UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Department of Electronic & Computer Engineering

Smartphone Medical Toolkit

Name:	Cian O'Dwyer
ID Number:	13127381
Supervisor(s):	Dr Sean McGrath / Dr Colin Flanagan
Course:	Electronic & Computer Engineering (LM118)
Academic Year:	2016/17
Project Number:	SMcG/CF 2

Contents

Table of Figures

Abstract

In an age where smartphones are widespread in even the poorest areas of the world, many people in these 3rd world countries lack access to the most basic levels of healthcare, and yet have direct access to this comparatively new technology.

With the huge rise of biomedical Android & iOS apps designed to cover the basics of patient care, many biomedical firms have identified these poor regions as a new market that could be revolutionised by the medical services provided by these biomedical applications. The overall aim of this project is to demonstrate the capabilities that the modern day smartphone has in a biomedical context.

Specifically, we will be designing a vein mapping system to assist with the process of injection. The usefulness of such a product could have huge benefits & implications for the medical sector at large, particularly in third world countries.

Due to the unsanitary nature of needles, and the importance of medicine administered through the process of injection, the aiding of said injection process will lead to numerous benefits. These benefits include reducing the spread of disease through needle re-usage, lower skill level required for the injection process, and a dramatic decrease in income spent on resources.

Introduction & Product Outline

The general purpose of this project is to design a mobile phone application that integrates smoothly with medical sensors & devices to provide a healthcare service that requires little training to utilise, and can be applied anywhere in the world. This Android Application & medical sensor integration culminates in a Vein Mapping system, which will allow any individual the ability to identify a network of veins within a person's body. This allows treatments to be injected straight into the person's bloodstream with minimal training required for nurses & doctors, as well as a huge increase in patient satisfaction.

To implement the entire system, it is best to view each of the functional blocks as separate entities. Each block will be developed and tested separately, before being combined together during the final stages of the project. These entities are:

- Sensor Development & Integration
- Application Development in Android Studio
- Machine Vision Implementation for Image Processing & Mapping

The overall aim of this project is to create a working prototype of the product specified above, while also becoming adept & increasing knowledge within the various concepts used to develop such a product. This involves learning how to develop Android applications within Android Studio, learning how to effectively implement computer vision systems by using Python for mathematical modelling, and well as correctly designing and calibrating an infrared transmission system on a Raspberry Pi board to make the whole project feasible.

The sensor development aspect of the project will work as follows:

- A Raspberry Pi board equipped with a transmitter (specifically, and LED circuit) will transmit Near-Infrared (NIR) light onto the part of the body in question.
- Because of the biological composition of the human body, and the various absorption rates of infrared light in the body, the veins will absorb these NIR rays, while the surrounding fat and muscle tissue will reflect it, thus providing a vein mapping.
- To be able to see this vein mapping however, a special purpose camera with the Infrared filter removed, aptly named NoIR camera, will be taking in the image produced. Almost all cameras will have an infrared filter on them, as the infrared spectrum is outside our field of vision and would distort images.
- The NoIR camera will be equipped with an optical band-pass filter in the form of a lens, which will only allow light within the band of interest to pass, while reflecting light of all other wavelengths. This is done to prevent interference from various other light sources when trying to obtain a usable image. This optical band-pass filter allows the sensor to be used at any time, regardless of the lighting and surroundings.

Product List

- Raspberry Pi 3 – Model B
- Raspberry Pi NoIR Camera – V2
- MidOPT BN850/27 – Narrow bandpass filter NIR
- Digikey HIR8323/C16 – 850nm LED
- Kingbright L-9294QBC-D – 465nm LED
- Sandisk 16GB SD Card – NOOBS pre-installed
- Raspberry Pi Power Supply – 13W, 5.1V, 2.5A
- Raspberry Pi Jumper Wires – Female to Male

Software Used

Python 2.7.9

Python is a high-level, object oriented programming language which is used for a wide variety of purposes [1]. It focuses on having a syntax that is easy to code in, while also having a huge amount of online support & open-source libraries, making it easily extendable by importing new modules.

In the context of this project, we will be using Python for implementing the machine vision aspects on the project at hand. There is a large amount of online support for Python in this specific application, and many of the standard Python libraries have built-in functions which will be used. Python also comes as standard with the Raspbian operating system on the Raspberry Pi.

Android Studio

Android Studio is the official IDE for Android app development [2], provided by Google. The software is written in Java and offers many powerful features such as a layout-editor which allows for drag-and-drop UI components, amongst many other features. Similar to Python, it is freely available software.

Android Studio will be used to design, build, and test the functionality of the Android app in which is being designed. Built into Android Studio is an Android emulator, which will emulate the actions & performance of the app within Android Studio.

Sketch 3

Sketch is a powerful graphics design tool used in various types of application & web design. It is a vector graphics editor, akin to Adobe Photoshop, with a focus on an intuitive easy-to-use layout that creates professional designs.

The UI of the Android app will be designed in Sketch, to create a visually appealing app interface that the average user will enjoy. Sketch also contains functionality which allows the user to export their designs. Android Studio is then able to import these directly into the app, thus providing further benefit of designing in Sketch.

Open CV 3.2.0

OpenCV (Open Source Computer Vision) is a library designed for computational efficiency with a strong focus on real-time machine vision applications [3]. It was initially developed in an Intel research centre, before eventually going on to be maintained as open-source software by Itseez. It is a cross-platform library that will be imported into Python for usage in this project.

Many of the image processing techniques to be used in this project (such as Canny Edge Detection) will be implemented through the usage of the OpenCV library.

Background Theory & Principles of Operation

Biological Background

When deciding on the specific wavelength of the Near-Infrared LEDs to be used, we must undertake an analysis of the light absorption properties of veins, arteries, & surrounding flesh.

Specifically, we are looking for a discernible difference in the absorption rates between the veins, and the surrounding tissue. It is from here we investigate further the NIR absorption rates in biological tissue, more specifically the Absorption Coefficient (u_a).

The Absorption Coefficient (u_a) of tissue is defined as the probability of photon absorption in tissue per unit path length [1] It is modelled by the following formulae:

$$u_a(\lambda_1) = [\ln(10) . \varepsilon_{Hb02(\lambda_1)} . C_{Hb02}] + [\ln(10) . \varepsilon_{Hb(\lambda_1)} . C_{Hb}]$$
$$u_a(\lambda_2) = [\ln(10) . \varepsilon_{Hb02(\lambda_2)} . C_{Hb02}] + [\ln(10) . \varepsilon_{Hb(\lambda_2)} . C_{Hb}]$$

Where:

- C = Molar concentration and
- ε = Molar extinction coefficient
- λ = wavelength value
- Hb02 = Oxygenated Haemoglobin
- Hb = Deoxygenated Haemoglobin

While molar concentration is not uniform throughout the bloodstream due to its nature, it should be easy to identify the difference between the two. This is because it is the veins, as opposed to the arteries that we are interested in. Because arteries are filled with blood which flows from the heart to the rest of the body, they will have the oxygenated blood, meaning it will have the oxygenated haemoglobin (HbO2). This blood flows towards the capillaries around the body to spread the oxygen evenly. Once the oxygen has been dispersed these capillaries begin to join together and form veins. These veins main purpose are designed to

bring the deoxygenated blood back to the heart and repeat the cycle. Because of this, the veins will be filled with deoxyhaemoglobin (Hb).

As we will see from the chart below (Figure.1) displaying the various absorption rates below, there is a discernible difference between HbO2 in the arteries, and the Hb in the veins which we are interested in mapping.

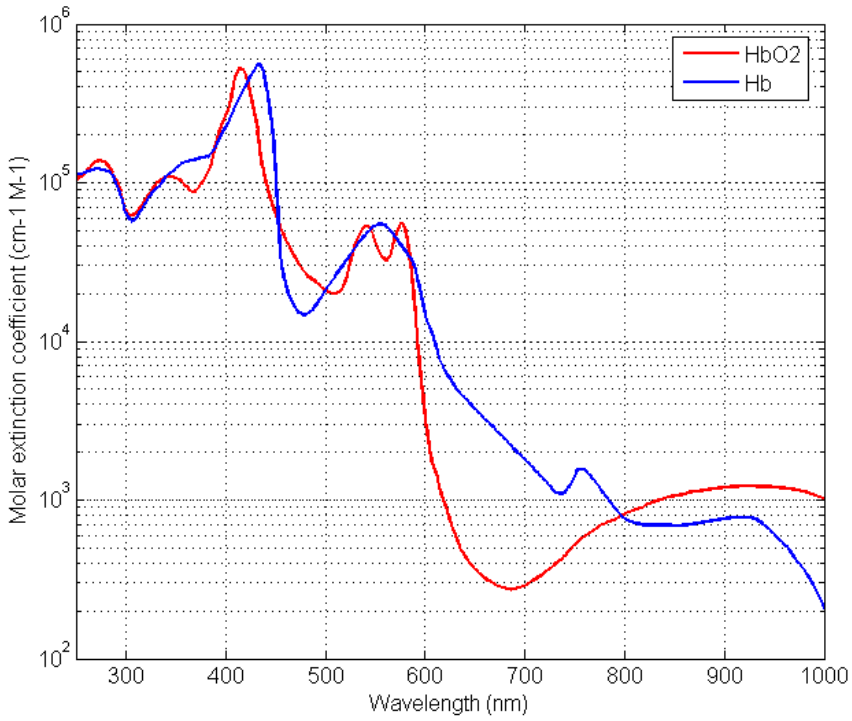


Figure 1 - Molar Extinction Coefficients of Hb & HbO2 [4]

The graph above shows the various values for the Molar Extinction Coefficient () for both haemoglobin and deoxyhaemoglobin. By multiplying the values for the molar extinction coefficient (ε) by the molar concentration (C), we get a value for the Absorption Coefficient. Ideally, the higher the value of u_a, the higher the Absorption Coefficient will be.

The Near-Infrared spectrum begins from about 700nm onwards. So the task at hand is to project an LED within the NIR spectrum, while maximising the difference between Hb and HbO2 absorption rates. From analysis, we can see that the wavelength of Near-Infrared light that we want to transmit onto the arm will be around the region of 750nm.

Image Processing & Machine Vision Background

Throughout this section, the principles under which these image processing techniques operate will be explained. While it may not be possible to explain all the intricacies of each, due to the breadth of material in which they would cover, a brief introduction into these concepts should provide enough of an understanding into how they operate and their importance to the project as a whole. As well as this, a description of certain terms that are continually referenced will also be provided.

Histogram equalisation

An image histogram is a plot which shows the distribution the various pixel values within an image. This distribution is represented by sorting each pixel value into 'bins', which refers to a specified range of pixel values. For example, if there are 8 bins within a greyscale image (8-bit integer) then each 'bin' would have a range of 32. So say if each pixel value was contained in the first bin of an 8-bin histogram of a greyscale image, then every value within the image would have a pixel value from 0-31. This image example would be quite dark and it would be hard to decipher what the image is. Histogram equalisation will edit the pixel values so that all intensities are as equally common as possible. This means that overly bright & washed out images will be adjusted to appear darker, and similarly overly dark images will appear brighter. This is probably best explained graphically below.

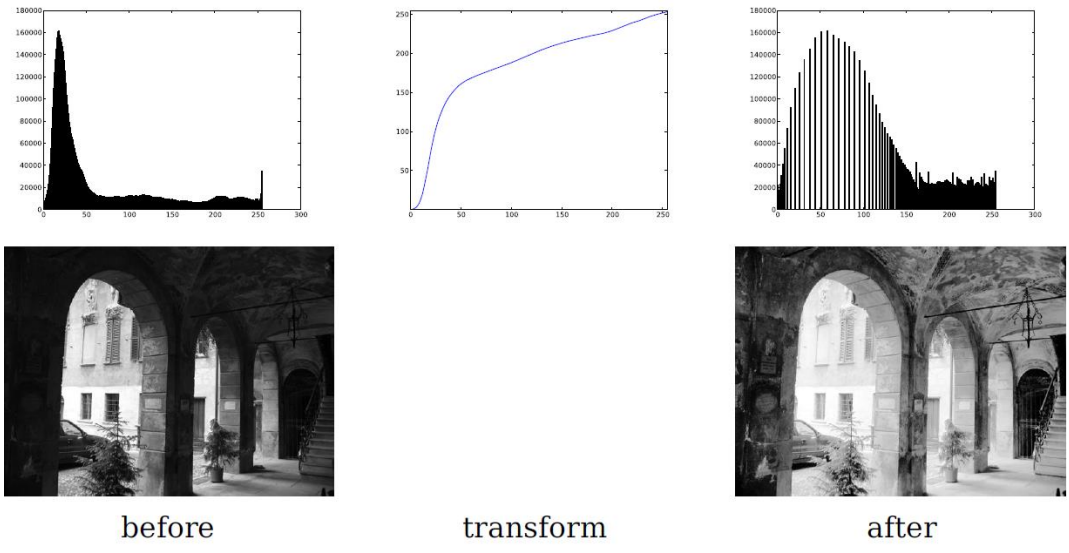


Figure 2 – Process of Histogram Equalisation- Courtesy 'Computer Vision with Python' [5]

Contrast Limited Adaptive Histogram Equilisation (CLAHE)

As we have seen previously, histogram equalisation does not work well in situations where the image consists mostly of extreme bright or dark values (i.e. very few values within the 30 – 210 pixel values within greyscale). This results in a loss in definition and information within the image, as the contrast between dark & bright values are reduced.

To solve this problem, adaptive histogram equalisation is used [6].As opposed to equalising the entire image at once, adaptive histogram equalisation will divide the image into small blocks of a specified size (opencv default is 8x8) and undergo separate equalisation for each block. Thus, in a small area, the histogram output should be confined to a small spread of values within the same bin region. If there is noise in the image however, the histogram output will be amplified. Here is where the contrast limiting aspect of CLAHE comes into play. Within CLAHE there is a specified contrast limit that cannot be exceeded (opencv default is 40). If there is a histogram bin which is above this specified contrast limit, the pixels in question are clipped and uniformly distributed to other bins before applying histogram equalisation.

Edge Detection – Canny Edge Detector

Canny Edge Detection is a specific form of edge detection algorithm designed by John Canny in 1986, which has become one of the industry standards for edge detection. The main aim of this algorithm is to maximise the probability of detecting real edge-points in an image, while minimising the likelihood of falsely detecting non-edge points [7]. It can be comprised of five separate steps, which are undertaken in the following order:

(1) Smoothing: Blurring of the image to remove noise in the image (through the use of a Gaussian filter).
(2) Identifying the gradients within the image: The edges will be marked where the gradients of the image has large magnitudes.
(3) Non-maximum suppression: Only local maxima should be marked as edges.
(4) Double thresholding: Retaining or removing edges from the image based on the strength of a threshold value specified
(5) Edge tracking through hysteresis: Final edges are determined by suppressing all edges that are not connected to a very certain (strong) edge.

Here we see the output of each stage of Canny Edge Detection [8]

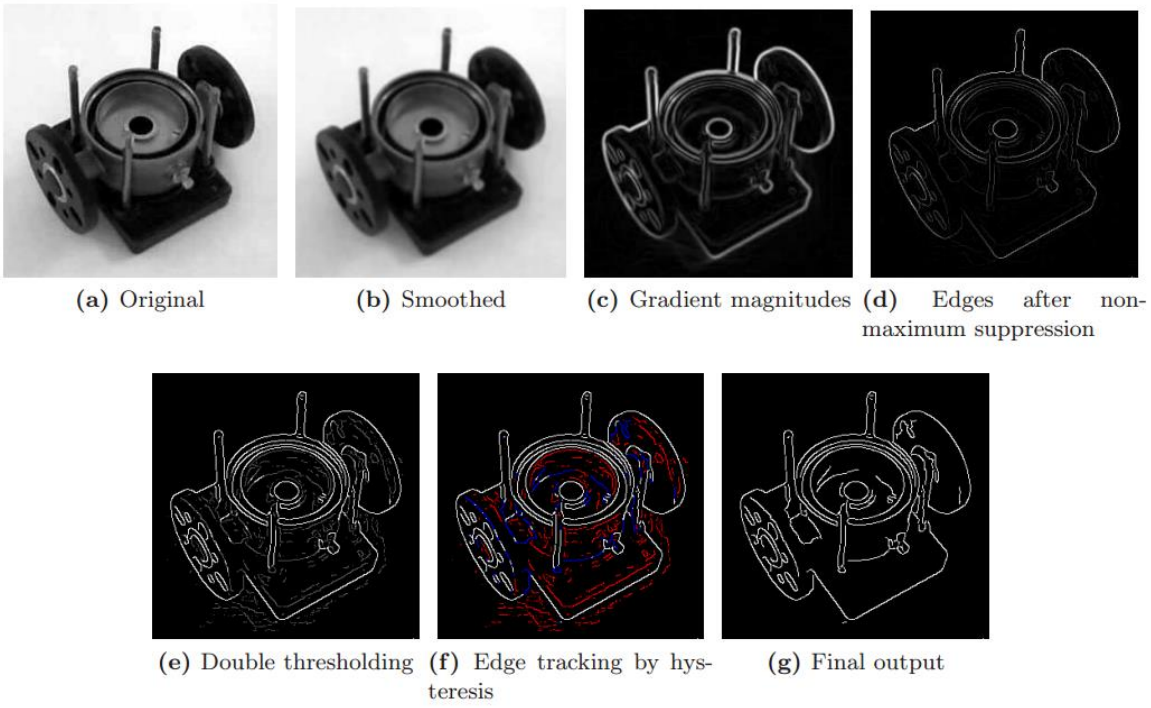


Figure 3 – Stages of Canny Edge Detection

These operations are all handled by the Canny Edge Detector designed in OpenCV, these steps explained briefly above will not have to be manually programmed. Instead, the 'canny' function can just simply be called from the OpenCV library.

SSH

SSH (known as Secure Shell) is a cryptographic network protocol for operating network services securely over an unsecured network. SSH provides a secure connection, through which you can have normal, interactive logins, as well as 'tunnelling' other ports (such as X11 connections) to or from a remote machine. Unlike other protocols such as '*telnet',* 'ssh' encrypts any data sent remotely, whether that be a sent command, a file, or even keystrokes from the remote terminal.

In the context of this project, we will be using '*ssh'* for establishing a remote connection from the Android application to the Raspberry Pi. The Raspberry Pi comes with the 'ssh' protocol as standard with the Raspbian operating system.

Design Implementation

Sensor Development

In the above diagram, we see the basic components of the 'sensor' aspect of the vein mapping system and how they interact with each other. The following system can be explained as follows

- An LED circuit, designed as shown in the PSpice circuit diagram, emits light at a 850nm wavelength. This circuit is powered by the Raspberry Pi GPIO, which provides a source voltage of 5V. This light is then shone upon the area we are interested in obtaining a vein mapping from.
- The idea is that multiple NIR LEDs are used together at various angles to ensure that there is a uniform distribution of infrared light on the area of interest.
- A Raspberry Pi camera module with its Infrared filter removed, is connected to the Raspberry Pi camera slot. This is done so it can interpret the Infrared light being shone upon the area of interest which would otherwise not be visible.
- The camera module is fitted with an infrared optical bandpass filter, which only allows light at a wavelength between 840 – 860nm pass through the lens. This means the camera will now only be able to take in light that operates at this wavelength (i.e. our infrared LED circuit).

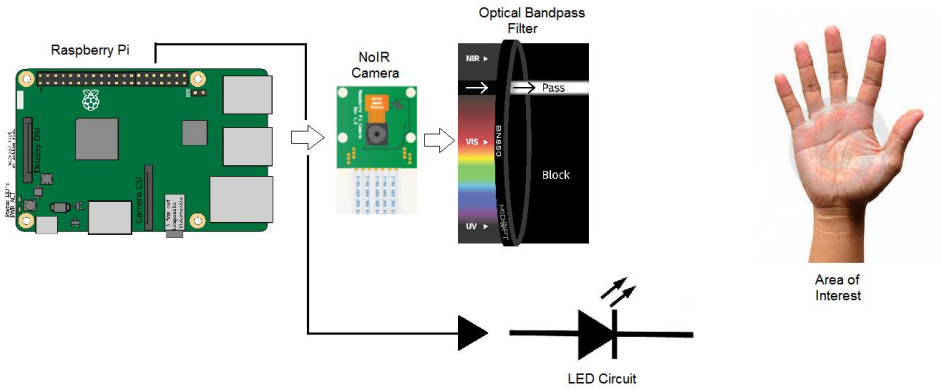


Figure 4 - Sensor Interaction Diagram

Based on the chart previous (Figure 5) which outlined the molar extinction coefficient of Hb02, the initial plan was to use LEDs which emitted light at a wavelength of 750nm. This was adjusted to 850nm as the project went on. This conclusion was reached for two main reasons

- Ease of access to 850nm wavelength LEDs
- 750nm wavelength being too close to the visible spectrum
- Cost

One of the main motivations behind this project was to create a vein-mapping system that would be much cheaper to design & implement than other products on the market. When comparing the price of the products, the 750nm LED's cheapest price was €3.10 p/p, compared to that of an 850nm LED which costs €0.35 p/p. As well as this, 750nm wavelength for an NIR LED is much more uncommon, due to how close the emitting wavelength is to the visible one. The fact that an 850nm emitting LED still gives the user a workable image means that the objective is not compromised, and thusly the decision was made to proceed with the 850nm wavelength LED.

The specific part ordered was the *Digikey HIR 8323/C16*, the datasheet of which is located in the appendices. The characteristics of this LED were the basis of which the LED emitter circuit was designed around. With the Raspberry Pi having a fixed voltage, and the LEDs having fixed characteristics, only the resistor values could be varied to adjust the luminosity of the LEDs. The circuit below was designed taking the following into account.

As well as this 850nm LED, a single LED that emits light within a visible wavelength was also added to the LED array circuit. The specific part ordered was the *Kingbright L-9294QBC-D*, which has a dominant wavelength of 465nm (the blue band of the visible spectrum). As stated previously, the only reason for this LED is to provide guidance for the user in regards to what direction the sensor is pointing in. The characteristics of the 465nm LED were obtained from the datasheet (also located in the appendices) and can be seen below, to be used for the design of the circuit.

Source Voltage (from Raspberry Pi): 5V

850nm LED: V_f = 1.45V 465nm LED: V_f = 3.3V

I_f = 20mA I_f = 20mA

Calculation of Resistor Values:

Branch 1 / 3: Two 850nm LEDs = 1.45V x 2 = 2.9V

5V – 2.9V = 2.1V

R = (V / I) = (2.1V / 0.02A) = 105Ω

Branch 2: One 850nm LED, One 465m LED

1.45V + 3.3V = 4.75V

5V – 4.75V = 0.25V

R = (V / I) = (0.25V / 0.02A) = 12.5Ω ≈ 13 Ω

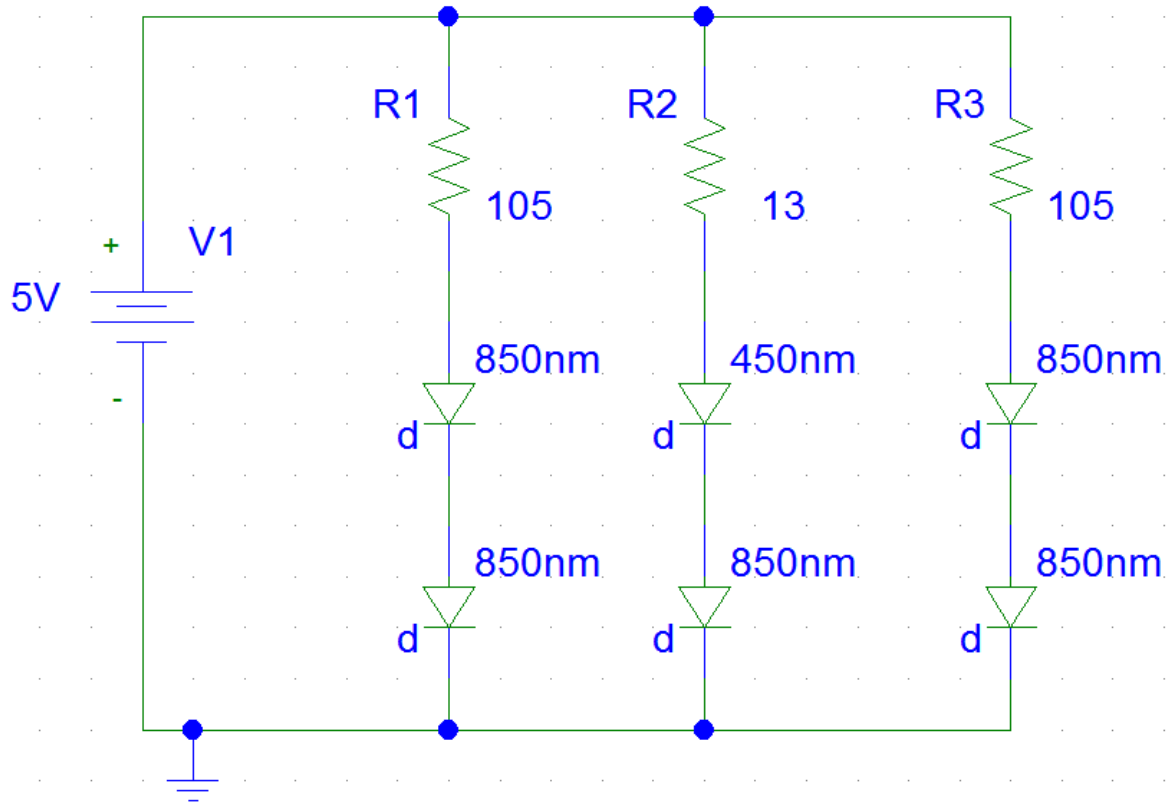


Figure 5 - Theoretical LED Circuit Design

Image Processing Design

For the computer vision and image processing task of the project, various algorithms will be applied so as to decipher the raw data received from the NoIR camera. In terms of what type of raw data we expect to receive, we will only be interested in the area of contact (i.e. the wrist), the rest of the image will be ignored. In terms of what we see in the wrist area, the raw image should show a prominently dark area over each vein (due to the absorbed NIR light) and much more colour around the surrounding areas with a much lower absorption coefficient.

From there, we will begin to implement the various image processing algorithms. While one aspect of this project aims to create a vein mapping system in the most effective & time efficient way as possible, this project also provides an opportunity to delve into new topics and concepts. From an image processing point of view, the aim is to learn many different image processing techniques and do a comparison to see which algorithms & processing models produce the most desirable results. While not being necessarily the most time efficient method, it is the one that provides the largest scope for success as there is a backup plan in place if the initial image processing technique does not work.

A similar project involving a vein mapping system was undertaken in 2007[9], from which Figure 6 is taken from. It shows the 4 stages of image processing that was undertaken to produce a stable & effective vein mapping for a user. The main concept behind this is edge detection.

As discussed previously in the theory behind the Canny Edge Detector, we're dealing with a mathematical modelling system which aims to identify points in a digital image at which the image brightness changes sharply [3]. Because of the fact that there should be a discernible difference between the point of interest (the veins) and the surrounding flesh, edge detection becomes very practical. It should help to dramatically reduce the amount of data that needs to be processed, by filtering out the less relevant information. While this is a very handy tool, it can be very difficult to implement for more complex images such as the one we are dealing with now.

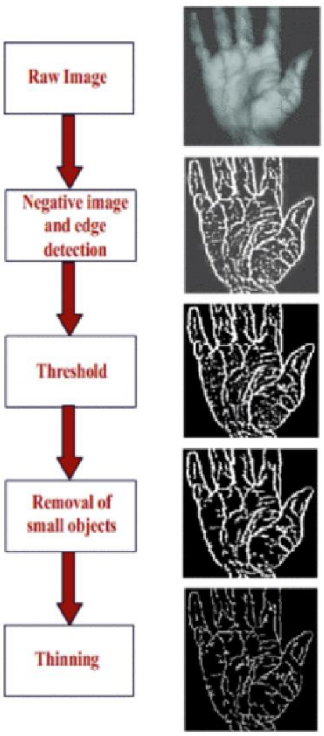


Figure 6 - Block Diagram Image Processing Idea [9]

Ideally, the image processing system that we will aim to create will have Canny Edge Detection being applied to live video being recorded from the NoIR camera. Within the OpenCV library is a Canny function which can be applied to live video streaming. If there is a strong enough contrast in veins vs. skin, then this system will work well. From there, the Android application will be able to receive the live feed from the Raspberry Pi camera.

Once there is an output from the Canny Edge Detector that is of a suitable standard, then other extra processing techniques can be added to the system. Unlike standard edge detection, the Canny Edge Detector will filter the image of noise using a Gaussian filter. If the edges are too thin and need to become more visible, then image 'dilation' can occur. This will work well as the Canny Edge Detector output will be a binary image (pixels are either black or white). Conversely, if the edges are too thick and more precision is needed, then edge thinning[4] can be applied to our image (as seen in the figure above). During this 'thinning' process, the detected edges have been smoothed, which removes the unwanted points [10].

However, this will not be the only type of image processing that we will be performing throughout the process of designing a working vein mapping system. As a backup plan, we shall also be analysing the effects of histogram equalisation on greyscale images of our area of interest. One of the main issues with histogram equalisation is that it cannot be applied to live video in our scenario. The way in which histograms work is based on the pixel value of each pixel within an image. Aside from being much harder to design, as well as being much more resource intensive, the results from the histogram would be almost impossible to interpret as the values would be constantly changing. Thusly, if we were to integrate such a processing technique into our vein mapping system, we would have to ensure that the system would only operate with still images as opposed to video.

Application Development & Design

Here we will discuss and analyse the User Interface of the Android Application. This will be the highest level of the project itself and be the only part in which the user will be interacting with. Firstly, we have the start-up screen for the app. This is what's displayed at the beginning for the purpose of loading the app, while also providing confirmation of what the user has just clicked into (or pressed in the case of a smartphone app). While not necessarily having any functionality, the look of the start-up screen for an app can make a major impression on the user so having a clean and sleek design will be important. The name and design are just for the sake of outline and will be subject to change. To proceed to the next activity, the user is required to press the main button in the centre of the screen with the application name on it ('Vein Mapper Lite')

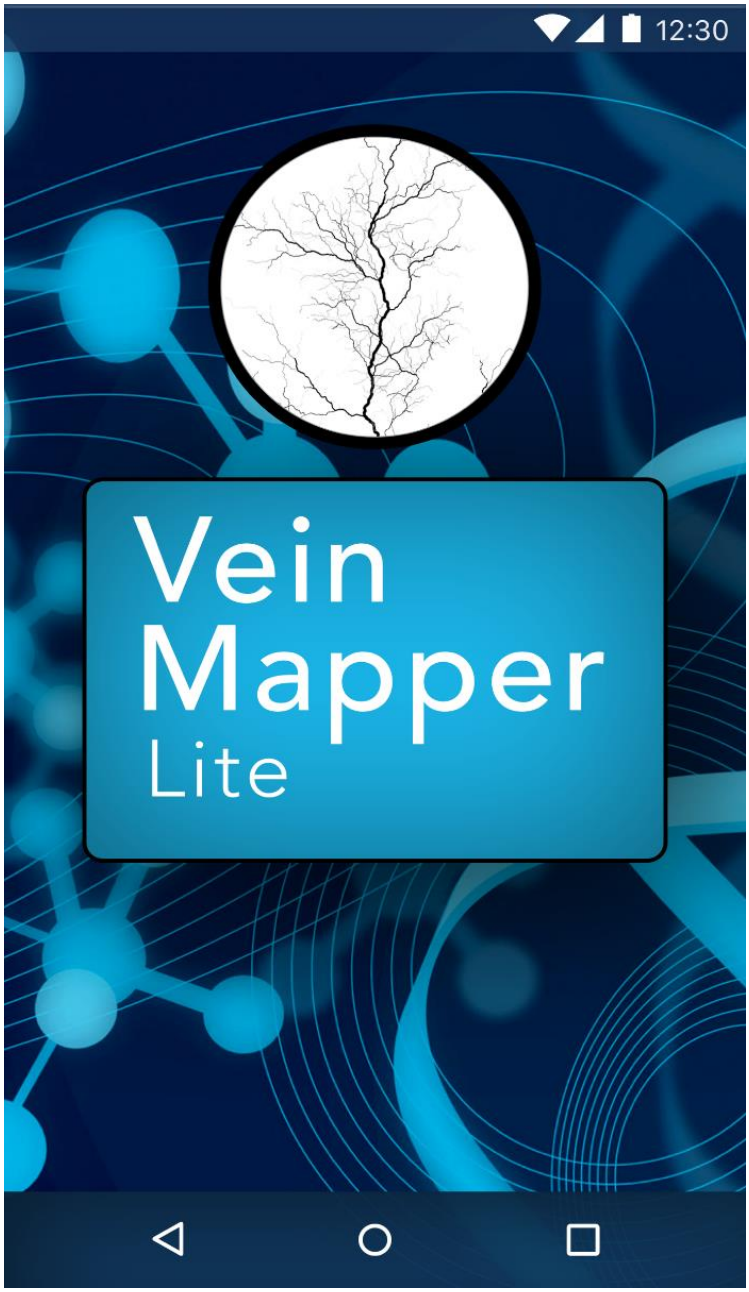


Figure 7 - Android App Startup Screen

Next we have the menu of the application. Within this menu, there are four selection boxes. The first one is the 'Open Scanner' option. This here is the main concept of the application itself and will send a signal to the Raspberry Pi to execute the Python script designed. The LED circuit will be running constantly so there will be no need to turn it on/off.

The 'Configure Sensor' button is responsible for one action, and that is ensuring that the Raspberry Pi and the Android application have established a connection with one another, and are able to communicate back and forth. The communication between the two nodes will

be done through ‘ssh’. As well as this, the check box at the bottom of the menu is also related to the actions of the ‘Configure Sensor’ activity. As is the case with all check boxes, our checkbox is in a Boolean state; if there is an established connection between the ‘Pi’ and the application, the check box is ticked. Otherwise, the box is unticked, and other activities such as ‘Open Scanner’ or ‘Show Previous Scans’ will not execute. Which brings us onto the next button…

The next option, ‘Show Previous Scans’, basically works in the general photo album format and shows the images taken for each of your previous scans. All the photos that have been created as a result of pressing ‘Open Scanner’ are stored on the Raspberry Pi. The ‘Show Previous Scans’ activity will essentially send a command through ‘ssh’ to the Raspberry Pi to send a list of all the photos within the subsequent directory.

Finally, there is the ‘Calendar’ option. While this does not necessarily perform any of the vein-mapping task, it provides a valuable tool to be included with such an app. Such is the nature of injections and the vital medication they provide, typically one would have to regularly receive a ‘shot’. Similar to any other calendar (including the one on every Android phone!), the user should be able to add recurring events at the appropriate time and date.

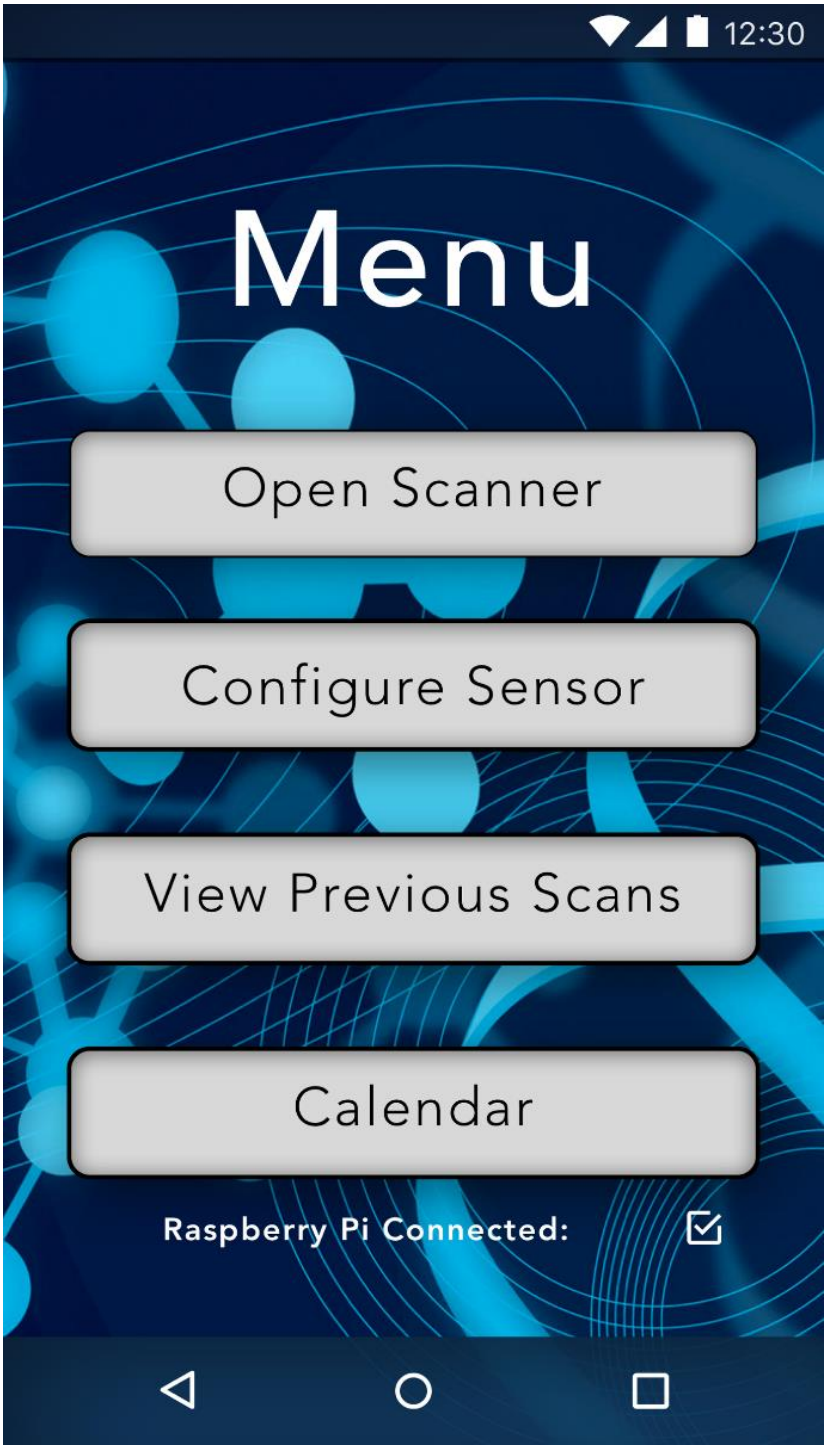


Figure 8 - Android App Main Menu

Here we see the 'Menu' screen as described on the previous page. The elements for this were designed in Sketch, which has fully customisable shapes and images. Features such as the added shadow on each button help to create depth within the screen which looks sleek & professional. Alignment tools help to ensure that every aspect of the design is symmetrical & even.

Sequence Diagram for Application & Raspberry Pi Interaction

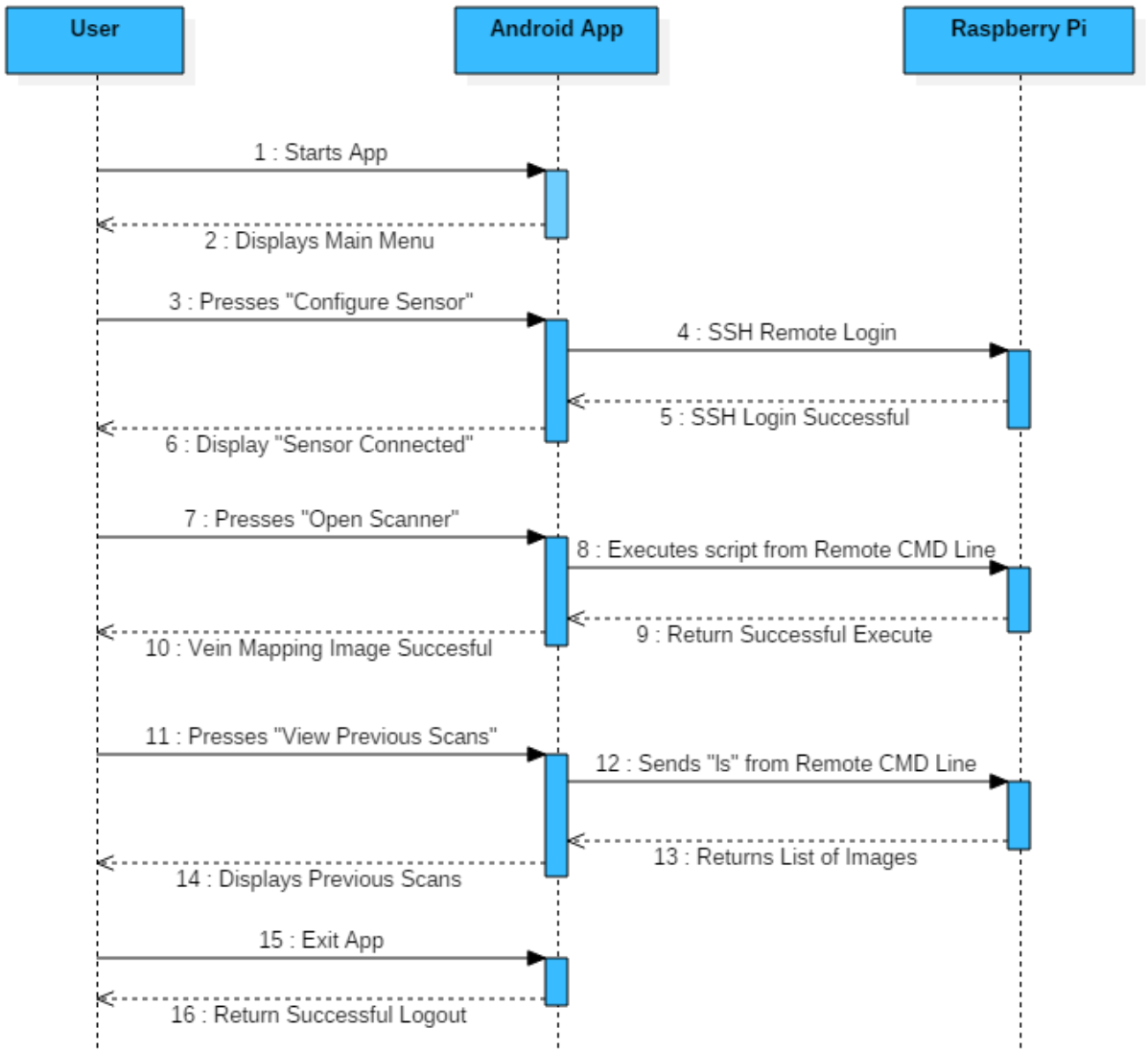


Figure 9 - Sequence Diagram - Android App

Here we see the sequence diagram the Android Application communicating with the Raspberry Pi. When the User presses a button on the App, it sends a signal / instruction for the app to undertake, which is programmed directly into the .java file of the application within Android Studio. Programmed into the .java file is the establishment of SSH communication between the App and the Raspberry Pi. The Android App will send a SSH request to the Raspberry Pi, and if successful the App will be able to communicate remotely with the Raspberry Pi.

Once the remote connection is established, the App will be able to execute the Python script from the Command Line which will take an image of the area of interest, and process the image to accentuate the veins. Similarly, the App will be able to get a list of all previous images taken as a result of running the same Python script by running the 'ls' command in the directory where the images are stored.

Hierarchy of Android Activities

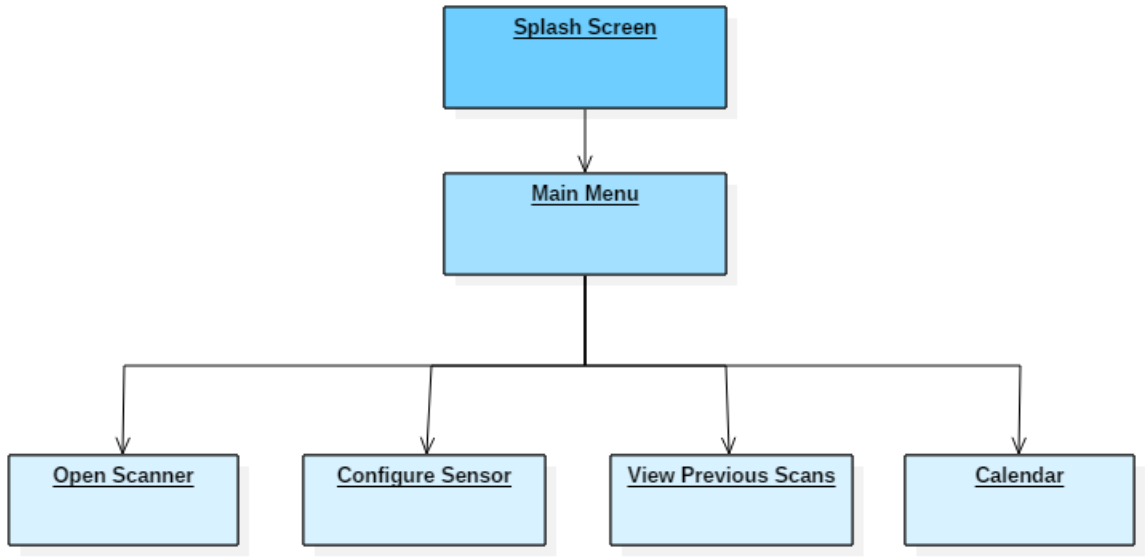


Figure 10 - Android App - Activity Hierarchy

Here we see the hierarchy of the individual activities to be created within Android Studio for the application we will be using. Hierarchy is a vital aspect of the design process within Android Studio. When creating a new activity, each new activity would typically have a parent assigned to it. In the context of this application, the hierarchy in place dictates the direction and mapping of the activities For example, the 'Menu' activity is the child of the splash screen activity. This means that within the *.java* class of the 'Splash Screen' activity, an intent is present where if the central button is pressed, the Application will then display the child process in which it is mapped to.

The same process occurs with ‘Main Menu’ in regards to its four child activities. The buttons within the ‘Menu’ are mapped to the four child processes. If/When an occurrence occurs where the button is pressed, the Application will then display the specific child activity (Open Scanner, etc.). Remember that each activity has an .xml & .java class associated, so it will display the xml output of the child activity as well as perform whatever action has been programmed within the .java class.

Design Testing & Evaluation

Sensor Development

Due to the V = IR relationship, the circuit above was designed so that the resistor values would provide the LEDs with their desired If & Vf values (which were obtained from the datasheet. However, there was an issue with the circuit provided in the context of illuminating veins in the area of interest. The light intensity of the LEDs was far too high at the values outlined in the circuit above. As a result, the images being processed by the Raspberry Pi were completely washed out due to the light-reflective properties of skin.

At this stage it became apparent that the ideal LED circuit to be used would have to be obtained through a few iterations of various circuit configurations to find the optimal luminosity. This means that two aspects of the circuit would have to be varied, that being

- The resistor values
- The number of LEDs in the circuit

To speed up this process, as opposed to going through a trial and error situation trying to obtain various luminosities values from the LED datasheet, variable resistors (also known as rheostats) were added to the circuit. These values could be adjusted in real time to see the changing effectiveness of the circuit. Once the optimal output was obtained, the resistance value of the rheostats were obtained, and resistors with that corresponding value were used.

Figure 11 - LED Circuit Build

This circuit shows only 3 LEDs, one of which is the 465nm LED which exists only to provide guidance for the user. The new characteristics of the new circuit can be seen on the next page. Unlike the previous design, we know the resistor values & the supplied voltage. So from there, we can calculate the current going through each LED.

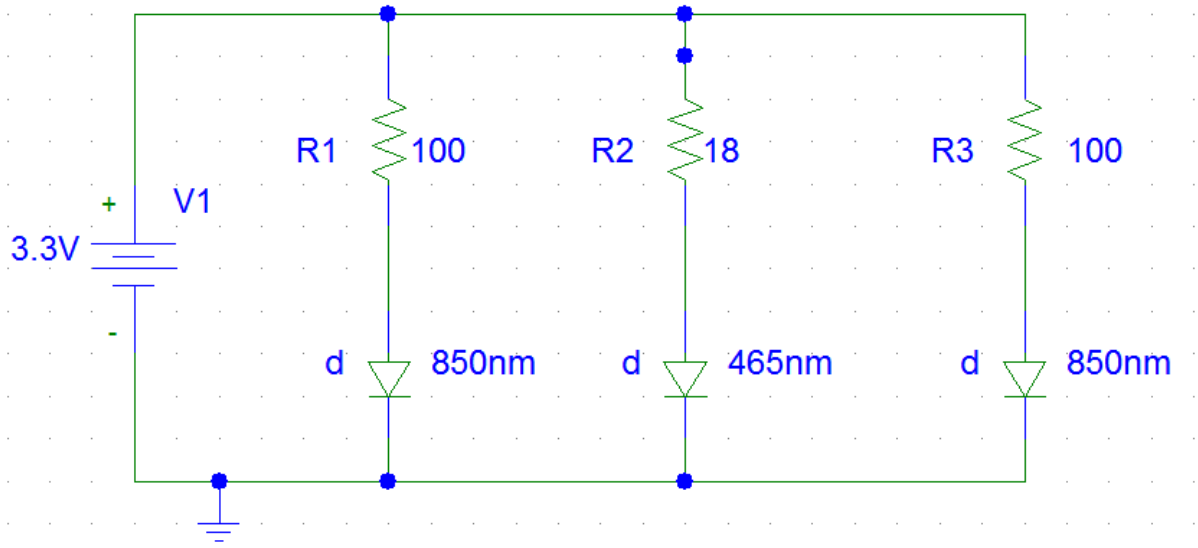
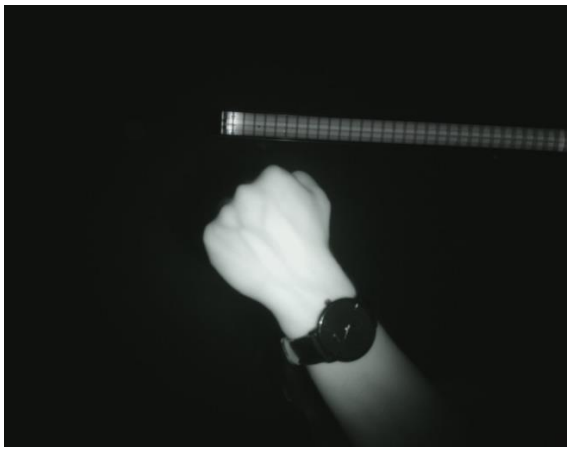


Figure 12 - Practical LED Circuit Design

Calculation of Current through Branch 1/3:

3.3V – 1.45V = 1.85V

I = (V / R) = (1.85V / 100Ω) = 18.5mA



Washed Out Image using Theoretical Circuit
6 LEDs with 5V supply voltage

More Defined Image Using Practical Circuit
3 LEDs with 3.3V supply voltage

Figure 13 - Theoretical vs. Practical Circuit - Image Output

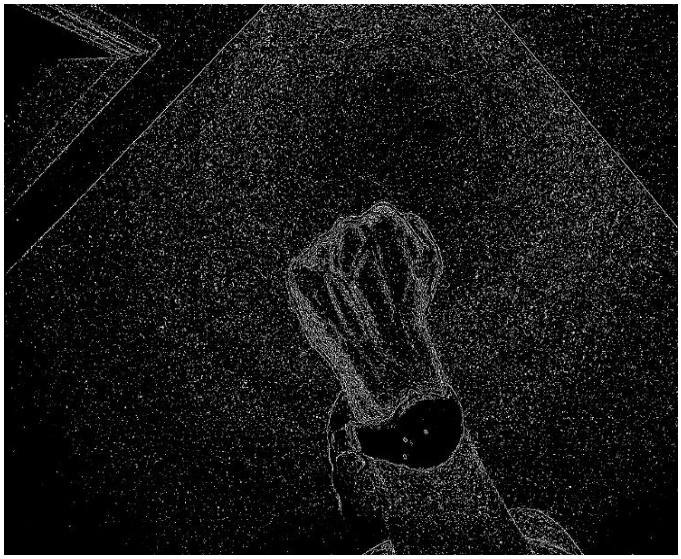
Image Processing Development & Evaluation

Canny Edge Detection – Image Comparison

When using the Canny Edge Detection function within OpenCV, we would first verify its functionality on a still image before seeing if it would be applicable to use on live video.

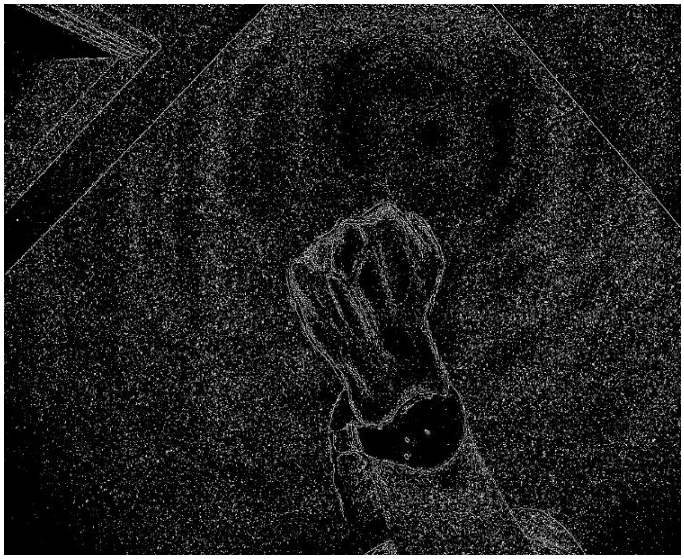
The process of determining what was the best level of thresholding for the edge detection was done through a trial and error process. This proved quite difficult to do, as it was very hard to determine a threshold low enough to identify the veins, while at the same time having a threshold high enough to not pick up noise within the image. Noise became a big factor, due to the resolution of the camera. You can see below, that even at very low thresholds it is hard to identify the veins, while at the same time displaying a huge amount of noise and giving us a much distorted signal. The Canny Edge Detector was applied to both the standard unprocessed images, and the CLAHE equalised images. While applying the Canny Edge Detector to the CLAHE equalised images did give more definition to the veins, it also created a large amount of noise. In essence, it exaggerated the Canny Edge Detector, and had the equivalent effect of lowering the threshold. The images from this should do a decent job of showing this:

The initial plan for the system was to have live video on the Raspberry Pi camera being streamed to the Android with the Canny Edge Detector applied. However, having obtained and reviewed the results obtained from this, it was decided to not have Canny Edge Detection as the main image processing software used to assist with the vein mapping process.



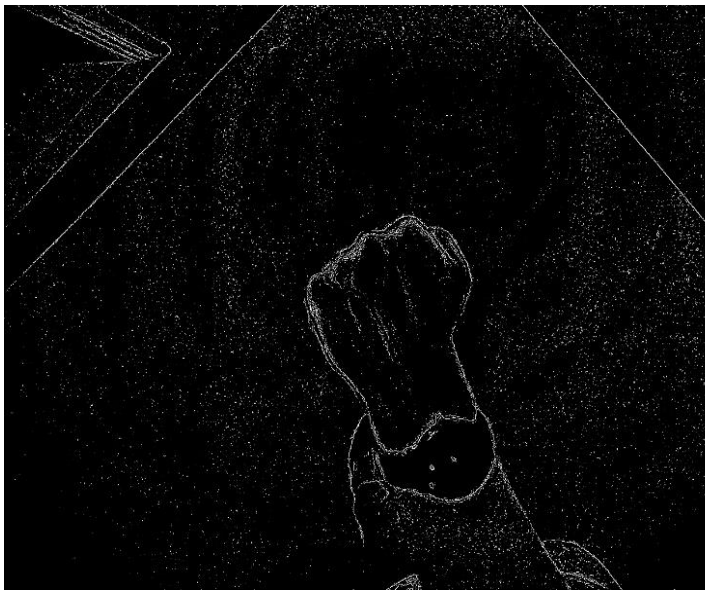
Canny of Standard Greyscale Image

Threshold (15, 15)



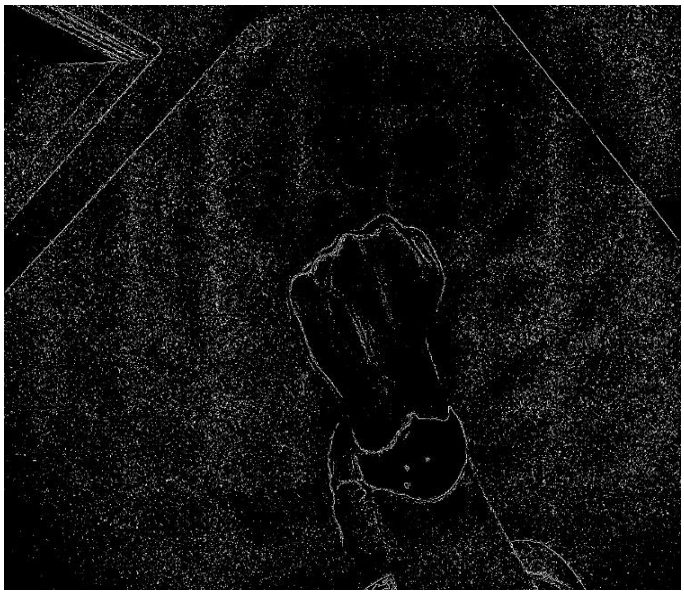
Canny of CLAHE x1 Image

Threshold (35, 35)



Canny of CLAHE x1 Image

Threshold (50, 50)



Canny of CLAHE x2 Image

Threshold (100, 100)

Figure 14 - Canny Edge Detection – Image Comparison

So having completed an analysis & implementation of Canny Edge Detection, we have witnessed the shortcomings of its performance in regards to what we want to achieve. Histogram Equalisation is the method that we will look to next, and see what results that can be achieved by undertaking said technique. It must also be noted that now, the way in which the Android device interprets the vein mapping has been altered. Initially, our plan was to apply a Canny Edge Detector to the live video on the Raspberry Pi camera, and stream that directly to the application. However, we are now going to have to analyse images instead of video. This is due to the shortcomings of histogram equalisation in the context of video, the reasons of which have been explained.

Histogram Equalisation – Image Comparison

Here we see the standard image of our area of interest compared with the same image underdoing different extents of histogram equalisation. The photo in the top right corner shows the effect of standard histogram equalisation. Because of the dark nature of the image, there was a low number of pixel values in the 'bins' with a high value (e.g. ≥ 200). Thus, when the equalisation process was undertaken, this resulted in much higher pixel values for the brightest area of the photo (i.e. the area of interest). Remember the higher the pixel value, the brighter the pixel. The result of this is the washed out image above which is not usable.

The photo in the bottom left hand corner, shows the result of Contrast Limited Adaptive Histogram Equalisation. As opposed equalising the entire image, we see the benefit of performing histogram equalisation on individual blocks and combining together. The block size in which we specified in our Python code is 8x8. What is interesting is that you can see artefacts in the shape of squares in the image above. This makes sense due to the fact the equalisation occurs in separate 8x8 blocks, thus creating a noticeable contrast in each block.

In the bottom right we see the characteristics of CLAHE further exaggerated as this image we are referring to has gone through this process twice. The 'boxy' artefacts are much more visible now. However, they are most prominent in the uniform background, and do not seem to be visible within our area of interest. The veins appear much more prominent than they do in the other photos so this definitely provides us with the most successful implementation of histogram equalisation.

Standard colour photo w/ no Image Processing

Image converted to greyscale w/ Histogram Equalisation

Greyscale image with CLAHE

Greyscale image with CLAHE applied twice

Figure 15 - Histogram Equalisation - Image Comparison

Implementation – Image Processing in Python

Previously we have shown the effects of implementing various image processing techniques in the overall aim of accentuating the veins of a user in their specified area of interest. These techniques have had various successes as we have seen previously. However, each user has their own personal preference, and allowing users to access the various images is not necessarily a bad thing. For this reason, the Python script to be executed remotely by the Android Application will produce 5 separate images, each available to the user in the 'View Previous Scans' activity. These images will be

- Standard Image (no image processing, displayed in Greyscale).
- Image with Histogram Equalisation.
- Image with Contrast Limited Adaptive Histogram Equalisation (CLAHE).
- Image with CLAHE applied twice.
- Image with Canny Edge Detection applied.

Each of these five image types will be stored in their own respective directory within the Raspberry Pi. Below is an explanation of the code used

```
camera = PiCamera()

camera.rotation = 180 # ensure camera has correct orientation
amera.start_preview()

sleep(5) # gives user 5 seconds to adjust camera before taking image


standardPhoto =
camera.capture('/home/pi/1_standardImg/Normal_{:%Y_%m_%d_%H:%M}.jpg'
.format(datetime.datetime.now()))

# convert standard photo to greyscale - less processing required
imgGray =
cv2.imread('/home/pi/1_standardImg/Normal_{:%Y_%m_%d_%H:%M}.jpg'
.format(datetime.datetime.now()), cv2.IMREAD_GRAYSCALE)
```

Here is the basic setup of the camera to take an image using the PiCamera library. The camera is rotated 180° to match the orientation in which we are using the camera. Upon executing the script, *camera.start_preview* will display the camera output of the Raspberry Pi camera onto whatever screen is connected to. It will do so for 5 seconds to give the user an opportunity to adjust the area in which the sensor is directed in.

Note how the name of the file is specific to the time in which it was taken (*format(datetime.datetime.now()*). This means that each time the script is run, a new photo is outputted into the directory specified (in the case of *standardPhoto*, it is saved directly into /home/pi/1_standardImg/). The *standardPhoto* image is also converted into Greyscale to create an image with an 8-bit range of pixel values that is easier to process.

```
# Equalised image
equalisedImg = cv2.equalizeHist(imgGray)
result1 = np.hstack((equalisedImg,imgGray))
cv2.imwrite('/home/pi/2_histImg/HistgramEq_{:%Y_%m_%d_%H:%M}.jpg'.format
(datetime.datetime.now()),equalisedImg)

# Contrast Limited Adaptive Histogram Equalisation
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))

cl1 = clahe.apply(imgGray)

cv2.imwrite('/home/pi/3_CLAHEimg/CLAHE1_{:%Y_%m_%d_%H:%M}.jpg'.format
(datetime.datetime.now()),cl1)
```

Thanks to OpenCV, the process of equalisation (whether it be standard histogram equalisation or CLAHE) is made simple. *cv2.equalizeHist* is a built-in function within the cv2 library, and all that is required is that we write the output to our specified file. The naming convention of the file is the same as previous, in that the date and time are specified within.

CLAHE is also relatively straightforward to implement. First the characteristics of the CLAHE are specified, such as the *clipLimit* and the block size. Once that is defined, your simply apply it to the image of interest, which in our case is the Greyscale version of our initial image.

```
CLAHEImg1 = cv2.imread('/home/pi/3_CLAHEimg/CLAHE1_{:%Y_%m_%d_%H:%M}.jpg'
.format(datetime.datetime.now()), 0)

# Applying CLAHE to the image that has already undergone CLAHE = CLAHE_2
cl12 = clahe.apply(CLAHEImg1)

cv2.imwrite('/home/pi/4_CLAHE_x2_img/CLAHE2_{:%Y_%m_%d_%H:%M}.jpg'.format
(datetime.datetime.now()),cl12)

# Canny Edge Detector - Applied to CLAHE img with specified
threshold of (50,50) - identified as best output
cannyPict = cv2.Canny(CLAHEImg1, 50, 50)
cv2.imwrite('/home/pi/5_CannyImg/Canny_{:%Y_%m_%d_%H:%M}.jpg'
.format(datetime.datetime.now()),cannyPict)

camera.stop_preview()
```

Now, unlike the previous three images produced, the final two processes will be applying their image processing techniques to the CLAHE output. Thus, we need to read in the image that we will be applying both the Canny Edge Detector, & another iteration of CLAHE. The Canny thresholds are defined as '50' based on our previous analysis. Finally, PiCamera exits once this has been completed.

Analysis – Histogram Analysis in Python

Seeing as how histogram equalisation is one of the more effective techniques being used in our vein mapping system, it is important that we actually see the effects that equalisation has on the histograms for our various images. *Matplotlib* is a built in graphics library within Python which is used extensively for producing mathematical figures & graphs in which to analyse. Despite being one of the more basic packages, it is quite effective in many scenarios. We shall be using it to plot the *Pixel Value* (x-axis) vs. *Frequency of Occurrence* (y-axis). The process & functionality of the code will be explained below.

```
imgGray = cv2.imread('/home/pi/Desktop/veinPhoto1.jpg',cv2.IMREAD_GRAYSCALE)

hist,bins = np.histogram(imgGray.flatten(),256,[0,256])
cdf = hist.cumsum()
cdf_normalised = cdf * hist.max()/ cdf.max()

plt.plot(cdf_normalised, color = 'b')
plt.hist(imgGray.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.show()
```

Similar to the previous script, we begin by reading in the file we wish to analyse, converted to Greyscale. As an aside, it should be noted that Python uses a US-American dictionary, so when converting to Greyscale, be wary of possible spelling mistakes. Next we create the histogram. '*Histogram'* is a built-in function within the *'numpy'* library, so all we need do is call it. The two arguments for the *'histogram'* function are the 'pixel values' and 'number of bins'. In our case we have 256 bins, which is the maximum for a Greyscale image (Greyscale = 8 bit). The image will also need to be '*flattened'*, because '*histogram'* only takes a one-dimensional array as input.

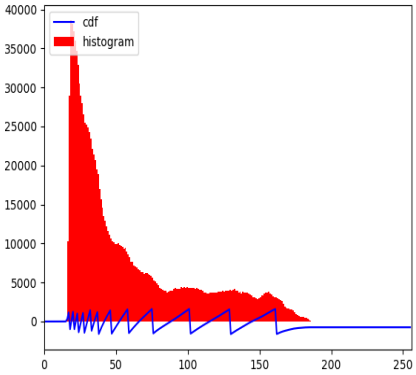
Next we obtain '*cdf',* also known as the Cumulative Distribution Function. Firstly, we obtain the cumulative sum of the histogram (*numpy.cumsum* returns the cumulative sum of the elements along a given axis [5]). This value is then normalised as a result of the calculation above.

Now that the actual calculations have occurred, we deal with the *matplotlib* functions to assist us in assembling a histogram plot. Extras like the addition of a legend, assigning colours & adjusting legend location are all small things that help to add clarity to the output.

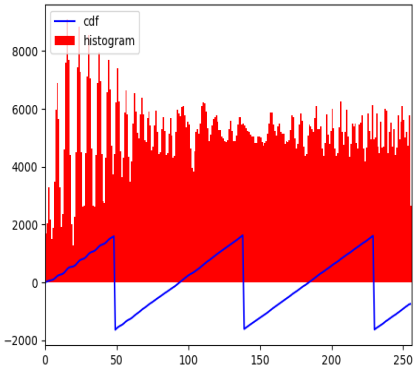
In Figure 16 on the following page, we see the histograms of the images discussed previously. The x-axis represents the 'bin' value, while the y-axis represents the number of pixels contained within its corresponding 'bin'. Each histogram has 256 bins from 0-255 to provide precision to the analysis. As we can see from the first histogram of the standard image with no equalisation, we're dealing with a dark image. This can be seen by the fact that most of our pixels have a pixel value of ≤ 50. The 'cdf' value shown here stands for Cumulative Distribution Function.

In the second histogram shown, we see the results of histogram equalisation, and it is exactly as one would expect. As opposed to having 'bin' values reaching 40,000 (as we can see in the histogram above for the unaltered photo), there is a relatively even spread of values. Almost all 'bins' are within 2,000 of the average value. What this results in, is the increasing of pixel values to values that are ≥ 170 in our area of interest. These high pixel values lead to the 'washed out' effect on the hand that we have already seen.

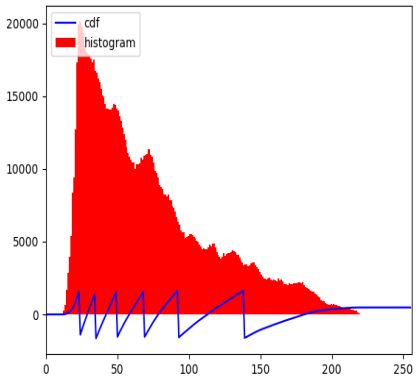
Finally, we look at the histogram has resulted from the effect of CLAHE on our original image. There is a better pixel value spread when compared to our original image. However, unlike the histogram that was the result of standard equalisation, this histogram still has a shape that resembles the original.



Original Histogram



Equalised Histogram



CLAHE Histogram

Figure 16 - Histogram Comparison - Plots

Android Studio – Implementation

Editing the UI - .xml Files

Because the functionality of the 'Startup Screen' & 'Menu Screen' are relatively straightforward, the first thing that we will look at is how we set up our graphical interface. The first task that was undertaken was adding the exported elements designed in Sketch, into the @drawable folder (/FYPapp1/app/res/drawable). These are simply .png files of the elements created within Sketch. These can now be referenced from within any .xml files in the project.

The layout of the xml was then altered to '*LinearLayout*'. '*RelativeLayout*' was the standard layout set for any new activity, and the Android Studio gives warning of this saying that said layout is now obsolete. *LinearLayout* prevents any objects from occupying the same space within an xml file. It can be specified to stack horizontally or vertically. In our case, we want our objects to stack vertically.

Our 'Startup Screen' only contains two elements, which are:

- Our Vein Logo
- Our Vein Mapper Lite button

The first thing we need to ensure, is that each element has a name (`android:id=`). This ensures that the graphical element can be mapped to certain functionality within a class in the .java file.

Now for both of these elements, all we need to do is set the background of each element to its corresponding Sketch imported element in the @drawable folder. Adding a '*layout_gravity*' attribute to each to ensure it stays in the '*central_horizontal*' position is also recommended. Using attributes like '*gravity*' or '*padding*' is much better practice than assigning an absolute (x,y) position to an element. This is because of the varying size of each Android device. The centre co-ordinates of a Nexus5 may not match that of a Huawei P9, for example. Our UI for the 'Startup Screen' should now match that of Fig.7 mentioned earlier

in the report. The MenuActivity .xml file will also have a relatively similar layout to the 'MainActivitySplah', with the major difference being that there are much more elements to deal with. The xml files of both are included in the Appendices for reference.

Creating Intents – Mapping Buttons to Activities

Now, to ensure that the 'Startup Screen' (referred to as MainActivitySplash within the project) accesses the Menu once the central button is pressed, we are going to have to create a new *'Intent'* within our MainActivitySplash.java class. This is done through the following code within the public MainActivitySplash class.

```
public void vein_map_lite_button(View view){
    Intent startNewActivity = new Intent(this, MenuActivity.class);
    startActivity(startNewActivity);
}
```

'*vein_map_lite_button'* refers to ID name of the button in our .xml file. From there, we utilise the '*Intent*' function to *startNewActivity*. The new activity in this case is our activity which we have already created, our 'MenuActivity'. When creating 'MenuActivity', it was also explicitly stated that 'MainActivitySplash' is the hierarchal parent of 'MenuActivity'. The four buttons within 'MenuActivity' which create the link to the four child activities of MenuActivity follow this exact same structure.

While 'MainActivitySplash' & 'MenuActivity' demonstrate the procedure required to interact between separate activities in an Android application, they don't provide a huge amount of functionality.

The main functionality in our 'Vein Mapper Lite' application is the establishment of an SSH connection between the application and the Raspberry Pi. To do this we will need to utilise two specific libraries, *Jsch* and *Jzlib*. 'Jsch' is a pure java implementation of SSH2, while 'Jzlib' is a java implementation of the ZLIB compression library. 'Jsch' will be used to establish the ssh connection, and 'Jzlib' will be utilised for the transfer of data between the two entities.

Neither of these libraries are in Android Studio, and thus will need to be downloaded and imported into the project. To do this, the downloaded .zip files were into the project's library folder (/FYPapp1/app/libs), and added as dependency files within the project structure. To verify that the specified libraries were added to the project, 'build gradle' was run and zero errors were displayed. Now the two libraries can be imported & utilised in each .java activity.

When trying to establish an ssh connection using Jsch, the following are the key functions to be called within the class you are using in your activity.

```
new AsyncTask <Integer, Void, Void>() {
    @Override
     Protected Void doInBackground(Integer, parameters, etc){
          try {
               String = raspPiIP = "192.168.0.53";
               executeRemoteCommand("pi", "<raspPiPassword>", raspPiIP, 22);
             }
          Catch (Exception e) {
             e.printStackTrace();
               }
```

This will attempt to establish a connection with the host machine (the RaspberryPi) whose IP is stated above. This is done outside the UI thread through the use of exception handling. The reason for this is because if the UI thread is executing a network call it can't repaint the UI so your users sees a frozen UI that doesn't respond to them while the app is waiting on the network call to finish. The onCreate() invokes this other thread using AsyncTask (as seen above). When, or if, the separate thread finishes it can post the results back to the UI thread and safely update your application's UI from the UI thread.

Review of Results & Conclusions

As of the writing of this report, the system has reached a stage where it is capable of producing a relatively prominent vein mapping of a human body. In the process of creating such a system, there has been a thorough investigation into the many different ways in which such a system could be achieved.

From an improvement point of view, there is still work that could be done to improve the cross communication between the Android App and the Raspberry Pi. This is where further work and analysis will be applied in future weeks to have the project working at an optimal level.

Although the initial plan of applying the Canny Edge Detector to the Raspberry Pi's live video feed did not end up producing the desired results, the fact that there were other options that were pursued in conjuncture with edge detection shows the benefit of having multiple options at hand. As well as this, the benefit of learning the operation of certain image processing techniques are very beneficial in helping to shape a broader understanding of computer & machine vision.

References

[1] python.org, Beginners Guide/Overview, 2017 https://wiki.python.org/moin/BeginnersGuide/Overview.html.

[2] Google, “developer.android.com/studio/intro,” 2017. [Online].

[3] opencv.org, About OpenCV[Online], 2017 http://opencv.org/about.html

[4] H. W. L.V. Wang, Biomedical Optics Principles & Imaging, Wiley, 2007.

[5] J. E. Solem, Computer Vision with Python, Creative Commons, 2012.

[6] Docs.opencv.org, Histogram Equalization, 2015 http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html.

[7] J. Canny, “A Computational Approach to Edge Detection. Pattern Analysis & Machine Intelligence,” IEEE, 1986.

[8] 09gr820, “Canny Edge Detection,” Lecture Notes, Kingston, 2009.

[9] J. T. T. S.Crisan, “A Low Cost Vein Detection System Using Near Infrared Radiation,” in Sensors Applications Symposium, 2007.

[10] Szeliski, Computer Vision: Algorithms & Applications, Springer, 2010.

Acknowledgements

First of all, I would like to thank both of my supervisors, Dr. Sean McGrath & Dr. Colin Flanagan, for all their help throughout the project timeline. They were both extremely helpful & approachable, and their expertise on the subject material

I would also like to thank the lab technicians who helped throughout the entirety of this project's development, most notably Jimmy O' Sullivan, who was extremely helpful with process of sourcing parts for the project.

Appendices

Smartphone Medical Toolkit

UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

Student name: Cian O'Dwyer

ID Number : 13127381

Programme of study: Electronic & Computer Engineering (LM118)

Introduction

The specification of this project is to demonstrate the capabilities that the modern day smartphone has in a biomedical context.

Specifically, we will be designing a vein mapping system to assist with the process of injection. The usefulness of such a product could have huge benefits & implications for the medical sector at large, particularly in third world countries.

Due to the unsanitary nature of needles, and the importance of medicine administered through the process of injection, the aiding of said injection process will lead to lower skill level required for performing such a task. As well as this, it would also lead to a dramatic decrease in capital spent on resources.

In an age where smartphones are widespread in even the poorest areas of the world, incorporating an Android application into the vein mapping system creates an effective product in a format that is easy to use & is familiar to the average person.

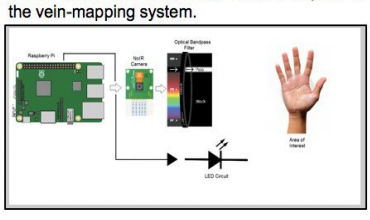
Aim

The overall aim of the system can be split into three separate sections:

- Create an LED circuit, powered by a Raspberry Pi, to emit infrared light at a certain wavelength to avail of the light-absorption properties of veins in the body.
- Using the same Raspberry Pi, take in images of the area of interest. Furthermore, incorporate various image processing techniques to further accentuate the veins within the area of interest.
- Create an Android application which integrates seamlessly with the Raspberry Pi, allowing users to see the results of the vein-mapping system in real time, while also provide an easy-to-navigate user experience.

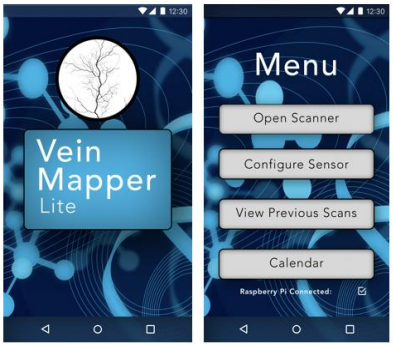
Method

- A Raspberry Pi NoIR camera was acquired, which is a camera module which allows the system to take in infrared light data, which would otherwise be not visible to a user.
- An Infrared LED circuit was designed, emitting light at an 850nm wavelength.
- Voltage applied to the system using the built-in Raspberry Pi GPIO (3.3V for Vin).
- An optical filter with a narrow band-pass was placed over the camera to ensure only light within our wavelength band of interest would be observed.
- The combination of the camera, filter, and LED circuit create the hand-held 'sensor aspect of the vein-mapping system.



Basic Diagram showing how the components of the 'sensor' interact with one another.

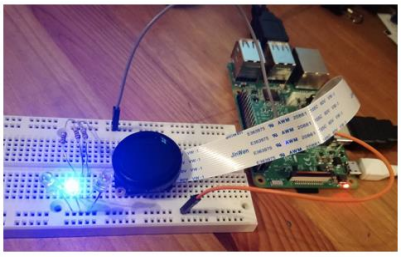
- Various mage processing techniques were applied to the obtained images.
- Specifically, histogram equalisation was undertaken on the obtained images to provide greater contrast between the veins & surrounding area.
- The Android application was designed within Android Studio, with the graphical UI and layout of the application designed in Sketch.
- The Android application will be communicating with the Raspberry Pi using the 'ssh' protocol.



User Interface of the Android App, displaying the Startup & Main Menu screens respectively.

Results

- A LED that emits light within the visible spectrum (i.e. 450nm) was also added to provide guidance for the user.
- Initial circuit design altered. Resistor values increased to reduce voltage & current into LEDs. Number of LEDs also reduced to remove light saturation on image.
- Variable resistors used to obtain optimal luminosity of LEDs (Optimal value R = 100Ω).
- Histogram equalisation performed on image, gives unsatisfactory results (image appears washed-out in areas of interest).
- Specific type of histogram equalisation known as 'CLAHE' performed, gives us our desired result (as shown).
- Python script created on the Raspberry Pi will take photo with the NoIR camera, undergo CLAHE on the image taken, and save the image on the system. Script designed so that no overwriting of images occurs (new photo w/ new name created each time script is executed).



Testing functionality of the IR LED circuit. Note the blue LED in between the two 850nm LEDs, helps user identify exactly where the system is directed at. NoIR camera only identifiable by ribbon, optical band-pass filter placed on top of it.

- 'View Previous Scans' function within the Android application successfully displays all previous photos stored on the Raspberry Pi.
- 'Configure Sensor' button is used to ensure the Raspberry Pi and application are able to communicate with each other. When this button is pressed, the application will 'ssh' into the Raspberry Pi. This is only necessary if the 'Raspberry Pi Connected' check-box at the bottom of the main menu is unticked.
- 'Open Scanner' button executes the Python script remotely from the Raspberry Pi Command Line to give image with accentuated vein mapping of the area of interest.
- Images below show the comparison of the before & after of the Vein Mapping system described here.

Image 1 = Standard photo (No IR Light, no Image Processing)
Image 2 = Processed image w/ IR LEDs & optical band-pass filter

Conclusion and personal reflection

I believe the design & implementation of this above system have been achieved effectively. While there is further room for improvement required for making this a marketable product, the overall functionality provide a solid foundation for which to refine and perfect.

Certain aspects of the project, such as the way the application and Raspberry Pi interacted with each other, could have been implemented in an easier way. However, I wanted to use this as an opportunity to get a better understanding of SQL databases and how they function.

Overall, I think this project as a whole was perfect for giving me a platform to go on to learn many skills (such as Python programming, machine vision, aspects of biomedical science). At the same time, it allowed me to apply these skills to a real-world application whose impacts can be understood by both myself and the general public.

Acknowledgements

I would like to thank both of my supervisors, Dr. Sean McGrath & Dr. Colin Flanagan. Their help & assistance proved vital within both the design & implementation process of the project. I would also like to thank the ECE technical staff who helped me along the way with their wealth of experience.

E&CE Department of Electronic & Computer Engineering

Electro-Optical Characteristics (Ta=25℃)

Parameter	Symbol	Condition	Min.	Typ.	Max.	Units
Radiant Intensity	I_E	I_F=20mA	20	30	50	mW/sr
		I_F=100mA Pulse Width≦100 μs ,Duty≦1%	--	120	--	
		I_F=1A Pulse Width≦100 μs ,Duty≦1%.	--	1200	--	
Peak Wavelength	λp	I_F=20mA	--	850	--	nm
Spectral Bandwidth	Δλ	I_F=20mA	--	40	--	nm
Forward Voltage	V_F	I_F=20mA	--	1.45	1.65	V
		I_F=100mA Pulse Width≦100 μs ,Duty≦1%	--	1.80	2.40	
		I_F=1A Pulse Width≦100 μs ,Duty≦1%.	--	4.10	5.25	
Reverse Current	I_R	V_R=5V	--	--	10	μA
View Angle	2θ 1/2	I_F=20mA	--	30	--	deg

Rank

Condition：I_F=20mA

Unit：mW/sr

Bin Number	A	B
Min	20	25
Max	30	50

Typical Electro-Optical Characteristics Curves

Fig.1 Forward Current vs. Ambient Temperature

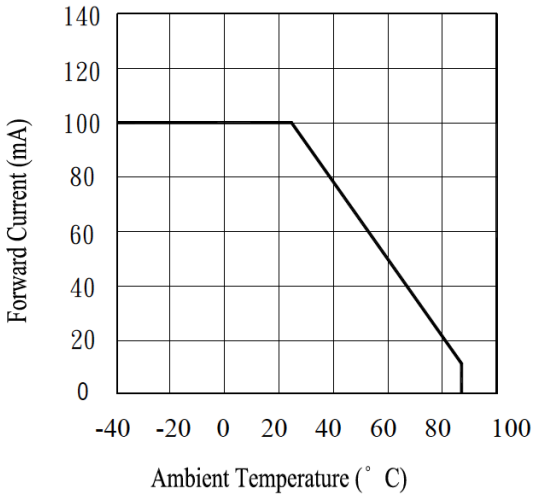


Fig.2 Spectral Distribution

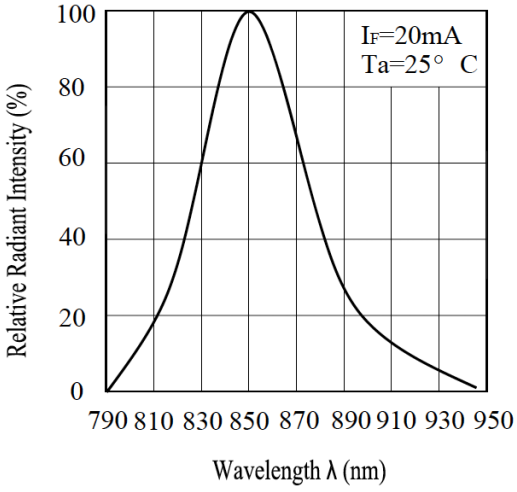


Fig.3 Peak Emission Wavelength Ambient Temperature

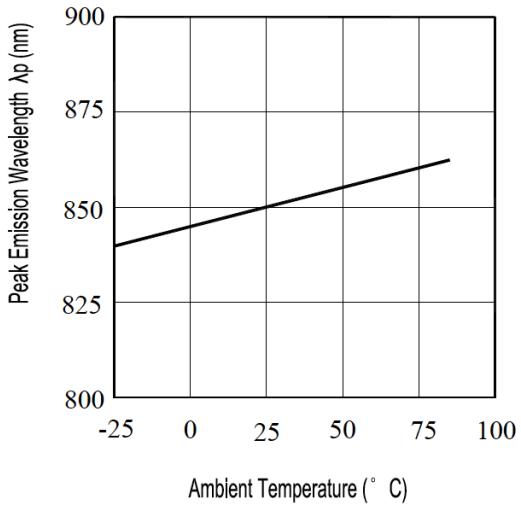
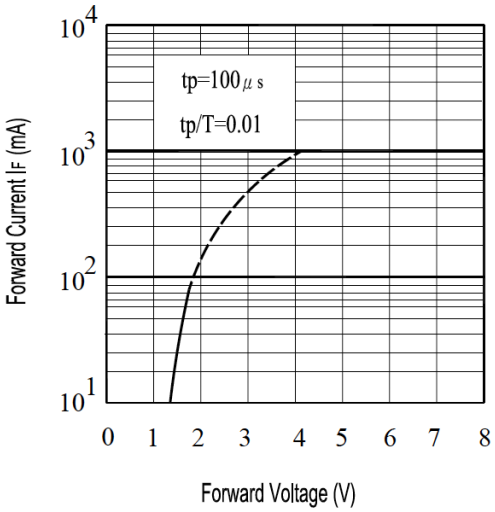


Fig.4 Forward Current vs. Forward Voltage



Kingbright

Selection Guide

Part No.	Dice	Lens Type	Iv (mcd) [2] @ 20mA		Viewing Angle [1]
			Min.	Typ.	2θ1/2
L-9294QBC-D	Blue (InGaN)	Water Clear	200	500	60°

Notes:
1. θ1/2 is the angle from optical centerline where the luminous intensity is 1/2 of the optical peak value.
2. Luminous intensity/ luminous Flux: +/-15%.
3. Luminous intensity value is traceable to the CIE127-2007 compliant national standards.

Electrical / Optical Characteristics at TA=25°C

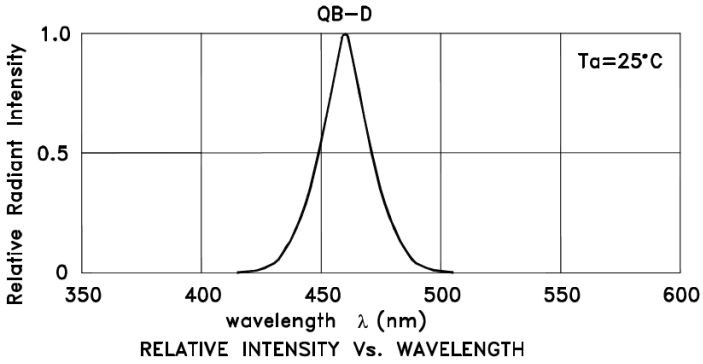
Symbol	Parameter	Device	Typ.	Max.	Units	Test Conditions
λpeak	Peak Wavelength	Blue	460		nm	I_F=20mA
λD [1]	Dominant Wavelength	Blue	465		nm	I_F=20mA
Δλ1/2	Spectral Line Half-width	Blue	25		nm	I_F=20mA
C	Capacitance	Blue	100		pF	V_F=0V;f=1MHz
V_F [2]	Forward Voltage	Blue	3.3	4	V	I_F=20mA
I_R	Reverse Current	Blue		50	uA	V_R = 5V

Notes:
1.Wavelength: +/-1nm.
2. Forward Voltage: +/-0.1V.
3. Wavelength value is traceable to the CIE127-2007 compliant national standards.

Absolute Maximum Ratings at TA=25°C

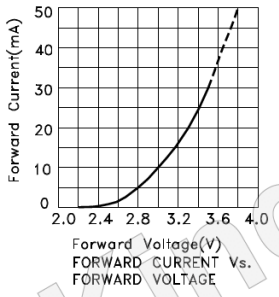
Parameter	Blue	Units
Power dissipation	120	mW
DC Forward Current	30	mA
Peak Forward Current [1]	150	mA
Reverse Voltage	5	V
Operating/Storage Temperature	-40°C To +85°C	
Lead Solder Temperature [2]	260°C For 3 Seconds	
Lead Solder Temperature [3]	260°C For 5 Seconds	

Notes:
1. 1/10 Duty Cycle, 0.1ms Pulse Width.
2. 2mm below package base.
3. 5mm below package base.

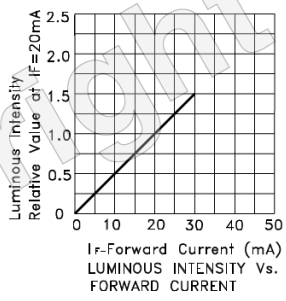


RELATIVE INTENSITY Vs. WAVELENGTH

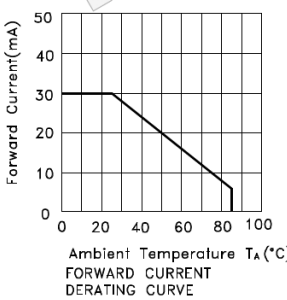
Blue **L-9294QBC-D**



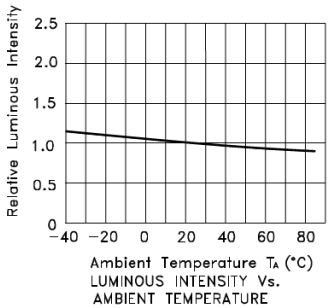
FORWARD CURRENT Vs.
FORWARD VOLTAGE



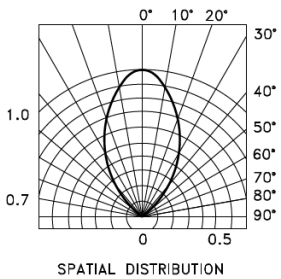
LUMINOUS INTENSITY Vs.
FORWARD CURRENT



FORWARD CURRENT
DERATING CURVE



LUMINOUS INTENSITY Vs.
AMBIENT TEMPERATURE



SPATIAL DISTRIBUTION

Android Studio XML file – Splash Screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="wrap_content"
  android:layout_height="match_parent"
  android:background="@drawable/dna_background2"
  android:paddingBottom="10dp"
  android:paddingLeft="10dp"
  android:paddingRight="10dp"
  android:paddingTop="10dp"
  android:visibility="visible"
  tools:context="com.example.patriciaodwyer.fypapp1.MainActivitySplash"
  android:orientation="vertical"
  android:weightSum="1">

  <ImageView
     android:id="@+id/vein_logo"
     android:layout_width="wrap_content"
     android:layout_height="200dp"
     android:paddingBottom="25dp"
     android:paddingTop="25dp"
     android:layout_gravity="center_horizontal"
     app:srcCompat="@drawable/veins_icon"
     tools:ignore="ContentDescription"/>

  <Button
     android:id="@+id/vein_map_lite_button"
     android:onClick="vein_map_lite_button"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:layout_gravity="center_horizontal"
     android:background="@drawable/vein_mapper_lite" />

</LinearLayout>
```

Android Studio XML file – Menu

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:maxLines="1"
  android:orientation="vertical"
  android:paddingBottom="10dp"
  android:paddingLeft="30dp"
  android:paddingRight="30dp"
  android:paddingTop="50dp"
  app:layout_behavior="@string/appbar_scrolling_view_behavior"
  tools:context="com.example.patriciaodwyer.fypapp1.MenuActivity"
  tools:showIn="@layout/activity_menu"
  android:weightSum="1">

  <ImageView
     android:id="@+id/menu_text"
     android:layout_width="200dp"
     android:layout_height="60dp"
```

```
    android:layout_gravity="center_horizontal"
    android:background="@drawable/menu"
    android:contentDescription="@string/title_activity_menu"
    android:paddingLeft="70dp"
    android:paddingRight="70dp"
    tools:layout_editor_absoluteX="80dp"
    tools:layout_editor_absoluteY="30dp" />

  <Space
    android:layout_width="match_parent"
    android:layout_height="20dp" />

  <Button
    android:id="@+id/button_open"
    android:layout_width="300dp"
    android:layout_height="90dp"
    android:layout_gravity="center_horizontal"
    android:background="@drawable/button_open"
    tools:layout_editor_absoluteX="30dp"
    tools:layout_editor_absoluteY="155dp" />

  <Button
    android:id="@+id/button_config"
    android:layout_width="300dp"
    android:layout_height="90dp"
    android:layout_gravity="center_horizontal"
    android:background="@drawable/button_config"
    tools:layout_editor_absoluteX="30dp"
    tools:layout_editor_absoluteY="245dp" />

  <Button
    android:id="@+id/button_view_prev"
    android:layout_width="300dp"
    android:layout_height="90dp"
    android:layout_gravity="center_horizontal"
    android:background="@drawable/button_previous"
    tools:layout_editor_absoluteX="30dp"
    tools:layout_editor_absoluteY="335dp" />

  <Button
    android:id="@+id/button_calendar"
    android:layout_width="300dp"
    android:layout_height="90dp"
    android:layout_gravity="center_horizontal"
    android:background="@drawable/button_calendar"
    tools:layout_editor_absoluteX="30dp"
    tools:layout_editor_absoluteY="425dp" />

  <CheckBox
    android:id="@+id/check_box_rasppi"
    style="@android:style/Widget.CompoundButton.CheckBox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:checked="true"
    android:duplicateParentState="true"
    android:shadowColor="#ffffff"
    android:text="@string/rasp_pi_connect"
    android:textColor="#ffffff" />

</LinearLayout>
```