entrena, *Hardware Architectures for Image Processing Acceleration*. INTECH Open Access Publisher, 2009.

[25] A. P. Reeves, "Parallel computer architectures for image processing," *Computer Vision, Graphics, and Image Processing*, vol. 25, no. 1, pp. 68–88, 1984.

[26] A. Humayun, O. Mac Aodha, and G. J. Brostow, "Learning to find occlusion regions," in *Proc. CVPR*. IEEE, 2011, pp. 2161–2168.

[27] D. Simon, "Kalman filtering," *Embedded Systems Programming*, vol. 14, no. 6, pp. 72–79, 2001.

[28] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.

[29] C. Suliman, C. Cruceru, and F. Moldoveanu, "Kalman filter based tracking in an video surveillance system," *Advances in Electrical and Computer Engineering*, vol. 10, no. 2, pp. 30–34, 2010.

[30] C. Wojek, S. Roth, K. Schindler, and B. Schiele, "Monocular 3d scene modeling and inference: Understanding multi-object traffic scenes," in *Proc. European Conference on Computer Vision*. Springer, 2010, pp. 467–481.

[31] D. Comaniciu, V. Ramesh, and P. Meer, "Kernel-based object tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 5, pp. 564–577, 2003.

[32] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *ACM Computing Surveys (CSUR)*, vol. 38, no. 4, p. 13, 2006.

[33] G. Welch and G. Bishop, "An introduction to the Kalman filter," University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. Rep., 1995.

[34] S. Xu and A. Chang, "Robust object tracking using Kalman filters with dynamic covariance," *Cornell University*, 2014.

[35] N. J. Gordon, D. J. Salmond, and A. F. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," in *Proc. IEE F-Radar and Signal Processing*, vol. 140, no. 2. IET, 1993, pp. 107–113.

[36] F. Castanedo, "A review of data fusion techniques," *The Scientific World Journal*, 2013.

[37] A. R. Mirza, "An architectural selection framework for data fusion in sensor platforms," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

[38] P. Axelsson, "Evaluation of six different sensor fusion methods for an industrial robot using experimental data," *IFAC Proceedings Volumes*, vol. 45, no. 22, pp. 126–132, 2012.

[39] J. L. Crowley, "Principles and techniques for sensor data fusion," in *Multisensor Fusion for Computer Vision*. Springer, 1993, pp. 15–36.

[40] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer Science & Business Media, 2008.

[41] F. Cremer, K. Schutte, J. G. Schavemaker, and E. den Breejen, “A comparison of decision-level sensor-fusion methods for anti-personnel landmine detection,” *Information Fusion*, vol. 2, no. 3, pp. 187–208, 2001.

[42] G. Hager, “Interval-based techniques for sensor data fusion,” University of Pennsylvania (MS-CIS), Tech. Rep., 1990.

[43] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. the MIT Press, 2006.

[44] A. P. Dempster, “A generalization of Bayesian inference,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 205–247, 1968.

[45] G. Shafer *et al.*, *A mathematical theory of evidence*. Princeton university press Princeton, 1976, vol. 1.

[46] F. Dernoncourt. (2013) Introduction to fuzzy logic (online course). [Online]. Available: http://francky.me/doc/course/fuzzy_logic.pdf

[47] M. Baaz, P. Hájek, F. Montagna, and H. Veith, “Complexity of t-tautologies,” *Annals of Pure and Applied Logic*, vol. 113, no. 1, pp. 3–11, 2001.

[48] B. Parhami, *Dependable Computing Systems*. John Wiley & sons inc., 2005.

[49] B. G. Buchanan, E. H. Shortliffe *et al.*, *Rule-based expert systems*. Addison-Wesley Reading, MA, 1984, vol. 3.

[50] J. Esteban, A. Starr, R. Willetts, P. Hannah, and P. Bryanston-Cross, “A review of data fusion models and architectures: towards engineering guidelines,” *Neural Computing & Applications*, vol. 14, no. 4, pp. 273–281, 2005.

[51] L. Chen, M. Cetin, and A. S. Willsky, “Distributed data association for multi-target tracking in sensor networks,” *International Conference on Information Fusion*, 2005.

[52] A. Zafeiropoulos, N. Konstantinou, S. Arkoulis, D.-E. Spanos, and N. Mitrou, "A semantic-based architecture for sensor data fusion," in *Proc. UBICOMM*. IEEE, 2008, pp. 116–121.

[53] E. Azimirad and J. Haddadnia, "The comprehensive review on JDL model in data fusion networks: Techniques and methods," *International Journal of Computer Science and Information Security*, vol. 13, no. 1, p. 53, 2015.

[54] D. L. Hall and J. Llinas, "An introduction to multisensor data fusion," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 6–23, 1997.

[55] A. N. Steinberg and C. L. Bowman, "Rethinking the JDL data fusion levels," *NSSDF JHAPL*, vol. 38, p. 39, 2004.

[56] B. V. Dasarathy, "Sensor fusion potential exploitation-innovative architectures and illustrative applications," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 24–38, 1997.

[57] M. Bedworth and J. O'Brien, "The omnibus model: a new model of data fusion?" *IEEE Aerospace and Electronic Systems Magazine*, vol. 15, no. 4, pp. 30–36, 2000.

[58] M. Á. García-Garrido, M. Á. Sotelo, and E. Martín-Gorostiza, "Fast road sign detection using Hough transform for assisted driving of road vehicles," in *Proc. International Conference on Computer Aided Systems Theory*. Springer, 2005, pp. 543–548.

[59] L. Kurnianggoro, J. Hariyono, K.-H. Jo *et al.*, "Traffic sign recognition system for autonomous vehicle using cascade SVM classifier," in *Proc. 40th Annual Conf. IECON*. IEEE, 2014, pp. 4081–4086.

[60] M. A. Garcia-Garrido, M. A. Sotelo, and E. Martin-Gorostiza, "Fast traffic sign detection and recognition under changing lighting conditions," in *Proc. IEEE Intelligent Transportation Systems Conference*. IEEE, 2006, pp. 811–816.

[61] Raspberry Pi 3 specs. [Online]. Available: https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/ (Last visited: 2016-04)

[62] UDOO DUAL and QUAD. [Online]. Available: http://www.udoo.org/features/ (Last visited: 2016-04)

[63] ODROID. [Online]. Available: http://www.hardkernel.com/main/main.php (Last visited: 2016-04)

[64] LinkSprite pcDuino. [Online]. Available: http://www.linksprite.com/linksprite-pcduino/ (Last visited: 2016-04)

[65] PiCamera documentation, section 10, API picamera.camera module. [Online]. Available: https://picamera.readthedocs.io/en/release-1.10/api_camera.html (Last visited: 2016-04)

[66] PiCamera documentation, section 4, Basic Recipes. [Online]. Available: https://picamera.readthedocs.io/en/release-1.10/recipes1.html (Last visited: 2016-04)

[67] PiCamera documentation, section 5, Advanced Recipes. [Online]. Available: http://picamera.readthedocs.io/en/release-1.10/recipes2.html#rapid-capture-and-processing (Last visited: 2016-05)

[68] PiCamera documentation, section 7, Camera Hardware. [Online]. Available: http://picamera.readthedocs.io/en/release-1.10/fov.html (Last visited: 2016-05)

[69] R. de Charette and F. Nashashibi, “Traffic light recognition using image processing compared to learning processes,” in *Proc. IROS*. IEEE/RSJ, 2009, pp. 333–338.