FEEG3003 Individual Project

A Vision Based Approach to Indoor Localisation using Raspberry Pi

Final Report

Author: Roshan Pasupathy

Supervisors: Professor Simon Cox, Dr. Steven Johnston

This report is submitted in partial fulfilment of the requirements for the Masters of Mechanical Engineering, Faculty of Engineering and the Environment, University of Southampton.

Date: 20th April 2016

Word Count: 10,043

Declaration

I, Roshan Pasupathy declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I confirm that:

1. This work was done wholly or mainly while in candidature for a degree at this University;
2. Where any part of this thesis has previously been submitted for any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission

Abstract

Localisation is the estimation of the three dimensional position of an object of interest. Precise coordinates of an object in an indoor environment are useful in a large number of engineering and commercial applications. Over the years various indoor localisation technologies have been investigated - Bluetooth, ultrasound, infra-red, vision based systems. Among these technologies, vision based systems standout as they are capable of providing much higher localisation accuracies.

Vision systems are limited by computational costs associated with image processing. This limitation is usually overcome by compensating on accuracy or increasing the cost of the device. This report details the design methodology and testing of a vision based indoor localisation system which aims to overcome this performance bottleneck by using software which exploits the capabilities of the Raspberry Pi without compromising on accuracy and cost.

Two cameras, each connected to a Beacon system, were used to localise a light emitting diode (LED) and a laser dot with an error of less than 3 cm and 5 cm respectively at a range of 5m. Increasing the number of cameras and implementing mathematical refinement algorithms can help improve the accuracy of the system.

The refresh rate of the system was limited by the camera frame rate (30 frames/second) and not the detection process. Thus the image processing overhead was overcome and localisation frequencies of above 2500 Hz were achieved. Use of newer revisions of the Raspberry Pi in future iterations of the system will greatly enhance performance.

The accuracy and speed of the proposed system makes it suitable for implementation in navigation systems for autonomous robots and as a 3D pointer device. As the first iteration of the device, the proposed system serves as a great foundation for future development in terms of technical specifications and applicability in various fields.

Acknowledgements

I would like to thank Dr. Steven Johnston and Professor Simon Cox for their continual guidance and supervision throughout the course of this project. The projects outcome was greatly influenced by their valuable input.

Table of Contents

List of Abbreviations

3D	3-Dimensional
LED	Light Emitting Diode
SIFT	Scale Invariant Feature Transform
ARM	Advanced Reduced instruction set computing Machine
RGB	Red, Blue, Green
HSV	Hue, Saturation, Value
LUT	Lookup Table
USB	Universal Serial Bus
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
I^2C	Inter-Integrated Circuit
LAN	Local Area Network
IP	Internet Protocol
SSH	Secure Shell

A Vision Based Approach to Indoor Localisation using Raspberry Pi

1. Introduction

"Accurate Indoor localisation has the potential to transform the way people navigate indoors, the same way GPS transformed the way people navigate outdoors"
- Microsoft Indoor Localisation Competition (April 13-17 2015)

Localisation is the estimation of the three dimensional position of an object of interest. Indoor localisation has been an objective of the ubiquitous computing research community due to its importance to several fields and industries such as robotics, gaming, human-computer interactions [1-4].

Numerous technologies and approaches to localisation have been investigated in the past 15 years. These include magnetic positioning, Bluetooth, ultrasound and vision based systems. A major consideration while deciding upon a technology is the tradeoff between range of the system and accuracy. This report presents the design methodology and implementation of a vision system which uses the pixel position of the object in the image planes of two cameras, each connected to a Raspberry Pi 2 model B, to estimate the object's three dimensional position with refresh rates of above 25 Hz.

1.1. Relevant Research and Devices

Despite the large-scale interest in indoor localisation, vision based systems have not enjoyed the same popularity as other technologies. The primary reason for this is the limitations of vision systems with respect to range and occlusion. Inertial sensors and signal strength based approaches are better suited in situations where occlusion is a prominent problem. However, vision systems do have a significant advantage over other technologies – high localisation accuracy is achievable [5].

Svedman et al. designed a system for reconstruction of a 3D object using two unsynchronised cameras (shutter timings are not synchronised). The scale invariant feature transform (SIFT) algorithm was used for matching points of interest located on the object (approximately 200 feature points) in the captured images. A median accuracy of 1.6 cm was achieved when the average depth was 3 m [6]. Although the underlying concept is similar to the system proposed in this report, accurate 3D reconstruction and SIFT based feature matching are computationally expensive and unfeasible on the Raspberry Pi if high speed localisation is required. Fujiyoshi et al. proposed a similar system specifically for localisation of objects with refresh rates of approximately 30 Hz [7]. Epipolar geometry was used to estimate the 3D position of a red ball, moving in a plane 50 cm away from the plane of the cameras, with an accuracy of 3 cm [8].

The Xbox Kinect, PlayStation Move and Nintendo Wii are examples of commercially available motion positioning systems developed for the gaming industry. The Nintendo Wii remote functions as a pointer device by combining data available from the on-board accelerometers and infra-red camera. The PlayStation Move tracks the movement of a controller in 3D space by detecting an LED illuminated ball mounted on the controller. A single camera is used and the change is depth is estimated by the corresponding change of size of the ball in the image. Similar to the Wii remote, the PlayStation move uses inertial sensors to better estimate the change in position of the controller. The Xbox Kinect creates a depth map (Figure 1(a)) and estimates the position of the joints of the body based on a decision tree (Figure 1(b)). From Figure 1, it is also clear that the Xbox Kinect is primarily used to infer body positions and not to localise objects. However, due to the fact that these commercial systems rely on depth information for tracking movement, it is possible to adapt them to localise objects. Gu et al. and Lopez et al. successfully implemented the Wii remote in an indoor navigation system for autonomous robots [9-10].

Figure 1: (a) Depth map generated by Xbox Kinect (left) and (b) Joint positions inferred by the Kinect (right)[11].

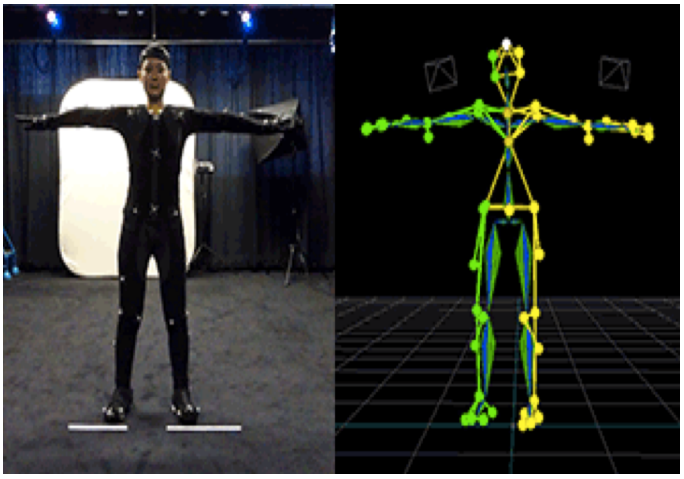

Figure 2: A test participant wearing a suit fitted with reflective dots (left). A 3D model generated by the Vicon system after localising the dots (right) [12].

The Vicon motion capture is a popular commercially available device which sets the gold standard for indoor localisation systems in terms of speed and accuracy. It uses multiple infra-red cameras to track and triangulate reflective markers. Vicon systems are capable of achieving localisation accuracies of less than a millimetre. However, these systems require high processing power, controlled environments and cost more than £100,000 [13].

1.2. Aim and Objectives

From section 1.1, summarised in Figure 3, it is clear that currently there is an absence of a high speed (25 Hz refresh-rate), accurate and low-cost vision based localisation system.

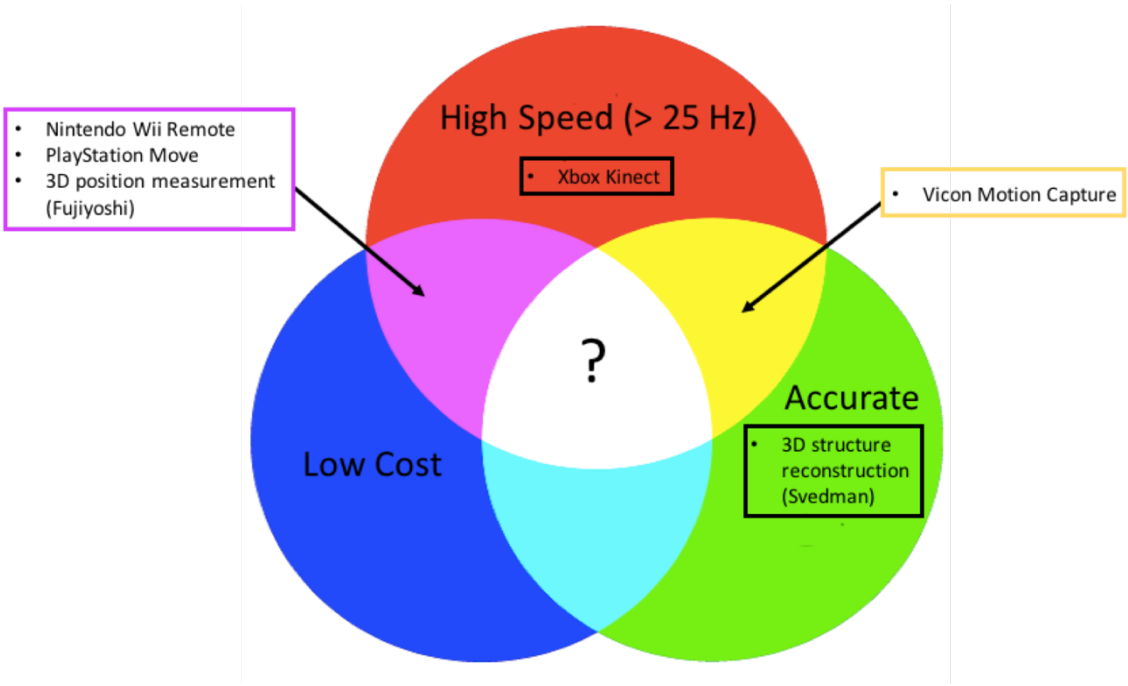

Figure 3: Illustration of current localisation systems and the need for a well-rounded system.

A new class of ARM based single-board computers has made the design and development of such a system possible. These credit card–sized computers provide relatively high performance at a very low cost. The Raspberry Pi 2 is one such system which was chosen for this project due to its large user base and abundance of software resources for developers.

1.2.1. Aim

To design a high speed vision based indoor localisation system using two or more cameras each connected to a Raspberry Pi 2.

1.2.2. Primary Objectives

- Minimise the computational cost of the image processing and detection software such that the refresh rate of the system is limited by the camera frame rate and not the image processing overhead - to achieve refresh rates of above 25 Hz.
- Achieve localisation accuracy of 3 cm up to a range of 5 meters from both cameras.
- Design a system which is expandable – more cameras can be added if needed.
- Design a system that can be built using both cheaply and easily available hardware such as Raspberry Pi, webcams and Wireless USB adapters.
- Design a system that can be effectively utilised in a broad spectrum of applications.

1.2.3. Secondary Objectives

- Design a low cost system within a budget of £200.
- Design a system in which the individual Beacons and receivers communicate with each other wirelessly thus allowing for greater flexibility with respect to placement and configuration of the system.
- Design a system which is user friendly and can be set up within a span of 2 minutes.
- Design a system which can interface with a wide variety of devices with different operating systems – Linux based systems, Macintosh, Windows.

1.3. Design Challenge

The main considerations while designing a system which meets the above requirements were:

Speed

Image processing involves iterating over a large number of pixels. For an image of resolution 640x480, the number of pixels being considered are 307,200.

Low cost high speed 3D-positioning using image processing on small-footprint computers has not been accomplished so far as most small-footprint computers lack the computing power. This performance bottleneck can be overcome by implementing efficient software which exploits the multicore ARM architecture of the Raspberry Pi 2.

Accuracy and Repeatability

The detection process is highly sensitive to noise in the image. The object of interest should have unique descriptors to make it distinguishable from various backgrounds. The high light intensity of RGB LEDs and laser dots make them an ideal choice as they have a uniform colour and are not susceptible to low lighting conditions.

Camera lens distortion significantly contributes to the localisation error. Thus undistortion algorithms have to be implemented to minimise error.

Range and Occlusion

The range of system is dependent on the camera resolution and field of view. A higher camera resolution and shorter camera focal length (increased field of view) improve the range of the system.

Occlusion takes place when there is an obstruction between the camera and the object of interest. A direct line of sight between each camera and the object is required for successful localisation. Occlusion can be mitigated by using more cameras.

2. Background

This chapter presents a brief overview of all the concepts and information necessary to develop a better understanding of the subject matter discussed later in this report.

2.1. Pinhole Camera Model

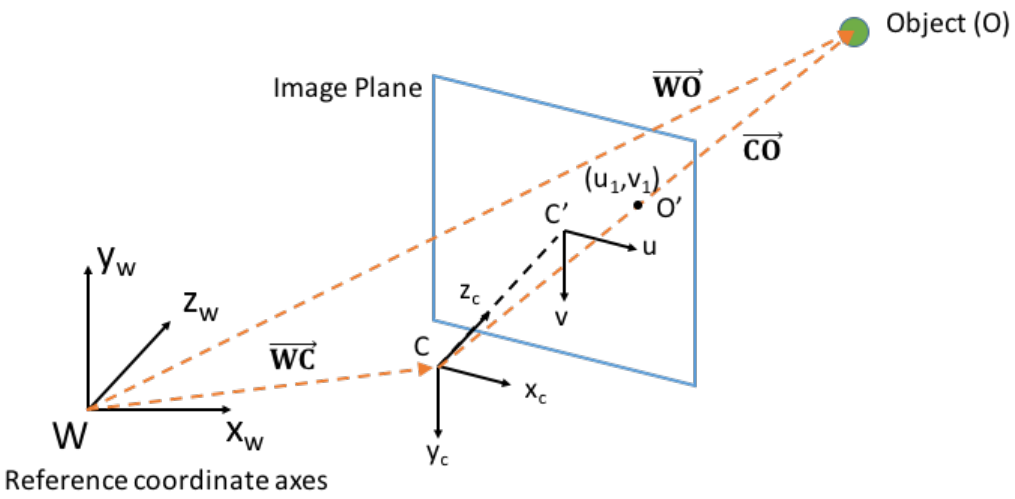

Figure 4: Pinhole camera model.

Each pixel in an image is obtained by projecting a point in 3D space onto the image plane of the camera. In Figure 4, the pixel positions u_1, v_1 are obtained by projecting the object O onto the image plane of camera C. Projected points are the intersection of the vector OC with the image plane. This mapping of 3D coordinates to 2D coordinates is called the perspective projection.

Thus each pixel position on the image plane is the projection of a certain point in 3D space. In Figure 4 above, the pixel position $\mathbf{m} = (u_1, v_1)$ is the perspective projection of the point represented by **wo**.

$$\mathbf{m} = \mathbf{P} \cdot \mathbf{wo} \tag{2.1}$$

where:

P is the perspective transformation matrix.

$\mathbf{wo} = [x \;\; y \;\; z]^T$ is the object position.

$\mathbf{m} = [u \;\; v]^T$ is the pixel position.

Homogeneous representation

$$\widetilde{\mathbf{wo}} = [x \;\; y \;\; z \;\; 1]^T \tag{2.2}$$

$$\widetilde{\mathbf{m}} = [u \;\; v \;\; 1]^T \tag{2.3}$$

$$\widetilde{\mathbf{m}} = \widetilde{\mathbf{P}} \cdot \widetilde{\mathbf{wo}} \tag{2.4}$$

In order to determine the components of $\widetilde{\mathbf{P}}$, the intrinsic parameters of the camera (represented by matrix **A**), the orientation of the camera (represented by matrix **R**) and the position of the camera (represented by **t**) have to be known.

The intrinsic matrix **A** is a 3 x 3 matrix:

$$\mathbf{A} = \begin{bmatrix} -f \cdot k_u & \gamma & u_0 \\ 0 & -f \cdot k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

f is the focal length of the camera given by the length of CC'. k_u, k_v are the number of pixels per millimeter along the u and v axes of the image plane. u_0, v_0 are the coordinates of principal point of the camera. The skew factor, γ represents the non-orthogonality of the u and v axes.

The $\mathbf{R}$ matrix is a 3 x 3 matrix and the $\mathbf{t}$ matrix is a 3 x 1 matrix They represent the orientation and displacement of the camera coordinate system at C with respect to the reference coordinate system at W.
Camera calibration is required to obtain the A, R matrix and t vector.

$\widetilde{\mathbf{P}}$ can be represented in terms of $\mathbf{A}, \mathbf{R}$ and $\mathbf{t}$:

$$\widetilde{\mathbf{P}} = \mathbf{A}[\,\mathbf{R} \mid \mathbf{t}\,] \tag{2.6}$$

$$\therefore \widetilde{\mathbf{m}} = \widetilde{\mathbf{P}} \cdot \widetilde{\mathbf{wo}} = \mathbf{A}[\,\mathbf{R} \mid \mathbf{t}\,] \cdot \widetilde{\mathbf{wo}} \tag{2.7}$$

$$\therefore \widetilde{\mathbf{m}} = \mathbf{A} \cdot \mathbf{R} \cdot \mathbf{wo} + \mathbf{A} \cdot \mathbf{t} \tag{2.8}$$

Using matrix algebra and multiplying $\mathbf{R^{-1}} \cdot \mathbf{A^{-1}}$ on both sides of Eq. (2.8) [14].

$$\mathbf{wo} = \left(-\mathbf{R^{-1}} \cdot \mathbf{t}\right) + \lambda \cdot (\mathbf{R^{-1}} \cdot \mathbf{A^{-1}} \cdot \widetilde{\mathbf{m}}) \tag{2.9}$$

Where λ is a scaling factor and $\lambda \in \Re$. λ accounts for the depth information lost while projection 3D points onto 2D.

Intuitively this can be seen in Figure 4 where the position of the object with respect to the world coordinate system can be represented as the sum of the camera position vector and the position of the object with respect to the camera.

$$\mathbf{wo} = \mathbf{wc} + \mathbf{co} \tag{2.10}$$

Thus comparing Eq. (2.9) to (2.10):

$$\mathbf{wc} = -\mathbf{R^{-1}} \cdot \mathbf{t} \tag{2.11}$$

$$\mathbf{co'} = \mathbf{R^{-1}} \cdot \mathbf{A^{-1}} \cdot \widetilde{\mathbf{m}} \tag{2.12}$$

Since $\mathbf{co}$ is parallel to $\mathbf{co'}$, $\mathbf{co}$ can be written as $(\lambda \cdot \mathbf{co'})$. λ determines where along the line $\mathbf{co'}$, the point O lies.

Thus for each calibrated camera (c_i), given the pixel position of the object ($\widetilde{\mathbf{m}}_i$) its 3D position ($\mathbf{wo}_i$) can be written as a function of the scaling factor λ_i in the parametric form.

$$\mathbf{wo}_i = \left(-\mathbf{R}_i^{-1} \cdot \mathbf{t}_i\right) + \lambda_i \cdot (\mathbf{R}_i^{-1} \cdot \mathbf{A^{-1}}_i \cdot \widetilde{\mathbf{m}}_i) \tag{2.13}$$

λ_i for each camera is estimated by calculating the point where the $\mathbf{wo}_i$ vectors meet or come closest to each other (if vectors are skew).

2.2. Colour Model

The ordinary human eye contains three types of cone cells in the retina. Each type is sensitive to a certain wavelength of light. The combination of these three types of cone cells enable us to perceive colour. Similarly, cameras contain a photosensitive sensor array fitted with a filter to enable them to "capture" colour as seen in Figure 5 below.

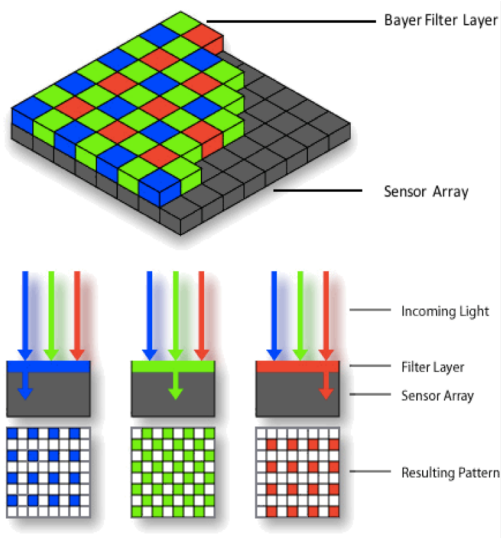

Figure 5: Schematic of a Bayer filter overlain on a camera sensor array (top) and the separation of red, blue and green light by the filter (bottom) [15].

The sensor array data is encoded and saved in images where each pixel consists of three channels. The format in which the colours are encoded is termed as the colour model.

Most 24-bit images captured by cameras today are in the Red, Green, Blue (RGB) format.

In the RGB model, each colour appears as components of the primary spectral colours red, green and blue. It is based on the Cartesian coordinate system.

As seen in Figure 6(a) below, RGB values are at the three corners of the cube, black is at the origin and white is at the corner of the cube farthest away from the origin. Different colours are represented as points on or inside the cube, defined by vectors extending from the origin.

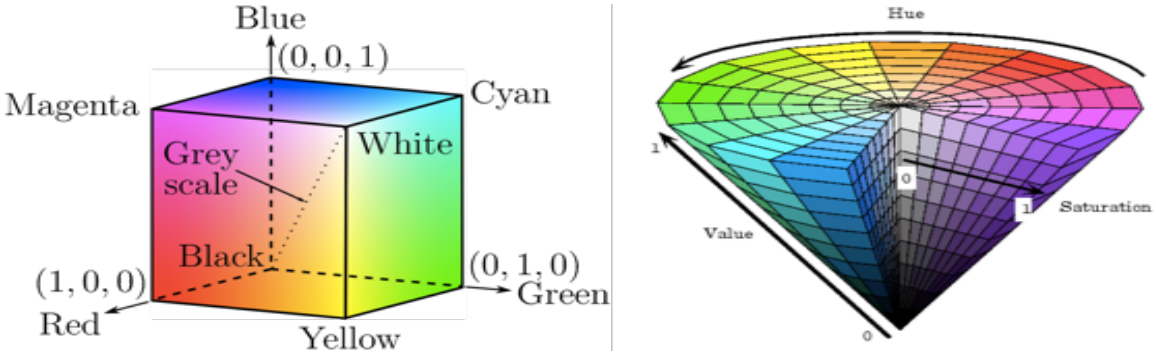

Figure 6: Graphical representation of (a) RGB Colour model (left) and (b)HSV colour model (right) [16,17].

The Hue, Saturation, Value (HSV) colour model shown in Figure 6(b) is another commonly used colour model. Hue represents the intrinsic colour. Saturation represents the percentage of 'pure' colour to 'grey'. Value represents the intensity of the object being represented. For the purpose of this project, the HSV colour model is more suited for detection of objects as it decouples the intensity component from the colour-carrying information (hue and saturation) in a colour image. The decoupling of intensity and colour information makes the development of detection algorithms more intuitive as the object can be described by its hue, intensity and saturation independently. For example, to detect a

green object all one has to do is threshold the hue channel for green hue values - 80 to 130 degrees. HSV thresholding is discussed in further detail in Section 3.1.
The RGB colour space can be converted to the HSV colour space using the following algorithm [18]:

$$V = \max(R, G, B)$$

$$S = \frac{255(V - \min(R, G, B))}{V} \quad if\ V \neq 0, 0\ otherwise$$

$$H = \begin{cases} \frac{60(G - B)}{S}, & V = R \\ 180 + \frac{60(B - R)}{S}, & V = G \\ 240 + \frac{60(R - G)}{S}, & V = B \end{cases}$$

if $H < 0$ then $H = H + 360$

2.3. Raspberry Pi 2 Model B Relevant Specifications

All image processing and calculations required in the proposed system were carried out on Raspberry Pis. The Raspberry Pis used had the following specifications.

Hardware specifications

- System on Chip (SoC): Broadcom BCM2836
- CPU: 900MHz quad-core ARM Cortex-A7
- GPU: 250Mhz Broadcom VideoCore IV
- RAM: 1 GB LPDDR2 SDRAM
- USB: 4 Ports, USB 2.0
- Storage: 32 GB SanDisk Extreme Plus, Class 10, microSDHC memory card

Software specifications

- Operating System: Raspbian Wheezy
- Libraries installed: OpenCV-2.4.11, Cython (0.23.4), Matplotlib (1.1.1rc2)

3. Design Methodology

This chapter describes in detail, the design process of the main sub-systems involved in the proposed localisation system.

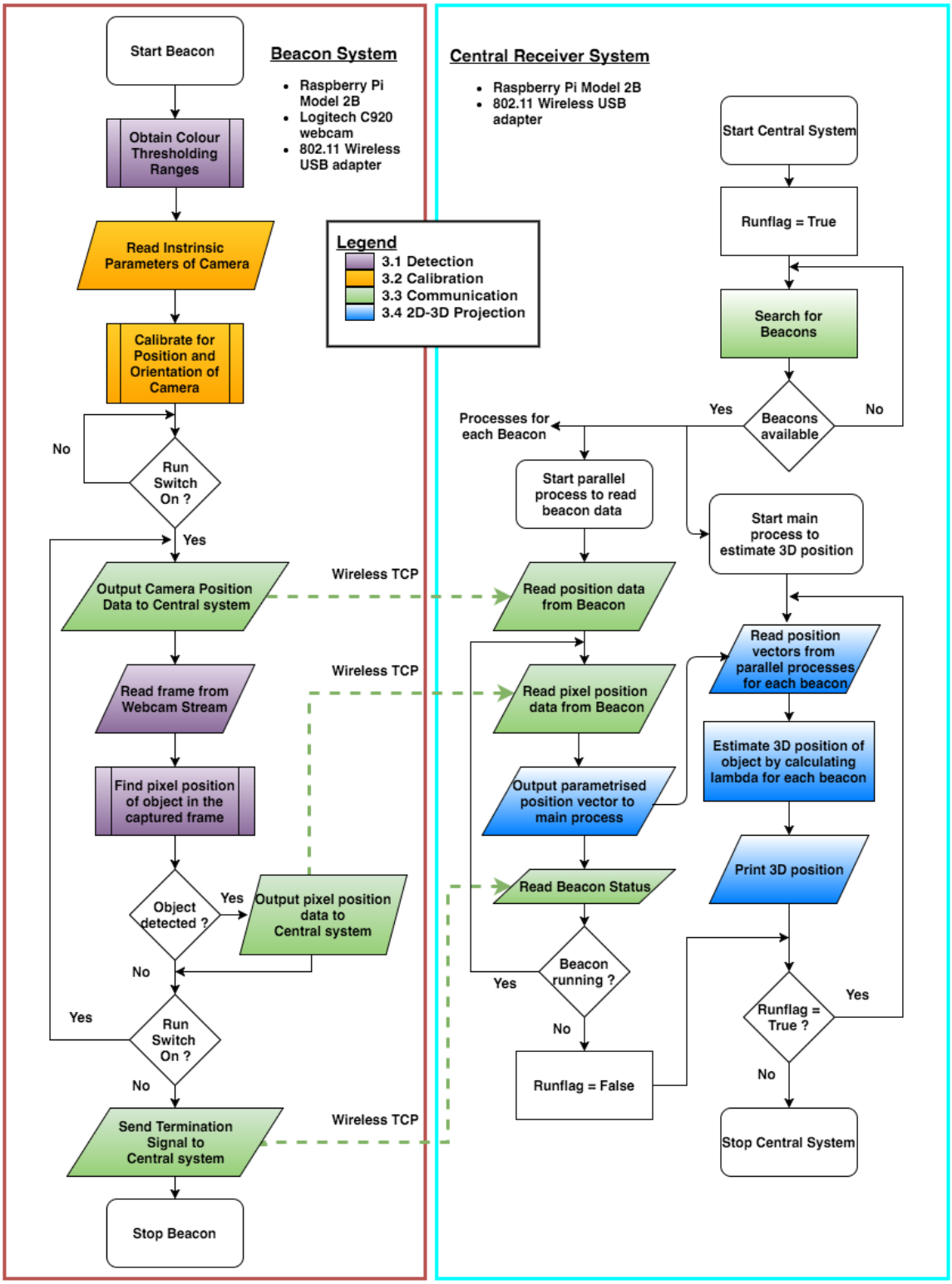

Figure 7: A flowchart representation of the proposed device illustrating a Beacon System (enclosed in red, left) and the Central Receiver System (enclosed in light blue, right). The legend shows the colour-coding for the main sub-systems and their corresponding section numbers.

The proposed system consists of two Beacons continuously sending data to a central system. The Beacons capture a video feed and detect the object of interest in each frame. Each Beacon consists of a Raspberry Pi 2 model B connected to a webcam and a wireless USB adapter. The central receiver receives data from the Beacons and uses this data to estimate the 3D position of the object being localised. The central receiver system consists of a single Raspberry Pi 2 which receives data via a wireless USB adapter. Figure 7 shows the workflow of one Beacon system and the central receiver system.

3.1. Detection

This section describes the detection sub-processes (shown in purple in Figure 7).

3.1.1. Object of interest

Detection is the process by which the pixel position of the object of interest in an image is obtained. The detectability of an object is determined by how easily it is distinguished from its surroundings. As explained in section 1.3, an RGB LED and a laser dot were chosen as objects of interest as they are easily distinguished from their surroundings due to their high light intensity values. Also in order to protect the user from the high intensities, the LED was placed in a diffused container and safety glasses were worn while operating the laser.

Specifications:

1. RGB LED – Star LED
Manufacturer – Solarbotics Limited.
Model number – 60160
Power – 3 Watt

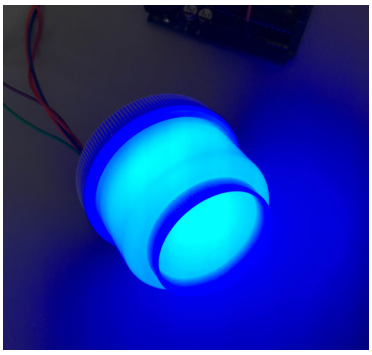

Figure 8: LED in diffused container.

2. Laser
Manufacturer – Skylaser
Model number – LP532-20
Wavelength – 532 nm (Green laser dot)
Power – 20 mW

3.1.2. Image Capture and Camera Settings

The Logitech C920 webcam was used to capture the video feed. The software package, OpenCV-2.4.11(Python 2.7), was used for video capture. Camera settings were adjusted using the Video for Linux (v4l2) library. The capture resolution was set to 480x640 pixels and the frame rate was set to the maximum value – 30 frames/second.

Auto-focus was switched off and the absolute focus of the webcam was set to the minimum value in order to get a diagonal field of view of 90 degrees. Auto-exposure was also switched off in order to prevent large variation of data in varying lighting conditions. Mesko et al. established that reducing camera exposure was advantageous while detecting high intensity objects [19]. Low camera exposure makes high intensity objects more distinguishable from their surroundings. This is evident in Figure 9(a) where the boundary of the laser dot is poorly defined. Refer to Appendix A for code.

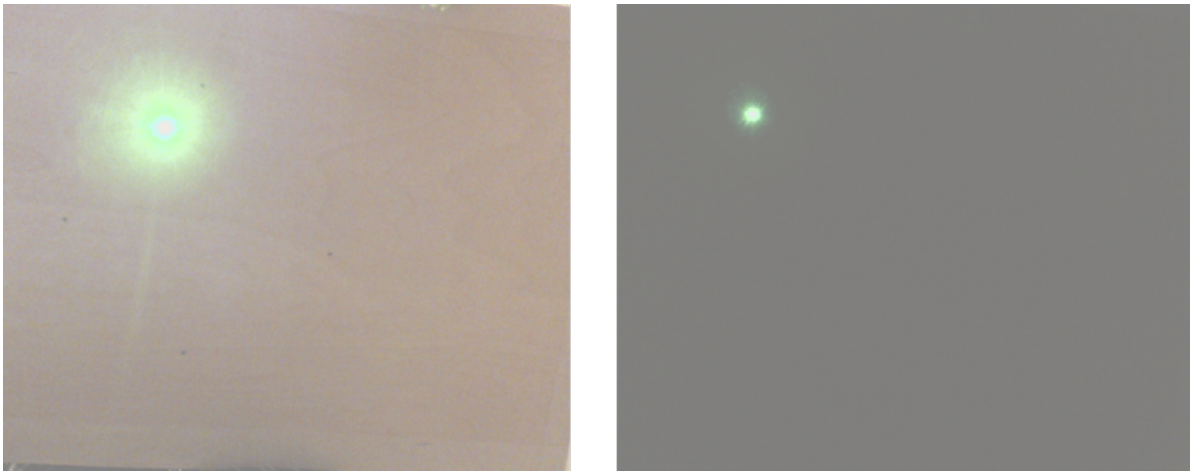

Figure 9: (a) Image of laser dot on plywood captured with high exposure settings (left) and (b) Image captured with low exposure settings (right).

Also from Figure 9(b), the difference in intensity between the laser dot and the plywood in the background is accentuated as the plywood now appears black. This greatly facilitates the selection of a detection method.

3.1.3. Object Data Acquisition from Images

An image differencing approach was adopted to extract object pixel data from the video feed.

Algorithm

Consider a Reference Image **I** and an altered image **A.**

Let Value of red channel of a pixel at position x, y be $r_{x,y,\mathbf{I}}$ and $r_{x,y,\mathbf{A}}$ for image **I** and image **A** respectively.

Similarly the green channel values will be $g_{x,y,\mathbf{I}}$ and $g_{x,y,\mathbf{A}}$,

And the blue channel values will be $b_{x,y,\mathbf{I}}$ and $b_{x,y,\mathbf{A}}$

Thus for every pixel at x, y position within the region of interest:

$$\Delta r_{x,y} = r_{x,y,\mathbf{A}} - r_{x,y,\mathbf{I}}$$

$$\Delta g_{x,y} = g_{x,y,\mathbf{A}} - g_{x,y,\mathbf{I}}$$

$$\Delta b_{x,y} = b_{x,y,\mathbf{A}} - b_{x,y,\mathbf{I}}$$

$$norm_{x,y} = \sqrt{\Delta r_{x,y}{}^2 + \Delta g_{x,y}{}^2 + \Delta b_{x,y}{}^2}$$

if $norm_{x,y} > cutoff\ value$ then pixel at x, y represent the object of interest.

Note: A cut-off value of 50 yields good results. This value was determined by experimental observation

Procedure

First a background image was captured to serve as a reference. The reference image for the RGB LED data extraction, Figure 10(a) included the diffused container with the LED switched off. The reference image for the laser dot data extraction, Figure 11(a) did not contain the laser dot. The reference images appear dark due to the low camera exposure. Another image was captured with the laser dot or with the LED switched on depending on the case. The object of interest was then manually selected as shown in the Figure 10(b) and Figure 11(b).

From Figure 10(c) and 11(c) it is clear that the only difference (shown in white) between the new image and the reference image is the object of interest. Thus object data can be extracted with great accuracy. Refer to Appendix B for code.

Figure 10: (a) Reference image (left-most), (b) Object Selection (middle) and (c) Image Difference of selected region (right-most) for the RGB LED.

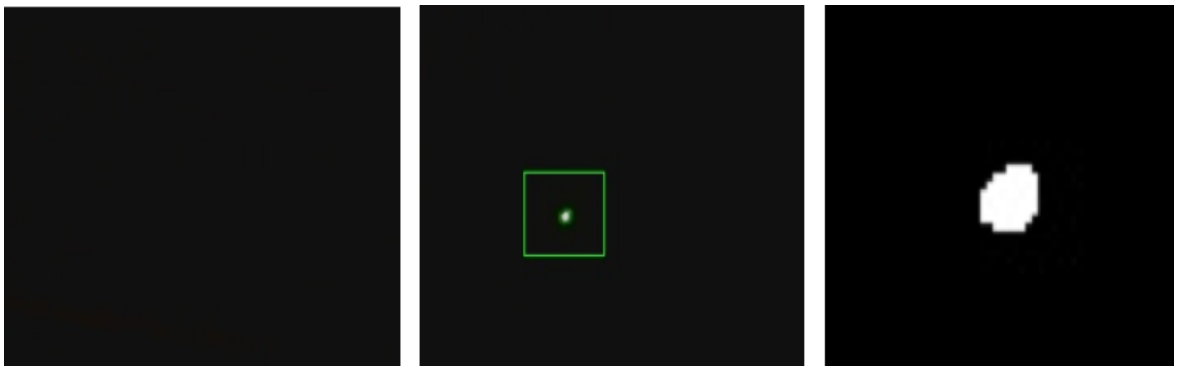

Figure 11: (a) Reference image (left-most), (b) Object Selection (middle) and (c) Image Difference of selected region (right-most) for the laser dot.

3.1.4. Feasibility of HSV Thresholding and Determining HSV Ranges

Figure 12 and 13 show three dimensional plots of the hue, saturation and intensity of pixels of the laser dot and the LED respectively. The plots are colour mapped to indicate the frequency of the occurrence of pixels with the same hue, saturation and intensity. Different colour maps are used for pixels representing the background objects and pixels representing the object of interest. From these graphs it is clear that HSV thresholding is a feasible approach for object detection as both the objects have unique hue, saturation and intensity values when compared to their backgrounds.

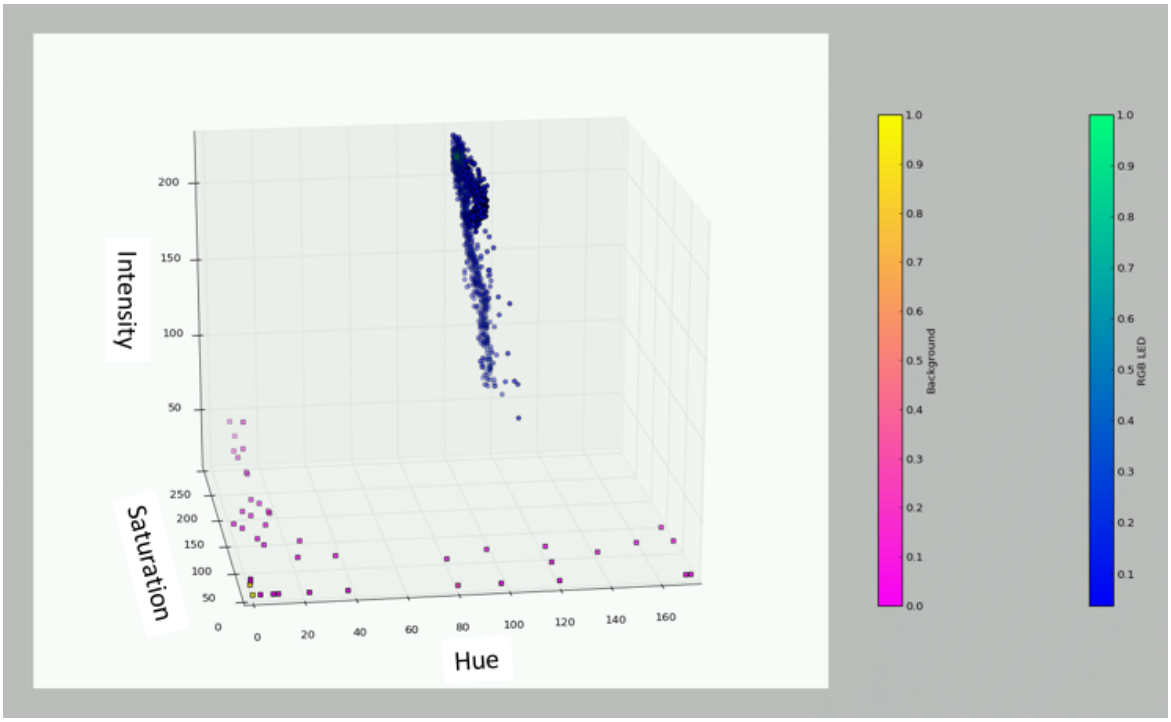

Figure 12: 3D graph displaying the heatmap of the Hue, Saturation and Intensity of the pixels representing the RGB LED (blue and green) and the Background (pink and yellow). Code in Appendix B.

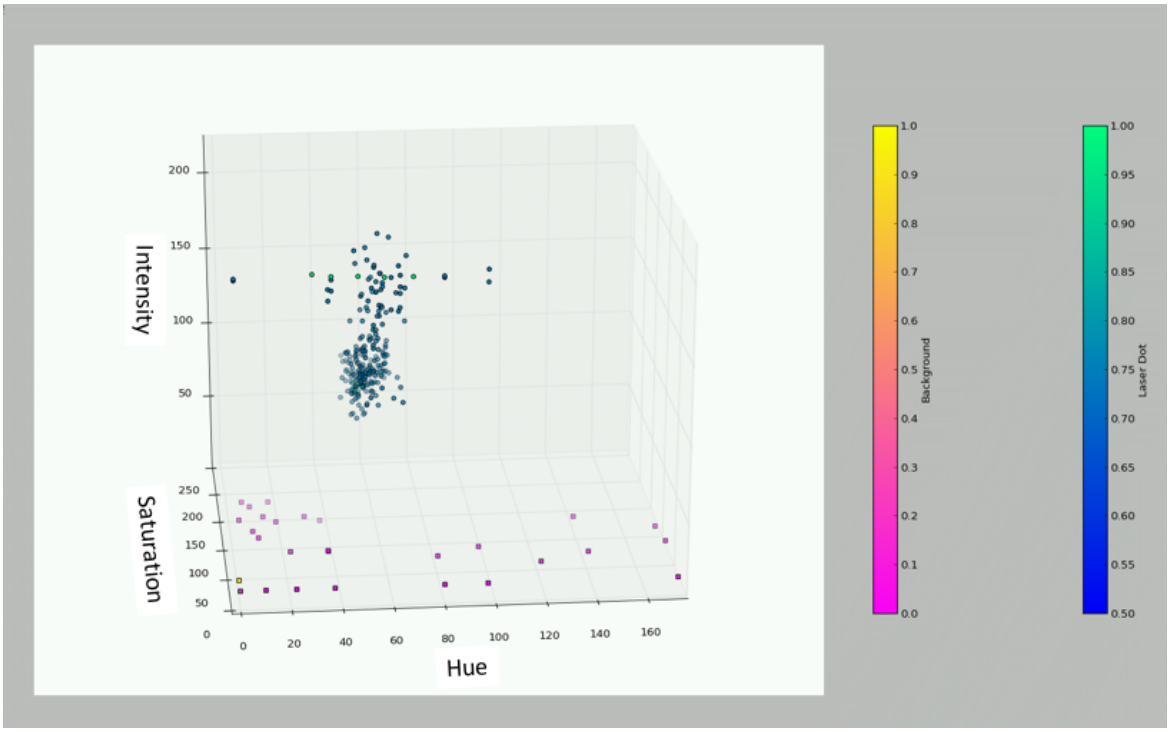

Figure 13: 3D graph displaying the heatmap of the Hue, Saturation and Intensity of the pixels representing the laser dot (blue and green) and the Background (pink and yellow). Code in Appendix B.

In order to detect objects, suitable hue, saturation and intensity ranges have to be determined. Hue and intensity are obvious descriptors of the LED and the laser dot. Thresholding is also necessary for saturation to remove false positives introduced by

natural light. Natural light has a high intensity and is polychromatic in nature - it is a superposition of light of various wavelengths. Thus an image filter with thresholds for only hue and value was insufficient as shown in Figure 14(b) below. Being a superposition of various wavelengths, natural light has a low saturation. This property is exploited by the saturation filter to greatly improve the detection process as seen Figure 14(c).

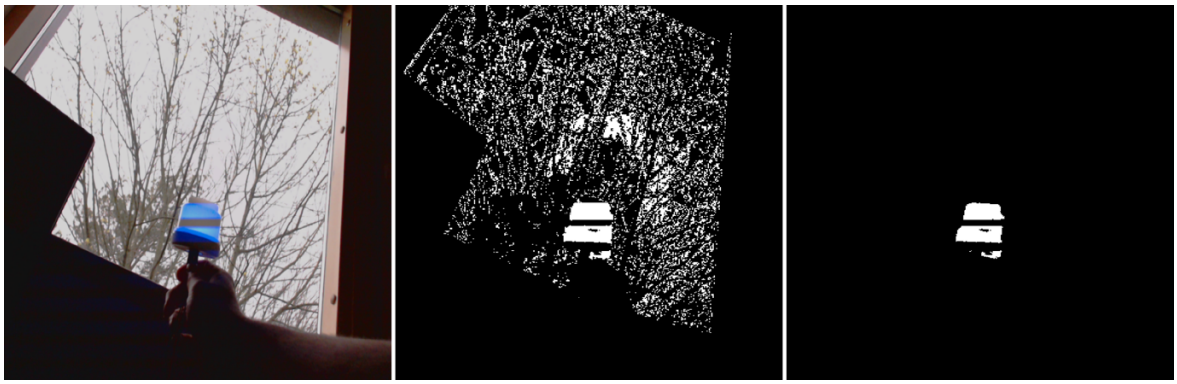

Figure 14: (a)Input image (left-most), (b)Filtered Image without thresholding (middle) for Saturation and (c)Filtered image with thresholding for Saturation(right-most).Code in Appendix C.

In Figure 12, it can be seen that most pixels which represent the RGB LED lie within the hue range 100–130. This is further illustrated by Figure 15, where the frequency of occurrence of hue values is plotted for pixels representing the LED.

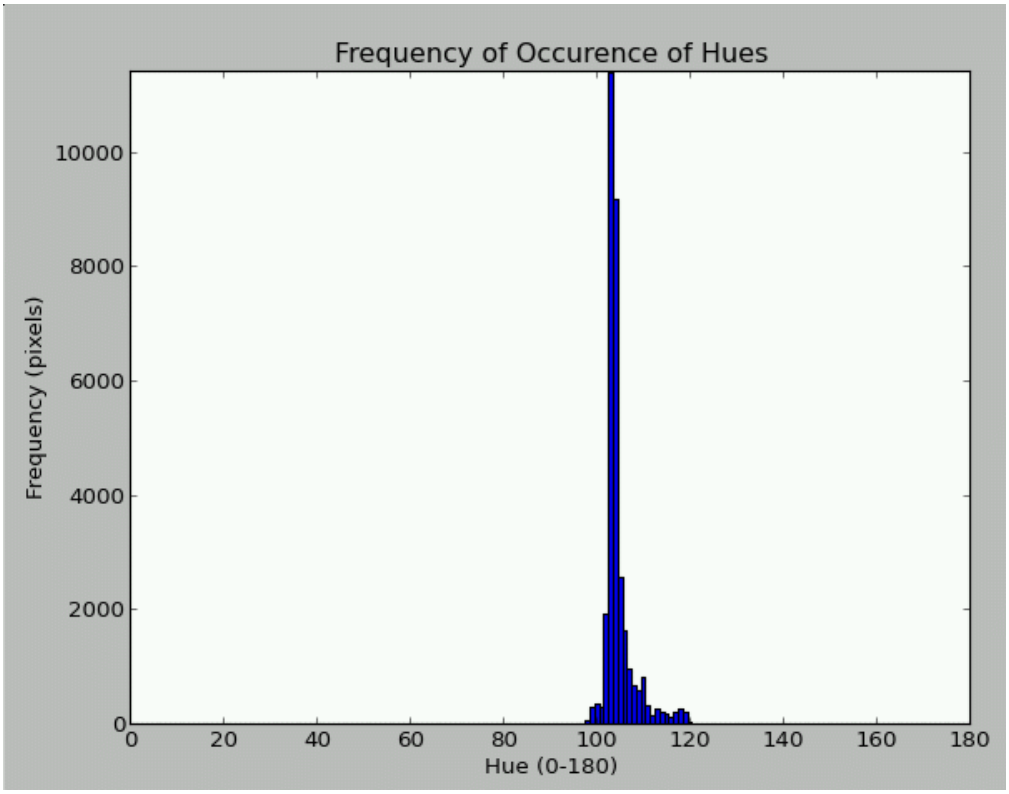

Figure 15: Frequency of occurrence of Hue values for the blue LED. Code in Appendix B.

Similarly, by observing Figure 13 and 16, a hue range of 40 – 80 was chosen for detecting the laser dot.

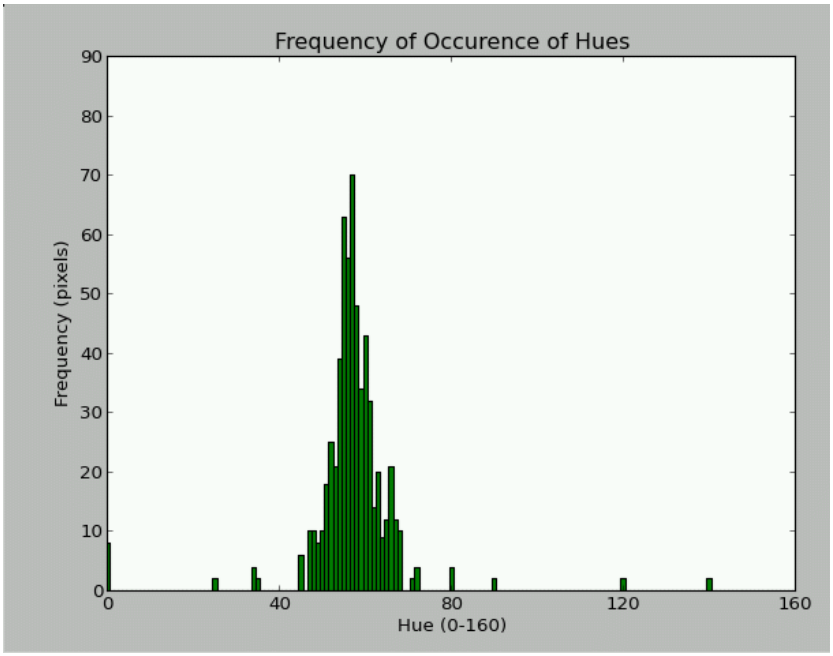

Figure 16: Frequency of occurrence of Hue values for the green laser dot. Code in Appendix B.

The major distinction between the detection processes for the laser dot and the LED is the type of thresholding used for saturation and intensity.

The saturation and intensity of the pixels, which represented the LED, were largely independent of the ambient conditions (lighting, colours in the background). There was no observable relationship between the saturation and intensity of the pixels which represented the LED. Thus based on Figure 17, static threshold ranges of 100 – 255 and 50 – 255 were chosen for saturation and intensity respectively.

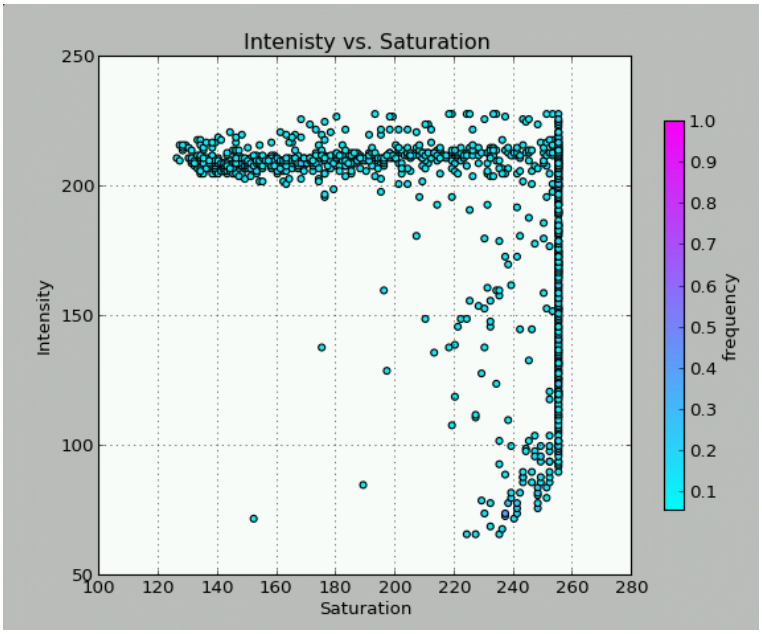

Figure 17: Saturation vs Intensity of pixels representing the RGB LED. Code in Appendix B.

The saturation and intensity of the laser dot were largely dependent on the surface at which the laser was pointed. Figure 18 illustrates the variation of saturation with respect to intensity of the laser dot for various surfaces. The intensity of pixels with low saturation was high and the intensity of the pixels with high saturation was low. Hence a dynamic threshold was needed for saturation and intensity of the pixels. This dynamic threshold is represented by lines l_{min} and l_{max}.

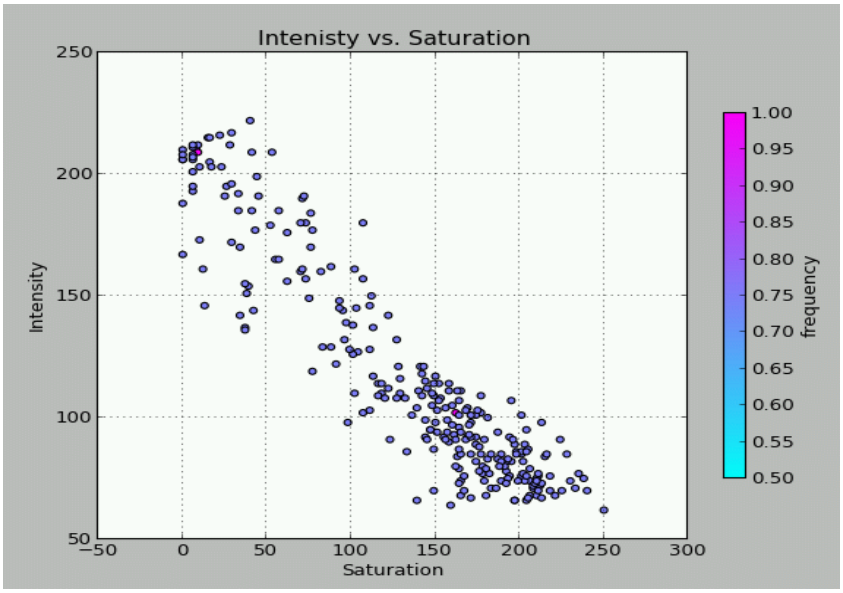

Figure 18: Saturation vs Intensity of pixels representing the laser dot. Code in Appendix B.

The following algorithm was used to calculate the equation of l_{min} and l_{max}:

Algorithm

Let the set of pixels representing the laser be S_1.

$S_1 \in \Re^2$ where each point in S_1 is represented by x_i and y_i. $x_i, y_i \in \Re$.

Step 1: Iterate over S_1 to find point (x_{min1}, y_{min1}) with minimum value of $x_i + y_i$.
(x_{min1}, y_{min1}) lies on l_{min}.

Step 2: Intuitively l_{min} is the line which makes the minimum slope with the line:
$y + x = y_{min1} + x_{min1}$
Since a minimum of two points from S_1 should lie on l_{min}, the other point (x_{min2}, y_{min2}) can be found by iterating over S_1 to find the point with minimum value of $\Delta m = \left|\frac{y_i - y_{min1}}{x_i - x_{min1}} + 1\right|$

Step 3: Thus equation of l_{min} is $y - y_{min1} = \left(\frac{y_{min2} - y_{min1}}{x_{min2} - x_{min1}}\right) \cdot (x - x_{min1})$

Step 4: Iterate over S_1 to find point (x_{max1}, y_{max1}) with maximum value of $x_i + y_i$.
(x_{max1}, y_{max1}) lies on l_{max}.

Step 5: Repeat step 2 to find point (x_{max2}, y_{max2}) which has minimum value of $\Delta m = \left|\frac{y_i - y_{max1}}{x_i - x_{max1}} + 1\right|$.
(x_{max2}, y_{max2}) lies on l_{max}.

Step 6: Thus equation of l_{max} is $y - y_{max1} = \left(\frac{y_{max2} - y_{max1}}{x_{max2} - x_{max1}}\right) \cdot (x - x_{max1})$

Applying this algorithm to the set of points in Figure 18 yields the following equations:
$l_{min}: y + 0.644 \cdot x = 155$
$l_{max}: y + 0.808 \cdot x = 326.14$
Figure 19 shows the plotted result after subtracting 10 from the intercept of l_{min} and adding 10 to the intercept of l_{max}. Refer to Appendix D for code.

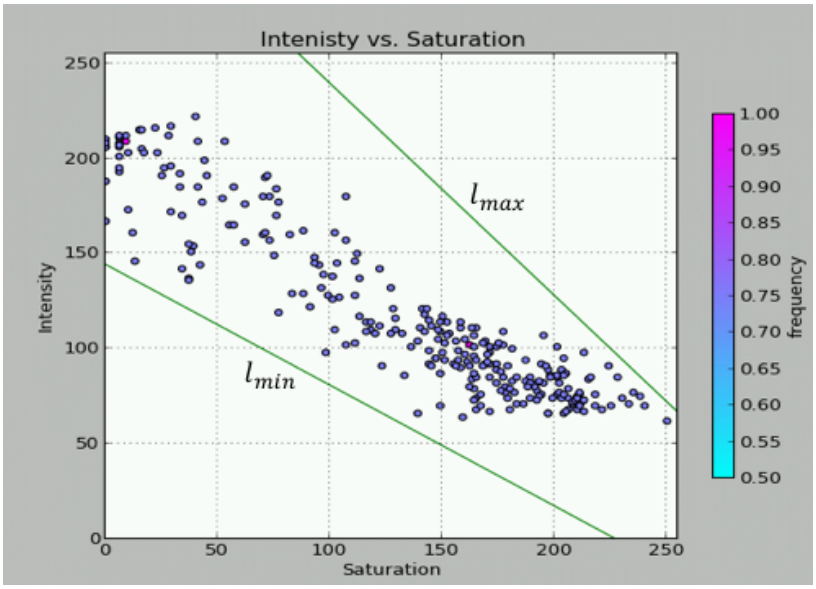

Figure 19: Saturation vs Intensity of pixels representing the laser dot with lmin and lmax.

3.1.5. Detection Results

Using the hue, saturation and intensity ranges found in Section 3.1.4, a threshold filter was applied to a set of input images for each object, Figure 20(top) and 21(top). The output images are illustrated in figure 20(bottom) and 21(bottom) (Code in Appendix E). A green box was drawn around the detected object in Figure 19 and a blue box was drawn around the detected object in Figure 20. These results substantiate the claim made in Section 3.1.4 that HSV thresholding is a feasible approach for accurately detecting the objects of interest.

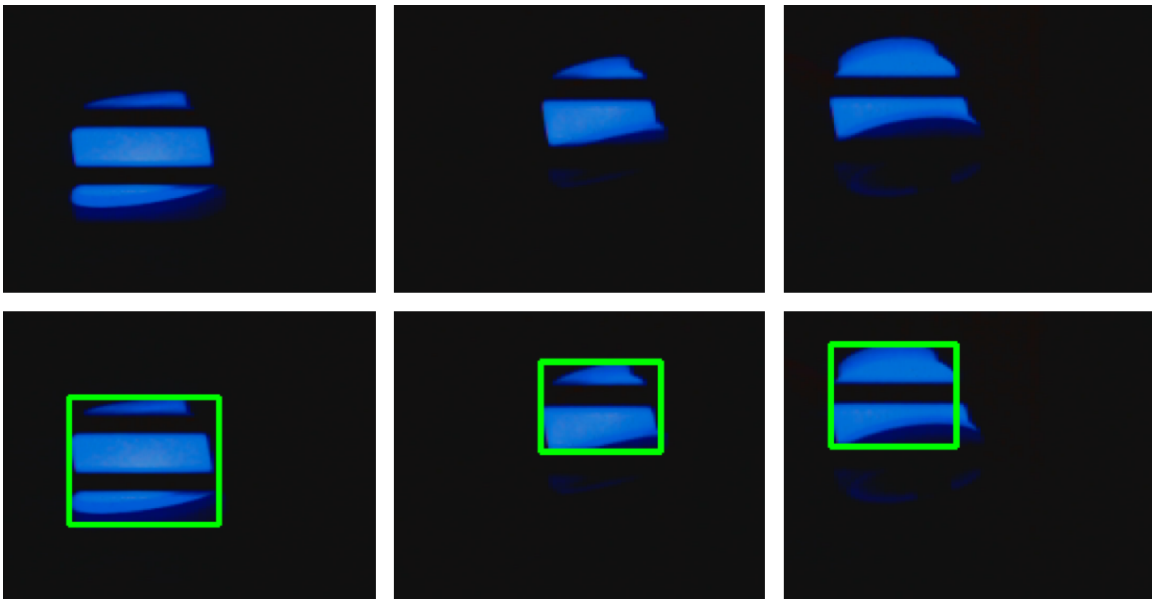

Figure 20: (top) Input images and (bottom) Detected result (enclosed in green) for LED.

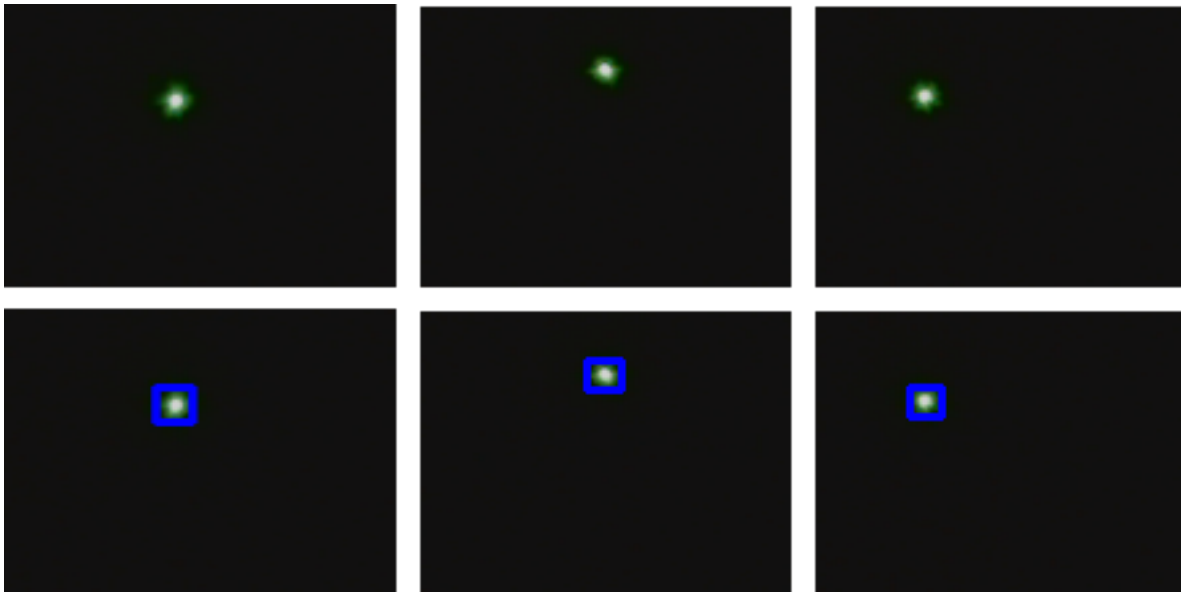

Figure 21: (top) Input images and (bottom) Detected result (enclosed in blue) for LED.

3.1.6. Performance Enhancement

Efficient software was required to implement image thresholding as the computational cost incurred by object detection could severely reduce the refresh rate for the proposed device.

Generic Algorithm

A generic algorithm was first tested to determine the feasibility of HSV thresholding and generate Figure 20, 21. This Algorithm was written with readability as the focus and not performance. The following steps describe this generic algorithm.

Algorithm

Step 1: Read the 24-bit colour image, containing the object of interest, as **I**. This is done by first creating a webcam streaming object called **cap** and then reading from it using **cap**.read().

Step 2: Image **I** is in the BGR (Blue, Green, Red) format which is a standard for webcam streaming using OpenCV. In order to carry out thresholding image **I** has to be converted to the HSV (Hue, Saturation, Value) format. This is done using: $\mathbf{I}_{HSV}$ = cv2.cvtColor(**I**, cv2.cv.COLOR_BGR2HSV)

Step 3: Each pixel of $\mathbf{I}_{HSV}$ is then passed through the HSV filter. If the hue, saturation and value (intensity) of the pixel are within the predetermined range, the pixel represents the object of interest. This is done using the numpy.where() function which return the pixel positions of the pixels.

Step 4: The pixel position of all the pixels which represent the object of interest are used to draw the bounding box (shown in red in Figure 20(bottom) and 21(bottom)). Corner points of the bounding box are calculated using the min() and max() functions. These corner points are passed to the cv2.rectangle() function which draws the box.

The Python Time module was used to test the performance of this algorithm. Figure 22 shows the results obtained upon implementing this algorithm.

```
**********************************************************************************
Number of frames: 300
Time taken: 24.4700 seconds
Number of frames processed per second: 12.26 frames
**********************************************************************************
```

Figure 22: Performance of generic algorithm for an image that has been read (Step 2-4). Code in Appendix E.

Performance Oriented Algorithm

In order to improve the performance of the generic algorithm, computationally expensive steps had to be recognised and the associated costs had to be reduced.

In the generic algorithm, converting each pixel from RGB to HSV format (Step 2) and checking if the pixel data is within the threshold ranges (Step 3) are operations which are performed for each and every pixel. Thus these operations are carried out 640 x 480 = 307,200 times.

Two common methods of improving the performance of image scanning processes are using Look Up Tables (LUT) and reducing the search zone in the image [20].

In order to test these assumptions a new detection function was written. The detection function was written in the Cython programming language. Cython enables writing of C extensions for Python by combining the readability of Python with the performance of C. The following steps describe the improved algorithm:

Algorithm

Step 1: Create a 3 dimensional LUT in which the first dimension represents the blue channel, the second dimension represents the green channel and the third dimension represents the red channel of the pixel. The LUT therefore consists of 255 x 255 x 255 positions representing every possible combination of blue, green and red values. It is rendered by checking if for each position the corresponding Hue, Saturation and Intensity value is within the predetermined ranges of the threshold. If the position is within the acceptable range, a value of 1 is assigned to it. Otherwise its value remains 0.

Step 2: Read the 24-bit colour image, containing the object of interest, as **I**. This is done by first creating a webcam streaming object called **cap** and then reading from it using **cap**.read().

Step 3: Establish a search region ($\mathbf{I}_{ROI}$) based on pixel positions of the object in previous frames. If the object wasn't found in the previous frame, the search region is expanded to the entire image.

Step 4: Iterate over each pixel in $\mathbf{I}_{ROI}$. Let $b_{x,y}$ = blue channel value of pixel. Similarly, $g_{x,y}$ = green channel value and $r_{x,y}$ = red channel value.
If the value of the LUT at position $b_{x,y}, g_{x,y}, r_{x,y}$ is 1, then the pixel at x, y represents the object of interest.

Step 5: The pixel position of all the pixels which represent the object of interest are used to draw the bounding box as described in Step 4 of the generic algorithm.

Figure 23 shows the results obtained upon implementing this algorithm.

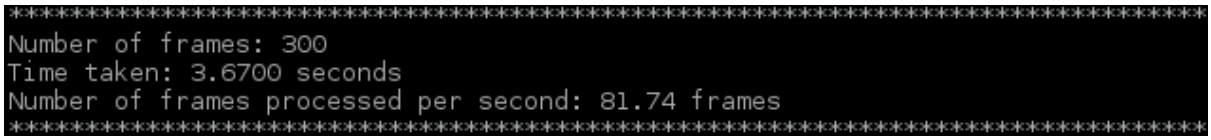

Figure 23: Performance of improved algorithm for an image that has been read (Step 3-5). Code in Appendix E.

The improvement in performance can be attributed to the following factors:

- The costs associated with conversion of pixel format from BGR to HSV is now incurred by the LUT rendering process carried out before running the main detection loop (Step 1).
- Reducing the search zone from 307,200 pixels to a smaller region based on the pixel positions of the object detected in previous frames greatly reduced the number of iterations per frame (Step 3).
- LUT access (Step 4 of improved algorithm) incurs much less overhead when compared to the six comparison operators used in the step 3 in the generic algorithm.

It is important to note that these performance results do not account for the overhead incurred by the image reading process. The following subsection discusses this issue

Python Multithreading for Video Capture

The **cap.**read() call in Step 2 of the improved algorithm was another major bottleneck. The call is a blocking operation i.e. the main thread of the Python Script is completely blocked until the frame is read from the camera device and assigned to object **I**. This introduces latency in the detection process.

Using threading to handle Input/Output-heavy tasks (such as reading frames from a camera sensor) is a programming model that has existed for decades [21]. By moving the reading of frames to a different thread from the main thread of the script, latency is significantly decreased. Thus after each detection in the main thread the new frame is grabbed from the video capture I/O thread without any delay being induced by the read operation.

Figure 26 shows the Python implementation of multithreading to carry out reading of frames in a parallel thread using the WebcamVideoStream class.

Figure 24 and 25 show the performance of the detection script before and after including the WebcamVideoStream class. The Raspberry Pi was used in the Graphical User Interface (GUI) mode.

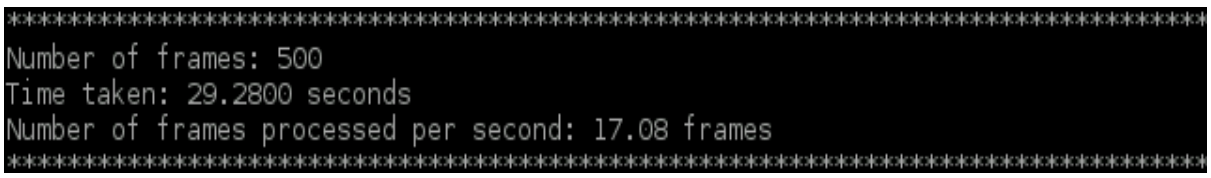

Figure 24: Performance of detection script without the WebcamVideoStream class.

```
**********************************************************************************
Number of frames: 500
Time taken: 11.9300 seconds
Number of frames processed per second: 41.91 frames
**********************************************************************************
```

Figure 25: Performance of detection script with the WebcamVideoStream class.

```python
# -*- coding: utf-8 -*-
"""
Multithreaded implementation of cv2.VideoCapture()
Author: Adrian Rosebrock
Modified by: Roshan Pasupathy
"""
import numpy as np
import cv2
from threading import Thread

class WebcamVideoStream:
    def __init__(self, src=0):
        # initialize the video camera stream and read the first frame
        # from the stream
        self.stream = cv2.VideoCapture(src)
        (self.grabbed, self.frame) = self.stream.read()

        # initialize the variable used to indicate if the thread should
        # be stopped
        self.stopped = False

    def start(self):
        # start the thread to read frames from the video stream
        t = Thread(target=self.update, args=())
        t.daemon = True
        t.start()
        return self

    def update(self):
        # keep looping infinitely until the thread is stopped
        while True:
            # if the thread indicator variable is set, stop the thread
            if self.stopped:
                return

            # otherwise, read the next frame from the stream
            (self.grabbed, self.frame) = self.stream.read()

    def read(self):
        # return the frame most recently read
        return self.frame

    def stop(self):
        # indicate that the thread should be stopped
        self.stopped = True
```

Figure 26: Python script for implementing threaded video capture.

The number of frames processed per second exceeds the maximum frame rate of the camera. It is clear that detection process is running more than once per frame and hence it is possible to detect multiple objects by running detections processes for each of them on every frame. The number of frames processed per second, when executed on the Beacon systems, will be greater than those obtained in Figure 24 and 25 as the GUI will not be used to run the scripts on the Beacons.

Thus by optimising the detection process, the challenges associated with image processing bottlenecks were overcome.

3.2. Calibration

This section describes the calibration sub-processes shown in orange in Figure 7.
Camera calibration was required to estimate the intrinsic and extrinsic parameters of the camera. A similar approach was adopted in both cases. One or more images are taken of a planar calibration pattern of known dimensions. The control points of the calibration pattern were then localised in each image and these control points were used to solve for the intrinsic parameters of the camera using 2D to 3D point correspondences.

The significant differences between the calibration process for intrinsic parameters and that for extrinsic parameters are:

- The type of control points of the calibration pattern.
- The algorithm used for localisation of control points

3.2.1. Calibration for Intrinsic Parameters

The intrinsic parameters of the camera are described by the **A** matrix and the distortion coefficients.

$$\mathbf{A} = \begin{bmatrix} -f \cdot k_u & \gamma & u_0 \\ 0 & -f \cdot k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \text{(Eq. 2.5)}$$

$Distortion_{coeff} = (k_1, k_2, p_1, p_2, k_3)$
$k_i = i^{th}$ radial distortion coefficient
$p_i = i^{th}$ tangential distortion coefficient.

Inaccurate localisation of control points is the primary source of error in the calibration process for intrinsic parameters. To mitigate this error, Datta et al. proposed an iterative refinement process. The accuracy of the iterative technique was found to be up to 50% better than non iterative techniques when tested on synthetic images [22].
Since accuracy was the main focus of the calibration processes, the iterative method proposed by Datta et al. was implemented.

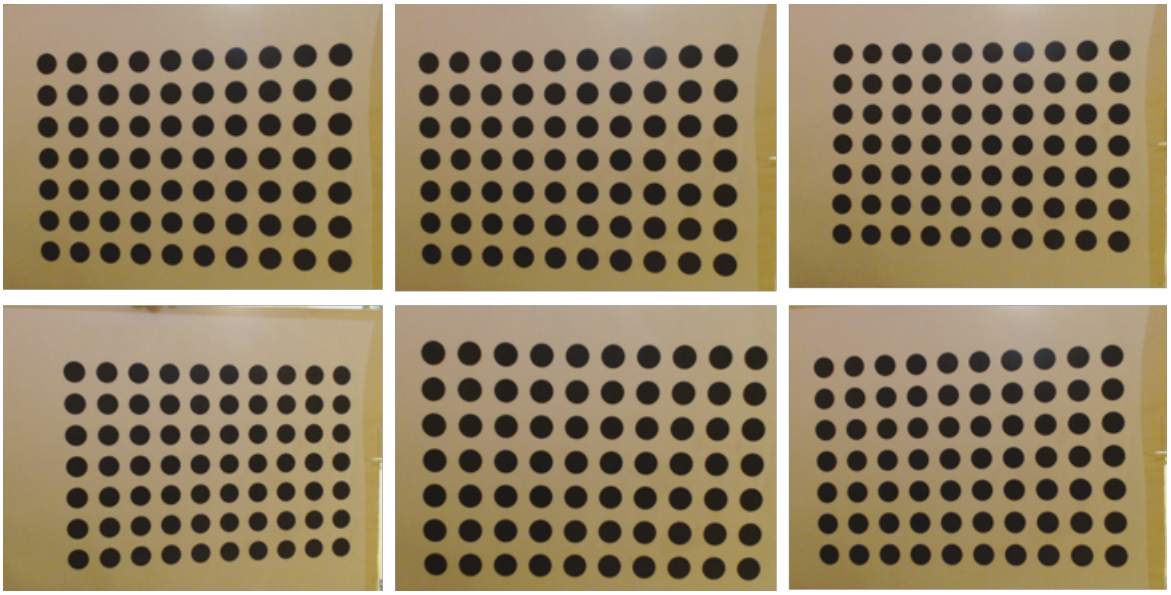

Figure 25: Input images of Calibration pattern taken from different angles.

Figure 27 shows that a calibration pattern with Circle control points was used since experimentally it was observed that iterative calibration with circle control points yielded the most accurate results [22].

The Algorithm as described in "Accurate Camera Calibration using Iterative Refinement of Control Points" is presented below [22].

Algorithm

Step 1: Detect calibration pattern control points (corners, circle or ring centers) in the input images.

Step 2: Use the detected control points to estimate camera parameters using Levenberg-Marquardt Algorithm [23].

Repeat the steps 3-6 until convergence

Step 3: Use the camera parameters to undistort and unproject input images to a canonical pattern.

Step 4: Localise calibration pattern control points in the canonical pattern.

Step 5: Project the control points using the estimated camera parameters.

Step 6: Use the projected control points to re-fine the camera parameters using Levenberg-Marquardt [23].

A Matlab software package was used to implement the algorithm shown above [24]. Ten images of the calibration pattern were captured.

The following **A** matrix and distortion coefficients were obtained:

$$\mathbf{A} = \begin{bmatrix} 620.54646 & 0 & 316.17234 \\ 0 & 621.10631 & 244.57960 \\ 0 & 0 & 1 \end{bmatrix}$$

$$Distortion_{coeff} = (0.13359\,, -0.29557\,, -0.0070\,, -0.00054, 0.00\,)$$

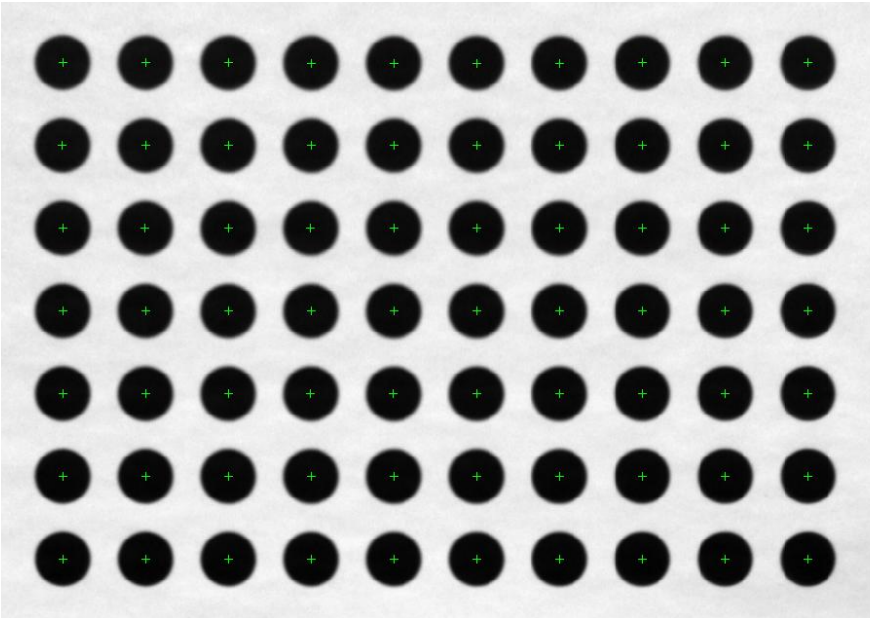

Figure 26: Control Point localisation.

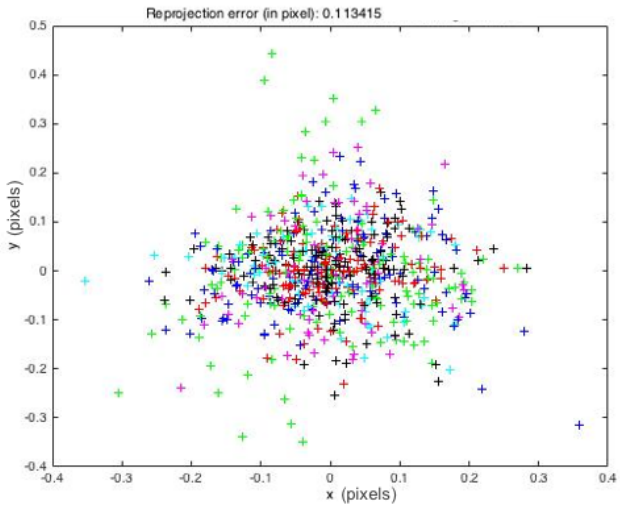

Figure 27: Scatter plot of reprojection error produced by the calibration process

Figure 28 shows the localisation of control of points and Figure 29 shows a scatter plot of the error obtained between the position of the reprojected control points and their actual positions. Reprojection error was within an acceptable range of 0.5 pixels

The distortion coefficients are used by OpenCV's undistortpoints() function to undistort the pixel positions obtained by the detection process (See Appendix F for code).

3.2.2. Calibration for Extrinsic Parameters

The extrinsic parameters of a camera are the rotation matrix ($\mathbf{R}$) and the translation vector ($\mathbf{t}$) (discussed in section 2.1) which describe the orientation and position of the camera respectively.

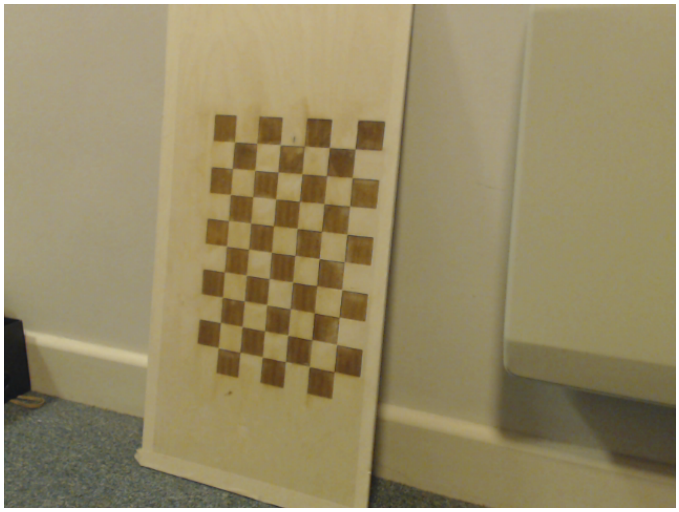

Figure 28: Image of chessboard pattern used to estimate the extrinsic parameters of camera.

As shown in Figure 30, a chessboard pattern was used for this calibration process. The control points in the chessboard pattern are the internal corners of the chessboard squares – the point of intersection of two black squares.

The Harris corner detector was used to obtain an initial estimate of the corner points [25]. It detects multiple pixels for each corner point. This initial estimate was further refined to sub-pixel accuracy using OpenCV's cornerSubPix() function [26]. cornerSubPix() calculates the centroid of the pixels, detected by the Harris Detector, for each corner (See Appendix X for code).

The intrinsic parameters of the camera are required to calibrate for the extrinsic parameters.

Figure 31 is a visual representation of the 3D-coordinate axes drawn on the chessboard pattern after the calibration process was completed using the code in Appendix G.

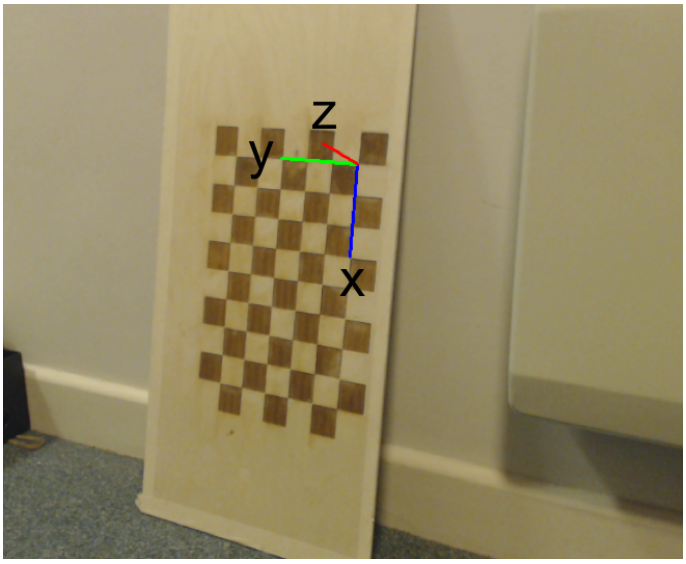

Figure 29: Visual representation 3D coordinate axis projected onto the input image.

3.3. Communication

The detection and calibration results can be used to write the 3D position of an object (**wo**) in the parametric form:

$$\mathbf{wo} = \left(-\mathbf{R_i}^{-1} \cdot \mathbf{t_i}\right) + \lambda_i \cdot (\mathbf{R_i}^{-1} \cdot \mathbf{A^{-1}}_{\boldsymbol{i}} \cdot \tilde{\mathbf{m}}_{\boldsymbol{i}})$$

where:

$\tilde{\mathbf{m}}_{\boldsymbol{i}}$ = pixel position of the object in homogeneous form

$-\mathbf{R_i}^{-1} \cdot \mathbf{t_i} = \mathbf{p_i}$ is the position vector of the camera.

$\mathbf{R_i}^{-1} \cdot \mathbf{A^{-1}}_{\boldsymbol{i}} \cdot \tilde{\mathbf{m}}_{\boldsymbol{i}} = \mathbf{q_i}$ is the 3D vector representation of the detected pixel position in the image plane with respect to the camera focal point. Refer to Appendix F for code.

A single camera cannot perceive the depth of an object. Hence parametric representations with respect to two or more cameras are required to solve for the 3D position of the object by calculating λ_i for each camera.

The communication sub-process (shown in green in Figure 7) involve the transmission of the components of the parametric equation, $\mathbf{p_i}$ and $\mathbf{q_i}$ from the Beacon system to the central receiver system.

3.3.1. Choice of Communication Technology

Initially, the central system was an Arduino Due. The following methods of communication to the Arduino were investigated:

Method 1: I2C communication via the I2C buses of the Raspi and the Arduino.

Method 2: Serial communication via USB cable.

Method 3: Serial communication via Bluetooth 4.0.

The first two methods to be investigated involve wired communication since it is robust in nature and is simple to implement. However, these approaches had several limitations. I2C communication is suitable only for short range communication. Serial communication via a USB cable is limited by the length of the cable. Also the Arduino Due has only two hardware I2C interfaces and three Serial interfaces. Thus the number of Beacons which can be used in the system is limited to 2 for I2C communication and 3 for Serial communication – limited expandability.

Range wasn't a problem for method 3 which involved wireless serial communication via Bluetooth. However, expandability was still an issue. Hence these solutions were not very elegant.

It was realised that using internet communication protocols and a Raspberry Pi as the central system was a much better solution than those previously investigated.

The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are two commonly used internet communication protocols. UDP is favourable in situations where high speed data transfer is required without ensuring reliability. The sender won't wait for an acknowledgement from the recipient, it will continue sending the next data packet.

TCP is slower than UDP because the sender waits for an acknowledgement from the recipient before sending the next packets. If the recipient does not receive the packet, the sender resends it. Thus reliable data transfer is ensured but with an additional overhead. Also, TCP requires a connection to be established between the sender and the recipient before data transfer can take place.

For the proposed system, the senders (Beacon systems) send less than 200 packets per second to the receiver (Central system). Thus the reliability of TCP is preferred over the speed of UDP.

3.3.2. TCP implementation

The Central system creates a wireless local area network (LAN) and assigns a fixed internet protocol (IP) address to each Beacon that connects to the network. This also enables the user to remote access the individual systems using Secure Shell (SSH) [27].

A single TCP connection requires a "client" and a "server". The client connects to the server using a "socket". The socket is defined by the IP address of the server, the port number on which the server is listening for connections and the communication protocol (TCP in this case).

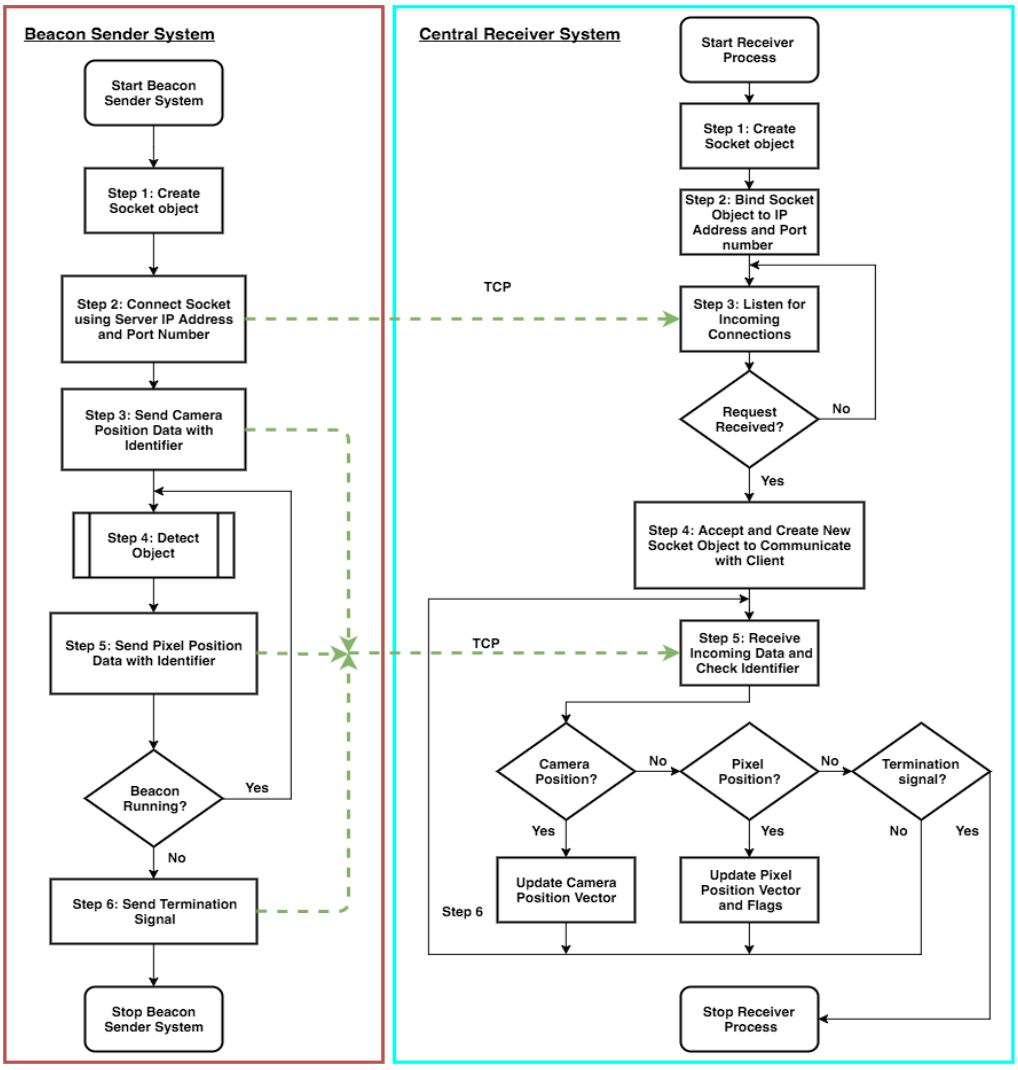

Figure 30: Detailed flowchart of communication sub-processes shown in figure 7.

TCP was implemented in the proposed system using Python's socket library. The central system functions as the server while the Beacons function as clients. For each Beacon, a parallel Python Process is created on the central system to receive data packets. The parallel process is created using the multiprocessing library. The multiprocessing library was used in conjunction with the multithreading library to prevent delays introduced by the Global Interpreter Lock (GIL). The GIL is released while reading data from the TCP socket since it is I/O bound. However, the Python code to update the main calculations is run within the GIL thus introducing delays. The multiprocessing library side-steps the GIL by using sub-processes instead of threads [28]. Figure 32 shows the workflow of a Beacon sender and this parallel Process. Refer to Appendix H for the code used.

In the receiver Process in Figure 32, a socket object is created and bound to the LAN IP address of the central system and a port number (Step 1 and 2). The port number was chosen arbitrarily from the available non standard port numbers (> 5000). The Process then listens for incoming connections. Once the Beacon connects to the socket, another socket object is created in the Process to receive data from the Beacon. This new socket object is not necessary for the proposed system and is created solely to allow the server to continue listening for connections from other clients.

The Beacons send 26-byte data packets. The first byte serves as the identifier. Table 1 below describes the identifiers.

Table 1: Description of packet identifiers used

Identifier	Description
c	The data packet represents the Beacon position data
u	The data packet represents the pixel position data
d	Object was not detected in latest frame
e	Termination Signal

The next 24 bytes represent either the camera position vector ($\mathbf{p}_i$) or the vector representation of the detected pixel positon vector of the object ($\mathbf{q}_i$). These vectors are of type – double (float64). Thus each component is represented by 8 bytes. The vectors are serialised by the Beacon using the Numpy function tostring() and deserialised by the receiver system using the Numpy function fromstring().

The last byte is the character '*l*' which represents the end of the data packet. This ending character prevents the receiver process from deserialising incomplete data packets.

Once the Central system's receiver processes deserialise the received data, they update the main process running on the same system. The updated values include the vector values and flags. The updated flags convey the state of the Beacons. The 'exec' flag determines if the main Process executes the 2D to 3D projection function. The runflag is set to 'True' only if the object was detected by both the Beacons – The packet identifier send by both Beacons is 'u' and not 'd' (Table 1). The main process is terminated if either Beacon sends the termination signal - 'e' packet identifier.

3.4. 2D to 3D Projection

3.4.1. Geometric Equation

The 3D position of the object can be calculated as the crossing point between the parametric lines drawn with respect to the camera for each Beacon. As discussed in Section 3.3, this parametric representation of the 3D position of the object for each camera is:

Eq. (2.11)

$$\mathbf{wo} = \mathbf{p_i} + \lambda_i \cdot \mathbf{q_i} \tag{3.1}$$

where:

$$\mathbf{p_i} = -\mathbf{R_i}^{-1} \cdot \mathbf{t_i}$$

$$\mathbf{q_i} = \mathbf{R_i}^{-1} \cdot \mathbf{A^{-1}}_i \cdot \widetilde{\mathbf{m}}_i$$

Parameterised forms for Beacon 1 and Beacon 2 are:

$$\mathbf{wo_1} = \mathbf{p_1} + \lambda_1 \cdot \mathbf{q_1} \tag{3.2}$$

$$\mathbf{wo_2} = \mathbf{p_2} + \lambda_2 \cdot \mathbf{q_2} \tag{3.3}$$

λ_1 and λ_2 are scaling factors belonging to the real domain. The position of the object is the midpoint of the transversal between $\mathbf{q_1}$ and $\mathbf{q_2}$. As seen in Figure 33, the transversal is the line drawn between the two vectors where they come closest to one another.

$$\therefore \mathbf{wo} = \frac{\mathbf{wo_2} + \mathbf{wo_1}}{\mathbf{2}}$$

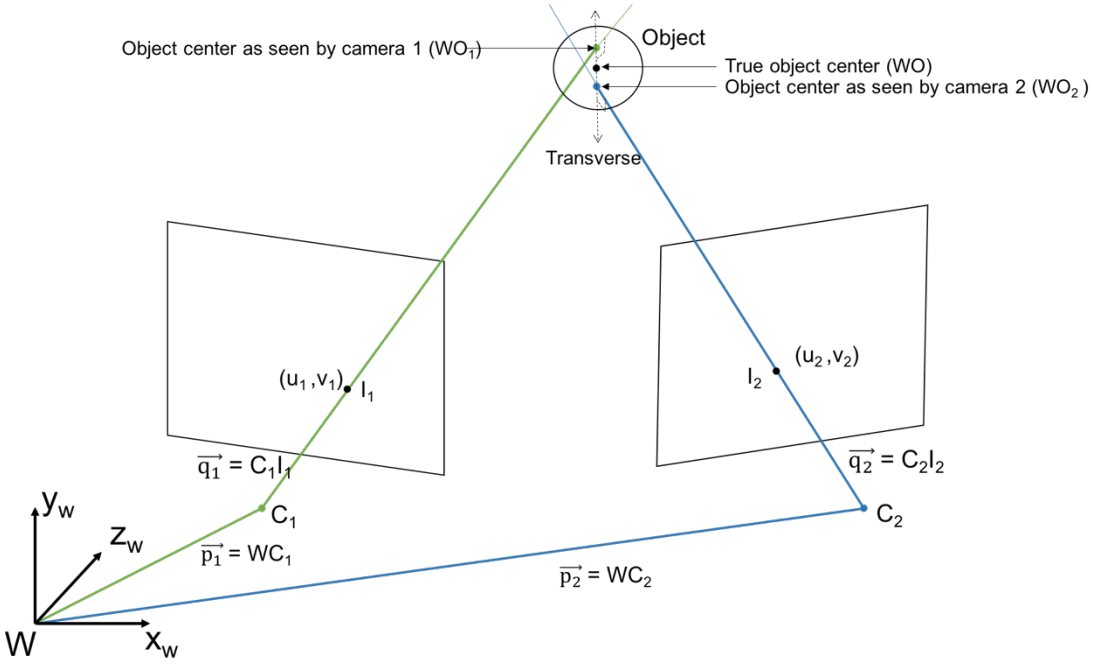

Figure 31: Geometric representation of 2D-3D projection problem.

From Eq. (3.2) and (3.3)

$$\mathbf{wo_2} - \mathbf{wo_1} = (\mathbf{p_2} - \mathbf{p_1}) + (\lambda_2 \cdot \mathbf{q_2} - \lambda_1 \cdot \mathbf{q_1}) \tag{3.4}$$

Also intuitively from vector properties it is known that:

$$\mathbf{wo_2} - \mathbf{wo_1} = a \cdot (\mathbf{q_2} \times \mathbf{q_1}) = a \cdot (\mathbf{m}) \tag{3.5}$$

Where a is also a scaling factor and $\mathbf{m}$ is the cross product of $\mathbf{q_1}$ and $\mathbf{q_2}$.

Thus from Eq. (3.4) and (3.5):

$$(\mathbf{p_2} - \mathbf{p_1}) + (\lambda_2 \cdot \mathbf{q_2} - \lambda_1 \cdot \mathbf{q_1}) = a \cdot (\mathbf{m}) \quad (3.6)$$

Let:

$$\boldsymbol{v_1} = \mathbf{q_1} \times \mathrm{m} \quad (3.7)$$

$$\boldsymbol{v_2} = \mathbf{q_2} \times \mathrm{m} \quad (3.8)$$

Since the dot product of perpendicular vectors is 0:

$\mathbf{q_1} \cdot \boldsymbol{v_1} = 0$
$\mathbf{q_2} \cdot \boldsymbol{v_2} = 0$
$\mathbf{m} \cdot \boldsymbol{v_1} = 0$
$\mathbf{m} \cdot \boldsymbol{v_2} = 0$

Taking the dot product of $\boldsymbol{v_1}$ and the right hand side and left hand side of Eq. (3.6):

$$(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_1} + (\lambda_2 \cdot \mathbf{q_2} \cdot \boldsymbol{v_1}) = 0 \quad (3.9)$$

$$(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_2} - (\lambda_1 \cdot \mathbf{q_1} \cdot \boldsymbol{v_2}) = 0 \quad (3.10)$$

Thus:

$$\lambda_1 = \frac{(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_2}}{\mathbf{q_1} \cdot \boldsymbol{v_2}} = \frac{(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_2}}{\mathbf{m} \cdot \mathbf{m}}$$

$$\lambda_2 = \frac{(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_1}}{-\mathbf{q_2} \cdot \boldsymbol{v_1}} = \frac{(\mathbf{p_2} - \mathbf{p_1}) \cdot \boldsymbol{v_1}}{\mathbf{m} \cdot \mathbf{m}}$$

Substituting λ_1 and λ_2 back in Eq. (3.2) and (3.3) and finding the midpoint:

$$\mathbf{wo} = \frac{\mathbf{p_2}+\mathbf{p_1}}{2} + \left(\frac{(\mathbf{p_2}-\mathbf{p_1})\cdot\boldsymbol{v_2}}{2\cdot\mathbf{m}\cdot\mathbf{m}}\right)\cdot\boldsymbol{v_1} + \left(\frac{(\mathbf{p_2}-\mathbf{p_1})\cdot\boldsymbol{v_1}}{2\cdot\mathbf{m}\cdot\mathbf{m}}\right)\cdot\boldsymbol{v_2} \quad (3.11)$$

3.4.2. Python Implementation

Eq. (3.11) derived in Section 3.4.1 was implemented in the main process on the central receiver system (blue boxes in figure 7). The values of $\mathbf{p_1}$, $\mathbf{q_1}$, $\mathbf{p_2}$ and $\mathbf{q_2}$ are updated by the parallel process for each Beacon.

The Calculations were carried out by functions written in Cython as it provided much better performance than the Numpy implementations. Figure 34 below shows the results on running the timeit module for 1 million iterations of both implementations. Refer to Appendix I for the detailed code.

```
********************************************************************************
Numpy implementation: 38.013261 seconds
Cython implementation: 2.191473 seconds
********************************************************************************
```

Figure 32: Time taken for Numpy and Cython implementations to run 1 million iteration of 2D-3D projection algorithm.

4. Results and Discussion

Figure 35 shows the complete localisation system all with the type of communication between individual devices. The Logitech C930 webcam is connected to the Central receiver system solely for testing purposes.

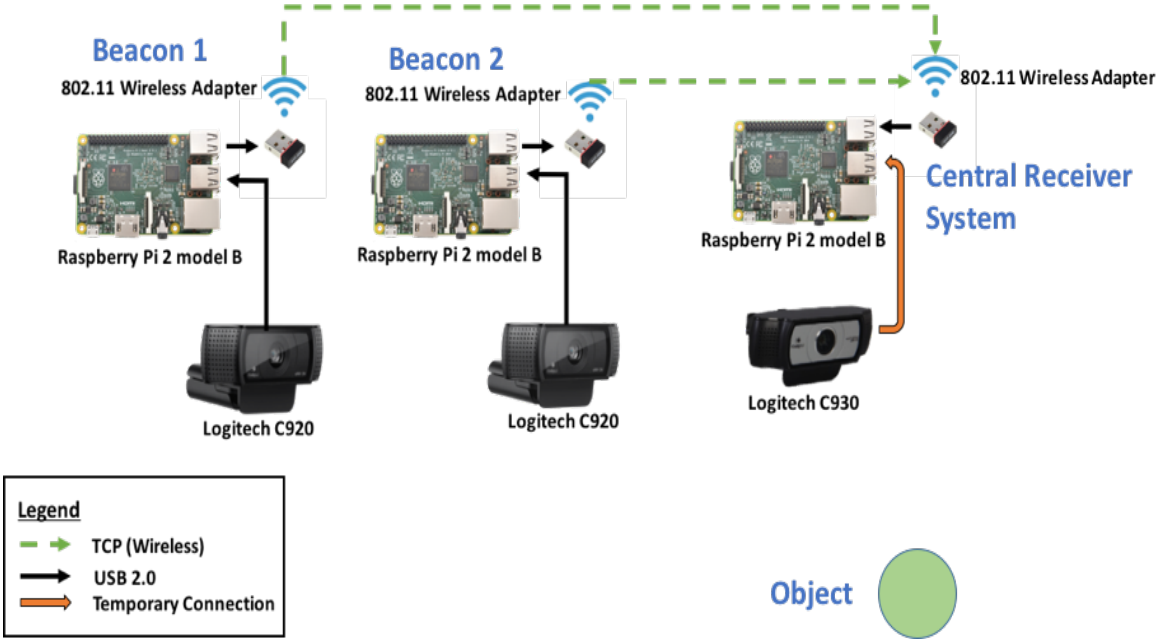

Figure 33: Connection diagram of Completed System

4.1. Reprojection Error

The Vicon system would serve as a great benchmark in order to determine the localisation error. However as described earlier, it was not financially feasible to use the Vicon system. Thus determining the absolute error of the localisation system was a tedious task since the absolute 3D coordinates of the object would have to be measured manually. The reprojection error was calculated instead. As seen in Figure 35, a webcam was connected to the central system. Once the 3D coordinates were determined by the system, these coordinates were projected onto the image plane of this webcam. The detection algorithm was also run on the Central receiver system. The reprojection error is the difference in pixels between the projection result and the detection result. Reprojection error is widely accepted as a good indicator of the accuracy of a localisation system.

4.1.1. Results

Figure 36 shows scatter plots of the reprojection error associated with localisation of the RGB LED and laser dot. The reprojection error is colourmapped according to the estimated distance from the webcam connected to the central system. Figure 37 further illustrates the variation of reprojection error with respect to distance from the camera. The following results were obtained by running the code in Appendix J.

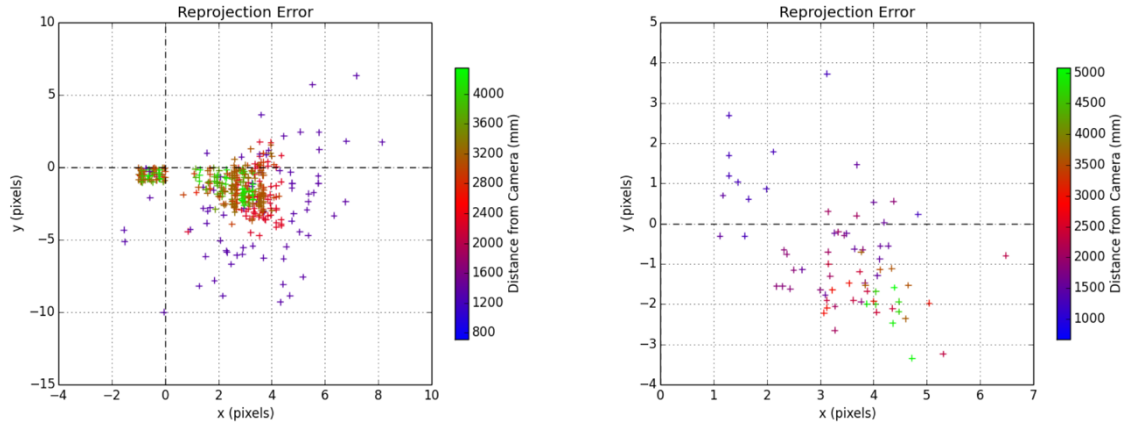

Figure 34: Reprojection error for localisation of the (a) LED (left) and (b) laser dot (right). The plots are colourmapped to represent the distance of the localised point from the camera.

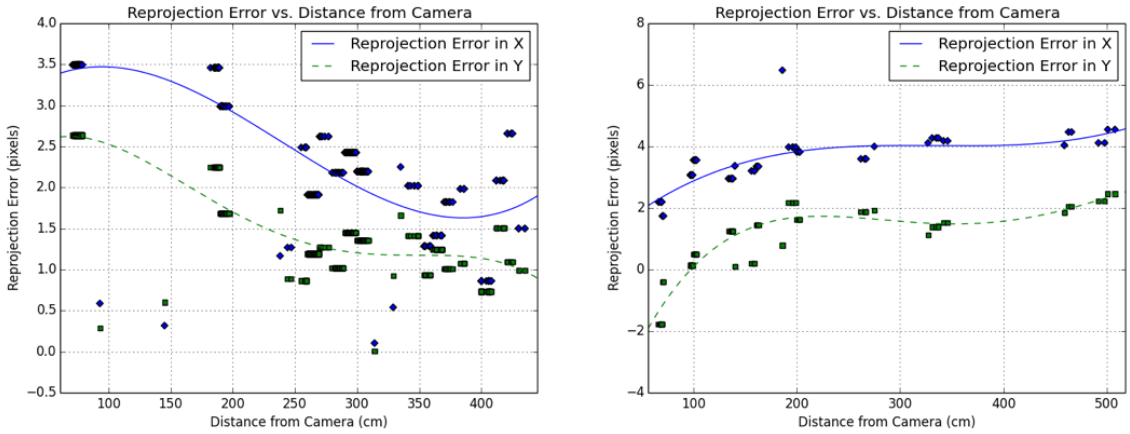

Figure 35: Relationship between reprojection error and distance from camera for localisation of the (a) LED (left) and (b) laser dot (right).

Table 2: Table of results for reprojection error associated with localisation of the RGB LED

Description	Value (pixels)
Average reprojection error in x direction for localisation of RGB LED	2.622
Average reprojection error in y direction for localisation of RGB LED	1.752
Maximum reprojection error in x direction for localisation of RGB LED	8.139
Maximum reprojection error in y direction for localisation of RGB LED	9.992

Table 3: Table of results for reprojection error associated with localisation of the laser dot

Description	Value (pixels)
Average reprojection error in x direction for localisation of laser dot	3.418
Average reprojection error in y direction for localisation of laser dot	1.367
Maximum reprojection error in x direction for localisation of laser dot	6.477
Maximum reprojection error in y direction for localisation of laser dot	3.734

4.1.2. Relationship between Reprojection Error and Absolute Error

In order to understand the significance of reprojection error, we refer to the pinhole camera model (Section 2.1).

Let's assume that the camera coordinate system and the world coordinate system are aligned and at the same position. Thus the **R** matrix is an Identity matrix and all elements of the translation matrix (**t**) are 0. The skew factor is 0 for the cameras used in the proposed system

From Eq. (2.5) and (2.6):

$$u = \frac{-f \cdot k_u \cdot x}{z} + u_0 \tag{4.1}$$

$$v = \frac{-f \cdot k_v \cdot y}{z} + v_0 \tag{4.2}$$

where:

u is the pixel position in the x direction

v is the pixel position in the y direction

The perspective projection of the actual 3D position $[x \;\; y \;\; z]^{\mathrm{T}}$ is $[u \;\; v \;\; 1]^{\mathrm{T}}$ and that of the localised position $[x' \;\; y' \;\; z']^{\mathrm{T}}$ is $[u' \;\; v' \;\; 1]^{\mathrm{T}}$.

The reprojection error in the x and y direction:

$$e_{x\_reprojection} = u - u' \tag{4.3}$$

$$e_{y\_reprojection} = v - v' \tag{4.4}$$

The absolute error in the x and y direction:

$$e_{x\_abs} = x - x' \tag{4.5}$$

$$e_{y\_abs} = y - y' \tag{4.6}$$

From Eq. (4.1) and (4.2):

$$e_{x\_reprojection} = -f \cdot k_u \cdot \left(\frac{x}{z} - \frac{x'}{z'}\right) \tag{4.7}$$

$$e_{y\_reprojection} = -f \cdot k_v \cdot \left(\frac{y}{z} - \frac{y'}{z'}\right) \tag{4.8}$$

The absolute error can be estimated roughly by assuming $z = z'$

$$e_{x\_reprojection} \approx -f \cdot k_u \cdot \left(\frac{e_{x\_abs}}{z'}\right)$$

$$\Longrightarrow e_{x\_abs} \approx z' \cdot \left(\frac{e_{x\_reprojection}}{-f \cdot k_u}\right) \quad (4.9)$$

$$e_{y\_reprojection} \approx -f \cdot k_v \cdot \left(\frac{e_{y\_abs}}{z'}\right)$$

$$\Longrightarrow e_{y\_abs} \approx z' \cdot \left(\frac{e_{y\_reprojection}}{-f \cdot k_v}\right) \quad (4.10)$$

$$\text{The magnitude of the reprojection error } (e_{x\_abs}) = \sqrt{{e_{x\_abs}}^2 + {e_{y\_abs}}^2} \quad (4.11)$$

Using the calibration procedure described in section 3.2.1, the intrinsic parameters of the camera connected to the central system were obtained (Code in Appendix J).

$-f \cdot k_u = 509.39049$

$-f \cdot k_v = 510.01682$

Figure 38 illustrates the relationship between the absolute error and the distance from the camera for the proposed system. The plot is a rough estimate as the localised depth (z') is assumed to be equal to the actual depth (z) which may not be the case most of the time.

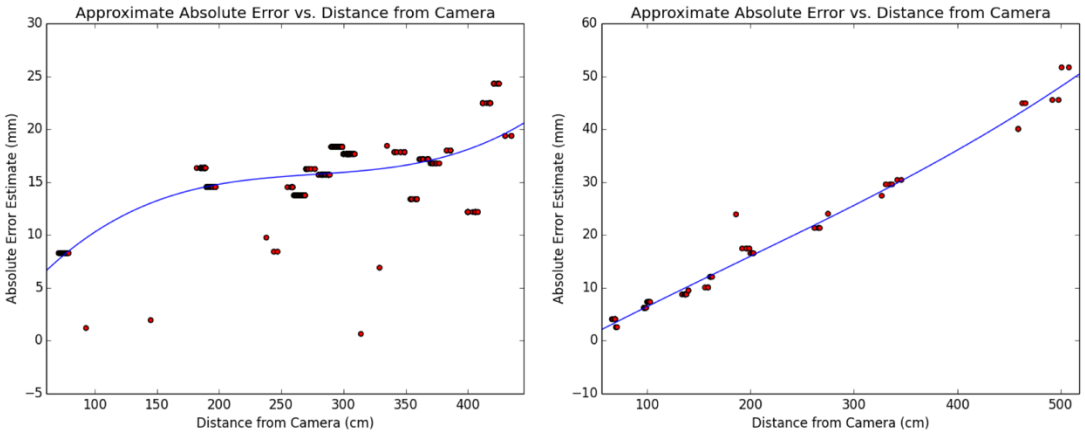

Figure 36: Approximate relationship between absolute error and distance from camera for localisation of the LED (left) and laser dot (right).

4.1.3. Discussion

Distribution of Reprojection error about Coordinate Axes

In Figure 36, the uneven distribution of reprojection error about the coordinate axes in both the plots can be explained by slight errors in the determination of the extrinsic parameters by the calibration process described in Section 3.2.2.

Variation of Reprojection error with distance from the Camera

Figure 37(a) indicates that the reprojection error associated with localisation of the LED decreases as the LED is moved away from the camera. Figure 37(b) indicates that the reprojection error associated with localisation of the laser dot increases when the laser pointer is pointed at a surface further away from the camera. It can be inferred that

reprojection error is dependent on the size of the object being localised. In Figure 33 in Section 3.4, it is shown that the object center as perceived by each camera is different and it is possible that none of the perceived centers is the true object center.

As the object size increases, the number of pixels which represent the object center also increases. Thus there is greater discrepancy between the perceived object centers for each camera and between the perceived centers and true center of the object. The LED appears smaller as it is moved away from the camera whereas the laser dot appears larger when it is pointed at a surface further away. This explains the decreasing trend observed in Figure 37(a) and the increasing trend observed in Figure 37(b).

Variation of Absolute error with distance from the Camera

From equation 4.9 and 4.10, it is clear that the absolute error is directly proportional to distance. In Figure 38(a), the absolute error for localisation of the LED increases less sharply than it does for localisation of the laser dot in Figure 38(b) because of the difference in the variation of reprojection error associated with both processes (seen in Figure 37). The decrease in reprojection error as seen in Figure 37(a) prevents the absolute error of localisation of the LED from increasing rapidly.

The maximum approximate error of localisation for the LED is less than that of the laser despite the maximum reprojection error of the LED being greater than that of the laser because unlike the laser dot, the maximum reprojection error for the LED occurs when the LED is closest to the camera.

4.2. Performance

```
**********************************************************
BEACON 1
Video Capture frame rate: 30 frames/second
Number of frames processed: 1000 frames
Frames in which object is detected:  999  frames
Packets sent:  1001
Frequency of packet transmission: 102.247 Hz
**********************************************************
```

Figure 37: Beacon1 performance results.

```
**********************************************************
BEACON 2
Video Capture frame rate: 30 frames/second
Number of frames processed: 1000 frames
Frames in which object is detected:  988  frames
Packets sent:  992
Frequency of packet transmission: 144.818 Hz
**********************************************************
```

Figure 38: Beacon2 performance results.

```
**********************************************************
CENTRAL RECEIVER
Packets received from Beacon1: 1001
Packets received from Beacon2: 992
2D-3D Projection loops run: 1608
Time taken: 0.64 seconds
Localisation Frequency: 2512.50 Hz
**********************************************************
```

Figure 39: Central receiver Performance results.

Figure 39, 40 and 41 indicate that localisation is performed multiple times for each data packet received from both the Beacons since frequency of localisation is approximately 15 times that of packet transmission. The frequency of packet transmission in turn is much greater than the camera frame rate. Thus the refresh rate of the system is limited by the camera frame rate and not the software being run on the systems. The above results were obtained by running the code in Appendix K.

5. Applications

5.1. Low Cost Navigation Systems for Autonomous Robots

The proposed system can be implemented in two types of navigation systems

Positioning and Error Rectification

Inertial sensors are commonly used for navigation of ground based autonomous robots. However, they suffer from cumulative error – the positioning error at any point in time is dependent on the positioning error at previous points in time. Slight wheel slip can lead to the robot straying off course. The proposed system as is, is capable of performing the active positioning necessary for error correction.

Beacons placed at fixed positions with respect to arbitrarily chosen coordinate axes will localise an LED placed on the robot. The central receiver placed on the robot will output the calculated position to the robot which can then perform corrective actions. Figure 42 below, illustrates this process.

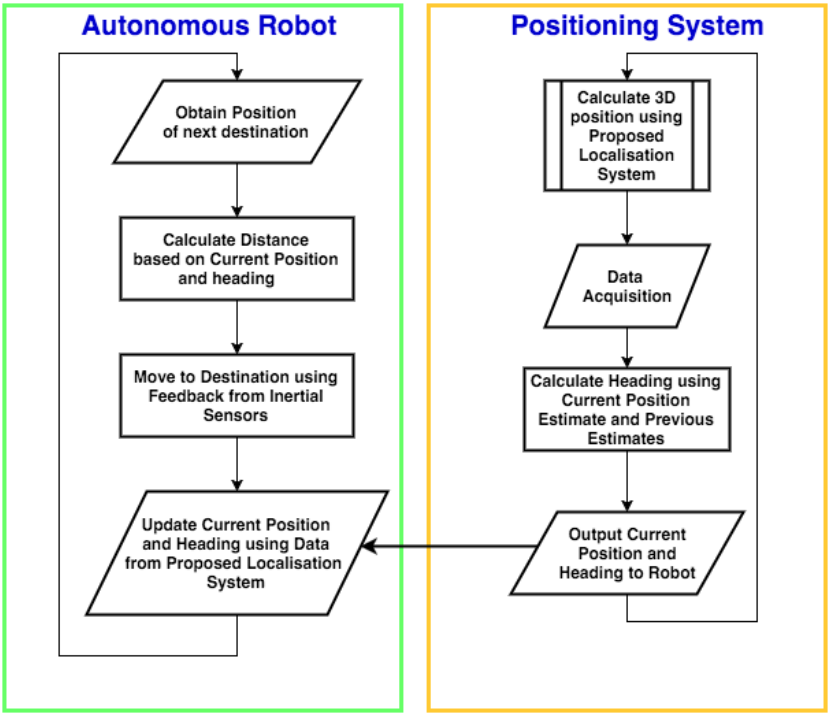

Figure 40: Flowchart of implementation of proposed system for positioning of autonomous robots.

The high speed and accuracy of the proposed system also make it an ideal solution for the navigation of autonomous drones in indoor environments.

Obstacle Avoidance and Path Planning

The localisation system can be used to detect obstacles in a certain direction by pointing the laser in that direction and localising the laser dot produced on the obstacle. This is illustrated in Figure 43. The reference frame **W** is stationary with respect to the robot and both Beacons are placed on the robot.

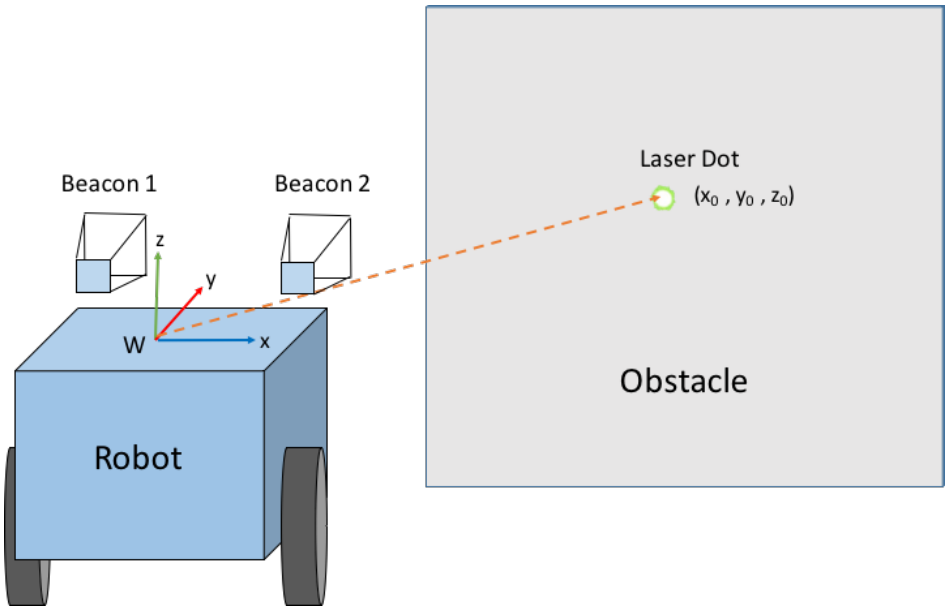

Figure 41: Implementation of proposed system in obstacle detection.

The distance to the obstacle (d) can be calculated as the Euclidean distance to the point (x_0 , y_0, z_0).

$$d = \sqrt{{x_0}^2 + {y_0}^2 + {z_0}^2}$$

By pointing the laser in various directions, the surroundings can be roughly mapped which is instrumental in path planning. This is similar to Light Detection and Ranging (LiDAR) systems which measure time of flight of emitted light to estimate the distance to an object. Obstacle avoidance can be performed in conjunction with positioning as described above to provide a low-cost robust navigation system.

An important health and safety concern is the use of lasers. For commercial applications, use of infra-red pointers and infra-red cameras would be a better option. Also, the proposed system is not suitable for implementation in large open areas.

5.2. 3D Pointer Device

Figure 44 illustrates the use of the localisation system as a 3D pointing device to draw a triangle shaped plane in 3D space. Three dimensional tracking capability enables a user to interface with an operating system or application more efficiently. Common applications of 3D pointing devices are:

1. Gesture Recognition
   - GestTrack3D [29]
2. Gaming
   - Nintendo Wii, Playstation Move
3. Computer aided design
   - 3Dconnexion Spacemouse

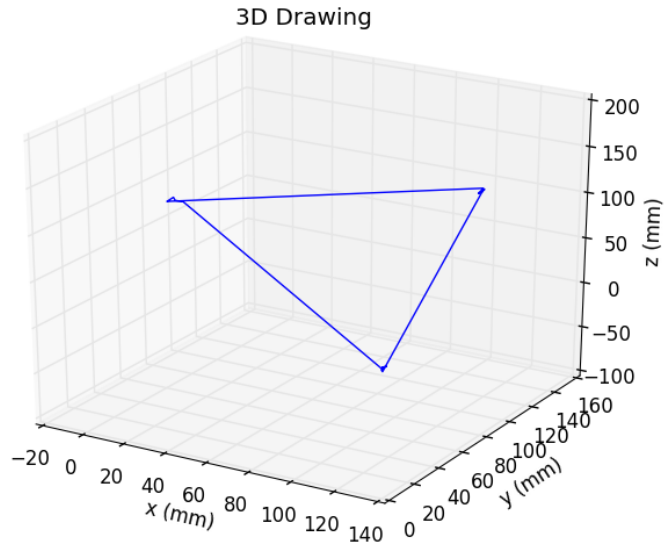

Figure 42: Triangle shaped plane drawn in 3D space using the localisation system.

The system was found to be highly sensitive to slight hand movements. More data processing has to implemented for the system to be used as a fully functional 3D pointing device

6. Future Work

Increasing the number of objects being localised

As described in Section 3.2, the refresh rate of the Beacon system is more than 3 times the frame rate of the camera. Thus instead of detecting one object multiple times in each frame, multiple objects can be detected once per frame. Increasing the number of objects being localised will greatly enhance the system's potential as a pointing device and a low cost navigation system.

Accounting for Image Capture Delay between Cameras

As shown in Figure 45 below, there is a delay (δ) between the shutter timings of the Beacon cameras. While localising fast moving objects, this delay can lead to errors as the 3D position of the object changes during the delay.

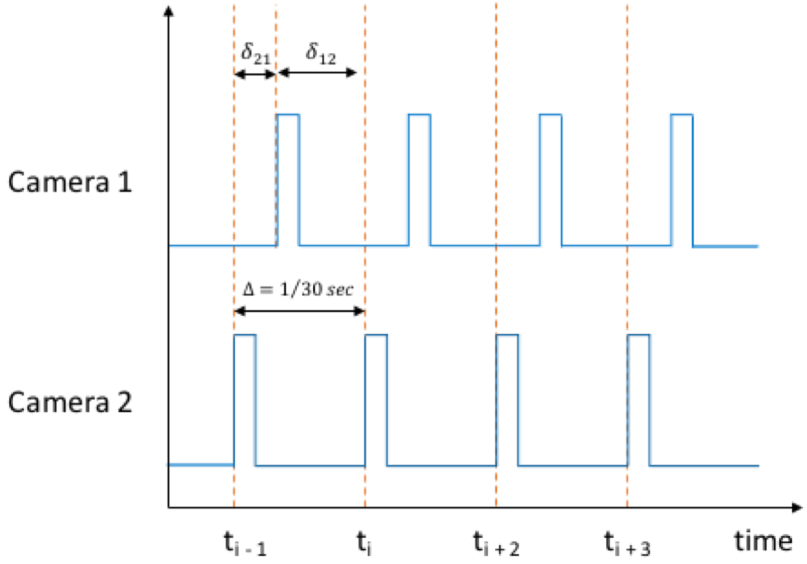

Figure 43: Shutter timings of camera 1 and camera 2. Image capture is represented by the pulses.

This problem can be solved in a number of ways:

Method 1: Synchronising the cameras will produce more accurate results but with a significant reduction in frame rate.

Method 2: Using cameras which capture video at a higher frame rate will reduce the delay.

Method 3: Using more Beacons and hence more cameras will help provide a more accurate estimate and also solve the occlusion problem.

Method 4: Fujiyoshi et al. used a linear estimation model to reduce the error without synchronising the cameras [7].

Methods 2-4 can be implemented in conjunction to provide a more robust device which can be used for localisation of objects moving at a high speed.

Iterative Calibration Processes to Determine the Extrinsic Parameters of the Camera

The iterative refinement process used to localise control points during the calibration for intrinsic parameters (Section 3.2.1) resulted in sub-pixel reprojection error. An iterative refinement process was not used in the calibration for extrinsic parameters of the camera (Section 3.2.2). As described in Section 4.1.3, the spread of the reprojection error around the origin is indicative of slight inaccuracies in the determined extrinsic parameters. Thus implementing an iterative refinement process in the calibration for these parameters may prove to be advantageous in term of accuracy.

7. Conclusion

A vision based indoor localisation system was designed, built and tested. The system consists of two cameras each connected to a Beacon. The Beacons detect the position of an RGB LED or a laser dot in the captured frame and wirelessly communicates this information to the central receiver system. Efficient thresholding techniques were implemented to speed up the detection process. The central receiver system then localises the object using its pixel position data received from the Beacons. All processing and calculations were performed by Raspberry Pis. The total cost of the components used in the system was £192 which is within the £200 budget (Component costs in Appendix L).

The system was designed such that multiple Beacons and hence cameras can be added by just by creating a parallel process, on the central system, to receive data from the new Beacon. It was found that the frequency of detection and localisation were not limited by the detection process or the other calculations but by the camera frame rate (30Hz). Thus the primary objective pertaining to performance was achieved.

The reprojection error was calculated. Reprojection error is widely accepted as a good indicator of the accuracy of a localisation system. For localisation of the LED, maximum reprojection errors of 8.14 pixels in the x direction and 9.99 pixels in the y direction were obtained. For localisation of the laser dot, maximum reprojection errors of 6.48 pixels in the x direction and 3.73 pixels in the y direction were obtained. This reprojection accuracy is acceptable for the proposed system to be implemented as a 3D pointer device and as a navigation system for low cost autonomous robots. It was experimentally proven that the reprojection error decreases as the size of the object being localised decreases.

An approximate relationship between the reprojection error and absolute 3D positioning error was derived. It was demonstrated that the absolute error is directly proportional to the distance of the object from the camera and the reprojection error, the former being the dominating factor. From the experimental results, an estimate of the 3D positioning error was obtained. At a distance of 4.5 m from the camera, connected to the central system, the localisation error for LED was approximately 2.5 cm which is well within the desired accuracy range of 3 cm. At a distance of 5 m from the camera, the localisation error for the laser dot was approximately 5 cm which is not within the desired accuracy range of 3 cm. The accuracy of laser dot localisation can be improved by using a focusing lens which will reduce the size of the laser dot being localised and hence reduce the reprojection error. Section 6 describes more methods to improve the accuracy of the system as a whole.

It is important to note that the primary and secondary objectives could not have been achieved if not for the low-cost and high performance of the Raspberry Pi 2. The Raspberry Pi 3 model B, launched in February 2016, features a 64-bit CPU and an integrated 802.11 wireless LAN. The integrated wireless adapter eliminates the need for an external Wireless

USB adapter. Undoubtedly the advent of this new class of single-board computers provides great scope for improvement of the proposed system.

8. References

[1] J. Brophy-Warren, "Magic wand: How hackers make use of their Wii-motes," in The Wall Street Journal, wsj.com, 2007. [Online]. Available: http://www.wsj.com/articles/SB117772630151685703. Accessed: Feb. 13, 2016.

[2] P. Arcara, L. Di Stefano, S. Mattoccia, C. Melchiorri, and G. Vassura, "Perception of depth information by means of a wire-actuated haptic interface," vol. 4, Proc. of 2000 IEEE Int. Conf. on Robotics and Automation, 2000, pp. 3443–4.

[3] Á. Cassinelli, S. Perrin, and M. Ishikawa, "Smart laser-scanner for 3D human-machine interface," vol. 4, Proc. of 2000 IEEE Int. Conf. on Robotics and Automation, 2000, 2005, pp. 3443–3444.

[4] S. Perrin, et al, "Laser-Based Finger Tracking System Suitable for MOEMS Integration," Image and Vision Computing, New Zealand, 26-28 Nov. 2003, pp. 131-136.

[5] R. Clark, N. Trigoni, and A. Markham, "Robust Vision-based Indoor Localisation," pp. 1–2, 2015.

[6] M. Svedman, L. Goncalves, N. Karlsson, M. Munich, and P. Pirjanian, "Structure from stereo vision using unsynchronised cameras for simultaneous localisation and mapping," 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2006, pp. 3069–3074.

[7] H. Fujiyoshi, S. Shimisu, T. Nishi, Y. Nagasaka, and T. Takahashi, "Fast 3D position measurement with two unsynchronised cameras," vol. 3, Proceedings 2003 lEEE International Symposium on Computational Intelligence in Robotics and Automation, 2003, pp. 1239–3.

[8] P. K. Ghosh and S. P. Mudur, "Three-Dimensional computer vision: A geometric viewpoint," The Computer Journal, vol. 38, no. 1, pp. 85–86, Jan. 1995.

[9] D. Gu, K. S. Ou, Y. T. Fu, and K. S. Chen, "Design of Wiimote indoor localisation technology for omni-directional vehicle trajectory control," vol. 1, IEEE, 2012, pp. 600–604.

[10] J. López, D. Pérez, E. Zalama, and J. Gómez-García-Bermejo, "Low cost indoor mobile robot localisation system," IEEE, 2011, pp. 1134–1139.

[11] J. Maccormick, "How does the Kinect work?,". [Online]. Available: http://users.dickinson.edu/~jmac/selected-talks/kinect.pdf. Accessed: Jan. 2, 2016.

[12] V. M. S. Ltd, "VICON,". [Online]. Available: http://www.vicon.com/visualisation#close. Accessed: Feb. 3, 2016.

[13] "Behance,". [Online]. Available: https://www.behance.net/gallery/16734909/Motion-Capture. Accessed: Feb. 2, 2016.

[14] A. Fusiello, E. Trucco, and A. Verri, "Machine vision and applications A compact algorithm for rectification of stereo pairs," Machine Vision and Applications, vol. 12, pp. 16–22, 2000.

[15] "Bayer filter," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/Bayer_filter. Accessed: Mar. 4, 2016.

[16] P. Cattin, "Digital Image Fundamentals: Introduction to Signal and Image Processing,". [Online]. Available: https://miac.unibas.ch/SIP/02-Fundamentals.html. Accessed: Mar. 4, 2016.

[17] "Color (image processing Toolbox),". [Online]. Available: http://www-rohan.sdsu.edu/doc/matlab/toolbox/images/color11.html. Accessed on: Nov. 16, 2015

[18] R. C. Gonzalez, R. E. Woods, and S. L. Eddins, Digital image processing, 2nd ed. United States: Gatesmark Publishing, 2009.

[19] M. Meško and Š. Toth, "LASER SPOT DETECTION," Journal of Information, Control and Management Systems, vol. 11, no. 1, 2013.

[20] "Look up Tables — OpenCV 2.4.11.0 documentation," in OpenCV 2.4.11, 2011. [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/core/how_to_scan_images/how_to_scan_images.html. Accessed: Oct. 20, 2015.

[21] A. Rosebrock, "Increasing webcam FPS with python and OpenCV," in PyImageSearch, 2015. [Online]. Available: http://www.pyimagesearch.com/2015/12/21/increasing-webcam-fps-with-python-and-opencv/. Accessed: Dec. 23, 2015.

[22] A. Datta, J.-S. Kim, and T. Kanade, "Accurate camera calibration using iterative refinement of control points," Sep. 2009.

[23] D. W. Marquardt, "An algorithm for least-squares estimation of Nonlinear parameters," Journal of the Society for Industrial and Applied Mathematics, vol. 11, no. 2, pp. 431–441, Jun. 1963.

[24] A. Datta, "Robotics institute: Software package for precise camera calibration," in Carnegie Mellon University Robotics Institute. [Online]. Available: http://www.ri.cmu.edu/research_project_detail.html?project_id=617&menu_id=261. Accessed: Sep. 5, 2015.

[25] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," The Plessey Company pic., 1988.

[26] "Feature detection," in OpenCV 2.4.11.0 documentation, 2011. [Online]. Available: http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html. Accessed: Nov. 28, 2015.

[27] "SSH using Linux or Mac OS," in Raspberry Pi documentation. [Online]. Available: https://www.raspberrypi.org/documentation/remote-access/ssh/unix.md. Accessed: Sep. 10, 2015.

[28] "multiprocessing — Process-based 'threading' interface," in Python-2.7.11 Documentation. [Online]. Available: https://docs.python.org/2/library/multiprocessing.html. Accessed: Dec. 2, 2015.

[29] "3D depth sensing/ 3D vision image control system / 3D gesture recognition software," in GestureTek. [Online]. Available: http://www.gesturetek.com/3ddepth/introduction.php. Accessed: Apr. 1, 2016.

9. Appendices

9.1. Appendix A: Image Capture

```
# -*- coding: utf-8 -*-
"""
Capture.py
Author: Roshan Pasupathy
"""
import numpy as np
import cv2
import os


os.system('v4l2-ctl -d 0 -c focus_auto=0')
os.system('v4l2-ctl -d 0 -c focus_absolute=0')
os.system('v4l2-ctl -d 0 -c exposure_auto=1')
os.system('v4l2-ctl -d 0 -
c exposure_absolute=3') #Change to 150 for high exposure
os.system('v4l2-ctl -d 0 -c contrast=100')
os.system('v4l2-ctl -d 0 -c brightness=100')
os.system('v4l2-ctl -d 0 -c white_balance_temperature_auto=0')
os.system('v4l2-ctl -d 0 -c white_balance_temperature=6500')

cap = cv2.VideoCapture(0)
cap.set(cv2.cv.CV_CAP_PROP_FPS, 30)
b = cap.get(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)
c = cap.get(cv2.cv.CV_CAP_PROP_FRAME_WIDTH)
print b,c
i = 0
while ( i < 1): #Capture only one image
    ret,frame = cap.read() #Read Frame
    cv2.imshow('frame',frame) #Display Captured Frame, Note* GUI needed
    if cv2.waitKey(1) & 0xFF == ord('c'): #Press C to capture image
        stringval = '/home/pi/ip/report/satimg' + str(i) +'.bmp'
        cv2.imwrite(stringval,frame) #Save image
        print 'img' + str(i) + ' taken and saved to camera 3 calibrationimages1'

        i += 1
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break


cap.release()
cv2.destroyAllWindows()
```

9.2. Appendix B: Object Data Extraction

Main Script – Object_Dat.py

```
# -*- coding: utf-8 -*-
"""
Object_Dat.py
Created on Wed Dec 16 13:16:07 2015
Author: Roshan Pasupathy
This Script is used for:
- Extracting Object Data
- Plotting the 3D graph for hue,saturation and intensity
  object and background
- Plotting the frequency of occurence of hue values to
  determine hue ranges
- Plotting the relationship if any between the satration
  and intensity of the object
"""
##Import modules
import numpy as np
```

```
17. import cv2
18. import os
19. from LUTptrall import bgrhsvarrayl
20. from LUTptrall import bgrhsvarraylc
21. from LUTptrall import bgrhsvarray3
22. from LUTptrall import cleanupf
23. from pylab import *
24. from mpl_toolkits.mplot3d import Axes3D
25. from matplotlib import pyplot as plt
26. 
27. #Set camera parameters
28. os.system('v4l2-ctl -d 0 -c focus_auto=0')
29. os.system('v4l2-ctl -d 0 -c focus_absolute=0')
30. os.system('v4l2-ctl -d 0 -c exposure_auto=1')
31. os.system('v4l2-ctl -d 0 -c exposure_absolute=3')
32. os.system('v4l2-ctl -d 0 -c contrast=100')
33. os.system('v4l2-ctl -d 0 -c brightness=100')
34. os.system('v4l2-ctl -d 0 -c white_balance_temperature_auto=0')
35. os.system('v4l2-ctl -d 0 -c white_balance_temperature=6500')
36. 
37. #Decalre flags
38. picturestaken = 0 #checks number of pictures calibrated
39. calibdone = False #status of calibration per picture
40. calibwindowopen = False #status of calibration window
41. running = True # status of webcam loop. False = terminate
42. scene = False #Is the selection window being displayed or video window
43. 
44. #declare calibration parameters
45. colourfreq = np.zeros((256,256,256))
46. colourscene = np.zeros((256,256,256))
47. 
48. #region of interest parameters and flags
49. rect = (0,0,1,1) #coordinates of corners
50. rectangle = False
51. rect_over = False
52. 
53. def onmouse(event,x,y,flags,params):
54.     """this function is called on mouse click. it plots colour,saturation and in
    tensity for slected region of interest"""
55.     # Declare global objects
56.     global sceneImg,backimage,rectangle,rect,ix,iy,rect_over, colourfreq,picture
    staken,scene,calibwindowopen
57.     #Copy img
58.     sceneCopy = sceneImg.copy()
59.     # Draw Rectangle
60.     if event == cv2.EVENT_LBUTTONDOWN:
61.         rectangle = True
62.         ix,iy = x,y
63. 
64.     elif event == cv2.EVENT_MOUSEMOVE:
65.         if rectangle == True:
66.             cv2.rectangle(sceneCopy,(ix,iy),(x,y),(0,255,0),1)
67.             cv2.imshow('mouse input', sceneCopy)
68.             cv2.waitKey(1)
69. 
70.     elif event == cv2.EVENT_LBUTTONUP:
71.         rectangle = False
72.         rect_over = True
73. 
74.         # Draw rectangle in copy
75.         cv2.rectangle(sceneCopy,(ix,iy),(x,y),(0,255,0),1)
76.         cv2.imwrite('/home/pi/ip/report/selectionLaser.jpg',sceneCopy)
77. 
78. 
79.         rect = (min(ix,x),min(iy,y),abs(ix-x),abs(iy-y))
80.         # Diffimage is binary
81.         diffimage = np.asarray(bgrhsvarrayl(backimage,sceneImg,rect[1],rect[1]+r
    ect[3],rect[0],rect[0]+rect[2],50.0))
```

```
82.         filteraxis = np.asarray(bgrhsvarraylc(diffimage,sceneImg,rect[1],rect[1]
    +rect[3],rect[0],rect[0]+rect[2],50.0))
83.         #colourscene contains the frequency of occurence of h,s,v of  Background
     pixels
84.         colourfreq += filteraxis
85.         #Save diff image
86.         cv2.imwrite('/home/pi/ip/report/diffLaser.jpg',diffimage)
87.         #Display copy with rectangle for 4 second
88.         cv2.imshow('mouse input', diffimage)
89.         cv2.waitKey(4000)
90.
91.         picturestaken += 1
92.         print rect
93.         print "%d picturestaken"%(picturestaken)
94.         scene = False
95.         cv2.destroyWindow('mouse input')
96.         calibwindowopen = False
97.
98.
99. # Named window and mouse callback
100.       cv2.namedWindow('video')
101.       # Start video stream
102.       cap = cv2.VideoCapture(0)
103.       keyPressed = None
104.
105.       # Print instructions
106.       print "Press b to capture background. Press o to obtain laser colour val.
     Press s to obtain colours of background"
107.
108.       while running:
109.           readOK, frame = cap.read()
110.           calibdone = False
111.           keyPressed = cv2.waitKey(5)
112.
113.           if keyPressed == ord('b'):
114.               backimage = frame
115.               #Capture Reference Image
116.               cv2.imwrite('/home/pi/ip/report/referenceLaser.jpg',frame)
117.               print "Background image taken"
118.
119.           if keyPressed == ord('o'):
120.               print "you pressed o. Please wait"
121.               scene = True
122.               cv2.destroyWindow('video')
123.               print "Select object of interest"
124.               sceneImg = frame.copy()
125.               cv2.imshow('mouse input', sceneImg)
126.
127.           if keyPressed == ord('s'):
128.               print "You pressed s. Please wait"
129.               Backgroundval = frame.copy()
130.               filterscene = np.asarray(bgrhsvarray3(Backgroundval))
131.               #colourscene contains the frequency of occurence of h,s,v of  Bac
    kground pixels
132.               colourscene += filterscene
133.               cv2.imshow('video', Backgroundval[0:480,0:640])
134.               cv2.waitKey(1000)
135.               print "background colours have been checked"
136.
137.           if not calibwindowopen:
138.               cv2.namedWindow('mouse input')
139.               cv2.setMouseCallback('mouse input',onmouse)
140.               calibwindowopen = True
141.           if not scene:
142.               cv2.imshow('video', frame)
143.
144.           if picturestaken == 3:
145.               running = False
146.
```

```
147.        fig = plt.figure(figsize = (8,6))
148.        ax = fig.add_subplot(111,projection = '3d')
149.        ax.grid(True)
150.
151.        #Plot 3D graph
152.
153.        #Object pixels
154.        #hue, saturation and value for which frequency > 0
155.        interestpos = np.nonzero(colourfreq)
156.        xo = interestpos[0] #Hue
157.        yo = interestpos[1] #Saturation
158.        zo = interestpos[2] #Intensity
159.
160.        #Colourmap according to frequency of occurence
161.        colval = colourfreq[interestpos]
162.        colors = cm.winter(colval/max(colval))
163.        colmap = cm.ScalarMappable(cmap = cm.winter)
164.        colmap.set_array(colval/max(colval))
165.
166.        #Background pixels
167.        #hue, saturation and value for which frequency > 0
168.        scenepos = np.nonzero(colourscene)
169.        xs = scenepos[0] #Hue
170.        ys = scenepos[1] #Saturation
171.        zs = scenepos[2] #Intensity
172.
173.        #Colourmap according to frequency of occurence
174.        colvals = colourscene[scenepos]
175.        colours2 = cm.spring(colvals/max(colvals))
176.        colmap2 = cm.ScalarMappable(cmap = cm.spring)
177.        colmap2.set_array(colvals)
178.
179.        ax.scatter(xo,yo,zo, c=colors, marker='o',label='Laser Dot ')
180.        ax.scatter(xs,ys,zs, c=colours2, marker='s',label='Background')
181.
182.        cb = fig.colorbar(colmap,shrink=0.75)
183.        cb.set_label('Laser Dot')
184.        cb2 = fig.colorbar(colmap2,shrink=0.75)
185.        cb2.set_label('Background')
186.
187.        ax.set_xlabel('Hue')
188.        ax.set_ylabel('Saturation')
189.        ax.set_zlabel('Intensity')
190.
191.        #########
192.
193.        fig1 = plt.figure(1)
194.        ax1 = fig1.add_subplot(111)
195.        ax1.set_title('Frequency of Occurence of Hues')
196.
197.        huefreq = np.sum(np.sum(colourfreq,axis=2),axis=1) #Frequency array
198.        ax1.set_xlim([0,180])
199.        ax1.set_ylim([0,max(huefreq) + 20])
200.        ax1.bar(np.arange(256) - 0.5,huefreq, width=1.0, color='b')
201.        ax1.set_xticks(np.arange(0,181,20))
202.        ax1.set_xlabel('Hue (0-180)')
203.        ax1.set_ylabel('Frequency (pixels)')
204.
205.        #########
206.        #Plot Saturation vs. Intensity
207.        plt.legend(loc='upper left')
208.        fig2 = plt.figure(2)
209.        ax2 = fig2.add_subplot(111)
210.        ax2.grid(True)
211.        hueaccept1 = colourfreq[:,:,:]
212.        hueaccept = np.nonzero(hueaccept1)
213.        satdata = hueaccept[1]
214.        valdata = hueaccept[2]
```

```
215.
216.        #Colourmap according to frequency of occurence
217.        densityfunc = hueaccept1[hueaccept]
218.        colours3 = cm.cool(densityfunc/max(densityfunc))
219.        colmap3 = cm.ScalarMappable(cmap = cm.cool)
220.        colmap3.set_array(densityfunc)
221.
222.        ax2.scatter(satdata,valdata, c=colours3,marker='o')
223.        cb3 = fig2.colorbar(colmap3,shrink=0.75)
224.        ax2.set_title('Intenisty vs. Saturation')
225.        cb3.set_label('frequency')
226.        ax2.set_xlabel('Saturation')
227.        ax2.set_ylabel('Intensity')
228.
229.        #######
230.        cleanupf() #Clear Memory
231.        cv2.destroyAllWindows()
232.        cap.release()
233.        plt.show()
```

Cython Extension - LUTptrall.pyx

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  LUTptrall.pyx
4.  Created on Wed Dec 16 2015
5.  Author: Roshan Pasupathy
6.  Extension file for Object_Dat.py
7.  """
8.
9.  @cython.boundscheck(False)
10. @cython.cdivision(True)
11. @cython.wraparound(False)
12. def bgrhsvarray3(unsigned char[:,:,::1] img_ptr,long x=480,long y=640):
13.     """Finds the frequency of occurency of a set of hue, saturation and Intensit
    y
14.     values in the BGR input image - img_ptr"""
15.     cdef:
16.         #Create empty array
17.         unsigned long[:,:,::1] colptr = np.zeros((256,256,256),dtype = np.uint32
    )
18.         long x0,y0
19.         double b,g,r
20.         long hue,saturation,val
21.         double chroma
22.     for x0 in range(x):
23.         for y0 in range(y):
24.             # Algorithm for converting BGR to HSV
25.             b = img_ptr[x0,y0,0]
26.             g = img_ptr[x0,y0,1]
27.             r = img_ptr[x0,y0,2]
28.             K = 0
29.             if g < b:
30.                 g,b = b,g
31.                 K = -6
32.             if r < g:
33.                 r,g = g,r
34.                 K =  -K - 2
35.             chroma = r - min(g,b)
36.             if chroma != 0:
37.                 hue = int(30 * abs(K +((g-b)/(chroma))))
38.             if r != 0:
39.                 saturation = int(255* chroma/(r * 1.0))
40.             else:
41.                 saturation = 0
42.             val = int(r)
43.             #Increase freuency for that set of hue, saturation and intensity
44.             colptr[hue,saturation,val] += 1
```

```
    return colptr


@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
def bgrhsvarrayl(unsigned char[:,:,::1] backi, unsigned char[:,:,::1] inputi,lon
g xmin,long xmax,long ymin, long ymax,double thresh):
    cdef:
        """Calculate the difference image between a reference image backi and th
e input image inputi"""
        #Channel difference between images
        unsigned char[:,:,::1] diffimage = cv2.absdiff(np.asarray(backi[xmin:xma
x,ymin:ymax]),np.asarray(inputi[xmin:xmax,ymin:ymax]))
        unsigned char* img_ptr = &diffimage[0,0,0]

        long deltax = xmax - xmin
        long deltay = ymax - ymin
        unsigned char pix0,pix1,pix2

        unsigned char[:,::1] outarray = np.zeros((deltax,deltay), dtype = np.uin
t8)
        unsigned char* outptr = &outarray[0,0]

        double normval

    for x0 in range(deltax):
        for y0 in range(deltay):
            pix0 = img_ptr[3*(x0*deltay + y0)]
            pix1 = img_ptr[3*(x0*deltay + y0)+1]
            pix2 = img_ptr[3*(x0*deltay + y0)+2]
            normval = ((pix0**2.0)+(pix1**2.0)+(pix2**2.0))**0.5
            if normval > thresh:
                outptr[x0*deltay + y0] = 255
            else:
                outptr[x0*deltay + y0] = 0
    return outarray


@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
def bgrhsvarraylc(unsigned char[:,::1] diffimage, unsigned char[:,:,::1] inputi,
long xmin,long xmax,long ymin, long ymax,double thresh):
    """Finds the frequency of occurency of a set of hue, saturation and Intensit
y
    values in the BGR input image - inputi, for pixels shown in white in diffima
ge"""
    cdef:
        unsigned char* diff_ptr = &diffimage[0,0]
        unsigned char* img_ptr = &inputi[0,0,0]

        unsigned long[:,:,::1] colourscat = np.zeros((256,256,256),dtype = np.ui
nt32)
        unsigned long* colptr = &colourscat[0,0,0]

        long x0,y0
        double b,g,r
        long hue,saturation
        double chroma

        long deltax = xmax - xmin
        long deltay = ymax - ymin

           for x0 in range(deltax):
               for y0 in range(deltay):
                   if diff_ptr[x0*deltay + y0] == 255:
                       b = img_ptr[3*((xmin +x0)* 640 + y0 + ymin)]
```

```
                        g = img_ptr[3*((xmin +x0)* 640 + y0 + ymin) + 1]
                        r = img_ptr[3*((xmin +x0)* 640 + y0 + ymin) + 2]
                        # Algorithm for converting BGR to HSV
                        K = 0
                        if g < b:
                                g,b = b,g
                                K = -6
                        if r < g:
                                r,g = g,r
                                K =  -K - 2
                        chroma = r - min(g,b)
                        if chroma != 0:
                                hue = int(30 * abs(K +((g-b)/(chroma))))
                        if r != 0:
                                saturation = int(255* chroma/(r * 1.0))
                        else:
                                saturation = 0
                        #Increase freuency for that set of hue, saturation and in
    tensity
                        colptr[(256*256*hue) + (256*saturation)+int(r)] += 1
            return colourscat
```

9.3. Appendix C: Saturation Threshold

```
# -*- coding: utf-8 -*-
"""
Sat_Thresh.py
Created on Dec 20th 2015
Author: Roshan Pasupathy
This Script is used for determining
if the saturation filter is necessary
"""
import numpy as np
import cv2

#Read BGR image
img = cv2.imread('/home/pi/ip/input_img.bmp')
#Convert from BGR to HSV
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

#Create empty binary image
nonsat = np.zeros((480,640),dtype=np.uint8)
sat = np.zeros((480,640),dtype=np.uint8)

#Find indices where threshold (excluding saturation threshold) is satisfied
ind = np.where((hsv[:,:,0] >= 100) * (hsv[:,:,0] <= 140) * (hsv[:,:,2] >= 70))
#Find indices where threshold (including saturation threshold) is satisfied
inds = np.where((hsv[:,:,0] >= 100) * (hsv[:,:,0] <= 140) * (hsv[:,:,1] >= 120)
    * (hsv[:,:,2] >= 70))

#Make pixels at object indices 255 i.e white
nonsat[ind] = 255
sat[inds] = 255

#Save Thresholded image
cv2.imwrite('/home/pi/ip/wo_sat_thresh.bmp',nonsat)
cv2.imwrite('/home/pi/ip/w_sat_thresh.bmp',sat)
```

9.4. Appendix D: Line Fitting

Main script - line_plot.py

```
# -*- coding: utf-8 -*-
"""
line_plot.py
Created on Dec 28th 2015
Author: Roshan Pasupathy
This script was included in the Object_Dat.py script
"""
#Import Libraries
from LUTptrall import lineplotter
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot as plt

#Data
hueaccept1 = colourfreq[:,:,:]
hueaccept = np.nonzero(hueaccept1)
satdata = hueaccept[1]
valdata = hueaccept[2]

#Colourmap according to frequency of occurence
densityfunc = hueaccept1[hueaccept]
colours3 = cm.cool(densityfunc/max(densityfunc))
colmap3 = cm.ScalarMappable(cmap = cm.cool)
colmap3.set_array(densityfunc)

##########
fig4 = plt.figure(4)
ax4 = fig4.add_subplot(111)
ax4.grid(True)
ax4.set_xlim([0,255])
ax4.set_ylim([0,255])

#obtain equations of lmin and lamx
slopes = lineplotter(satdata,valdata,30)
print 'slopes =', slopes

#Subtract 10 from intercept of lmin
slopes[1] = slopes[1] - 10

#Add 10 to intercept of lmax
slopes[3] = slopes[3] + 10

#lmin points
x0min = 0
x1min = -slopes[1]/(1.0 * slopes[0])
y0min = slopes[1]
y1min = 0

#lmax points
x0max = 0
x1max = -slopes[3]/(1.0 * slopes[2])
y0max = slopes[3]
y1max = 0

#Plot scatter
ax4.scatter(satdata,valdata, c=colours3,marker='o')
#Plot lmin
ax4.plot([x0min,x1min],[y0min,y1min],'-g')
#Plot lmax
ax4.plot([x0max,x1max],[y0max,y1max],'-g')

#Colourbar
cb4 = fig4.colorbar(colmap3,shrink=0.75)
```

```
cb4.set_label('frequency')

ax4.set_title('Intenisty vs. Saturation')
ax4.set_xlabel('Saturation')
ax4.set_ylabel('Intensity')
```

Cython Extension – lineplotter() (LUTptrall.pyx)

```
# -*- coding: utf-8 -*-
"""
This script is part of the LUTptrall.pyx script
Created on Wed Dec 16 2015
Author: Roshan Pasupathy
The lineplotter function is described
"""

@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
def lineplotter(np.ndarray saturation, np.ndarray val, int valthresh):
    assert saturation.dtype == np.int and val.dtype == np.int
    cdef:
      #Find data points which have intensity greater than an intensity threshold

      np.ndarray newsat = saturation[np.where(val[:] >= valthresh)]
      np.ndarray newval = val[np.where(val[:] >= valthresh)]
      #Add stauration and intensity for each point
      np.ndarray carray = newsat + newval
      #create slope array for lmin and lmax
      np.ndarray outputmin = 2.0 * np.ones(len(newsat),dtype = np.float64)
      np.ndarray outputmax = 2.0 * np.ones(len(newsat),dtype = np.float64)
      #Calculate first point on lmin and lmax
      int posmin = np.where(carray == min(carray))[0][0]
      int posmax = np.where(carray == max(carray))[0][0]

      int i, maxlinepos,minlinepos
      np.ndarray outarray = np.zeros(4, dtype = np.float64)
    for i in range(len(carray)):
      #update slope array for every point in data array
      if newsat[i] != newsat[posmin]:
        outputmin[i] = abs(((newval[i] - newval[posmin])/(1.0*(newsat[i] - newsa
t[posmin]))) + 1.0 )
      if newsat[i] != newsat[posmax]:
        outputmax[i] = abs(((newval[i] - newval[posmax])/(1.0*(newsat[i] - newsa
t[posmax]))) + 1.0 )
    #Calculate second point on lmax
    maxlinepos = np.where(outputmax == min(outputmax))[0][0]
    outarray[2] = (newval[maxlinepos] - newval[posmax])/(1.0 * (newsat[maxlinepo
s] - newsat[posmax]))
    outarray[3] = (newsat[maxlinepos]*newval[posmax] - newval[maxlinepos]*newsat
[posmax])/(1.0 * (newsat[maxlinepos] - newsat[posmax]))
    #Calculate second point on lmin
    minlinepos = np.where(outputmin == min(outputmin))[0][0]
    outarray[0] = (newval[minlinepos] - newval[posmin])/(1.0 * (newsat[minlinepo
s] - newsat[posmin]))
    outarray[1] = (newsat[minlinepos]*newval[posmin] - newval[minlinepos]*newsat
[posmin])/(1.0 * (newsat[minlinepos] - newsat[posmin]))
    return outarray
```

9.5. Appendix E: Detection Performance

Generic Algorithm Detection Script

```
# -*- coding: utf-8 -*-
"""
detection_numpy.py
Author: Roshan Pasupathy
This script runs the detection algorithm for the LED
"""
#import libraries
import numpy as np
import cv2
import time

iterations = 1

#read image from disk onto stack memory
img = cv2.imread('/home/pi/ip/report/readimg1.bmp')
img1 = img.copy()

#Initialise output array
output = np.zeros(4)

#start timer
start = time.clock()
while iterations <= 300:
  #Convert from BGR to HSV
  hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
  #Find indices where threshold (excluding saturation threshold) is satisfied
  ind = np.where((hsv[:,:,0] >= 100) * (hsv[:,:,0] <= 140) * (hsv[:,:,1] >= 120) * (hsv[:,:,2] >= 70))
  output[0] = min(ind[0])
  output[1] = max(ind[0])
  output[2] = min(ind[1])
  output[3] = max(ind[1])
  iterations += 1
  #Draw enclosing rectangle. Comment out lines 34 - 39 for performance testing
  cv2.rectangle(img,(output[2],output[0]),(output[3],output[1]),(0,255,0),2)
  #Show detection result
  cv2.imshow('frame',img)
  if cv2.waitKey(1) & 0xFF == ord('c'): #save image if 'c' is pressed
      cv2.imwrite('/home/pi/ip/input_img.bmp',img1)
      cv2.imwrite('/home/pi/ip/detected.bmp',img)
timetaken = time.clock() - start
#stop timer

print '*' * 80
print 'Number of frames: 300'
print 'Time taken: %.4f seconds'%(timetaken)
print 'Number of frames processed per second: %.2f frames'%(300/timetaken)
print '*' * 80
```

Improved Algorithm Detection Script

Detection_cython.py

```
"""
detection_cython.py
Author: Roshan Pasupathy
This script runs the detection algorithm for the LED
"""
#import libraries
import numpy as np
import cv2
import time
from LUTptrall import squarelut8
```

```
11. from LUTptrall import cleanupf
12. 
13. iterations = 1
14. 
15. #read image from disk onto stack memory
16. img = cv2.imread('/home/pi/ip/report/readimg1.bmp')
17. img1 = img.copy()
18. 
19. #Initialise output array
20. output = np.array([0,640,0,480,0,480])
21. 
22. #start timer
23. start = time.clock()
24. while iterations <= 300:
25.     output = np.asarray(squarelut8(output,480,640,10,img[output[4]:output[5],:,:
    ]))
26.     iterations += 1
27.     #Draw enclosing rectangle, comment out lines 28 - 33 for performance testing

28.     cv2.rectangle(img,(output[0],output[2]),(output[1],output[3]),(0,255,0),2)
29.     #Show detection result
30.     cv2.imshow('frame',img)
31.     if cv2.waitKey(1) & 0xFF == ord('c'): #save image if 'c' is pressed
32.         cv2.imwrite('/home/pi/ip/input_img.bmp',img1)
33.         cv2.imwrite('/home/pi/ip/detected.bmp',img)
34. timetaken = time.clock() - start
35. #stop timer
36. 
37. print '*' * 80
38. print 'Number of frames: 300'
39. print 'Time taken: %.4f seconds'%(timetaken)
40. print 'Number of frames processed per second: %.2f frames'%(300/timetaken)
41. print '*' * 80
42. cleanupf()
```

LUTptrall.pyx – tablegenlaser(), tablegenLED() and squarelut8()

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  This is a part of the LUTptrall.pyx script
4.  Author: Roshan Pasupathy
5.  This script shows the creation and rendering
6.  of the Look Up Table
7.  """
8.  #Import libraries
9.  cimport cython
10. import numpy as np
11. cimport numpy as np
12. import cv2
13. from libc.stdlib cimport calloc,free
14. from cpython.mem cimport PyMem_Malloc as malloc
15. from cpython.mem cimport PyMem_Free as freep
16. 
17. # Initialises LUT. Creates pointer to memory stack and initiates all values to 0

18. cdef bint *tablelut_ptr = <bint *>calloc(256*256*256,sizeof(bint))
19. 
20. @cython.boundscheck(False)
21. @cython.cdivision(True)
22. @cython.wraparound(False)
23. cdef void tablegenlaser(long hmin,long hmax,double m1, double c1, double m2, dou
    ble c2, double thresh):
24.     """function for rendering the LUT for laser detection"""
25.     cdef:
26.         long x0,y0,z0
27.         double b,g,r,saturation
28.         int K
29.         double chroma
```

```
        long hue
    global tablelut_ptr
    #Run thresholding for every set of pixels
    for x0 in range(256):
        for y0 in range(256):
            for z0 in range(256):
                #Convert BGR to HSV
                b,g,r = x0,y0,z0
                K = 0
                if g < b:
                    g,b = b,g
                    K = -6
                if r < g:
                    r,g = g,r
                    K =  -K - 2
                chroma = r - min(g,b)
                if r != 0:
                    saturation = 255 * chroma/(r * 1.0)
                else:
                    saturation = 0.0
                if chroma != 0:
                    hue = int(30 * abs(K +((g-b)/(chroma))))
                    if (hue>= hmin) & (hue <= hmax) & (r >= thresh):
                        if (r >=  (m1 * saturation) + c1) & (r <=  (m2 * saturat
ion) + c2):
                            #If pixel combination satisfies threshold then m
ake it 1
                            tablelut_ptr[(256*256*x0) + (256*y0)+z0] = 1

@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
cdef void tablegenLED(long hmin,long hmax,long smin, long smax, long v, long vma
x):
    """function for rendering the LUT for LED detection"""
    cdef:
        long x0,y0,z0
        double b,g,r
        int K
        double chroma
        long hue, saturation
    global tablelut_ptr
    #Run thresholding for every set of pixels
    for x0 in range(256):
        for y0 in range(256):
            for z0 in range(256):
                #Convert BGR to HSV
                b,g,r = x0,y0,z0
                K = 0
                if g < b:
                    g,b = b,g
                    K = -6
                if r < g:
                    r,g = g,r
                    K =  -K - 2
                chroma = r - min(g,b)
                if r != 0:
                    saturation = int(255 * chroma/(r * 1.0))
                else:
                    saturation = 0
                if chroma != 0:
                    hue = int(30 * abs(K +((g-b)/(chroma))))
                    if (hue>= hmin) & (hue <= hmax) & (r >=v) & (r <= vmax) & (s
aturation >= smin) & (saturation <= smax):
                        #If pixel combination satisfies threshold then make it 1

                        tablelut_ptr[(256*256*x0) + (256*y0)+z0] = 1

```

```
93. #Render LUT
94. #tablegenlaser(40,80,-1.0,100.0,-
    0.651162,242.7906,40.0) #uncomment for laser detection
95. tablegenLED(100,140,20,255,70,255) #uncomment for LED detection
96. 
97. @cython.boundscheck(False)
98. @cython.cdivision(True)
99. @cython.wraparound(False)
100.        def squarelut8(int[::1] output,int x, int y,unsigned char v,unsigned char
    [:,:,::1] image):
101.            """Returns the pixel position of the object of interest and the regio
    n of interest for
102.            the next detection"""
103.            cdef:
104.                #Create pointer to outputarray
105.                int* outputptr = &output[0]
106.                unsigned char* img_ptr = &image[0,0,0]
107. 
108.                #Range of pixels in y direction which could contain the object
109.                int yminscan = (outputptr[0]<= outputptr[1])*((v+1)*outputptr[0]
    > (v*outputptr[1]))*((v + 1)*outputptr[0] - (v*outputptr[1]))
110.                int ymaxscan = ((outputptr[0]<= outputptr[1])*((v+1)*(outputptr[1
    ]+1) < y + (v*outputptr[0]))*(((v+1)*(outputptr[1]+1)) - (v*outputptr[0]) - y))
    + y
111.                int deltay = ymaxscan-yminscan
112.                #Range of pixels in x direction which could contain the object
113.                int deltax = outputptr[5] - outputptr[4]
114. 
115.                #Increment variable which is fed the LUT value
116.                bint inc
117.                bint xpos = 0
118. 
119.                #pointer to LUT. Read only
120.                bint *tablelut_ptr1=tablelut_ptr
121. 
122.                #Allocate memory to x array
123.                int *x_outptr =<int *>malloc(2*sizeof(int))
124.                int *y_outptr =<int *>malloc(deltay*sizeof(int))
125. 
126.                int x0,y0,i0
127.                int i = 0
128. 
129.                #offset while iterating
130.                long *posptr =<long *>malloc(2*sizeof(long))
131.            outputptr[0] = ymaxscan-1
132.            outputptr[1] = 0
133. 
134.            x_outptr[0] = deltax
135.            x_outptr[1] = deltax - 1
136. 
137.            y_outptr[1] = ymaxscan - 1
138.            for x0 in range(deltax):
139.                #update offset with x0
140.                posptr[0] = (x0 *y)
141. 
142.                #store value of i after iterating over a row
143.                i0 = i
144.                #iterate over every 3rd pixel until object pixels are reached
145.                for y0 in range(yminscan,ymaxscan,3 - (2*xpos)):
146.                    #update offset with y0
147.                    posptr[1] = 3 * (posptr[0] + y0)
148. 
149.                    #inc = 1 if pixel belongs to object else it is 0
150.                    inc = tablelut_ptr1[256*256*img_ptr[posptr[1]] + 256*img_ptr[
    posptr[1] +1] + img_ptr[posptr[1] + 2]]
151.                    #increase i by 1 if object pixel
152.                    i += inc
153. 
154.                    #if pixel not object pixel, pix pos sent to addr 0
```

```
                    y_outptr[inc*(i - i0)] = y0
                #i > i0 if row contains a valid pixel. add 0 contains first value
    . add 1 contains latest value
                if i > i0:
                    #if condition is triggered for the first, xmin = x0 fed to ad
    dr 0.
                    x_outptr[xpos] = x0
                    #each subsequent detection fed to addr 0, xmax = x0
                    xpos = 1

                    #Update min and max y pixel positons
                    outputptr[0] += (y_outptr[1] - outputptr[0])*(outputptr[0] >
    y_outptr[1])
                    outputptr[1] += (y_outptr[i-
    i0] - outputptr[1])*(outputptr[1] < y_outptr[i - i0])
                    # change search area
                    ymaxscan += (outputptr[1] + 20 -
    ymaxscan) * (outputptr[1] + 20 < ymaxscan)
                #break out of loop if no pixel of ineterest detected in 3 lines a
    fter xpos switched on
                elif x0 - x_outptr[1] > 100:
                    break
            outputptr[2] = x_outptr[0] + outputptr[4] #if none detected outputptr
    [2] = deltax + image start
            outputptr[3] = x_outptr[1] + outputptr[4] #if none detected outputptr
    [3] = delatx - 1 + imagestart
            #if none detected outputptr[4] = 0 and outputptr[5] = x
            #if search area crosses x or becomes negative second condition switch
    es off
            outputptr[4] = (outputptr[2]<= outputptr[3])  *  (((v+1)*outputptr[2]
    ) > (v*outputptr[3]))  *  ((v + 1)*outputptr[2] - (v*outputptr[3]))
            outputptr[5] = ((outputptr[2]<= outputptr[3]) *  ((v+1)*(outputptr[3]
    +1) < x + (v*outputptr[2]))  * (((v+1)*(outputptr[3]+1)) - (v*outputptr[2]) - x)
    ) + x
            freep(x_outptr) #clear memory
            freep(y_outptr) #clear memory
            freep(posptr) #clear memory
            return output
```

9.6. Appendix F: Undistortion and Parameterisation of Pixel Position

Pixel Position Parameterisation – final_return() (LUTptrall.pyx)

```
"""
This is part of the LUTptrall.pyx script
author: Roshan Pasupathy
This script calculates the 3D vector form of the
detected pixel position after undistorting
"""

#Create source array for undistortion
srcarray =  1.0 * np.mgrid[0:640,0:480].T.reshape(1,480*640,2)

#Create LUT of undistorted points for every pixel. input: srcarray
cdef double[:,:,::1] undistortptr = cv2.undistortPoints(srcarray,mtx,dst,R = Non
    e,P= mtx)

def final_return(rect,Qinv):
    ""Return 3D vector of detected pixel after undistorting"""
    #find center pixel of detected object
    xcord = int((rect[2] + rect[3])/2) #find x pixel
    ycord = int((rect[0] + rect[1])/2) #find y pixel

    #undistorted pixel position
    res = undistortptr[0,(xcord*640) + ycord]

    #Create homogenous vector
```

```
    ppos = np.array([[res[0]],[res[1]],[1]])

    #calculate u vector 3 x 1
    u = np.dot(Qinv,ppos)

    #3D vector to be sent
    return u
```

Camera Position Parameterisation – Beacon1.py

```
# -*- coding: utf-8 -*-
"""
This is part of the Beacon1.py script
Created on Feb 3 2016
author: Roshan Pasupathy
This script caculates the 3D vector which represents camera position
"""
#Intrinsic Parameters
mtx = np.array([[ 620.54646,0, 316.17234],[0,621.10631,244.57960],[0,0,1]],dtype=np.float64) #Camera matrix
distcoeff = np.array([ 0.13359,-0.29557, -0.00070 ,-0.00054,0.00000 ],dtype=np.float64) #Distortion coefficients
pose = np.load('/home/pi/ip/pose.npz') #Read extrinsic param calibration file
R = pose['R'] #Rotation matrix
tvec = pose['tvecs'] #translation vector

#Calculats A * R
Q = np.dot(mtx,R)
#calculate A  * translation. last column of matrix
q = np.dot(mtx,tvec)
#calculate inverse of Q
Qinv = np.linalg.inv(Q)
#calculate - inverse of Q
_Qinv = -1.0*Qinv
#calculate c vector 3 x 1
c = np.dot(_Qinv,q).ravel()
```

9.7. Appendix G: Calibration for Extrinsic parameters

Posecalibration.py

```
# -*- coding: utf-8 -*-
"""
Posecalibration.py
Created on Thu Dec 10 19:27:52 2015
author: Roshan
This script calculates the extrinsic parameters of the camera
(position and orientation)
"""
#Import libraries
import numpy as np
import cv2
import os
import time

#Camera settings
os.system('v4l2-ctl -d 0 -c focus_auto=0')
os.system('v4l2-ctl -d 0 -c focus_absolute=0')
os.system('v4l2-ctl -d 0 -c zoom_absolute=100')
os.system('v4l2-ctl -d 0 -c exposure_auto=3')
os.system('v4l2-ctl -d 0 -c contrast=128')
os.system('v4l2-ctl -d 0 -c brightness=128')
os.system('v4l2-ctl -d 0 -c white_balance_temperature_auto=1')

#Camera Intrinsic Parameters
mtx = np.array([[ 511.78922,0, 314.76367],[0,512.44537,268.54549],[0,0,1]],dtype=np.float64)
```

```
dist = np.array([0.07220,-0.15304,0.00269 ,-0.00064,0.00000],dtype=np.float64)

#Termination criteria for cornersubpix
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 1000, 0.0001)

#3D positon of control points
objp = np.zeros((6*9,3), np.float32)
objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
objp = 30.0 * objp

#3D positon of coordinate axes
axis = np.float32([[90,0,0], [0,90,0], [0,0,-90.0]]).reshape(-1,3)

#Create video capture object
cap = cv2.VideoCapture(0)
cap.set(cv2.cv.CV_CAP_PROP_FPS, 30)
b = cap.get(cv2.cv.CV_CAP_PROP_FRAME_HEIGHT)
c = cap.get(cv2.cv.CV_CAP_PROP_FRAME_WIDTH)
print b,c

l = 0
#Start timer
start = time.clock()
while (True) & ( l < 1): #One input image
    #Read image
    ret,frame = cap.read()
    #Show image
    cv2.imshow('frame',frame)
    if cv2.waitKey(1) & 0xFF == ord('p'): #Press 'p' to capture image
        print "stopped"
        frame1 = frame.copy()
        break
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
frameorig = frame1.copy()

#conver to grayscale
gray = cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY)
#Harris corner detector
ret, corners = cv2.findChessboardCorners(gray, (9,6),None)
#If corners detected
if ret == True:
    print "ret true"
    #Sub-pixel refinement
    cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
    #obtain orientation and pose using pnp method
    rvecs, tvecs, inliers = cv2.solvePnPRansac(objp, corners, mtx, dist)
    R = cv2.Rodrigues(rvecs)[0]
    print "Rmatrix",R
    print "translation vector",tvecs
    #Save intrinsic parameters
    np.savez('/home/pi/ip/pose.npz',R=R,rvecs=rvecs,tvecs=tvecs)
    #project coordinate axes onto image plane
    imgpts, jac = cv2.projectPoints(axis, rvecs, tvecs, mtx, dist)
    corner = tuple(corners[0].ravel())
    #x axis
    cv2.line(frame1, corner, tuple(imgpts[0].ravel()), (255,0,0), 2)
    #y axis
    cv2.line(frame1, corner, tuple(imgpts[1].ravel()), (0,255,0), 2)
    #-z axis
    cv2.line(frame1, corner, tuple(imgpts[2].ravel()), (0,0,255), 2)
    #save images
    cv2.imwrite('/home/pi/ip/report/posecalibrate4.bmp',frame1)
    cv2.imwrite('/home/pi/ip/report/withoutaxes.bmp',frameorig)
    l = 1
    print 'image taken and saved'
else:
```

```
    "A problem has occured. Please Recalibrate..."
cv2.imshow('result',frame1)

end = time.clock()
#Stop timer
print 'time taken', end - start, 'seconds'
      print 'frame rate',1/(end - start),'frames/second'
```

9.8. Appendix H: Communication

Beacon Sender – Beacon1.py

```
# -*- coding: utf-8 -*-
"""
This is part of the Beacon1.py script
Created on Feb 15 2016
author: Roshan Pasupathy
This script contains code relevant to communication
- Creation of Socket object
- Connecting to a server
- Serilisation os data and adding flags
- Sending data
"""
#Import libraries
import numpy as np
import socket

#Create socket object
soc = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
#Connect to server
soc.connect(('192.168.42.1',8000)) #Port number 8080 for Beacon 2

#Send camera position vector
soc.send(''.join(['c',c.tostring(),'l'])) #len26

#main loop code
    #Detect object
    output = squarelut8(output,480,640,10,frame[output[4]:output[5],:,:])
    #Check if object detected
    if output[0] <= output[1]:
        #Comment out following line if visual feedback needed. GUI has to be use
d***
        #cv2.rectangle(frame,(output[0],output[2]),(output[1],output[3]),(255,0,
0),2)
        #SSH is fine. GUI not necessary
        #print np.asarray(output)[0:4]
        #comment pixel positon 3D vector
        u = final_return(np.asarray(output)[0:4], Qinv).ravel()
        #Send packet with flag u
        soc.send(''.join(['u',u.tostring(),'l']))
        #Succesful send then failtimes set to 0
        failtimes = 0
        detected += 1
    else:
        #Increment failtimes
        failtimes += 1
        #If object not detected in 5 iterations, send d flag with default array

        if failtimes > 5:
            soc.send(''.join(['d',failarr.tostring(),'l']))
            failtimes = 0
            pack_send += 1

#Send temrination signal from Beacon
soc.send(''.join(['e',failarr.tostring(),'l']))

#Close sockets
soc.close()
```

Central Receiver – Receiver.py

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  This is part of the Receiver.py script
4.  Created on Feb 20 2016
5.  author: Roshan Pasupathy
6.  This script Creates multiple processes to
7.  receive data from each Beacon. Communication script
8.  """
9.  ################## IMPORT LIBRARIES ##################
10.
11. import socket
12. from multiprocessing import Pipe,Process,Array,Value,sharedctypes
13. from threading import Thread
14.
15. ################## OBJECT DECLARATIONS ##################
16. #Port numbers
17. port1 = 8000 #Beacon1
18. port2 = 8080 #Beacon2
19. #Host address
20. _TCP_SOCKET_HOST = "192.168.42.1"
21.
22. ################## SOCKET READER CLASS ##################
23. class SocketReader:
24.     """Create a socket class which is initialised by each process
25.     - Receives packet
26.     - Checks flag
27.     - Updates 3D arrays for the Beacon concerned"""
28.     #Constructor
29.     def __init__(self, sock,dat_size,pipecal):
30.         """ Takes the socket object from which to receive data,
31.         The data size to be read in each loop, and a pipe to send data
32.         to process"""
33.         #create array object which is updated
34.         self.arr = np.array([0,0,0],dtype=np.float64)
35.         #create runflag object which is updated
36.         self.flag = 1
37.         #create client socket object
38.         self.client = sock
39.
40.         # initialise the variable used to indicate if the thread should
41.         # be stopped
42.         self.running = True
43.
44.         #pipe to send caldata
45.         self.pipe = pipecal
46.         self.read_size = dat_size
47.
48.
49.     def start(self):
50.         """Exit main thread of process"""
51.         t = Thread(target=self.threadedloop,args=())
52.         t.daemon = True
53.         t.start()
54.         return self
55.
56.     def threadedloop(self):
57.         """this function runs in a parallel thread"""
58.         #loop to update flag and arrays
59.         while self.running:
60.             # otherwise, read the next frame from the stream
61.             content = self.client.recv(self.read_size)
62.             while len(content) < (self.read_size):
63.                 content += self.client.recv(self.read_size - len(content)) #add
    truncated data
64.             if (content[0] == 'u') and (content[-1] == 'l'): #valid data
```

```
65.                 #Deserialise
66.                 self.arr = np.fromstring(content[1:25],np.float64)
67.                 #run flag update. 2 indicates run localisation code
68.                 self.flag = 2
69.             elif content[0] == 'd': #invalid data
70.                 #run flag update. 1 indicates Beacon running but data invalid
71.                 self.flag = 1
72.             elif content[0] == 'c': #calibration
73.                 self.pipe.send(np.fromstring(content[1:25],np.float64))
74.             elif content[0] == 'e': #loop stop
75.                 #run flag update. 0 indicates terminate
76.                 self.flag = 0
77.                 self.running = False
78.         return
79.
80.     def read(self):
81.         """Return array and runflag"""
82.         return (self.flag,self.arr)
83.
84.     def stop(self):
85.         """Stops thread"""
86.         self.running = False
87.
88. ################## PROCESS DEFINTION ##################
89. def socketcomm(port,pipec, flags,uarr,dat_size = 26):
90.     """Callback function to deal with incoming tcp communication
91.     for each Beacon
92.     pipec: describes position of camera sent to main process
93.     port: port number
94.           - 8000 for Beacon1
95.           - 8080 for Beacon2
96.     uarr: shared mem array to send array info to main process
97.     """
98.     #Flags
99.     sockets = []
100.           arr = np.array([0,0,0],dtype=np.float64)
101.           #initialise socket object
102.           serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
103.           serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
104.           serversocket.bind((_TCP_SOCKET_HOST, port))
105.           serversocket.listen(1)
106.
107.           #add server socket to list of sockets
108.           sockets.append(serversocket)
109.
110.           #client socket creator
111.           clientsocket,clientaddr = serversocket.accept()
112.           sockets.append(clientsocket)
113.
114.           #start thread which reads sockstream
115.           sockr = SocketReader(clientsocket,dat_size=dat_size,pipecal=pipec).st
    art()
116.           datr = (1,arr)
117.           #run loop
118.           while datr[0]:
119.               # Read socketreader class to get newest data
120.               datr =  sockr.read()
121.               with uarr.get_lock():
122.                   uarr.get_obj()[:3] = datr[1]
123.               flags.value = datr[0]
124.           #Close socket objects
125.           for i in sockets:
126.               i.close()
127.
128.       ################## INITIALISATIONS ##################
129.       #f is for function and l is for loop
130.       #pipes for carr
131.       pipecl1, pipecf1 = Pipe(False)
132.       #set process
```

```
133.        uarray1 = Array('d',3)
134.        uflag1 = Value('B',1)
135.
136.        #create process instance which updates uarray1,uflag1, pipecf1 at port 80
    00
137.        proc1 = Process(target=socketcomm,args=(port1,pipecf1,uflag1,uarray1))
138.
139.        #f is for function and l is for loop
140.        #pipes for carr
141.        pipecl2, pipecf2 = Pipe(False)
142.        #set process
143.        uarray2 = Array('d',3)
144.        uflag2 = Value('B',1)
145.
146.        #create process instance which updates uarray2,uflag2, pipecf2 at port 80
    80
147.        proc2 = Process(target=socketcomm,args=(port2,pipecf2,uflag2,uarray2))
148.
149.        ################### RUN MAIN LOOP###################
150.        """
151.        .
152.        . Localisation loops
153.        .
154.        """
155.        ################### CLOSE PROCESSES ################
156.        proc1.join()
157.        proc2.join()
```

9.9. Appendix I: 2D-3D Projection

Main script – 2D_3Dproj.py

```
1.  """
2.  2D_3Dproj.py
3.  author: Roshan Pasupathy
4.  Run script to compare performance between python numpy
5.  and Cython implementation for the 2D-3D projection
6.  """
7.  #Import libraries
8.  from arryop import calc3d, update_c #All calculations performed in extension fil
    e
9.  import numpy as np
10. import timeit
11.
12. #input test uarrays - pixel position array
13. ut1 = np.array([[52.45],[62.34],[56.78]],dtype=np.float64)
14. ut2 = np.array([[32.75],[92.24],[76.28]],dtype=np.float64)
15. ul1 = ut1.ravel()
16. ul2 = ut2.ravel()
17.
18. #input test carray - camera position array
19. c1 = np.array([12.45,610.34,74.10],dtype=np.float64)
20. c2 = np.array([80.45,12.34,56.78],dtype=np.float64)
21.
22. p21 = c2 - c1
23.
24. def numpycross(p21, c1p2h,ul1,ul2):
25.     """Calculates 3D projection from two arrays in parametric form
26.     numpy implementation"""
27.     #M vector
28.     m = np.cross(ul2,ul1)
29.     baseval = np.cross(p21,m)/float(np.dot(m,m))
30.     #Calculate lmbda 1 and 2
31.     lmda1 = np.dot(baseval,ul2)
32.     lmda2 = np.dot(baseval,ul1)
33.     #3D positon
34.     return (c1p2h) + (((lmda1*ul1) + (lmda2*ul2))/2.0)
```

```
35.
36. #Uodate c array in extension file
37. update_c(c1,c2)
38.
39. #Check if results are the same for test data
40. print "*" * 80
41. print "calculated value pythonimp", numpycross(p21,(c2+c1)/2.0,ul1,ul2)
42. print "calculated value Cython",np.asarray(calc3d(ul1,ul2))
43.
44. print "*" * 80
45. print "calculated value pythonimp", numpycross(p21,(c2+c1)/2.0,ul2,ul1)
46. print "calculated value Cython",np.asarray(calc3d(ul2,ul1))
47. #Setup files
48.
49. timerSetup1="""
50. import numpy as np
51. from __main__ import numpycross
52. ul1 = np.array([52.45,62.34,56.78],dtype=np.float64)
53. ul2 = np.array([32.75,92.24,76.28],dtype=np.float64)
54.
55. c1 = np.array([12.45,610.34,74.10],dtype=np.float64)
56. c2 = np.array([80.45,12.34,56.78],dtype=np.float64)
57.
58. p21 = c2 - c1
59. """
60.
61. timerSetuppointer="""
62. import numpy as np
63. from arryop import calc3d,update_c
64. from __main__ import c1,c2
65. update_c(c1,c2)
66. ul1 = np.array([52.45,62.34,56.78],dtype=np.float64)
67. ul2 = np.array([32.75,92.24,76.28],dtype=np.float64)
68. """
69.
70. #Timeit objects
71. numpyt = timeit.Timer ('numpycross(p21,(c2+c1)/2.0,ul1,ul2)' , timerSetup1)
72. pointert = timeit.Timer ('mainrecv(ul1,ul2)' , timerSetuppointer)
73.
74. print "*" * 80
75. print 'Numpy implementation: %4.6f seconds'  %numpyt.timeit(1000000)
76. print 'Cython implementation: %4.6f seconds'  %pointert.timeit(1000000)
77. print "*" * 80
```

Cython Extension – arryop.pyx

```
1.  """
2.  arryop.pyx
3.  Created on Feb 25 2016
4.  author: Roshan Pasupathy
5.  This file is called to calculate 3D position of the object.
6.  Performs all computationally expensive steps.
7.  Alternative to using numpy functions( np.dot() and np.cross())
8.  """
9.  #Import libraries
10. cimport cython
11. import numpy as np
12. cimport numpy as np
13.
14. #Initialise M and pointer to M
15. cdef double[::1] marr= np.array([0.0,0.0,0.0],dtype=np.float64)
16. cdef double* m =  &marr[0]
17.
18. #Initialise basevallarr and pointer to basevalarr
19. cdef double[::1] basevalarr= np.array([0.0,0.0,0.0],dtype=np.float64)
20. cdef double* baseval =  &basevalarr[0]
21.
22. #c1 - c2 and its pointer
```

```
cdef double[::1] p21 = c2 - c1
cdef double* c21ptr =  &p21[0]

cdef double[::1] c2p1harr = (c2 + c1)/2.0
cdef double* c2p1h = &c2p1harr[0]

cdef double[::1] outarr= np.array([0.0,0.0,0.0],dtype=np.float64)
cdef double* outptr =  &outarr[0]


@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
def update_c(double[::1] c1,double[::1] c2):
    """Update c1 + c2 and c1 - c2"""
    c2p1h[0] = (c2[0] + c1[0])/2.0
    c2p1h[1] = (c2[1] + c1[1])/2.0
    c2p1h[2] = (c2[2] + c1[2])/2.0
    c21ptr[0] = c2[0] - c1[0]
    c21ptr[1] = c2[1] - c1[1]
    c21ptr[2] = c2[2] - c1[2]

@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
cdef void crossc(double* arr1,double* arr2):
    """Called by calc3d. Compute m and baseval and updates it in heap memory"""

    global m,baseval
    #cross product of arr1 and arr2
    m[0] = (arr1[1] * arr2[2]) - (arr1[2] * arr2[1])
    m[1] = (arr1[2] * arr2[0])-(arr1[0] * arr2[2])
    m[2] = (arr1[0] * arr2[1])-(arr1[1] * arr2[0])

    #calcuates M.M
    cdef double m2 = (m[0] * m[0]) + (m[1] * m[1]) + (m[2] * m[2])

    #calculates baseval
    baseval[0] =   ((c21ptr[1] * m[2]) - (c21ptr[2] * m[1]))/m2
    baseval[1] = ((c21ptr[2] * m[0])-(c21ptr[0] * m[2]))/m2
    baseval[2] = ((c21ptr[0] * m[1])-(c21ptr[1] * m[0]))/m2

@cython.boundscheck(False)
@cython.cdivision(True)
@cython.wraparound(False)
def calc3d(double[::1] u1ptr,double[::1] u2ptr):
    """ Main function called. Returns 3D position"""
    crossc(&u2ptr[0],&u1ptr[0]) #update m and baseval
    cdef:
        #Calculate lmbda 1 and lmbda 2
        double lmda1 = ((baseval[0] * u2ptr[0])  +   (baseval[1] * u2ptr[1]) + (
baseval[2] * u2ptr[2]))
        double lmda2 = ((baseval[0] * u1ptr[0])  +   (baseval[1] * u1ptr[1]) + (
baseval[2] * u1ptr[2]))
    #Update outputarray
    outptr[0] = c2p1h[0] + ((u1ptr[0] * lmda1) + (u2ptr[0] * lmda2))/2.0
    outptr[1] = c2p1h[1] +((u1ptr[1] * lmda1) + (u2ptr[1] * lmda2))/2.0
    outptr[2] = c2p1h[2] +((u1ptr[2] * lmda1) + (u2ptr[2] * lmda2))/2.0
    return outarr
```

9.10. Appendix J: Error

```
# -*- coding: utf-8 -*-
"""
This is part of the Receiver.py script
Created on Feb 20 2016
author: Roshan Pasupathy
```

```
6.  This script contains code relevant to communication
7.  - Creation of Socket object
8.  - Connecting to a server
9.  - Serilisation os data and adding flags
10. - Sending data
11. """
12. ########## Import libraries ################
13. import numpy as np
14. import cv2
15. from arryop import calc3d
16. from LUTptrall import squarelut8
17. from pylab import *
18. from mpl_toolkits.mplot3d import Axes3D
19. from matplotlib import pyplot as plt
20.
21. ########## Main loop to calculate Reprojection error#################
22. while runflag:
23.     #Read flag status from processes
24.     dat1 = uflag1.value
25.     dat2 = uflag2.value
26.     #Read image frame
27.     frame = vs.read()
28.     if (dat1 == 2) and (dat2 ==2): #Object detected by both Beacons
29.         #update camera position vectors
30.         with uarray1.get_lock():
31.             arr1 = np.frombuffer(uarray1.get_obj())
32.         with uarray2.get_lock():
33.             arr2 = np.frombuffer(uarray2.get_obj())
34.         #Calculate 3D position
35.         pos3d =  calc3d(arr1,arr2)
36.         #Project 3D points onto image plane
37.         imgpts, jac = cv2.projectPoints(np.float32([np.asarray(pos3d)]).reshape(
    -1,3), rvecs, tvecs, mtx, dist)
38.         #Show projected result
39.         cv2.rectangle(frame,(int(imgpts[0,0,0]) - 2,int(imgpts[0,0,1]) - 2),(int
    (imgpts[0,0,0]) + 2 ,int(imgpts[0,0,1]) + 2),(255,0,0),1)
40.         #Detect object. run detection script
41.         output = squarelut8(output,480,640,10,frame[output[4]:output[5],:,:])
42.         if (output[0] <= output[1]) * (output[3] - output[2] < 2 * (output[1] -
    output[0])): #object detected
43.             #Show detection result
44.             cv2.rectangle(frame,(output[0],output[2]),(output[1],output[3]),(0,0
    ,255),2)
45.             #Calculate reprojection error in x
46.             reprojectx.append(((output[0] + output[1])/2.00) - imgpts[0,0,0])
47.             #Calculate reprojection error in y
48.             reprojecty.append(((output[2] + output[3])/2.00) - imgpts[0,0,1])
49.             pos3da = np.asarray(pos3d)
50.             #Calculate distance from camera
51.             reprojectz.append(np.linalg.norm(pos3da - c))
52.     #One of the Beacons send temrination signal
53.     elif not dat1 or not dat2:
54.         runflag = False
55.     cv2.imshow('frame',frame)
56.     if cv2.waitKey(1) & 0xFF == ord('q'):
57.         break
58.
59. ################## Plot results ##########################
60.
61. # Data to be plotted
62. z_dec = reprojectz/float(100) #z in decimeters
63. z_cm = reprojectz/float(10) #z in centimeters
64. z_cm_sort = z_cm[np.argsort(z_cm)] #z cm sorted
65. z_dec_sort = z_dec[np.argsort(z_dec)] # dcm sorted
66. z_cm_int = z_cm_true.astype(int) #zcm integer type, redunandant can be replaced

67. z_cm_plot = z_cm_int[np.argsort(z_cm_int)] #zcm integer type sorted
68.
69. x_sort = reprojectx[np.argsort(z_dec)] #sort x according to distance
```

```
y_sort = reprojecty[np.argsort(z_dec)] #sort y according to distance
z_sort = reprojectz[np.argsort(reprojectz)] #sort z (distance)

############
#Plot scatter plot of reprojectione error
fig = plt.figure(0)
ax = fig.add_subplot(111)
ax.grid(True)
#Colour map according to distance from camera
colours = cm.brg(reprojectz/max(reprojectz))
colmap = cm.ScalarMappable(cmap = cm.brg)
colmap.set_array(reprojectz)

ax.scatter(reprojectx,reprojecty, c=colours,marker='+',s=40)
cb = fig.colorbar(colmap,shrink=0.75)
cb.set_label('Distance from Camera (mm)')
ax.set_title('Reprojection Error')
ax.set_xlabel('x (pixels)')
ax.set_ylabel('y (pixels)')
ax.axhline(c='k',ls='--')
ax.axvline(c='k',ls='--')

############
#Plot reprojection error with respect to distance. Calculated in both x and y di
rection
zp = np.linspace(min(z_cm_plot) - 10,max(z_cm_plot) + 10,num=600)
#polyfit data
pfit_x = np.poly1d(np.polyfit(z_cm_plot,x_sort,4)) #4
pfit2_x = pfit_x(zp)

pfit_y = np.poly1d(np.polyfit(z_cm_plot,y_sort,4)) #4
      pfit2_y = pfit_y(zp)

      fig2 = plt.figure(2)
      ax2 = fig2.add_subplot(111)
      ax2.grid(True)

      ax2.scatter(z_cm_plot, abs(x_sort), c='b',marker='D')
      x2_plot = ax2.plot(zp,abs(pfit2_x), c='b',ls='-
',label='Reprojection Error in X')

      ax2.scatter(z_cm_plot, abs(y_sort), c='g',marker='s')
      y2_plot = ax2.plot(zp,abs(pfit2_y), c='g',ls='--
',label='Reprojection Error in Y')
      ax2.legend()
      ax2.set_title('Reprojection Error vs. Distance from Camera')
      ax2.set_xlabel('Distance from Camera (cm)')
      ax2.set_ylabel('Reprojection Error (pixels)')
      ax2.set_xlim(min(z_cm_plot) - 10,max(z_cm_plot) + 10)

      ############
      #Plot Relationship between absolute error and distance
      #Calculate abs_error
      x_abserror = np.multiply(x_sort,z_sort)/509.39049
      x_ae_2 = np.multiply(x_abserror,x_abserror)

      y_abserror = np.multiply(y_sort,z_sort)/510.01682
      y_ae_2 = np.multiply(y_abserror,y_abserror)

      abs_err_euc = np.sqrt(x_ae_2 + y_ae_2)

      #polyfit data
      pfit = np.polyfit(z_cm_sort,abs_err_euc,4) #5 and avg for laser
      pform = np.poly1d(pfit)

      #Plot graph
      fig3 = plt.figure(3)
      ax3 = fig3.add_subplot(111)
```

```
135.        x3_plot = ax3.scatter(z_cm_plot,abs_err_euc, c='r',marker='o')
136.        pfit3 = ax3.plot(zp,pform(zp), c='b')
137.        ax3.set_xlim(min(z_cm_plot) - 10,max(z_cm_plot) + 10)
138.        ax3.set_title('Approximate Absolute Error vs. Distance from Camera')
139.        ax3.set_xlabel('Distance from Camera (cm)')
140.        ax3.set_ylabel('Absolute Error Estimate (mm)')
141.
142.        plt.show()
```

9.11. Appendix K: Performance

Beacon Code

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  Beacon1.py
4.  Created on Feb 20 2016
5.  author: Roshan Pasupathy
6.  This is main script run on Beacon 1
7.  """
8.  #Import Libraries
9.  import numpy as np
10. import cv2
11. import os
12. from LUTBeacontest1 import squarelut8
13. from LUTBeacontest1 import cleanupf
14. from LUTBeacontest1 import final_return
15. import time
16. from threading import Thread
17. import socket
18. import sys,traceback
19.
20. os.system('v4l2-ctl -d 0 -c focus_auto=0')
21. os.system('v4l2-ctl -d 0 -c focus_absolute=0')
22. os.system('v4l2-ctl -d 0 -c exposure_auto=1')
23. os.system('v4l2-ctl -d 0 -c exposure_absolute=3')
24. os.system('v4l2-ctl -d 0 -c contrast=100')
25. os.system('v4l2-ctl -d 0 -c brightness=100')
26. os.system('v4l2-ctl -d 0 -c white_balance_temperature_auto=0')
27. os.system('v4l2-ctl -d 0 -c white_balance_temperature=6500')
28.
29. output = np.array([0,640,0,480,0,480])
30. ############################
31.
32. mtx = np.array([[ 620.54646,0, 316.17234],[0,621.10631,244.57960],[0,0,1]],dtype
    =np.float64)
33. distcoeff = np.array([ 0.13359,-0.29557, -0.00070 ,-
    0.00054,0.00000 ],dtype=np.float64)
34. pose = np.load('/home/pi/ip/pose.npz')
35. R = pose['R']
36. tvec = pose['tvecs']
37.
38. ############################
39. Q = np.dot(mtx,R)
40. #calculate A  * translation. last column of matrix
41. q = np.dot(mtx,tvec)
42. #calculate inverse of Q
43. Qinv = np.linalg.inv(Q)
44. #calculate inverse of Q
45. _Qinv = -1.0*Qinv
46. #calculate c vector 3 x 1
47. c = np.dot(_Qinv,q).ravel()
48. failarr = np.array([0,0,0],dtype=np.float64)
49.
50. class WebcamVideoStream:
51.     def __init__(self, src=0):
52.         # initialise the video camera stream and read the first frame
53.         # from the stream
```

```
        self.stream = cv2.VideoCapture(src)
        (self.grabbed, self.frame) = self.stream.read()

        # initialise the variable used to indicate if the thread should
        # be stopped
        self.stopped = False
        self.fps = self.stream.get(cv2.cv.CV_CAP_PROP_FPS)

    def start(self):
        # start the thread to read frames from the video stream
        t = Thread(target=self.update, args=())
        t.daemon = True
        t.start()
        return self

    def update(self):
        # keep looping infinitely until the thread is stopped
        while True:
            # if the thread indicator variable is set, stop the thread
            if self.stopped:
                return

            # otherwise, read the next frame from the stream
            (self.grabbed, self.frame) = self.stream.read()

    def read(self):
        # return the frame most recently read
        return self.frame

    def stop(self):
        # indicate that the thread should be stopped
        self.stopped = True

try:
    soc = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    soc.connect(('192.168.42.1',8000)) #8080 for Beacon 2

    soc.send(''.join(['c',c.tostring(),'l'])) #len156

    vs = WebcamVideoStream(src=0).start()
    failtimes = 0
    l = 1
    i = 1
    detected = 0
    pack_send = 2
    start = time.clock()
          while (True) & ( l <= 1000):
              #ret,frame = cap.read()
              frame = vs.read()
              output = squarelut8(output,480,640,10,frame[output[4]:output[5],:
    ,:])
              if output[0] <= output[1]:
                  #cv2.rectangle(frame,(output[0],output[2]),(output[1],output[
    3]),(255,0,0),2)
                  #print np.asarray(output)[0:4]
                  u = final_return(np.asarray(output)[0:4], Qinv).ravel()
                  soc.send(''.join(['u',u.tostring(),'l']))
                  failtimes = 0
                  detected += 1
              else:
                  failtimes += 1
                  if failtimes > 5:
                      soc.send(''.join(['d',failarr.tostring(),'l']))
                      failtimes = 0
                      pack_send += 1
                  #print "Ball Not detected"
              l += 1
              ##Uncomment lines 111 - 118 if GUI is available
```

```
120.                #cv2.imshow('frame',frame)
121.                #if cv2.waitKey(1) & 0xFF == ord('c'):
122.                #   stringval = 'img' + str(i) +'.bmp'
123.                #   cv2.imwrite(stringval,frame1)
124.                #   i += 1
125.                #   print stringval + ' taken'
126.                #if cv2.waitKey(1) & 0xFF == ord('q'):
127.                #   break
128.            end = time.clock()
129.            print '*' * 80
130.            print 'BEACON 1'
131.            print 'Video Capture frame rate:', vs.fps, ' frames/second'
132.            print 'Number of frames processed: 1000 frames'
133.            print 'Frames in which object is detected: ',detected,' frames'
134.            print 'Packets sent: ',detected + pack_send
135.            print 'Frequency of packet transmission: %.3f Hz'%((detected+pack_sen
    d)/(end-start))
136.            print '*' * 80
137.        except:
138.            typ,val,tb = sys.exc_info()
139.            traceback.print_exception(typ,val,tb)
140.
141.
142.        finally:
143.            soc.send(''.join(['e',failarr.tostring(),'1']))
144.            soc.close()
145.            cleanupf()
146.            cv2.destroyAllWindows()
147.            vs.stop()
```

Central Receiver

```
1.  # -*- coding: utf-8 -*-
2.  """
3.  Receiver_fin.py
4.  Created on Feb 20 2016
5.  author: Roshan Pasupathy
6.  This is the main script run on the central receiver
7.  """
8.  ################### IMPORT LIBRARIES ##################
9.  import socket
10. import select
11. import os.path
12. import numpy as np
13. import cv2
14. import os
15. import time
16. import sys,traceback
17. # import time
18. from multiprocessing import Pipe,Process,Array,Value,sharedctypes
19. from threading import Thread
20. # from logging import debug, info, warn, error
21. from arryop import calc3d,update_c #********************
22.
23. ################### OBJECT DECLARATIONS ##################
24. port1 = 8000
25. port2 = 8080
26. #Host address
27. _TCP_SOCKET_HOST = "192.168.42.1"
28.
29. ################### FOR TESTING ##################
30. mtx = np.array([[  509.39049,0, 312.71212],[0,510.01682,269.33614],[0,0,1]],dtyp
    e=np.float64)
31. dist = np.array([0.07151,-0.15763,0.00257,-0.00169,0.0000],dtype=np.float64)
32. rvecs = np.array([[-0.33025405],[-0.25113913],[-1.52710343]])
33. tvecs = np.array([[-33.85189337],[189.50452354],[ 609.96533569]])
34. ################### SOCKET READER CLASS ##################
35. class SocketReader:
```

```
36.     def __init__(self, sock,dat_size,pipecal):
37.         #create array object which is updated
38.         self.arr = np.array([0,0,0],dtype=np.float64)
39.         #create flag object which is updated
40.         self.flag = 1
41.         #create client socket object
42.         self.client = sock
43.
44.         # initialise the variable used to indicate if the thread should
45.         # be stopped
46.         self.running = True
47.
48.         #pipe to send caldata
49.         self.pipe = pipecal
50.         self.read_size = dat_size
51.         #packet counter
52.         self.count = 0
53.
54.     def start(self):
55.         t = Thread(target=self.threadedloop,args=())
56.         t.daemon = True
57.         t.start()
58.         return self
59.
60.     def threadedloop(self):
61.         """this function runs in a parallel thread"""
62.         #loop to update flag and arrays
63.         while self.running:
64.             # otherwise, read the next frame from the stream
65.             content = self.client.recv(self.read_size)
66.             self.count += 1
67.             while len(content) < (self.read_size):
68.                 content += self.client.recv(self.read_size - len(content)) #add
    truncated data
69.                 #self.trash = self.client.recv(self.dat_size - len(content))
70.             #if len(content) == self.dat_size:
71.             if (content[0] == 'u') and (content[-1] == 'l'): #valid data
72.                 self.arr = np.fromstring(content[1:25],np.float64)#np.loads(''.j
    oin([self.encstr,content[1:self.read_size-1]))
73.                 self.flag = 2
74.             elif content[0] == 'd': #invalid data
75.                 self.flag = 1
76.             elif content[0] == 'c': #calibration
77.                 self.pipe.send(np.fromstring(content[1:25],np.float64))
78.             elif content[0] == 'e': #loop stop
79.                 self.flag = 0
80.                 self.running = False
81.                 print 'Beacon packets',self.count
82.         return
83.
84.     def read(self):
85.         return (self.flag,self.arr)
86.
87.     def stop(self):
88.         # indicate that the thread should be stopped
89.         self.running = False
90.
91. ################### PROCESS DEFINTION ###################
92. def socketcomm(port,pipec, flags,uarr,dat_size = 26):
93.             """Callback function to deal with incoming tcp communication
94.             for each Beacon
95.             pipec: describes position of camera sent to main process
96.             port: port number
97.             - 8000 for Beacon1
98.             - 8080 for Beacon2
99.             uarr: shared mem array to send array info to main process
100.            """
101.            #Flags
```

```
102.            sockets = []
103.            arr = np.array([0,0,0],dtype=np.float64)
104.            #initialise socket object
105.            serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
106.            serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
107.            serversocket.bind((_TCP_SOCKET_HOST, port))
108.            serversocket.listen(1)
109.
110.            #add server socket to list of sockets
111.            sockets.append(serversocket)
112.
113.            #client socket creator
114.            clientsocket,clientaddr = serversocket.accept()
115.            sockets.append(clientsocket)
116.
117.            #start thread which reads sockstream
118.            sockr = SocketReader(clientsocket,dat_size=dat_size,pipecal=pipec).st
    art()
119.            datr = (1,arr)
120.            #run loop
121.            while datr[0]:
122.                datr =  sockr.read()
123.                with uarr.get_lock():
124.                    uarr.get_obj()[:3] = datr[1]
125.                flags.value = datr[0]
126.            for i in sockets:
127.                i.close()
128.
129.        ################## WEBCAM CLASS ##################
130.        # uncomment for testing
131.        class WebcamVideoStream:
132.            def __init__(self, src=0):
133.                # initialise the video camera stream and read the first frame
134.                # from the stream
135.                self.stream = cv2.VideoCapture(src)
136.                (self.grabbed, self.frame) = self.stream.read()
137.
138.                # initialise the variable used to indicate if the thread should
139.                # be stopped
140.                self.stopped = False
141.
142.            def start(self):
143.                # start the thread to read frames from the video stream
144.                t = Thread(target=self.update, args=())
145.                t.daemon = True
146.                t.start()
147.                return self
148.
149.            def update(self):
150.                # keep looping infinitely until the thread is stopped
151.                while True:
152.                    # if the thread indicator variable is set, stop the thread
153.                    if self.stopped:
154.                        return
155.
156.                    # otherwise, read the next frame from the stream
157.                    (self.grabbed, self.frame) = self.stream.read()
158.
159.            def read(self):
160.                # return the frame most recently read
161.                return self.frame
162.
163.            def stop(self):
164.                # indicate that the thread should be stopped
165.                self.stopped = True
166.
167.        ################## SET THREAD PARAMETERS ##################
168.
169.        #f is for function and l is for loop
```

```
170.        #pipes for carr
171.        pipecl1, pipecf1 = Pipe(False)
172.        #set process
173.        uarray1 = Array('d',3)
174.        uflag1 = Value('B',1)
175.
176.        #create process instance which updates uarray1,uflag1, pipecf1 at port 80
    00
177.        proc1 = Process(target=socketcomm,args=(port1,pipecf1,uflag1,uarray1))
178.
179.        #f is for function and l is for loop
180.        #pipes for carr
181.        pipecl2, pipecf2 = Pipe(False)
182.        #set process
183.        uarray2 = Array('d',3)
184.        uflag2 = Value('B',1)
185.
186.        #create process instance which updates uarray2,uflag2, pipecf2 at port 80
    80
187.        proc2 = Process(target=socketcomm,args=(port2,pipecf2,uflag2,uarray2))
188.
189.        ################### START PROCESSES ###################
190.        try:
191.            print '*' * 80
192.            runflag = True
193.            proc1.start()
194.            proc2.start()
195.            #uncomment nextline for testing
196.            #vs = WebcamVideoStream(src=0).start()
197.
198.            ################### BEGIN MAIN LOOP ###################
199.
200.            print "Udating c..."
201.            c1 = pipecl1.recv() #Blocking
202.            c2 = pipecl2.recv() #Blocking
203.            update_c(c1,c2) #########
204.            loop_no = 0
205.            print "Running main loop..."
206.            start = time.clock()
207.            while runflag:
208.                dat1 = uflag1.value
209.                dat2 = uflag2.value
210.                #frame = vs.read() #for testing
211.                if (dat1 == 2) and (dat2 ==2):
212.                    loop_no += 1
213.                    with uarray1.get_lock():
214.                        arr1 = np.frombuffer(uarray1.get_obj())
215.                    with uarray2.get_lock():
216.                        arr2 = np.frombuffer(uarray2.get_obj())
217.                    pos3d =  calc3d(arr1,arr2)
218.                    #print np.asarray(pos3d)
219.                    #imgpts, jac = cv2.projectPoints(np.float32([np.asarray(pos3d
    )]).reshape(-1,3), rvecs, tvecs, mtx, dist)
220.                    #cv2.rectangle(frame,(int(imgpts[0,0,0]) - 2,int(imgpts[0,0,1
    ]) - 2),(int(imgpts[0,0,0]) + 2 ,int(imgpts[0,0,1]) + 2),(255,0,0),1)
221.                elif not dat1 or not dat2:
222.                    runflag = False
223.                #cv2.imshow('frame',frame)
224.                #if cv2.waitKey(1) & 0xFF == ord('q'):
225.                #   break
226.            end = time.clock()
227.            print "Waiting for both processes to stop..."
228.            while uflag1.value or uflag2.value:
229.                continue
230.
231.            print "Script executed without issue"
232.            print 'loops run', loop_no
233.            print 'time taken', end-start, ' seconds'
```

```
234.            print 'frequency', loop_no/(end-start),' Hz'
235.
236.        except:
237.            typ,value,tb = sys.exc_info()
238.            traceback.print_exception(typ,value,tb)
239.
240.        finally:
241.            print "Cleaning up......"
242.            proc1.join()
243.            proc2.join()
244.            #-V- uncomment for testing
245.            #vs.stop()
246.            #cv2.destroyAllWindows()
```

9.12. Appendix L: Costing

Component	No. of units	Per unit cost(£)
Raspberry Pi 2 model B	3	24.50
Logitech C920 webcam	2	50.00
Wirless USB adapter	3	6.00
Total Cost		£191.50