NUI Galway
OÉ Gaillimh

B.E. Electronic & Computer Engineering Project Report

ECE

April 2017

Design and Implementation of an Object Tracking Engine for Video

Aaron Regan

Supervisor: Dr Peter Corcoran

Abstract

This report details the method developed to implement an object tracking engine for video. The idea was drawn up to track pedestrians on a live video feed on an embedded device. The project was carried out to implement an efficient pedestrian tracker at a low cost, on an easily available device. The applications for pedestrian detection and tracking expand into many fields such as security, automotive and in public transport. For an image processing task, the primary goal is to extract useful information from an image or video and analyse a scene; the project's overall outcome achieves this.

Statement of Originality

I hereby declare that this thesis is my original work except where stated

Signature_____ Date: _____

Acknowledgements

I would like to thank Dr Peter Corcoran for his guidance and mentoring during the project. I would also like to thank my co-supervisor Dr Eddie Jones for his review of my progress midway through the project. Finally, I would like to extend my gratitude to the staff and technical officers of the Electronic Engineering Department for their support throughout the project.

Table of Contents

List of Figures

List of Charts

List of Tables

Glossary

RPi – Raspberry Pi

HOG – Histogram of Orientated Gradients

OEM – Original Equipment Manufacturer

SVM – Support Vector Machine

FPS – Frames per Second

NMS – Non-Max Suppression

MOG – Mixture of Gaussian

ROI – Region of Interest

Chapter 1. Introduction

The project requires the implementation of a real-time tracking engine on an embedded system as the end goal. A software architecture will be developed which can extract key features from a video frame and determine the presence of a pre-determined object.

The use of object tracking is becoming ever more important in the modern world as the use of cameras in everyday life becomes more prevalent. Implementing software in these cameras to allow for real-time image processing is becoming more practical as cheaper components with faster processing power are more accessible.

There is three steps in object tracking as stated by Yilmaz et al. “Detection of interesting moving objects, tracking of such objects from frame to frame, and analysis of object tracks to recognise their behaviour” [1]. This project aims to implement each step in a real-time system, which is common in such systems as automotive [Figure 1.1], traffic surveillance and security systems.

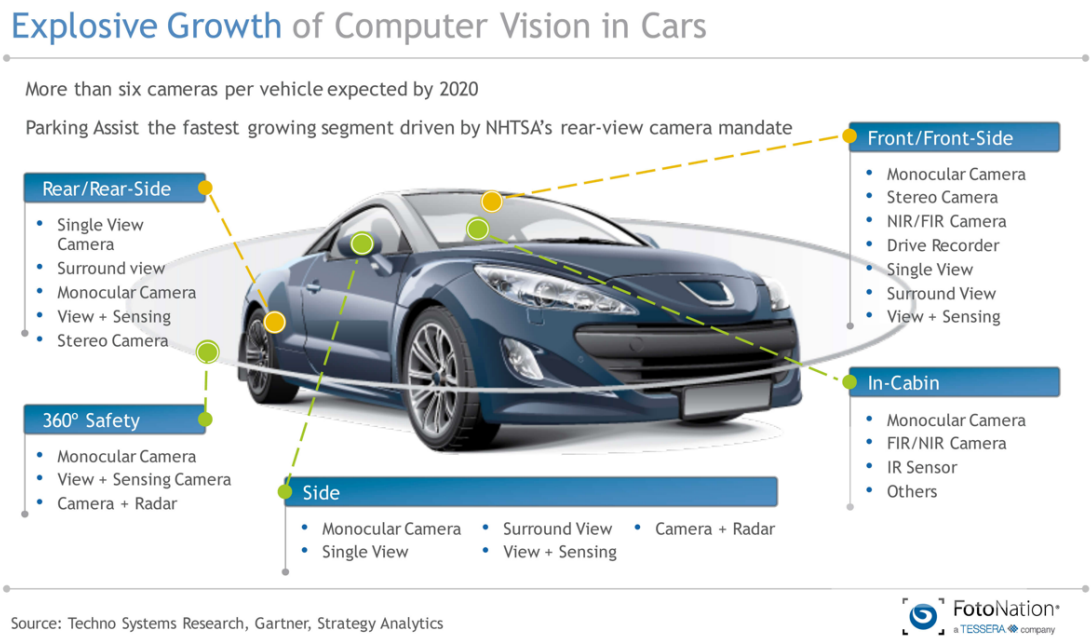

Figure 1.1 - The use of camera systems in cars

My initial interest came about from my placement at Valeo where I worked as part of the automated parking team in research and development. The exposure to the latest camera systems with various computer vision software used for detection and

tracking appealed to me. Currently “90% of accidents are caused by preventable human errors” [2], with increased semi-automated systems which safeguard various aspects of driving, this number can significantly reduce. Reduction in accidents can be achieved through the use of multiple sensors and cameras. Using object tracking systems, we would be able to predict and warn drivers of hazards on the road. This semi-automatic car would pave the way for an automated driving experience, which would hopefully reduce the number of deaths on the road.

The applications of this software can integrate into many areas, which companies such as Fotonation have set out to achieve. Founded in 1997, they create intelligent camera systems which include object detection and tracking software. According to Tractica’s report [3] on the growing computer vision market shows the interest in research and industry to be growing rapidly. The market itself is expected to grow from $5.7 billion in 2014 to $33.3 billion by 2019 showing promising investment in the area [Chart 1.1] from companies such as Google, Nvidia and Microsoft.

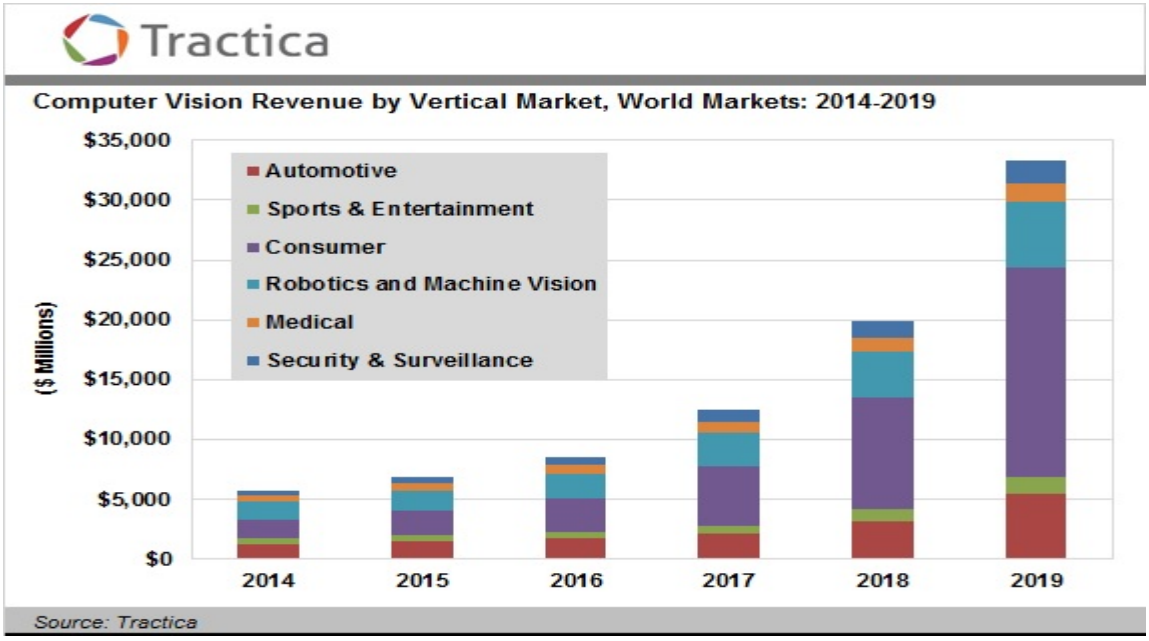

Chart 1.1 - Computer Vision Market 2014 - 2019 [3]

My interest in this area is what drove me to work on a project such as this one, With the intention of learning computer vision theory and techniques to implement an object tracking engine.

Chapter 2. Literary Review & Existing Technologies

2.1. Background

The field of computer vision technologies has seen a substantial increase in activity over the last two decades; this is in part due to the increase in affordable hardware for the field, with high-quality cameras and computer systems capable of handling the requirements of vision software. The increase has opened up research and development beyond those with substantial financial backing for expensive research materials and placed into the hands of students, hobbyists and enthusiasts allowing a community to form, which is constantly trying to improve upon the already existing technologies.

Secondly, Open source communities such as OpenCV have developed, allowing for a central repository where new implementations of different computer vision ideas get stored. Beginning in 1999 as an “Intel Research initiative to advance CPU-intensive applications” [4] it now has over “2500 optimised algorithms” [5]. The coordination within the community has created a hub of research and project work, where academics and hobbyists can jump start their work and build real-world applications using computer vision.

An extensive area of research carried out in this area is with object detection and tracking. The reasons behind this are the same reason computer vision as a whole saw a huge increase in research with the addition of the “increasing need for automated video analysis” [1]. These three conditions combined created a need and ability for object detection and tracking software to be designed and implemented. Though there are many different methodologies for tracking and how to determine our object of interest, in this project the focus is on detection of a pedestrian, which quickly narrowed down the technologies that would be used.

For the scope of this work to be carried out, we will define object tracking as “the problem of estimating the trajectory of an object in the image plane as it moves around a scene.” [1] from here we apply this to our pedestrian as the ‘object’. There have been many methods of determining this through the OpenCV libraries. The

steps and techniques used to achieve this object detection and tracking have been researched previously with various approaches to each level of the system.

2.2. **Academic Research**

The first step in tracking an object is to determine how to represent this object. An object can be defined as “anything that is of interest for further analysis.” [1] It is important to select the representation that best defines the object. The various methods of representation include:

- Points (Figure 2.1 A, B)
- Primitive geometric shapes (Figure 2.1 C, D)
- Object Silhouette and contour (Figure 2.1 G, H, I)
- Articulated shape and models (Figure 2.1 E)
- Skeletal models (Figure 2.1 F)

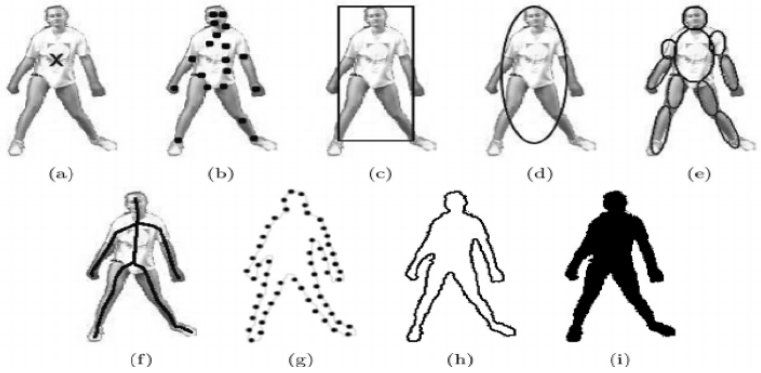

Figure 2.1- Various Object Representation techniques

The second step is to determine the best feature selection for the region of interest (ROI). As with any machine learning task, choosing a good feature to represent the ROI is imperative, as Shi et al. point out in their paper “the right features are exactly those that make the tracker work best.” [6] The type of feature selection used varies depending on the object we desire to detect and its environments such as uniqueness of the feature to allow the system to distinguish it from irrelevant articles and noise, or illumination of the object and how this can vary. Some features which can be used for selection include:

- Color

- Edges
- Optical flow
- Texture

Colour is the most widely used feature selection but is susceptible to illumination changes, whereas Edge is less sensitive to this.
Once the representation and feature selection have been determined, we can apply it to an object detection algorithm. Every tracker needs a method of detecting the ROI. This can be achieved by analysing the first frame the object appears in, or every frame in the system. Some of the popular object detection techniques include:

- Point detection
- Background subtraction
- Segmentation
- Supervised Learning

The background subtraction technique, for example, tracks based on motion. It does this by creating a model image, where each successive image is compared, and differences are noted and marked. Its advantages come from its general ease of implementation and high true positive rate. However, its major pitfall comes from its basic premise of tracking motion; there is no way of differentiating our objects of interest. Another problem stems from the need for a stable camera, As discrepancies between the model and successive frames can lead to a breakdown of the systems ability to track. There have been successful attempts to overcome this, Toyama et al. designed the Wallflower, creating a three-tiered system whereby if there is a sudden change in the pixels at the frame level, it is assumed the pixel-based model created is no longer useable. If this happens the previously stored model is swapped in, or the model is reinitialized. [7] This technique can avoid changes in the environment such as leaves moving or clouds, giving the detector a greater accuracy.
Some supervised learning methods include utilising Viola Jones Haar Cascade [8][1] which employs AdaBoost and training sets of pedestrians to create a cascade used to identify the object of interest in images. This method has advantages over

[1] Haar Cascade is described in further detail in Section 7.3

techniques such as background subtraction thanks to its ability to differentiate between objects of interest and other objects in the frame.
Another implementation of supervised learning within object tracking techniques is a histogram of orientated gradients(HOG) from Dalal et al. were a support vector machine(SVM) is trained using a training set of pedestrians. [9] This method again has significant benefits against background subtraction, thanks to its use of gradient vectors to determine the presence of a person in an image. [10]
Now that the system can identify an object and map it through feature representation we can begin to track the object of interest. “The aim of an object tracker is to generate the trajectory of an object over time by locating its position in every frame of the video.” [1] this can be split into three categories of tracking:

- Point tracking
- Kernel Tracking
- Silhouette Tracking

The tracking methods differ by their method of determining the location of the detected object in successive frames. An example of point tracking is the Kalman Filter. Based on a point from the centre of the region of interest, using a statistical model, the Kalman generates a predicted future movement based on velocity and location. An example of its use can be seen in Broida et al. paper on object tracking. [11]
Implementation of these techniques to generate object detection and tracking systems can be seen in the patent for a Real-time face tracking in a digital image acquisition device [12]. Corcoran et al. use a face detector on raw images, which are then passed to a tracker for tracking the detected faces. It is noted that working with an image that has been reduced in size significantly speeds up the memory intensive sections of the system as “1/4 of the main image e.g. only has a ¼ the number of pixels and or ¼ of the size , then the processing time involved is only about 25% of that of the full image” [12]. Something which will heavily feature when developing the object tracker for live video.

2.3. Industry

Prevalence of computer vision in modern industry is becoming more and more the norm as everything from automotive to social media is finding ways of incorporating it into their functionality. Facebook AI Research use image recognition software to determine the content of photos uploaded to the social media platform. Using a combination of Deepmask, Sharpmask and multiPathNet an image is segmented, the objects of interest are singled out, and classification of the object takes place. This technique makes use of neural networks, a form of machine learning used to understand and classify images. [13]

Another area investing heavily in computer vision is the automotive industry. The benefits of pedestrian detection regarding road safety can lead to fewer pedestrian deaths as it is implemented in auto-braking and early warning systems as part of advanced driver safety systems. The number of Original equipment manufacturers(OEM's) incorporating this feature into their production cars has rapidly increased over the past few years as camera equipment, and computer capabilities have improved.

Some companies carrying out research in this area and developing software and hardware for production include Fotonation [14], Valeo [15] and Mobileye [16]. In 2006 Mobileye and Volvo released their "Collision Warning with Auto Brake", a system designed to detect pedestrians and automatically apply the brakes to avoid an impact.

2.4. Conclusion

Although a quite a growing area of research, the benefits of computer vision and more specifically object detection and tracking have to lead to tremendous advancements in this field of study. The versatile applications of such a system have created a project with vested interests from many different areas from automotive, social media and security. The goal of this project of implementing a pedestrian detector on a low cost easily available device will hopefully serve as an example of how far the field of computer vision has come.

Chapter 3. Software

3.1. Introduction

This chapter details the software tools and packages used to implement the system

3.2. OpenCV

OpenCV is an open-source programming library used for real-time computer vision applications. The library has “more than 2500 optimised algorithms” [5] which can be used in conjunction with Python to allow the manipulation of images in the form of matrices which can then be processed to provide object tracking.

OpenCV 3.1.0

3.3. Python

Python was chosen as the programming language for this project for a number of reasons. Firstly, its ability to implement OpenCV libraries gave its suitability for the work to be carried out, as this was the primary importance of the chosen programming language. Other languages which could implement OpenCV included Java, C and C++. Secondly, it is a high-level language which makes it easier to identify what is happening behind the scenes of the object tracking algorithm, making it easier to debug and build on. Libraries such as numpy are useful for this type of application, as image processing requires the manipulation of matrices to allow image analysis and comparison to take place, and is the core process carried out in object tracking. For this reason, it was chosen over the other OpenCV compatible programming languages.

The python version used on the system is Python 2.7.12

Chapter 4. Hardware

4.1. Introduction

This chapter details the hardware used to implement the object tracking system. It is split between the initial testing of the system on a PC and the implementation of a system on the embedded system using the Raspberry Pi

4.2. MacBook Pro

For initial work, the use of a MacBook Pro [Figure 4.1] with considerably higher processing power than a Raspberry Pi was used to provide an efficient method of coding and image processing. The front facing 720p FaceTime HD camera is used to capture/stream video. This is in line with the task list for implementing a tracking algorithm, as it allows for faster testing of the various methods.

Figure 4.1 - MacBook Pro Specifications

4.3. Raspberry Pi

The embedded system on which the chosen tracking algorithm will be implemented on is the Raspberry Pi 3. It was chosen due to its low cost and the ease with which it can be obtained. The Model 3 has a 1.2 GHz 64-bit quad-core ARM Cortex-A53 with a 15-pin MIPI camera interface (CSI) connector where a camera module can be

connected. Processing power is lower than the MacBook, so the implementation of the tracking algorithm on the embedded system would take place after initial testing.
OS: Raspbien Jesse

4.4. Pi Camera NoIR

The Cameras chosen for the embedded system was the compatible Pi Camera module which is attached to the Pi 3 via a 15-pin MIPI CSI. It can capture 1080p videos, with a resolution of 3280 × 2464 pixels [17]. These specifications can be appropriately scaled down to suit the optimised performance of the tracking algorithm.
The Noir camera module allows for ‘night-vision’ by not filtering out infrared, allowing for night use if illuminated by infrared light. This is to allow the possible expansion into night-time object tracking if time permitting.

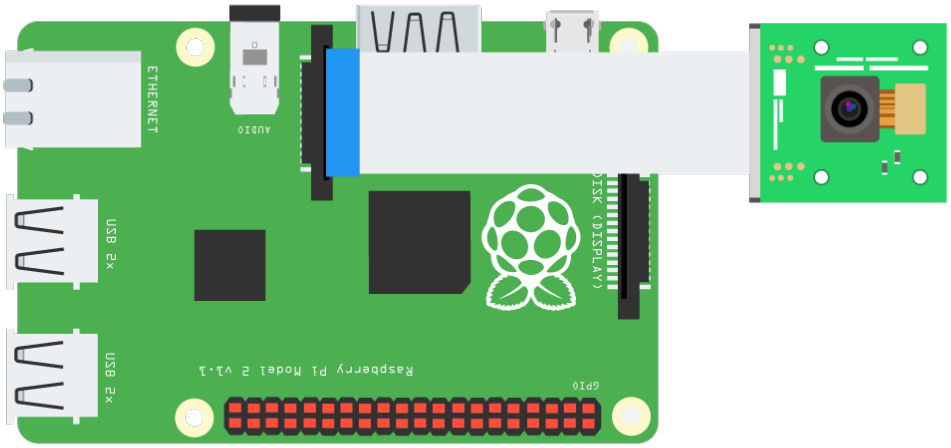

Figure 4.2-Raspberry Pi with Pi Camera attached [18]

Chapter 5. Installation

5.1. Introduction

This chapter details the installation of the software onto both the PC and embedded system.

5.2. Installation

The first step carried out was to install Python and OpenCV on the embedded system and MacBook. Both were very similar process however the faster processing power of the Mac meant it was far quicker.
A tutorial [19] on how to install the language and libraries was found for both processes. HomeBrew, a package manager to help download packages needed to support the build, was installed. Then a virtual environment was created to allow us to build separate Python projects in the future. This virtual environment can be accessed through the command ‘workon cv’ [Figure 5.2].

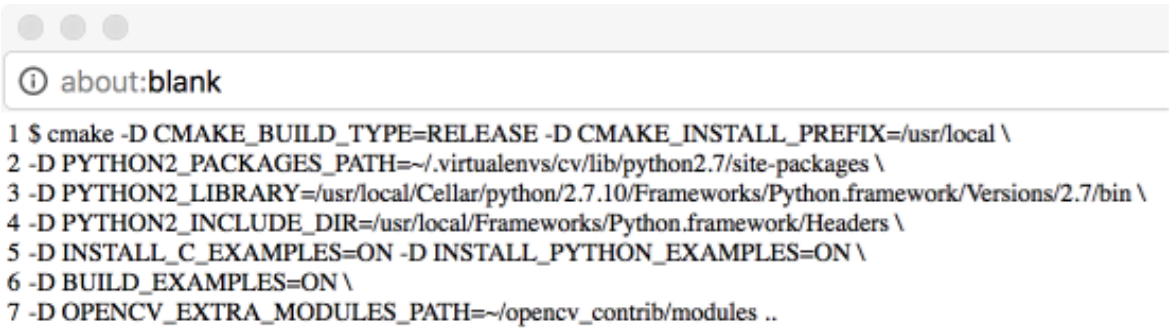

Figure 5.1 - Screenshot of CMake configuration

OpenCV is then cloned from their Github repository at https://github.com/Itseez/opencv.git. The build directory is then created, and CMake is used to configure the build. This contains various paths to where our python installation is kept, the ‘cv’ virtual environment, base directory, etc. [Figure 5.1] once CMake is properly configured we can begin to compile OpenCV and install it on our

Osx/Raspbien. The process took ~30 minutes with the compile taking the most time; this is quite short compared to the ~1.5 hours needed to carry out the process on the Raspberry Pi.

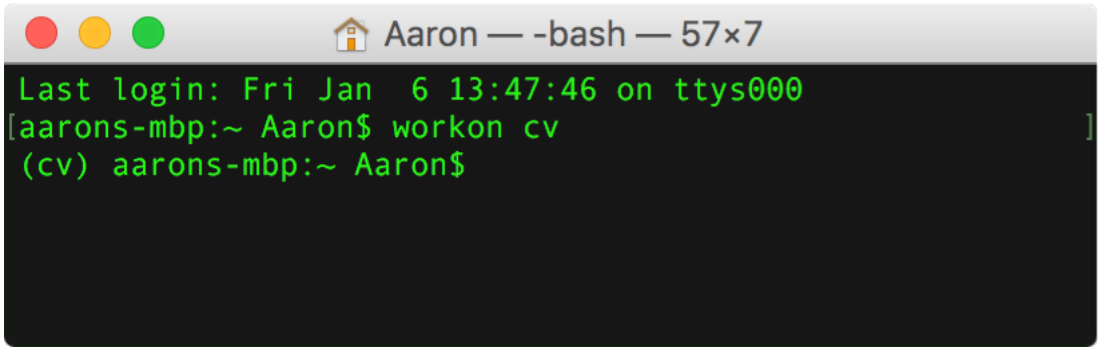

Figure 5.2 - Screenshot showing 'cv' virtual environment

5.3. Github

Distributed version control was used to save work and provide a back up in an online repository. The service used is Github; a web-based hosted service for git repositories. The code for both the testing of algorithms and the final implementation is available at https://github.com/AaronRegan/object_detection.

Chapter 6. Architecture

6.1. Introduction

This section outlines the overall system flow and design chosen to implement the object tracking system.

6.2. Pipes and Filters

The chosen architecture for the overall system and sub-subsystems was the Pipes and filters method. The structure consists of inputs and outputs of manipulating functions(Filters) passed between each other through the connections between these functions(Pipes).
Its advantages stem from its simple design approach, making it easy to follow the flow of the grabbed frames throughout the object tracking process providing valuable debugging properties. Secondly, the modular design of the architecture provides a separation of concerns during the running of the code and the handling of the manipulated frame. This provides greater system reliability allowing for a cohesive program that can be easily debugged when errors are thrown.
The major pitfalls of Pipes and filters design stem from its ease of use, as bottlenecks occur at slower processes, leading to an overall high level of program latency. As Alexandre R.J. François points out the “overall throughput of a Pipes and Filters system is imposed by the transmission rate of the slowest filter in the system” [20]. Because of this, the less memory intensive functions are delayed by more memory intensive functions, such as the object detector itself.
The structure of the architecture can be seen in Figure 6.1; The data being passed through the pipes is the image collected from the Pi Camera. The filters shown take the image as their input and output a manipulated version, which is passed to the final user monitor. The Inner loop is shown bounded by the blue box. Here the image is moved into an image resizing filter, where the number of pixels is reduced to a threshold point of peak performance. Performance is represented by the speed at

which the image is processed and the ability of the system to identify and track a pedestrian.

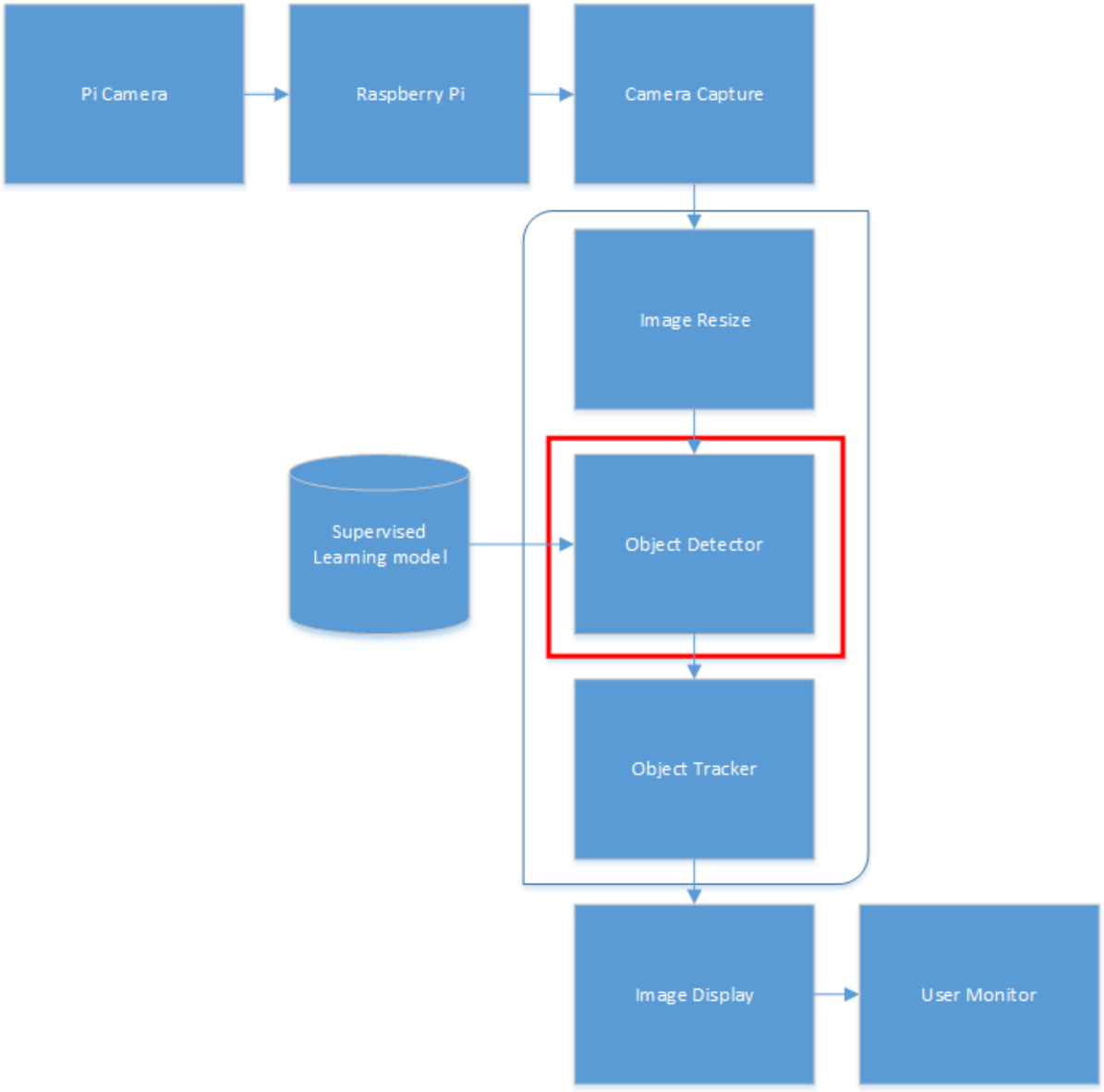

Figure 6.1 - Pipes & Filters Architecture

The next filter is the object detector subsystem, where a machine learning model is utilised. This is generally in the form of a cascade or descriptor stored as an XML file. This data will be called and initialized within the system to compare the image results with to determine whether any objects within the frame do in fact represent a person. It is bounded by a red box to indicate it as an area for the potential bottleneck. This is due to it being a memory intensive process, and as explained

above, can lead to a slowing of other sub-systems, and therefore leading to a slowing of the overall system.

The frame is then passed to the next filter, the object tracker. This step utilises a tracking method to determine the location of the detected object in successive frames. It too has potential to be a memory intensive filter depending on the tracking method used. However, some tracking methods such as Point tracking can be quite fast compared to others, so its potential of being a choke point in the system is lower, and therefore not marked as such.

Leaving the central image processing loop, we then pass the frame, with manipulations imposed, to the display image filter, where the image is piped to the user monitor for viewing. These are non-memory intensive filters, but they are only as fast as the filters that precede them due to the structure of the architecture.

Having chosen to implement this architecture due to its relative simplicity, it is important to ensure choosing an efficient object tracker to provide maximum performance while not sacrificing the throughput of the entire system.

Chapter 7. Other Algorithms Tested

7.1. Introduction

This Chapter Details the various algorithms implemented that were not used in the completed tracking engine design. A total of two techniques were tested for pedestrian detection and later discarded, both of which are discussed in more depth below.

The First sub-system to implement within the tracking engine was an object detector. The following sub-chapters outline various methods that were implemented and tested.

7.2. MOG Background Subtraction

7.2.1. Introduction

Background subtraction is a model based detection method, where a model is created from the first frame the camera captures. Successive frames are then compared to this, with any deviations from the model being marked as an object of interest. The process was the first algorithm selected to test as it as suitable for the task of detection within the tracking engine and provided a high detection rate.

The algorithm was implemented first on a PC and tested on stock footage[2] of a man walking across a pier; this video was chosen as it had a single person in the frame for the entire length of the video, except for the first frame, allowing for the creation of a suitable model.

[2] The Video used for testing the background subtraction was downloaded from PixaBay.com [26].

7.2.2. Implementation

The frame is first passed through a resizing function from the imutils library [21] to decrease the number of pixels in the image, reducing the time taken for the object detector to scan the image, thus increasing the overall speed of the of the system. This is followed by converting the RGB image to grey scale, as determining the difference between the model and successive images does not require colour and will decrease processing time. Followed by Gaussian smoothing to smooth the frame using the Gaussian Blur function as seen in Figure 7.1. The images being compared to the model will not be identical due to illumination changes and slight movements in the environment. By applying Gaussian smoothing, we can eliminate these minute differences and noise, preventing false positives when comparing with our model.

```
# resize the frame, convert it to grayscale, and blur it
frame = imutils.resize(frame, width=700)
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (21, 21), 0)
```

Figure 7.1 - Extract of code showing Gaussian Blur function

Figure 7.2-The Resulting image after generating the absolute difference and applying the threshold

Next, the smoothed image is compared to the model using OpenCV absolute difference function, where any difference in the two images will be displayed to the user as seen in Figure 7.2 above. This function applies a subtraction between the pixel values between the two images, returning the absolute value of the differing pixels. To eliminate pixels below a certain threshold, we apply a limit using cv2.Threshold, where the pixels with an absolute value lower than the set threshold are removed, leaving a mask of only the moving object.

```
#Computing the absolute difference bewteen the model and the current frame
frameDelta = cv2.absdiff(firstFrame, gray)
thresh = cv2.threshold(frameDelta, 25, 255, cv2.THRESH_BINARY)[1]
```

Figure 7.3- Extract of code showing absolute difference function & applying threshold

Figure 7.4-The Resulting image after using dilation on the masked image

The image is then passed to a dilation function as seen in Figure 7.5, dilating the threshold area to fill in any gaps or missing pixels which can be seen above in Figure 7.4. By applying the find Contours function the resulting image is then searched for shapes that are above a minimum size, by using a for loop to eliminate any objects that are too small. The find Contours function returns the contours to a list with the corresponding height and width, from these a bounding box encloses the object as

seen in Figure 7.6 signifying detection and the final image is displayed to the user as seen in Figure 7.7.

```
#dialation is carried out fill in any pixel gaps within the contours
#find contours within the final frame
thresh = cv2.dilate(thresh, None, iterations=2)
cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[-2]
```

Figure 7.5 - Extract of code showing dilation function & finding contours

```
# checking every contour returned by check contours function
for c in cnts:
    # if the contour is too small, ignore it
    if cv2.contourArea(c) < args["min_area"]:
        continue

    #for each contour surround with bounding box
    (x, y, w, h) = cv2.boundingRect(c)
    cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
    i=i+1
```

Figure 7.6 - Extract of code for bounding contours

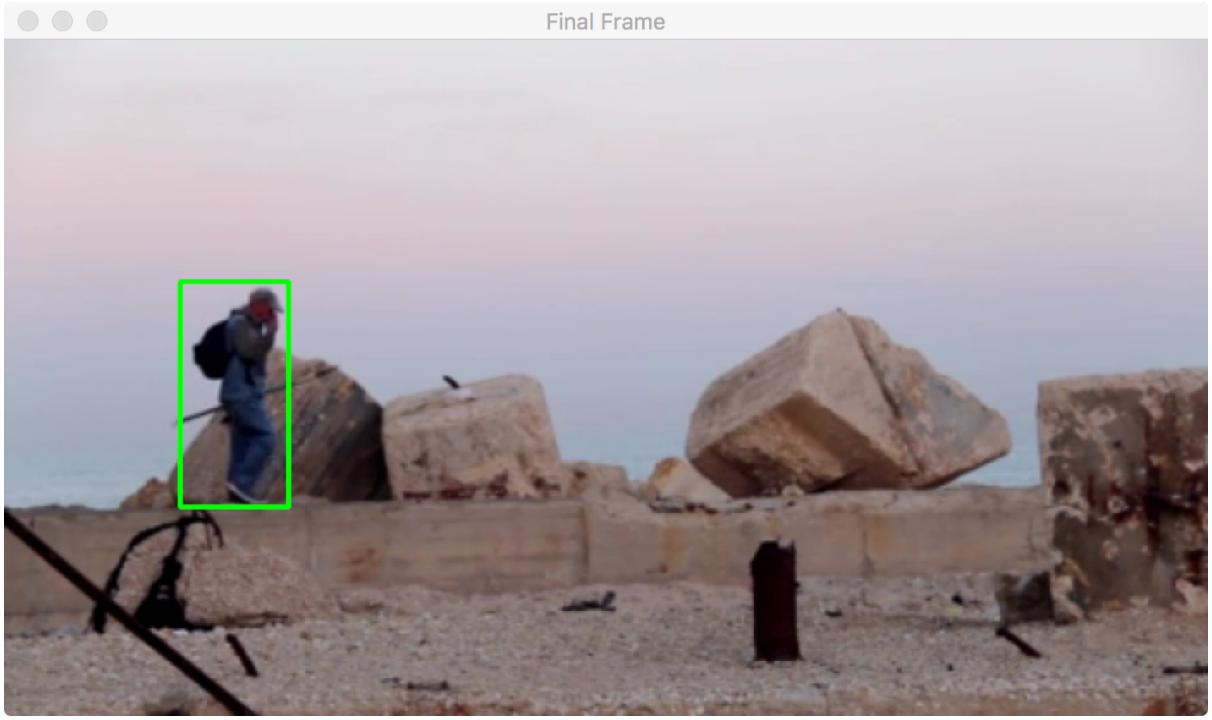

Figure 7.7- Final image displaying object in motion within bounding box

7.2.3. Review

The background subtraction technique proved very efficient regarding processing speed and detection. The method of detection did not create a bottleneck of too vast a scale, and in turn, lead to a fast system. It was not implemented in the final system due to its inability to determine between objects of interest (pedestrians) and any other moving object. Instead opting for a pattern matching approach, which provided the ability to determine regions of interest within frames.

7.3. Haar Cascade

7.3.1. Introduction

Haar wavelets were first theorised as a method of object detection by Papageorgiou et al. [22] where it was suggested a set of Haar wavelets of the object of interest could be put into a support vector machine to produce a model of the object. This was then put into practice by Viola et al. [23] by developing Haar-like features. A Haar-like feature is calculated by summing the pixel intensities of adjacent rectangles and getting the difference between the sums. The difference is then used to categorise sections of the image. For example, the brow is generally of a different pixel intensity to the eye region as seen in Figure 7.8, leading it to be a Haar-like feature.

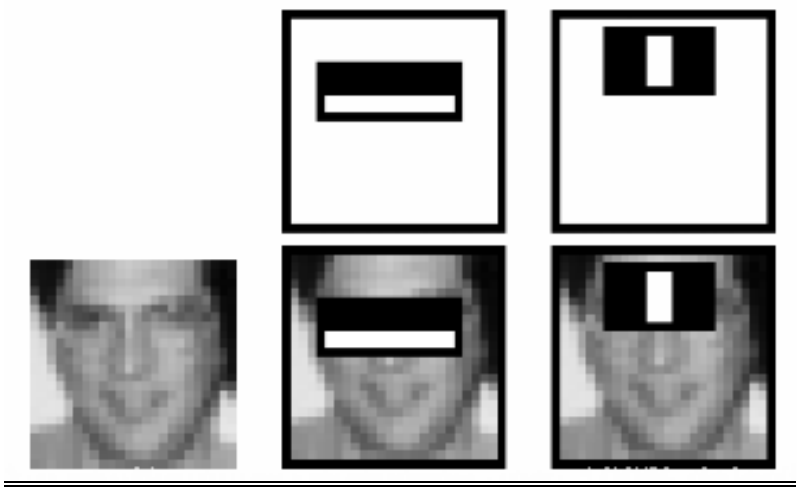

Figure 7.8 - Example of using Haar wavelets for face detection [5]

By applying these same methods to a pedestrian, and building a cascade, we could develop a pedestrian detection system using Haar-like features. The video used for testing was used for its suitability, as it had a person to be detected in every frame. [24]

7.3.2. Implementation

Implementing the Haar Cascade method begins by assigning a variable to the cascade[3] by directing it to the cascade path as can be seen below in Figure 7.9. The cascade is an XML file containing the Haar-like features from a pre-prepared model derived from training data sets of pedestrians. The model is built using AdaBoost, a supervised machine learning algorithm which adds weights to the classification of objects in the training set. The model then selects a base classifier with the least error. The error is proportional to the weights of the misclassifications. The process is repeated, choosing a different classifier that will work better on the data than the previous selection.

```
fullbody_cascade = cv2.CascadeClassifier(
    '/Users/Aaron/Documents/College/Fourth_Year/Final_Year_Project/Datasets/body10/haarcascade_fullbody.xml')
```

Figure 7.9 - Assigning the Haar Cascade for Full body detection

This classifier will be used to determine the location of pedestrians in the frame being passed into the programme using a sliding window. The sliding window is a defined $m \times n$ matrix of pixels which moves across the incoming frame, taking the Haar like features and passing them to our model for analysis. The performance of the detector is largely based on image size and sliding window size, so to help increase processing speed, the image is resized.

```
frame = imutils.resize(frame, width=300)
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
fullbody = fullbody_cascade.detectMultiScale(gray, 1.03, 1, 0 | 1, (40,60), (80, 400))
for (x, y, w, h) in fullbody:
    i=i+1
    cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
```

Figure 7.10 - Extract of detectmultiScale function and drawing bounding box

[3] The Cascade was downloaded from Ale Reimondo's online webpage [27]

The frame is converted to grayscale, as we are dealing with pixel intensity, and this can speed up processing time. Using the detect multiScale function, we pass the sliding window over the image searching for Haar-like features that represent a person. If any people are detected, dimensions for a bounding box are returned, and the enclosure is drawn. The result the end user can be seen in Figure 7.11 below.

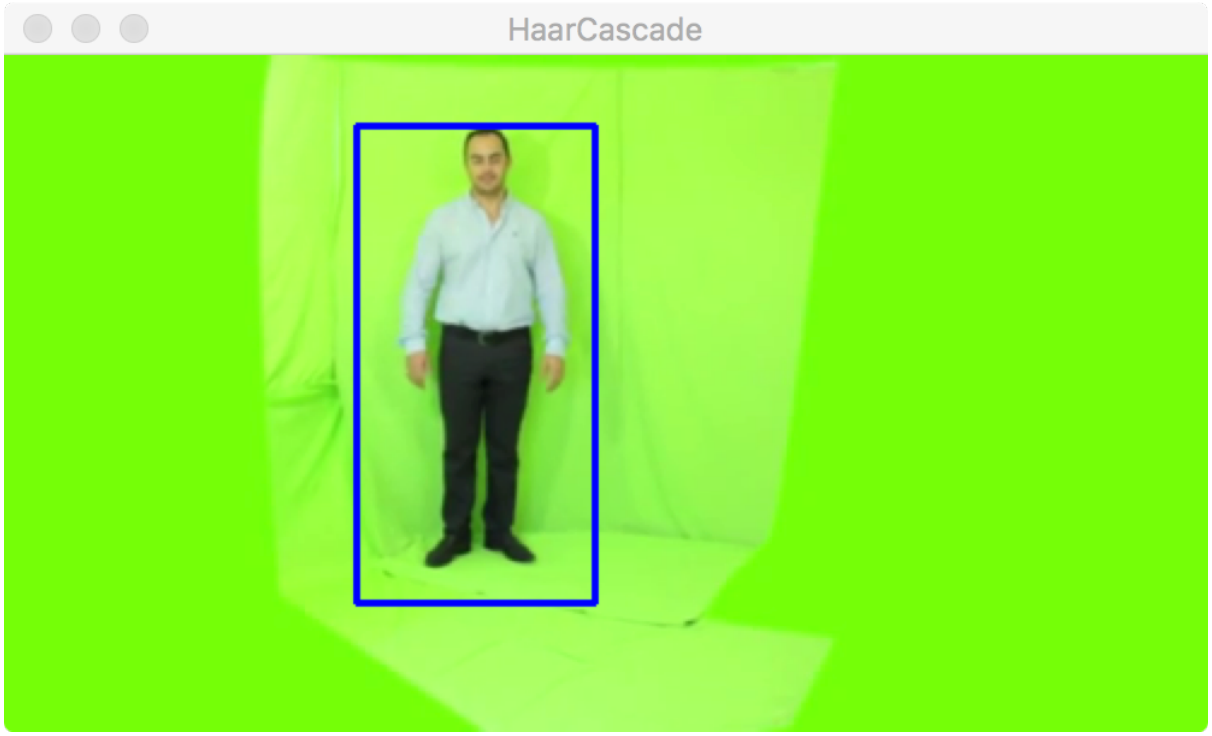

Figure 7.11 – implementation of Haar Cascade

7.3.3. Review

The Haar cascade was effective at identifying the object of interest for half the time in the video. The Haar was developed by Viola et al. to identify faces, which contain many suitable Haar-like features, whereas the human body can be harder to identify from these. Because of this, the Haar classifier was not chosen for the final system.

Chapter 8. Object Tracking Engine Implementation

8.1. Introduction

This chapter details the final implementation of the object tracking engine on the Raspberry Pi using a PiCamera, with all systems and subsystems discussed.

8.2. Histogram of Orientated Gradients

Histogram of orientated gradients(HOG) is a supervised classifier implementation of an object detector which utilises pattern matching techniques to determine the presence and location of an object of interest. First proposed by Dalal et al. [10] it uses the gradients within frames to determine the presence of a person based off of an SVM model. Its accuracy and consistency throughout development and testing lead it to be chosen for the object detector in the tracking engine.

8.2.1. HOG Descriptor

The HOG descriptor is built using large datasets of positive and negative images. Positive being images containing the object of interest and negative images being of some other scene, as long as they do not contain an object of interest. The dataset used to build the HOG descriptor with OpenCV is the INRIA dataset, a collection of positive and negative training and testing datasets for creating the HOG descriptor. The data collected is in the form of histograms, with a set amount of bins. These histograms contain gradients within the photo which can be used to match common features when they are found.

To create an object detector, the HOG descriptors of all the positive images are extracted and stored. The process is repeated for the negative images, and both are used to train a linear SVM. The resultant model created can now be used when applying HOG to the incoming frames within our object tracking engine.

8.2.2. Linear Support Vector Machine

SVM's are supervised learning models used for classification purposes. Trained using the training set from the INRIA [10] data set it separates each HOG passed to it by a linear classification line. Samples are mapped based on whether or not there is a pedestrian present within the HOG. New examples are then mapped to this sample space, determining whether or not it contains a pedestrian. Weights are returned, based on the likely hood of the classification being correct. Data from the test video [24] shows the returned weights after performing HOG classification. [Chart 8.1]

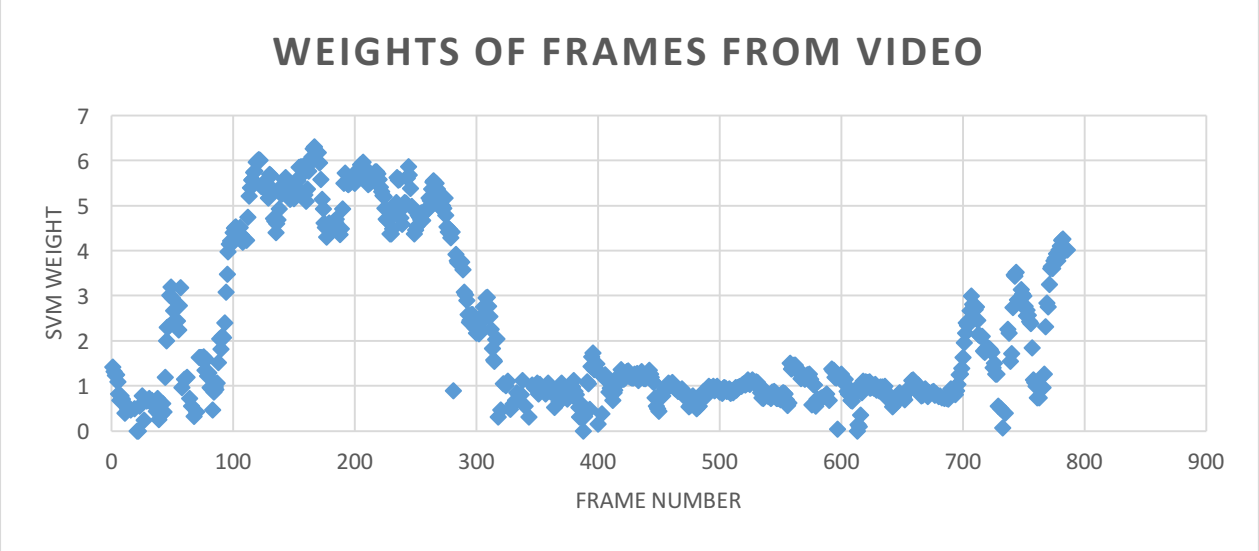

Chart 8.1 - Weights returned from performing HOG classification on Test Video

We can clearly see that areas where the model was most confident that the detection was a person. Between frames 100 to 300 the person is standing in plain view facing the camera leading to a high level of confidence. From frames 300 to 700 the person is now walking towards the camera, turning around and facing away. This is reflected in the drop in the classifiers confidence. Finally, between frames 700 and 800 the person returns facing the camera increasing the weighted confidence of the classification[Figure 8.1]. It should be noted that the drops in confidence still return a positive match for the presence of a pedestrian.

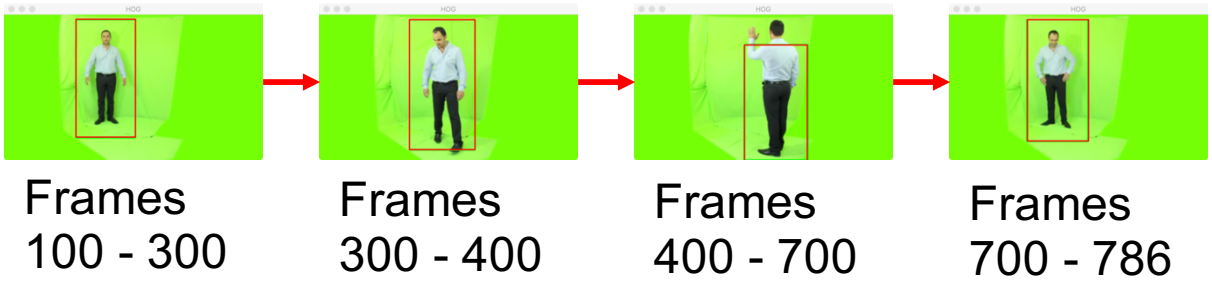

Figure 8.1 - Visual representation of Chart 8.1

8.2.3. Pedestrian Detection

The HOG was implemented on a PC webcam initially for testing as with Haar and Background subtraction. After its better performance over the other detectors used, it was chosen to be implemented on the raspberry Pi with PiCamera for live video tracking.[4]

To begin, the PiCamera is initialized to allow for a camera feed to open; this is done by setting the parameters for resolution, frame rate and flipping the image to take into account the cameras upside down orientation. Finally, the RGB array is setup and assigned to a variable to store the incoming raw image matrices.

```
# initialize the camera and grab a reference to the raw camera capture
camera = PiCamera()
camera.resolution = tuple(conf["resolution"])
camera.framerate = conf["fps"]
camera.vflip = True
rawCapture = PiRGBArray(camera, size=tuple(conf["resolution"]))
```

Figure 8.2 - Extract of code showing camera initialization

A configuration file was used to pre-store all values needed to set up the camera. The file was called within the functions as can be seen above in Figure 8.2.Following this, the HOG descriptor was initialized by calling the cv2.HOGDescriptor() function and setting up the default people detector provided by OpenCV [5].

---

[4] The extracts are featured in the main body of code, which can be found in the Appendix

To loop over the incoming frames for analysis the raw camera stream was put into a for loop, as each frame passed, it was grabbed and assigned to a variable to be processed.

```
# capture frames from the camera
for f in camera.capture_continuous(rawCapture, format="bgr", use_video_port=True):
        frame = f.array
```

Figure 8.3 - Extract of code showing for loop over incoming camera stream

The extracted frames are then re-sized, to ensure maximum performance without sacrificing accuracy and passed to the detectMultiScale function[5]. The Four parameters as seen below in Figure 8.4 dictate the actions of the sliding window used to extract the HOG features from the incoming frame. winStride() represents the step size taken by the sliding window. At each step, the sliding window retrieves the HOG features and passes them to the SVM for analysis. This parameter is important as it can affect the speed and accuracy of the detector significantly.

```
frame = imutils.resize(frame, width=250)
e1 = cv2.getTickCount()
    # detect people in the image
(rects, weights) = hog.detectMultiScale(frame, winStride=(8, 8),
                                        padding=(24, 24), scale=1.05)
e2 = cv2.getTickCount()
timeHOG = (e2-e1)/cv2.getTickFrequency()
```

Figure 8.4 - Extract of code showing the detectMultiScale function

The padding parameter determines how many pixels in the x-direction and y-direction are padded before hog extraction. This value does not affect performance as much as other parameters but is necessary for the overall effectiveness of the detector. The 4th and final parameter used, like window stride, can significantly affect the performance of our detector. The scale determines how much our image is downsized at each at each layer of the image pyramid. A smaller scale increases the

[5] Testing of the parameters for the HOG classifier can be found in Testing

number of layers in our pyramid, therefore increasing our detection abilities, but can increase the time taken to compute in the process. As can be seen in Figure 8.4 the function has a timer wrapped around it, to allow for testing of various parameters and their execution time. The function returns an array of co-ordinates enclosing the detected objects in the frame, and weights for the confidence value returned by the SVM.

```
rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in rects])
pick = non_max_suppression(rects, probs=None, overlapThresh=0.65)
```

Figure 8.5 - Extract of code detailing the use of the non-maxima suppression function

Using the imutils [21] library we pass the output of the detectMultiScale function to the non-max suppression function, as seen in Figure 8.5 above, to suppress excess bounding boxes, ensuring only one bounding box surrounds the object of interest. This function is described in detail in section 8.3 below. The output of this function passes the new (x, y) locations to the loop seen in Figure 8.6 where the bounding boxes are drawn. This frame is displayed to the user on screen [Figure 8.7] and then cleared from the rawCapture container using the truncate function to make way for the next frame, which is loaded into the loop for processing. The user can exit the loop by pressing ‘Q’ on the keyboard.

```
# draw the final bounding boxes
for (xA, yA, xB, yB) in pick:
    cv2.rectangle(frame, (xA, yA), (xB, yB), (0, 255, 0), 2)
```

Figure 8.6 - Extract of code detailing the bounding box loop

Figure 8.7 - Screenshot of Live object detection using HOG and RPi

8.3. Non-Maxima Suppression

Non-Max suppression is used to reduce the number of overlapped bounding boxes returned by the HOG classifier. The imutils [21] library provided the functionality to ensure that our detector only bounded the objects once, preventing detection clutter. The function works by taking all bounding boxes and calculating their area from the bottom right (x, y) co-ordinate. From here depending on the overlap threshold set, we eliminate boxes with an overlapping area as seen in Figure 8.8 below.

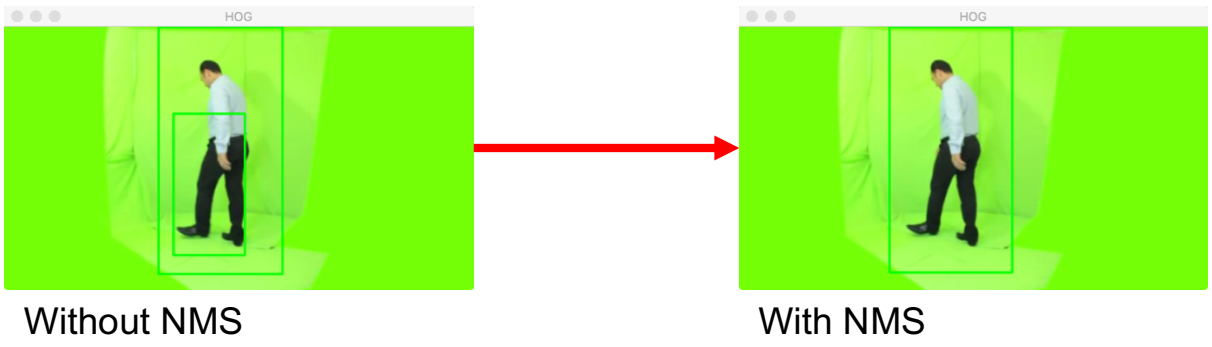

Figure 8.8 - Showing difference before and after NMS implementation

8.4. Kalman Filter

8.4.1. Introduction

The Kalman Filter is a point tracking predictive tracker. It is known as an optimal estimator due to its assumption that the point being monitored is random, and Gaussian distributed. In this project it was implemented as a 2D tracker, predicting the movement of the centre pixel of the bounding box surrounding the pedestrian. To capture the anticipated movement, we take what we can measure, in this case, the current state, to generate a result for something we cannot measure, the predicted state.

$$x_k = \begin{bmatrix} Position \\ Velocity \end{bmatrix}$$

Taking Position and velocity as our Gaussian distributed functions, we can first make an initial estimate of where the object is going to be [A]. Then we take an actual measurement, with noise included [B]. Following these two measurements, we can then get an optimised best estimate of the location of the object [C]. The actual measurement is taken, and the known location is used to correct the predicted state [D]. This process repeats itself cyclically to give the best estimate of our object. [Figure 8.9]

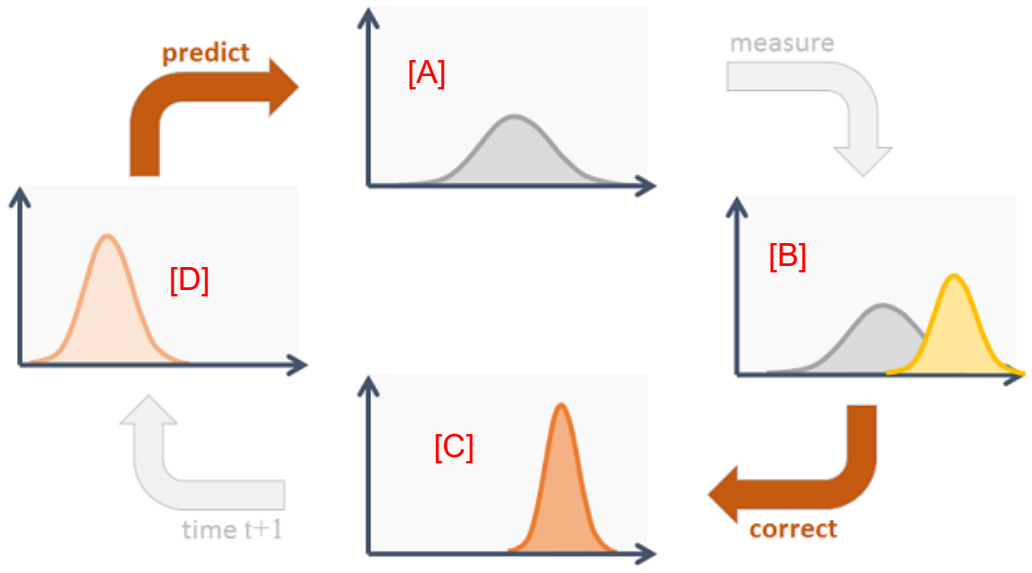

Figure 8.9 - Flow graph of Kalman Filter [25]

8.4.2. Implementation

The Kalman filter was implemented with a dependent relationship to the HOG detector. The functions would only be called after an object had been identified, this was done to avoid unnecessary code execution if no object has been detected to track. We begin by initializing the data stores, to hold the predicted and measured (x, y) value of the tracked object. meas and pred are created [Figure 8.10] to store all values calculated from the Kalman function, the tp and mp arrays store the real-time tracking locations generated by the prediction function and the measurement function.

```
#initliase our matrices for storing Kalman values
meas = []   #all collected measured values
pred = []   #all collected predicted values
mp = np.array((2, 1), np.float32)  # measurement
tp = np.zeros((2, 1), np.float32)  # tracked / prediction
#intialzing variables for storing clock times
totalHOG = 0
totalKAL = 0
```

Figure 8.10 - Extract of code showing initialization of data stores

```
kalman = cv2.KalmanFilter(4, 2)
kalman.measurementMatrix = np.array([[1, 0, 0, 0], [0, 1, 0, 0]], np.float32)
kalman.transitionMatrix = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0], [0, 0, 0, 1]], np.float32)
kalman.processNoiseCov = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]], np.float32) * 0.03
```

Figure 8.11 - Extract of code showing initialization Kalman matrices

Following this, we initialize the Kalman filter function provided by OpenCV and assign it to a variable. This variable is then called to set up the measurement matrix; this initialises our function to work in a 2D space, recording the (x, y) location of our pixel. Similarly we set up the transition matrix, used to initialise the function predicted transition to the next state, taking into account (x, y) location and velocity in the x direction and y direction. Finally the process noise matrix is initialised for position and velocity, to take into account noise affecting the predicted location.[Figure 8.11]

```
for (xA, yA, xB, yB) in pick:
    cv2.rectangle(frame, (xA, yA), (xB, yB), (0, 255, 0), 2)
    centerX = (xB + xA) / 2
    centerY = (yB + yA) / 2
    #collect the center value and pass this into the kalman mesaured function
    onPed(centerX, centerY)
    #updatekalmanwith the measured point
    updateKalman(mp)
    #the returned predicted co-ordinate is passed into the paint function
    paint(tp, xA, yA, xB, yB)
```

Figure 8.12 - Extract of code showing the calling of Kalman methods

The Kalman filter is called whenever a bounding box is created by the HOG detector as seen in Figure 8.12, where the centre pixel of the bounding box is calculated using simple arithmetic. This (x, y) value is passed into the real time array we have created called mp for passing into the prediction function, and appended to the end of the measured matrix for storing. This function returns the real time measured value, which is passed into the KalPredict() function. Here the measured value is passed into the Kalman predict function, which returns tp as seen in Figure 8.13, the predicted transition (x, y) location.

```
def onPed(x, y):#on detection of pedestrian we pass in the x,y co-ordinates
    global mp, meas
    mp = np.array([[np.float32(x)], [np.float32(y)]])
    meas.append((x, y))

def updateKalman(mp):   #updateKalman as measured is passed into kalman
    global pred, tp     #the predicted kalman is then returned , the rectangle is drawn
    kalman.correct(mp)  #aroundthis  point
    tp = kalman.predict()
    pred.append((int(tp[0]), int(tp[1])))
```

Figure 8.13 - Extract of code showing the Kalman measure method and predict method

Finally, the paint function is called which paints a rectangle around the returned transition (x, y). The end user is now shown the detection rectangle in green and the Kalman predicted movement by the red bounding box. It can be noted that the movement of the pedestrian is now predicted by the red bounding box, moving

slightly ahead of the detection bounding box as seen in Figure 8.14 below marked by the red arrow showing trajectory.

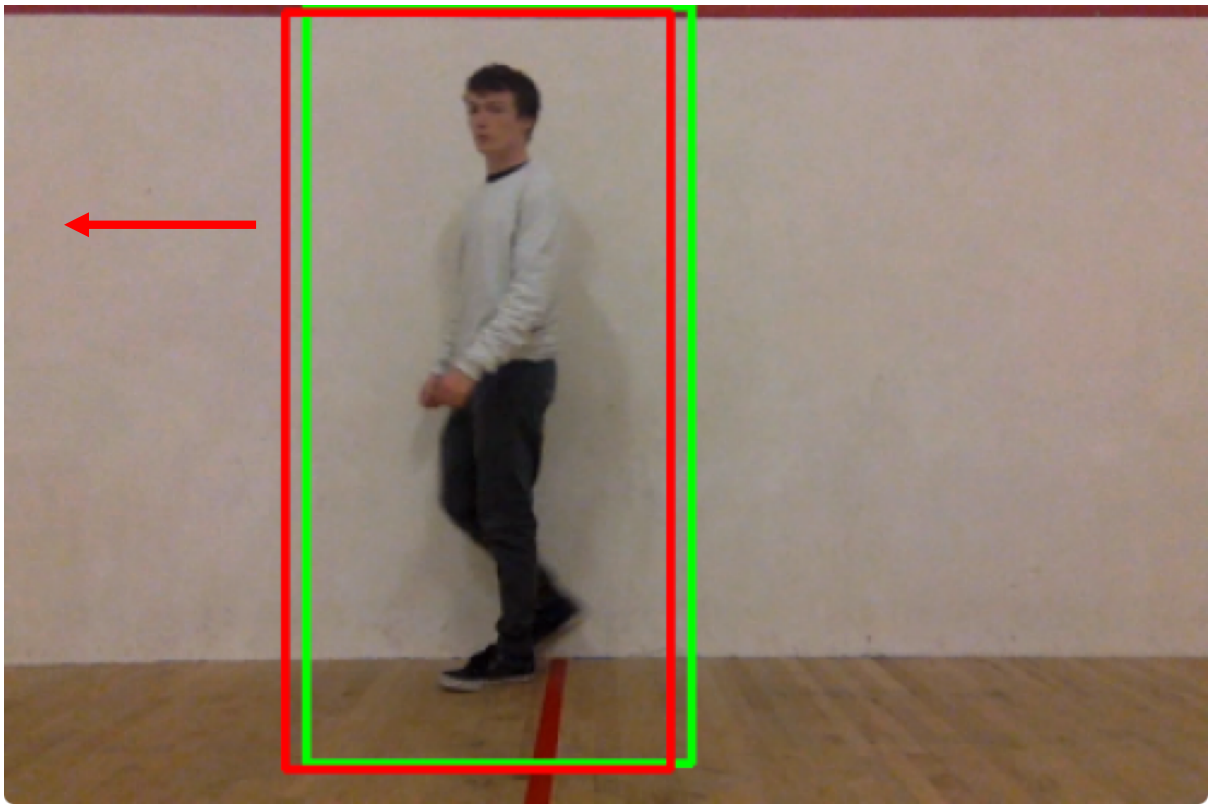

Figure 8.14 - Screenshot showing the Kalman bounding box moving ahead of detection bounding box, predicting trajectory

The displayed frame would be re-sized after the memory intensive functions had completed. By re-sizing, the user views a larger window of the live tracking system without causing a large bottleneck during detection and tracking. This process would be looped continuously until the user presses ‘Q’ on the keyboard, where the loop is broken, and the process ends.

```
k = cv2.waitKey(1) & 0xff
#if Q is pressed quit the loop of frames
if k == ord("q"):
    break
```

Figure 8.15 - Extract of code showing the break from the loop

8.5. System Flow

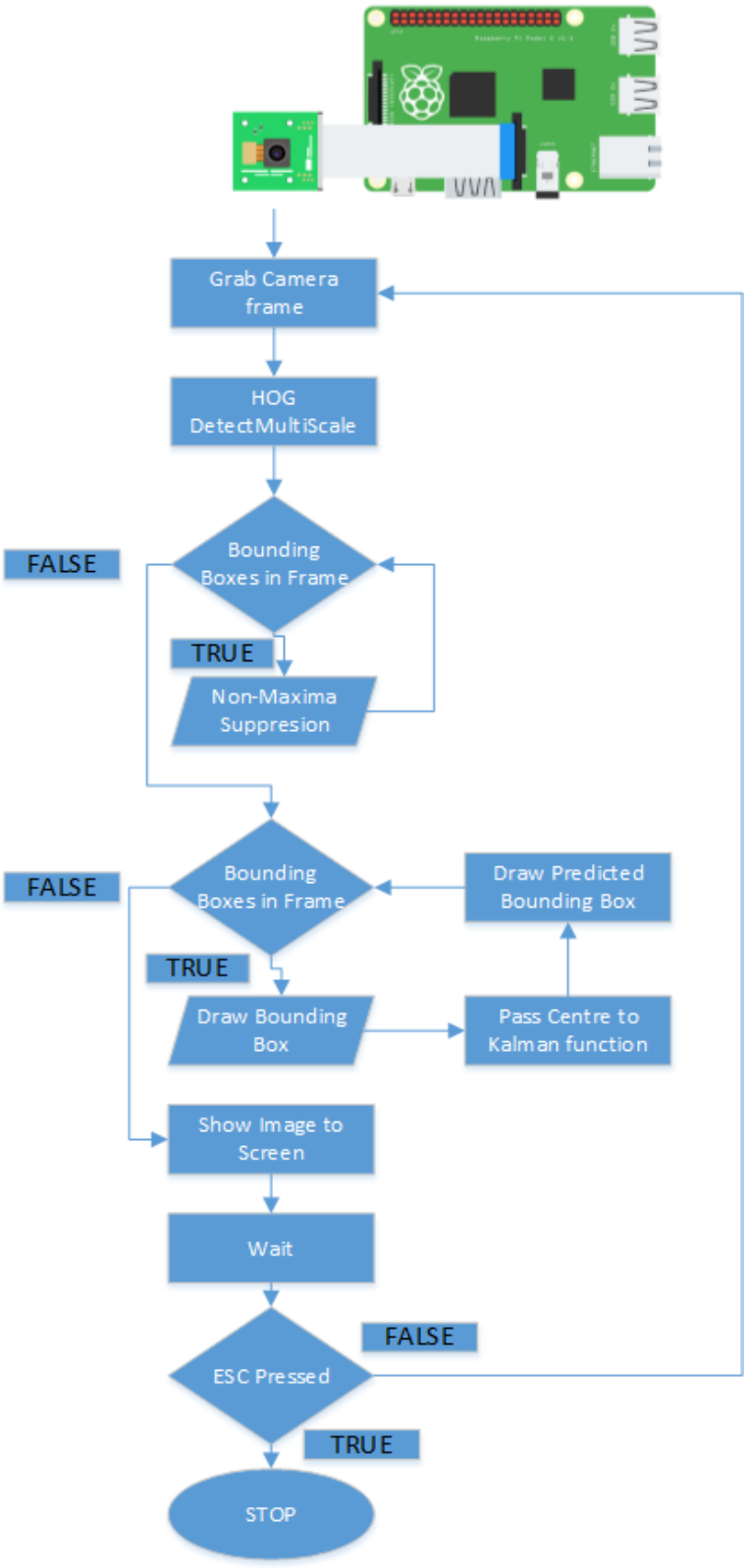

Figure 8.16 - System Flow of object tracking engine

Chapter 9. Testing

9.1. Introduction

This Section details the testing that was undertaken on the implemented object tracking engine to determine the best parameters to decrease processing time and maximise accuracy.

9.2. Timer

To measure processing speed various timers were placed throughout the code. Using OpenCV get tick count function the time taken to complete a task was logged and printed to screen, these values were transferred to a CSV file for analysis.

```
e1 = cv2.getTickCount()
 #run detection with HOG over frame (specified parameters best tested
(rects, weights) = hog.detectMultiScale(frame, winStride=(8, 8),
                                        padding=(16, 16), scale=1.05)
e2 = cv2.getTickCount()
timeHOG = (e2-e1)/cv2.getTickFrequency()
```

Figure 9.1 - Extract of code showing the timer surrounding the HOG detect multiscale function

9.3. Frame Resolution

Table 9.1 - Resolution Test Case

Test Conditions	
Parameters	detectMultiScale(winStride=(8, 8),padding=(16, 16), scale=1.05)
Hardware	Raspberry Pi
Frames	100
Source	Live Video
Resolution	Variable

Increasing the processing time of the main bottleneck in the object tracking pipeline, the HOG detection, could be achieved through the adaption of the resolution of the incoming frame. By decreasing the number of pixels for the sliding window to analyse we can decrease the processing time. This reduction in pixels is proportional to the decrease, as ¾ of the original frame size has 25% fewer pixels, leading to a reduction in processing time of 25%. [12] Finding the optimum size giving faster run time, with a high accuracy for detection was the purpose of this test.

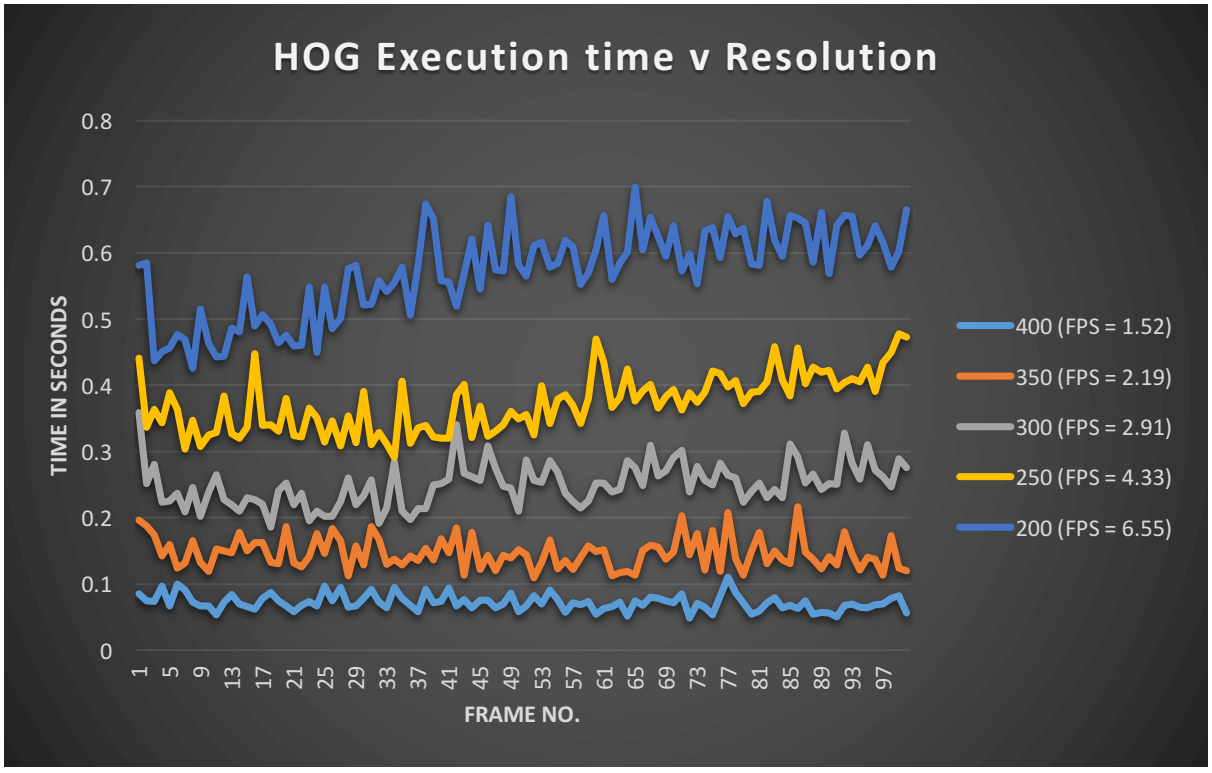

Chart 9.1 - Showing the difference in processing times depending on frame resolution

Chart 9.1 above shows the large differences in time between using a frame with a resolution of 200 vs. the resolution of 400, with an average FPS difference of 5.03. The trade off however for such a fast FPS with a resolution at a width of 200 is the inability for the sliding window to detect any pedestrians. This is the trade-off between accuracy and throughput. The best performing resolution was a width of 300, providing an acceptable FPS of 2.91 and not sacrificing accuracy in the process.

To test performance, a simple counter was used to count each bounding box present in each frame. By counting the frames on a test video with one person present in every frame, we can calculate the accuracy regarding detections[Figure 9.2]. The

performance for each resolution was recorded and graphed as seen in Chart 9.2 below. The ideal threshold regarding detections made can be seen to be 300 frames, which correlates with our maximal FPS rate as well.

```
Performance = (float(detection)/(frames))*100
if Performance >= 100:
        Performance = 100
```

Figure 9.2 - Extract of code showing the performance measure used

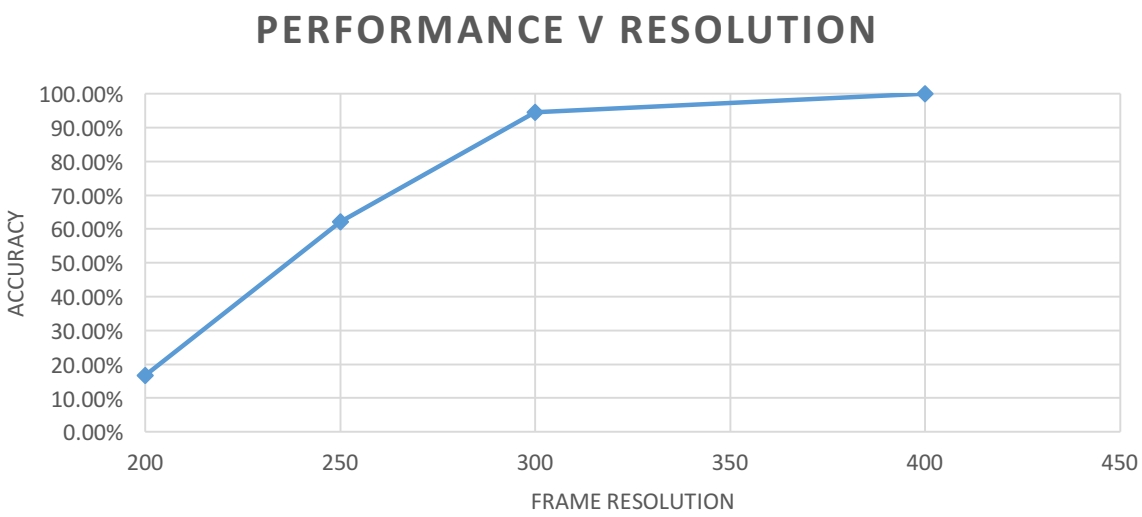

Chart 9.2 - Performance v Frame Resolution

The general increase in average processing time shown in Chart 9.1 by the graphed lines can be attributed to the temperature increase in the RPi due to the process using 100% of the CPU during execution. This overheating was reduced with an external fan below the heat sinks on the RPi motherboard.

9.4. Scale

Table 9.2 - Scale Test Case

Test Conditions	
Parameters	detectMultiScale(winStride=(8, 8),padding=(16, 16), scale=Variable)
Hardware	MacBook
Frames	300
Source	Video Sequence [24]
Resolution	600

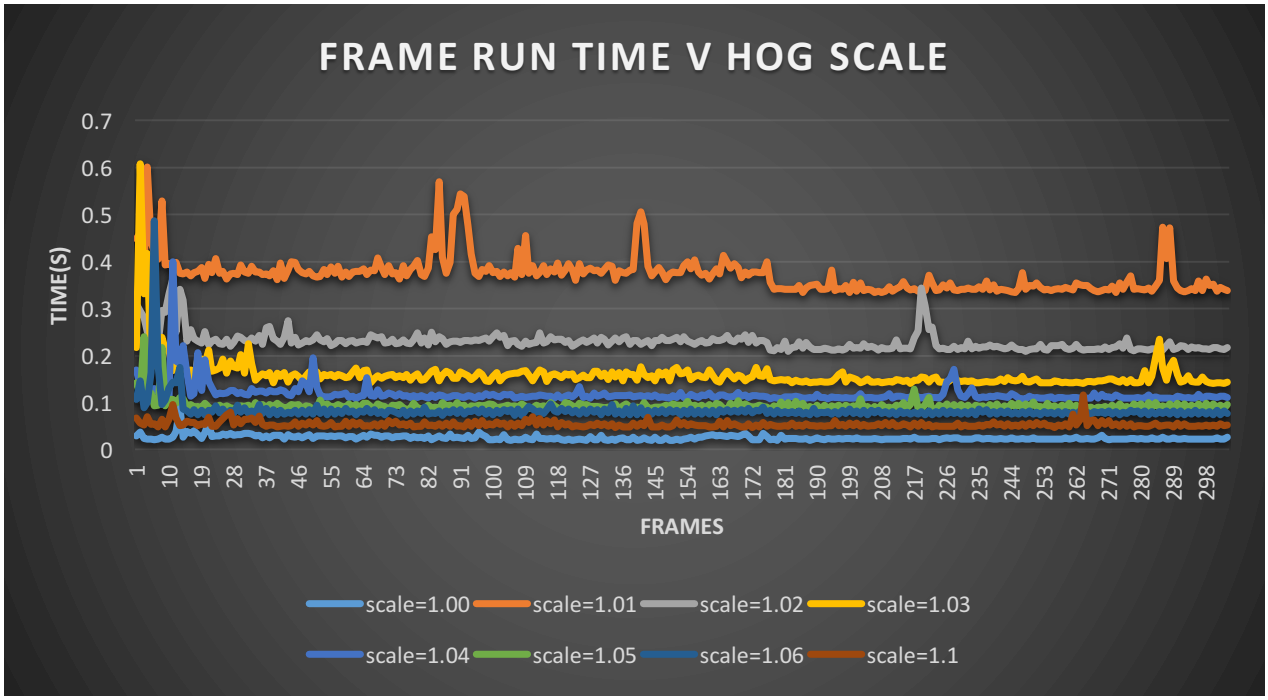

Chart 9.3 - Showing the change run time per frame as scale is changed

The scale parameter is very influential for the Frame per second(FPS) processed. It determines the size of the image pyramid created, therefore increasing the length of time spent scanning the incoming frame. By decreasing the scale, we increase the number of scaled-down images that the sliding window runs over. The reason for the reduction in scale is to allow the HOG classifier to find pedestrians that are far away within the initial frame, and would otherwise be missed. We can see from Chart 9.3 above as we decrease the scale parameter the overall FPS increases. As we can clearly see from the chart, FPS increases as we approach a large-scale parameter,

but with this accuracy decreases. The most suitable parameter value for scale was determined to be 1.05 due to the average Processing time of 0.0937 seconds and its overall ability to accurately detect pedestrians.

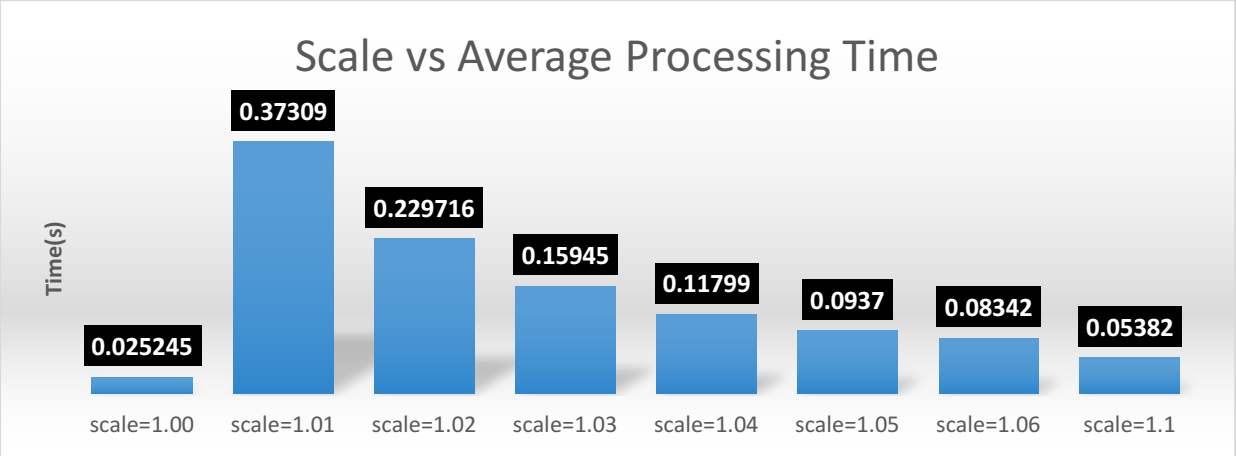

Chart 9.4 - Showing Average processing time vs. Scale

9.5. Window Stride

Table 9.3 - Window Stride Test Case

Test Conditions	
Parameters	detectMultiScale(winStride=(variable),padding=(16, 16), scale=1.05)
Hardware	MacBook
Frames	300
Source	Video Sequence [24]
Resolution	600

The second most influential parameter on the processing time of the HOG function is window stride. Window stride determines the size of the sliding window used to extract the HOG descriptors from each frame. A larger window stride makes for faster processing of the incoming frame but can lead to a loss in accuracy due to the histograms of gradients being over a larger subset of pixels. A smaller window stride can increase the accuracy of the detector but can create a large bottleneck in the processing pipeline due to the same area of pixels being scanned by a smaller window.

Having tested three different window sizes, (4,4), (8,8) and (16,16) it could be conferred that the optimal window stride was (8,8) due to its efficient processing time coupled with a strong performance on the detection of pedestrians. (4,4) Window stride was omitted from the below Chart 9.5 due to its time being such an outlier and skewing the data.

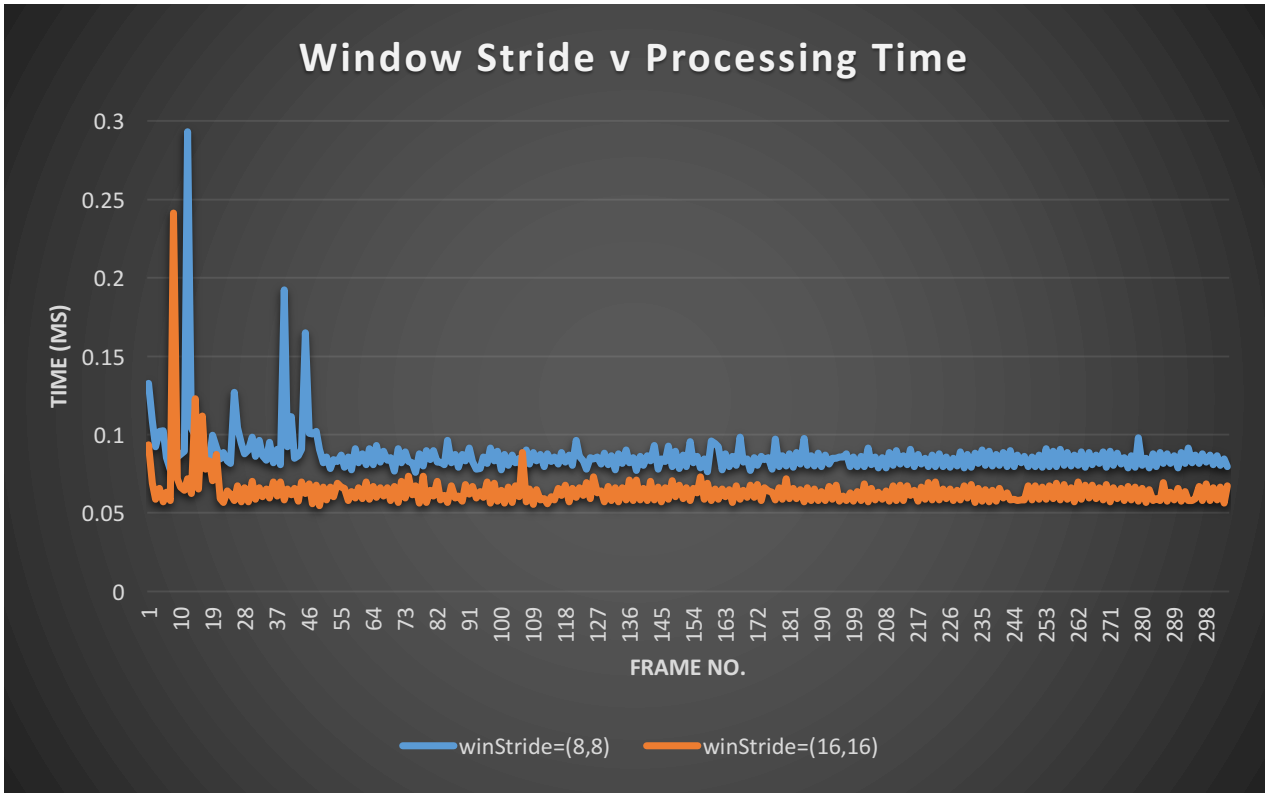

Chart 9.5 - Showing the change in processing time from different window strides

9.6. Padding

Table 9.4 - Padding Test Case

Test Conditions	
Parameters	detectMultiScale(winStride=(8,8),padding=(variable), scale=1.05)
Hardware	MacBook
Frames	300
Source	Video Sequence [24]
Resolution	600

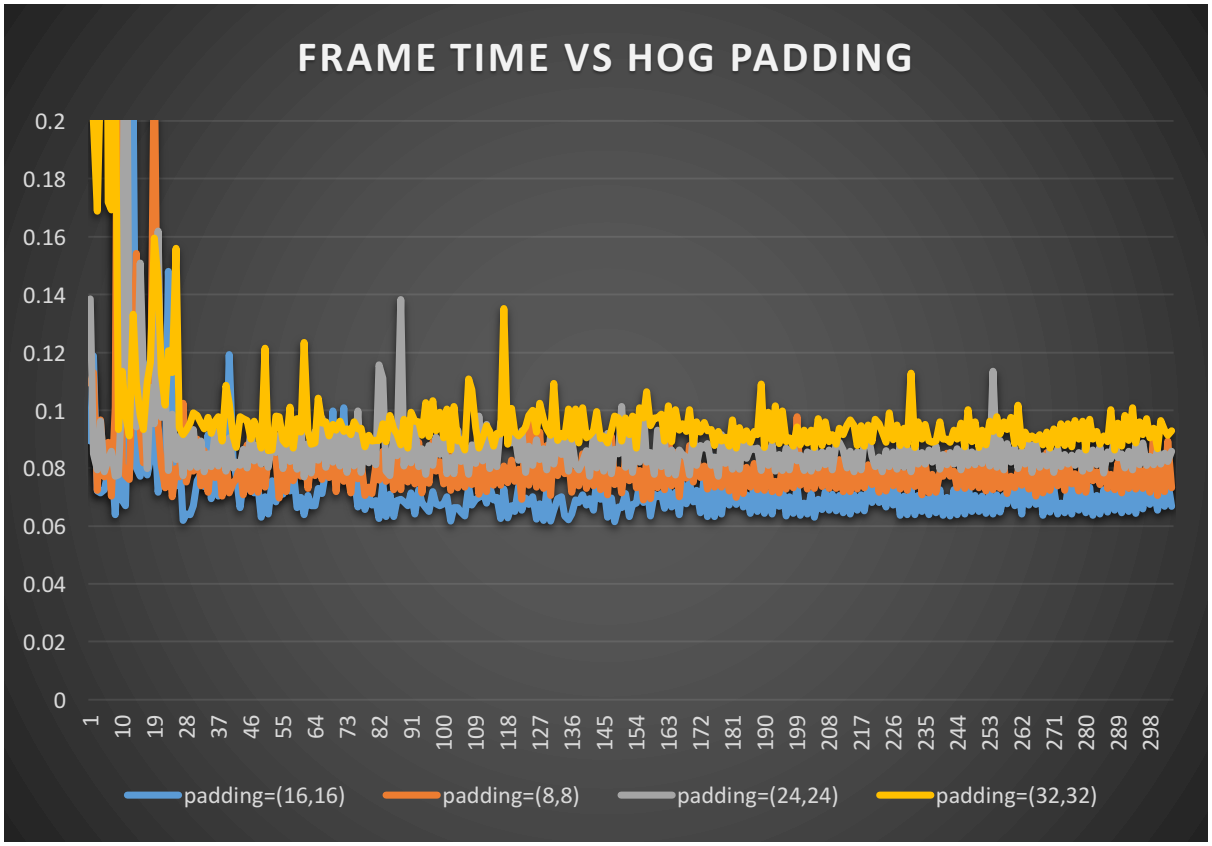

Chart 9.6 - Showing the effect Padding size has on HOG processing time

The padding parameters changes how much buffering pixels in the x and y direction are used for the sliding window. From Chart 9.6 above we can see that overall the change in processing time isn't that great with the average time between the fastest setting (16,16) and slowest setting (32,32) being 0.027 seconds as seen below in Table 9.5. For this reason, a padding size of (16,16) was chosen due to its speed and ability to detect a pedestrian on the edge of the frame, increasing accuracy.

Table 9.5 - Average processing times in seconds of different padding sizes

Frames	padding=(8,8)	padding=(16,16)	padding=(24,24)	padding=(32,32)
304	0.080440727	0.071605954	0.087047368	0.09864948

9.7. Analysis

From the testing carried out it was determined the best parameter values to set for the HOG detection function where a window stride of (8,8), a padding size of (16,16) with a scale of 1.05. For the resolution of the input frame, a width of 300 was chosen to allow for a maximum efficiency and accuracy while not deteriorating the user's display too much. To allow the user a larger screen to view before calling the image show function another re-size was carried out to create a larger output screen after the memory intensive filters had been conducted. [Table 9.6]

Table 9.6 - Testing Results

Testing Analysis		
Parameter	**Input**	**Reference**
Scale	1.05	Chart 9.4
WinStride	(8,8)	Chart 9.5
Padding	(16,16)	Chart 9.6
Source	Live Video Stream	-
Resolution	300	Chart 9.1
Hardware	Raspberry Pi w/ PiCamera	-
Expected Performance	~90%	Chart 9.2

Chapter 10. Conclusions

10.1. Introduction

This section is a discussion on what was set out for the project, and what was achieved. Any Future work that could be carried out is outlined here also.

10.2. Work Summary

All work was done with the aim of meeting the specifications set out to develop an object tracking engine for live video. Each specification leads to its own area of research, implementation, and testing, together building an entire project.
This begun with selecting various techniques of detection and tracking and the methods needed to implement this. This was undertaken through online paper resources and books, where variants in the process where noted and researched further. In the end, a combination of detection and tracking methodologies were used to reach the goal. These processes were built into the chosen architecture style, where each sub-system was selected and implemented. The pattern matching technique of utilising Histogram of orientated gradients was coupled with the predictive tracking Kalman filter. The Raspberry Pi was then used as the embedded system for the project, where live tracking was done using the PiCamera.
Testing was carried out for the core areas of the system, Processing time and how pre-processing of the image can improve the speed of the tracker, and an evaluation methodology was developed to measure the accuracy of each performance. Both of these required testing and analysis of playback and live video and improved the end result.
The project specifications were met leading to a working object tracking system on an embedded device. There are areas that improvements could be made to make the overall system more effective and efficient. These are detailed in section 10.3 below.

10.3. Future Work

10.3.1. Hungarian Algorithm

The implementation of the Kalman filter predictive tracking was unable to effectively track whenever there was more than one object of interest within the frame. This was due to the storing of the centre points of bounding boxes within an array for passing into the Kalman functions. Two or more bounding boxes lead to conflicting (x, y) co-ordinates being passed into prediction and measurement functions. This skewed results and lead too the Kalman bounding boxes being drawn in incorrect areas, not enclosing the object of interest.

A method for implementing the Kalman filter on multiple objects involves utilising the Hungarian algorithm. It is an optimisation algorithm which would allow for the points of least distance from the current frame being compared with locations of objects of interest in previous frames. This builds a tracking map, reducing the new points to their most likely last position, and assigning them to that particular object of interest. By using this method, we can implement predictive tracking methods on multiple objects

10.3.2. Real-time Video

The system was run on a live video stream from a PiCamera. However, real-time video acquisition was never achieved. This was due to the memory intensive HOG detection sub-system causing a bottleneck in the pipeline. For future work methods of improving the frame rate and throughput of the system could be researched. One such method that could be implemented is an architecture that allows for threaded processes separating the memory intensive functions from the faster processes.

10.3.3. Night Vision

The camera used was a PiCamera NoIR, a camera with no infrared filter. Using infrared lights to illuminate dark environments, improvements to the system could be made to allow for night time testing of an object tracker and analysis. This would

open the scope for pedestrian detection and tracking in poorly light scenes, an important area of research for implementation of the system in automotive and security systems.

10.4. **Concluding Remarks**

Pedestrian detection and tracking is a growing area of interest for many companies such as Facebook, Google, Apple and in the automotive industry where millions are spent on research in computer vision. This project aimed to implement a system on a small, cheap and easily obtainable device, to show that research in this area is no longer limited to research institutes and industries with significant investment. Real world problems can be solved with this technology regarding safety, automation and security and for it to be done within the criteria of this project shows the leaps and bounds the area has made in the past two decades.

Chapter 11. **Bibliography**

[1] A. Yilmaz, O. Javed and M. Shah, “Object tracking:A Survey,” *ACM Computing Surveys,* vol. 38, no. 4, pp. 13-es, 2006.

[2] “Human error as a cause of vehicle crashes,” 2017. [Online]. Available: http://cyberlaw.stanford.edu/blog/2013/12/human-error-cause-vehicle-crashes. [Accessed 07 April 2017].

[3] “Computer Vision Technologies and Markets,” Tractica, Boulder, CO, 2016.

[4] G. R. Bradski and A. Kaehler, Learning OpenCV, 1st Edition ed., Farnham: O'Reilly, 2013.

[5] OpenCV, “About - OpenCV library,” 2017. [Online]. Available: http://opencv.org/about.html. [Accessed 3 April 2017].

[6] C. Tomasi and J. Shi, “Good features to track,” *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94,* vol. X, no. X, pp. 593-600, 1994.

[7] T. K, K. J, B. B and M. B, “Wallflower: principles and practice of background maintenance,” *Proceedings of the Seventh IEEE International Conference on Computer Vision,* 1999.

[8] P. Viola, M. J. Jones and D. Snow, “Detecting Pedestrians Using Patterns of Motion and Appearance,” *International Journal of Computer Vision,* vol. 63, no. 2, pp. 153-161, 2005.

[9] B. Triggs, N. Dalal and C. Schmid, “Human Detection Using Oriented Histograms of Flow and Appearance,” *Computer Vision – ECCV 2006,* pp. 428-441, 2006.

[10] N. Dalal and B. Triggs, “Histograms of Oriented Gradients for Human Detection,” *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05),* 2005.

[11] T. J. Broida and R. Chellappa, “Estimation of Object Motion Parameters from Noisy Images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vols. PAMI-8, no. 1, pp. 90-99.

[12] P. Corcoran, E. Stienberg, S. Petrescu, A. Drimbarean, F. Nanu, A. Pososin and P. Biglol, “Real-time face tracking in a digital image acquisition device”. U.S Patent 7315631, 01 Jan 2008.

[13] “Learning to Segment,” 2017. [Online]. Available: https://research.fb.com/learning-to-segment/. [Accessed 03 April 2017].

[14] “Automotive - FotoNation,” 2017. [Online]. Available: https://www.fotonation.com/products/automotive/. [Accessed 03 April 2017].

[15] “Electronic & Production Engineers in Galway - Valeo Vision,” 2017. [Online]. Available: http://www.valeovision.com/. [Accessed 03 April 2017].

[16] “our technology,” 2017. [Online]. Available: http://www.mobileye.com/our-technology/. [Accessed 03 April 2017].

[17] “Camera Module - Raspberry Pi Documentation,” 2017. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/README.md. [Accessed 03 April 2017].

[18] “how to build a picamera,” 2017. [Online]. Available: https://www.allaboutcircuits.com/projects/how-to-build-a-picamera/. [Accessed 03 April 2017].

[19] A. Rosebrock, “PyImageSearch,” [Online]. Available: http://www.pyimagesearch.com/.

[20] A. R. Francois, “Software Architecture for Computer Vision: Beyond Pipes and Filters,” UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES INST FOR ROBOTICS AND INTELLIGENT SYSTEMS, Los Angeles, JUL 2003.

[21] “jrosebr1/imutils,” 2017. [Online]. Available: https://github.com/jrosebr1/imutils. [Accessed 04 April 2017].

[22] C. Papageorgiou, M. Oren and T. Poggio, “A general framework for object detection,” *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271).*

[23] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001,* 2001.

[24] F. S. Footage, *Man standing, walking in front modern green screen environment. Free stock footage,* 2017.

[25] D. Jurić, “Object Tracking: Kalman Filter with Ease - CodeProject,” 2017. [Online]. Available: https://www.codeproject.com/Articles/865935/Object-Tracking-Kalman-Filter-with-Ease. [Accessed 10 April 2017].

[26] “Free video,” 2017. [Online]. Available: https://pixabay.com/en/videos/fishing-fisherman-rod-fishing-rod-1022/. [Accessed 04 April 2017].

[27] A. Reimondo, “Haar Cascades,” 2017. [Online]. Available: http://alereimondo.no-ip.org/OpenCV/34. [Accessed 05 April 2017].

Appendix

The Following Code is the implemented object tracking engine on the Raspberry Pi:

```
from imutils.video.pivideostream import PiVideoStream
from non_max_suppression import non_max_suppression
from picamera.array import PiRGBArray
from imutils.video import FPS
from picamera import PiCamera
import numpy as np
import datetime
import argparse
import warnings
import imutils
import time
import cv2
import json


#get path to JSON file
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--conf", required=True,help="path to the JSON configuration file")
args = vars(ap.parse_args())
#load JSON file from path
warnings.filterwarnings("ignore")
conf = json.load(open(args["conf"]))
client = None
#initliase our matrices for storing Kalman values
meas = []   #all collected measured values
pred = []   #all collected predicted values
mp = np.array((2, 1), np.float32)  # measurement
tp = np.zeros((2, 1), np.float32)  # tracked / prediction
#intialzing variables for storing clock times
totalHOG = 0
totalKAL = 0
#initiliase performance variables for counting frames
Detection=0
Frames=0
#assign the descriptor to its variable & call the people Detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
#initialising Kalman filter and assign the measurement,transition and noise values
kalman = cv2.KalmanFilter(4, 2)
kalman.measurementMatrix = np.array([[1, 0, 0, 0], [0, 1, 0, 0]], np.float32)
kalman.transitionMatrix = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0], [0, 0, 0,
1]], np.float32)
kalman.processNoiseCov = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0,
1]], np.float32) * 0.03
```

```
def onPed(x, y):#on detection of pedestrian we pass in the x,y co-ordinates
    global mp, meas
    mp = np.array([[np.float32(x)], [np.float32(y)]])
    meas.append((x, y))


def updateKalman(mp):   #updateKalman as measured is passed into kalman
    global pred, tp     #the predicted kalman is then returned , the rectangle is drawn
    kalman.correct(mp)  #aroundthis  point
    tp = kalman.predict()
    pred.append((int(tp[0]), int(tp[1])))


def paint(tp, xA, yA, xB, yB):#draw rectangle for predicted state
    global frame, pred
    cv2.circle(frame, ((tp[0]), (tp[1])), 3, (0, 0, 255), -1)
    cv2.rectangle(frame, ((tp[0]) - ((xB - xA) / 2), (tp[1]) + (yB - yA) / 2),
                  (((tp[0]) + ((xB - xA) / 2)), ((tp[1]) - (yB - yA) / 2)), (0, 0, 255),
2)

#start the stream , wait for 2 seconds for the camera to warm up
vs = PiVideoStream().start()
print("[INFO] warming up...")
time.sleep(2.0)
fps = FPS().start()
avg = None


#perform this loop until the user exits
while(True):
        frames = frames+1
        #grab the frame from the top of the queue
        frame = vs.read()
        #resize to 300
        frame = imutils.resize(frame, width=300)
        frame = cv2.flip(frame,0)
        # detect people in the image
        e1 = cv2.getTickCount()
         #run detection with HOG over frame (specified parameters best tested
        (rects, weights) = hog.detectMultiScale(frame, winStride=(8, 8),
                                            padding=(16, 16), scale=1.05)
        e2 = cv2.getTickCount()
        timeHOG = (e2-e1)/cv2.getTickFrequency()
        #the HOG returns rectangles, for every rectangle
        #carry out non_max_suppresion
```

```
        rects = np.array([[x, y, x + w, y + h] for (x, y, w, h) in rects])
        pick = non_max_suppression(rects, probs=None, overlapThresh=0.65)


        #loop over every returned co-ordinate and draw the bounding box
        f1 = cv2.getTickCount()
        for (xA, yA, xB, yB) in pick:
            cv2.rectangle(frame, (xA, yA), (xB, yB), (0, 255, 0), 2)
            centerX = (xB + xA) / 2
            centerY = (yB + yA) / 2
            #collect the center value and pass this into the kalman mesaured function
            onPed(centerX, centerY)
            #updatekalmanwith the measured point
            updateKalman(mp)
            #the returned predicted co-ordinate is passed into the paint function
            paint(tp, xA, yA, xB, yB)
        f2 = cv2.getTickCount()
        timeKAL = (f2-f1)/cv2.getTickFrequency()
        # check to see if the frames should be displayed to screen
        output = imutils.resize(frame, width=600)
        #resize frame to make it easier to view
        print_HOG_time = "[INFO] HOG RUN TIME: "+ str(timeHOG)
        cv2.putText(output, print_HOG_time, (10,55), cv2.FONT_HERSHEY_SIMPLEX, 0.6,
(0,255,0), 2)
        cv2.namedWindow("HOG", cv2.WND_PROP_FULLSCREEN)
        cv2.setWindowProperty("HOG",cv2.WND_PROP_FULLSCREEN,1)
        cv2.imshow("HOG", output)
        k = cv2.waitKey(1) & 0xff
        #if Q is pressed quit the loop of frames
        if k == ord("q"):
            break
        fps.update()
        #print ("{}".format(timeHOG))
        totalKAL= timeKAL+totalKAL
        totalHOG= timeHOG+totalHOG
#print all stored Data for analysis
totalAvgHOG = totalHOG/fps._numFrames
totalAvgKAL = totalKAL/fps._numFrames
fps.stop()
Performance = (float(detection)/(frames))*100
if Performance >= 100:
        Performance = 100
print("INFO elapsed time: {:.2f}".format(fps.elapsed()))
print("INFO approx FPS {:.2f}".format(fps.fps()))
print("INFO Performance: {:.2f}".format(Performance))
print("INFO Average time to complete HOG: {:.2f} Seconds".format(totalAvgHOG))

print("INFO Average time to complete NMS: {:.2f} Seconds".format(totalAvgKAL))
cv2.destroyAllWindows()
vs.stop()
```