MOBOFOTO: A MOBILE PLATFORM FOR CONCENTRATION MEASUREMENT THROUGH COLORIMETRIC ANALYSIS

BY

CHENGPENG HU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Logan Gang Liu

Abstract

Traditional colorimetric measurement is widely used in chemical concentration estimation. However, compared to the laboratory solutions for accurate measurements with professional measuring equipment, colorimetric measurement is often more qualitative than quantitative. It would be most productive if we were only looking for a range of readings to draw a qualitative analysis conclusion. This means the traditional colorimetric measurement would only be an appropriate implementation for pre-medical self-diagnoses at home. Since it can only provide limited information about the tested chemical solutions, we are now featuring MoboFoto, an integrated mobile platform applying image analysis based on traditional colorimetric method to provide a more quantitative measurement, which can be used clinically. The mobile integration enables high-accuracy measurement and data visualization with a more affordable cost in a user-friendly setup environment. And the power of mobile computing also provides interfaces to capture, extract and aggregate measurement data for trend analysis from a medical perspective. Specifically, this research focuses on the exploration of the feasibility of high-accuracy colorimetric measurement as well as hardware-software implementation for the entire mobile platform.

Acknowledgments

The author wishes to express sincere appreciation to his advisor, Prof. Gang Logan Liu, for his assistance in preparing this thesis. Also, thanks to X. Zhou, X. Wang and G. Lin, who genuinely helped me in 3D printing, peripherals design and sample preparation. Finally, thanks to my family and my girlfriend, Yue Wang, for consistent support to help me overcome the difficulties in the process of earning my master's degree.

Contents

1. Introduction

1.1 Overview

The sensing of solution concentration that directly relates to pre-medical diagnosis is now an important topic of research. Numerous products have been developed for patients at home or medical staff at the clinic to monitor health condition indicators such as concentration of glucose. Compared to those sophisticated analysis devices in the lab, the most widely used tools, such as chemical sensing strips, can only provide a qualitative result due to the availability. However, the rising awareness of health in common people and the demand for finer data for clinical analysis require a new solution. Thus, the goal for this thesis project is to develop a sensing platform that can provide quantitative data for further analysis and at the same time still retain the advantages of qualitative methods such as low cost.

According to International Data Corporation (IDC), the worldwide smartphone market grew 13.0% for the year ending in 2015 Q2 with 341.5 million shipments. And with the demand for high-end smartphones, fine camera peripherals are now more accessible with cheaper cost. Meanwhile, the development of mobile computing power largely enables image processing in real-time. Therefore, the integration of cheap mobile technology with chemical strip sensing now becomes a popular research topic, especially for the colorimetric sensing of chemical concentrations on a smartphone [1]. Some research uses smartphone-based colorimetry for multi-analyte sensing arrays [2], some develops a mobile-platform to monitor the chlorine concentration in water [3], [4] and the others rely on color sensing to determine the pH and O_2 with mobile devices [5], [6]. Meanwhile, we are seeking an opportunity to measure the health index chemical concentrations such as glucose using existing sensing strip technology on a smartphone.

This thesis introduces the MoboFoto platform, which has two major functionalities: the sensing of health condition indices such as glucose concentration and the sensing of pregnancy indices such as ovulation concentration. These functionalities were enabled by utilizing the camera component on the smartphone to capture a single image for the sensing strip under test. Then we applied mobile image processing techniques and colorimetric analysis to directly extract color information out of the image. Finally, we map the readings to empirical data for visualization to the end user. All the user needs to do

is insert the sensing strip into the peripheral package for analysis through the associated mobile application.

1.2 Roadmap

Section 1.3 will discuss the background of the thesis and previous work related to colorimetric calibration under certain environments. Chapter 2 will focus on the system design including the hardware utilized in detail. Chapter 3 focuses on building and testing the application on an Android phone. Chapter 4 will focus on evaluating the experimental result. Chapter 5 will conclude the thesis with our expectation of MoboFoto's future usage.

1.3 Background

The earliest initiative for this project came from the demand of measuring a sensing strip. On this sensing strip, there were deployed 10 types of sensing array, in each of which was embedded a type of enzyme used for chemical reaction. When the arrays directly contact the targeted solutions, the chemical reaction process induces the change of the color on the top layer of the array. Under normal/indoor lighting environment, these changes of color can be easily viewed by the naked eye. When the reaction is complete, using the provided reference chart, we can read out the approximate range of the solution concentration immediately. However, it is difficult to precisely quantify the analyte amount, so tremendous improvement is needed.

There are many research works on colorimetric sensing. One of the most valuable works is that by Li Shen [7], which develops a tool that allows high-accuracy measurements for quantifying colors of colorimetric diagnostic assays in a wide range of ambient conditions. Instead of directly using the RGB intensities, Shen used chromaticity values extracted from images taken by smartphone to construct calibration curves of analyte concentrations to overcome inadequacies of the simple RGB analysis, for example, the low sensitivity to dark colors. This setup first requires a standard color reference chart to calibrate the lighting conditions, then it measures the RGB value of the point of interest and maps it to the International Commission on Illumination (CIE) 1931 color space for chromaticity value

quantification [8]. Finally, it performs regression on 3D space to compute the measured value. Shen provides a good way to perform the colorimetric measure and thoroughly discusses the compensation of lighting conditions with the reference chart; however, the tool still has several drawbacks that hinder its application at a clinical level. First, the non-linear regression in 3D space to compute the measured value is an expensive method, which means it takes a significant time to perform the algorithm. Second, the paper discusses the compensation for ambient light and concludes that the position of the light source, light temperature, or outdoor lighting environment can be compensated using a color reference chart because the measured intensities between different ambient light conditions have a linear relationship. However, this conclusion is not universal. Based on our observation, unless the lighting conditions including exposure time, exposure rate, ISO rate and even focal length, etc., are consistent, the calibration reference is not necessarily linear. Last, the position and angle of the handheld smartphone camera in relation to the testing strip had a major effect on measurement value that Shen ignored.

We are trying to build a system that can overcome the major drawbacks described above.

2. System Design

2.1 Peripherals Design

The peripherals comprised three major parts: the smartphone, the lighting control case and the sensing strip holder.

For the smartphone, we choose to use HM2A from Xiaomi Inc., since it has a noticeable advantage in the selling price while the camera mounted is sufficient for the colorimetric analysis. This smartphone has an 800 million-pixel BSI (back side-illuminated) camera, an LED flash light and autofocusing functionality. It also has 1 GB RAM with a 1.5 GHz core which provides sufficient computing power for image processing in real-time. The OS is a deep-customized Android system that almost enables the full privilege to control the camera hardware manually such as adjusting the ISO value, focal length or forcing the exposure time.

Figure 1. Front view of the case

The lighting control case, shown in Figure 1 and 2, is made of aluminum and designed to isolate the sensing strip from illumination except for the flash light. In the front, a control panel is embedded from which the user can launch our application. Between the back camera of the smartphone and the sensing strip, there is a tunnel with reflective material on the tunnel wall.

Figure 2. Back view without back plate

Figure 3. Reflective bubbles

As shown in Figure 3, this reflective material actually consists of stickers with bubble-like bulges. The purpose of these bulges is to increase the reflectivity of the tunnel wall so it can scatter the lighting from the flash light. Since we are trying to simulate a controllable daylight environment when taking the testing images, scattering the flash light is very critical to reduce the effect of light focusing as a single ray model.

Besides controlling the light conditions while taking the picture, another usage of the case is to provide a skeleton to hold the sensing strip with the sensing holder. The user can put the strip under test on the

strip holder and, because the case design has a 30° reclining angle, can smoothly slide the strip holder into the device while keeping the sensing strip uncontaminated.

On the back plate, there is another component that serves for lighting condition calibration. It is a paper holder that supports the color reference chart printed using filter paper. The reason we use filter paper is that it is decently coarse on the surface and thus can reduce the reflection from the flash light.

The color reference chart contains a series of color blocks which cover the color range from standard RGB to greyscale. It has been used to calibrate the lighting conditions in order to adjust the linear regression parameters when we calculate the colorimetric value. In order to calibrate the lighting conditions, we first want to know what is the real RGB value of Black (0, 0, 0) and White (255, 255, 255) being captured through the camera. After extracting this information, we can use Equation 1 to compute the corresponding pixel value for the point of interest.

$$R_{corr} = \left(\frac{256}{R_W - R_B} \right) \left(R_{meas} - R_B \right) \tag{1}$$

Besides the case, we also designed a strip holder, shown in Figure 4, for both sensing strips. The holder was designed in SolidWorks and fabricated using a 3D printer in the lab. The average fabrication time for each holder is approximately 20 minutes.

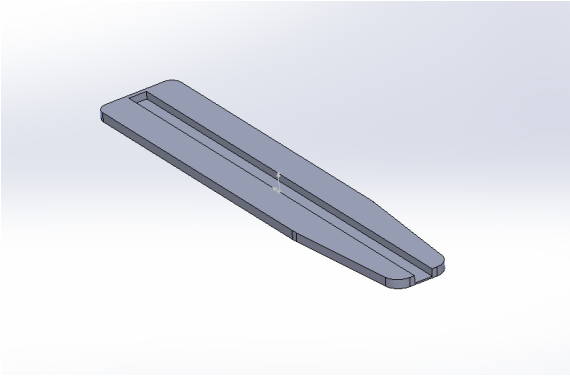

Figure 4. Side view of strip holder

2.2 Sensing Strips

We first need to clarify that all the testing strips are coming from current commercial products. We did not adjust any chemical property of these strips. That means as soon as you have the sensing strip, you would be able to conduct quantitative analysis immediately.

2.2.1 Glucose

The Glucose sensing strip we used is a urine reagent strip for urinalysis. It is a firm plastic strip with several different reagent areas affixed which were designed to conduct the tests for glucose, bilirubin, ketone, pH, etc. These concentration indexes are intended to provide information regarding the status of carbohydrate metabolism, kidney and liver function, acid-base balance and bacteriuria. In Figure 5, the reagent area circled is the test area for glucose concentration.

The glucose test is based on a double sequential enzyme reaction. One enzyme, glucose oxidase, catalyzes the formation of gluconic acid and hydrogen peroxide from the oxidation of glucose. A second enzyme, peroxidase, catalyzes the reaction of hydrogen peroxide with potassium iodide chromogen to oxidize the chromogen to colors ranging from blue-green to greenish-brown through brown and dark brown. This information is essential for the colorimetric estimation, because based on our experiments, the double sequential enzyme reaction most likely will affect one or two color channels significantly and linearly. This means we can directly extract the channel information to perform a linear regression estimation, which can later combine with other interesting indexes to produce an accurate concentration reading.

To use the testing strip, we need to follow the following procedure:

1. Obtain only enough strips for immediate use from the bottle, and replace the cap tightly since dampness can cause them to malfunction during the enzyme reaction.
2. Completely immerse the glucose reagent area of the strip in fresh, well-mixed urine and immediately remove the strip. This is to avoid dissolving out the reagent area into the testing solution and causing bias.
3. While removing, you should touch the side of the strip against the rim of the solution container to remove excess urine. This is due to the prevention of chemical over-reaction. From our experiment

and empirical experience, we suggest simply putting the strip horizontally on the table for 5 to 10 seconds to remove most of the excess solution.

4. Originally, you should use the result strip to compare with the reference chart listed on the bottle to get a range of concentration reading. There are total 6 ranges provided based on the color. They are 0 ~ 5 mmol/L, 5 ~ 15 mmol/L, 15 ~ 30 mmol/L, 30 ~ 60 mmol/L, 60 ~ 110 mmol/L. However, in our setup, the user needs to put the strips into the strip holder and then insert it into the device. The next step is to launch the application on the smartphone to get a quantitative result.

Figure 5. Glucose strip and container

2.2.2 Ovulation

For the ovulation strip, we used a commercial product called Wondfo. It is FDA-approved, highly sensitive (detecting a level of 25 mIU/ml), and provides a high degree of accuracy in pinpointing the user's most fertile time of the month. It is first designed to increase the user's chances of becoming pregnant by correctly detecting the "LH surge". The "LH surge" here means a dramatic increase in luteinizing hormone present in the urine just before ovulation. Normally, the best time to fertilize the egg is within 6-24 hours of ovulation.

The setup for this strip, shown as the left image in Figure 6, is simple. The user immerses the testing side of the strip into the urine for about 5 seconds. Keep the maximum line above the solution. Then, wait for the strip to completely absorb the solution, which may take approximately 1-2 minutes. While reading the result, first make sure the control line on the strip is in red, which means this strip is not malfunctioning. Then, check if there is another red line below the control line. The luteinizing hormone (LH) level can be roughly estimated from the darkness of the red line.

The interesting part here is that this strip can only indicate an LH surge condition. It requires the LH concentration to reach a certain high level in order to show a dark red line. Below this certainty, we can only say the test result is negative but not what is the concentration of LH in this sample. This is where MoboFoto comes into play. Rather than simply find this red line on the testing strip, the user can again put it into the device and run the application. This application can estimate and log the level of LH in the urine, which can be helpful for clinical research to monitor the ovulation process.

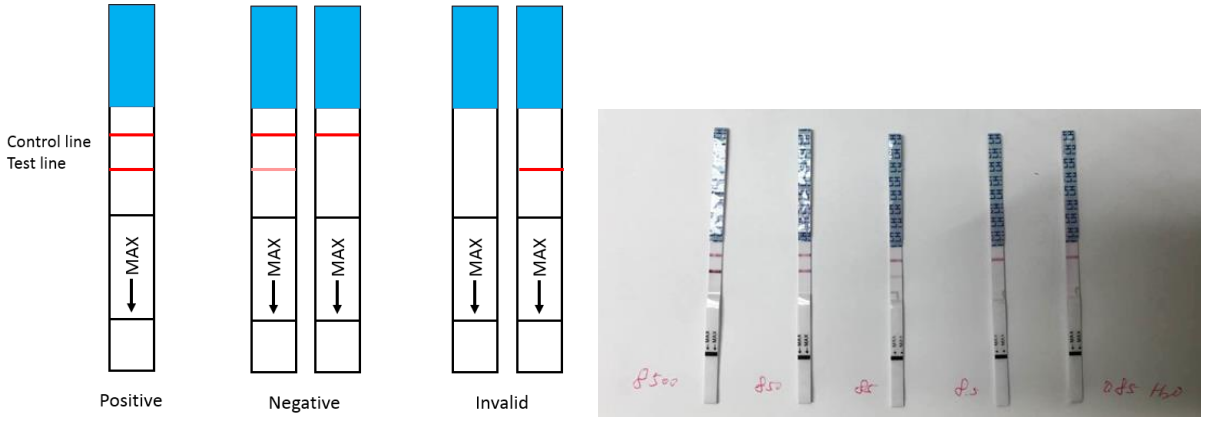

Figure 6. Ovulation strip instruction and real tests

3. Application Design

3.1 Design Overview

The main purpose of the application design is to detect and compute the concentration of the solution from the image input. At the same time, it will try to minimize the effect of variation by normalizing the image data and automatically selecting only the reasonable pixel value. This application was developed for Android OS targeting API level 15 and above, which means it can run on most current Android phones without an upgrade. The development process took place on an Eclipse ADT using Android SDK on a Macbook Pro (late 2013) laptop. External libraries include 'OpenCV for Android', 'Apache common math', 'Google gson' and 'GraphView-4.0.1'.

The application can be divided into three major parts.

1) Photo Taking

The first part is photo taking. It utilizes the default Android camera interfaces to let the user take a photo of the testing strip with one click. In the implementation, rather than use the system default camera UI, we created my own camera space (called surface holder). This camera space can provide the user more information about current focal length, ISO value and exposure compensation because these parameters have direct effect on concentration estimation. It also prompts the user to input a name for the photo in order to easier track the image data. Besides taking the photo, the user can also choose to load the photo from system disk space or external storage. No matter how you load the input photo, the image data will be sent to the second part to perform image processing.

2) Image Processing

This part heavily uses OpenCV library to provide various feature detection operations to locate the area of interest in the input image and extract the pixel information. In our current setup, the location of the testing area is fairly consistent in the input images, which means we can simply compute the image offset to locate and gather the pixel information. However, this is just a hardcode solution because our design will change in the future in order to adopt more tests. Thus, automatically recognizing the point of interest by detecting the feature is very important. Also, this method can compensate for the small variation in strip placement due to the flaw in peripheral design. The data extraction part for glucose

and ovulation is different and will be discussed in detail in Section 3.6, Data Extraction. The data extracted from the input image will then be sent to the third part for analysis and visualization.

3) Analysis and Visualization

After image processing has extracted the correct pixel information, the analysis part will calibrate the data first in order to convert it into the current lighting conditions. Then it will estimate the concentration by fitting it into the pre-computed linear regression model. This model is pre-computed based on the sensing result using known selected sample concentrations. After obtaining the estimation, the application will automatically visualize it based on the name given by the user. This helps us to gather each individual user's trend based on test date to plot a trend graph. In the current version, the application will also visualize image data generated during the computation for verification purpose.

3.2 Camera interface in Android OS

The camera class in Android OS is used to set image capture settings and start/stop preview, and it is a client for the camera service, which manages the actual camera hardware. To enable the class to access the hardware, we need to first declare the CAMERA permission in the Android Manifest. And we also need to add the 'camera' and 'camera.autofocus' in the 'uses feature' class field.

To use the camera class capture image, we need to first initialize a camera instance and set related parameters. A detailed list of parameter values we used is in Figure 7. After setting the parameters, the most important thing is to use a fully initialized SurfaceHolder instance. This instance would be the display visualization used for the camera class. Any UI interaction functionality, such as 'Press' button to take the photo, should be included in this SurfaceHolder. It is also the right place to display additional information about the current camera parameters. In our case, we displayed the focal length, ISO and exposure compensation information. After successfully loading the SurfaceHolder, we need to start the preview from the camera interface. This will let the camera class gather the system resource and launch the phone camera to start capturing the image. By default, it will use the back camera. While the camera is in preview status, the user is free to press the button to capture the image. The user can then choose to save the image into external storage by default or retake it.

Among the parameters set up in the camera interface, there are two worth some discussion. First is the picture size parameter. This parameter controls the size or resolution of the image captured by the camera. Theoretically, you can define an arbitrary image resolution that can fit into the phone screen. In our case we set it to 3264x2448. However, when retrieving the image back from the external storage, the true resolution of the image is just 1024x720. We believe this is due to the limitation of the phone screen, which only has resolution of 1280x720. Thus to fit the width of the screen, the image has been scaled to 1024x720. Another parameter is the white balance. White balance is the process of removing unrealistic color casts, so that objects which appear white in person are rendered white in your photo. In our case, while extracting the pixel information, the white balance can be a critical issue that causes bias in recognizing the RGB channel values since it will make compensation on certain channels. In the program, there are the following choices for white balancing: auto, cloudy_daylight, daylight, fluorescent, incandescent, shade, twilight, warm_fluorescent. Under current circumstances, we can simply use auto white balancing which lets the camera decide which mode should be applied. What we care about is the consistency of the white balance. Because in our setup the only light source is the flash light, the auto white balance will always invoke the same mode. An example of camera SurfaceHolder and a list of camera parameters in MoboFoto are shown as Figure 7.

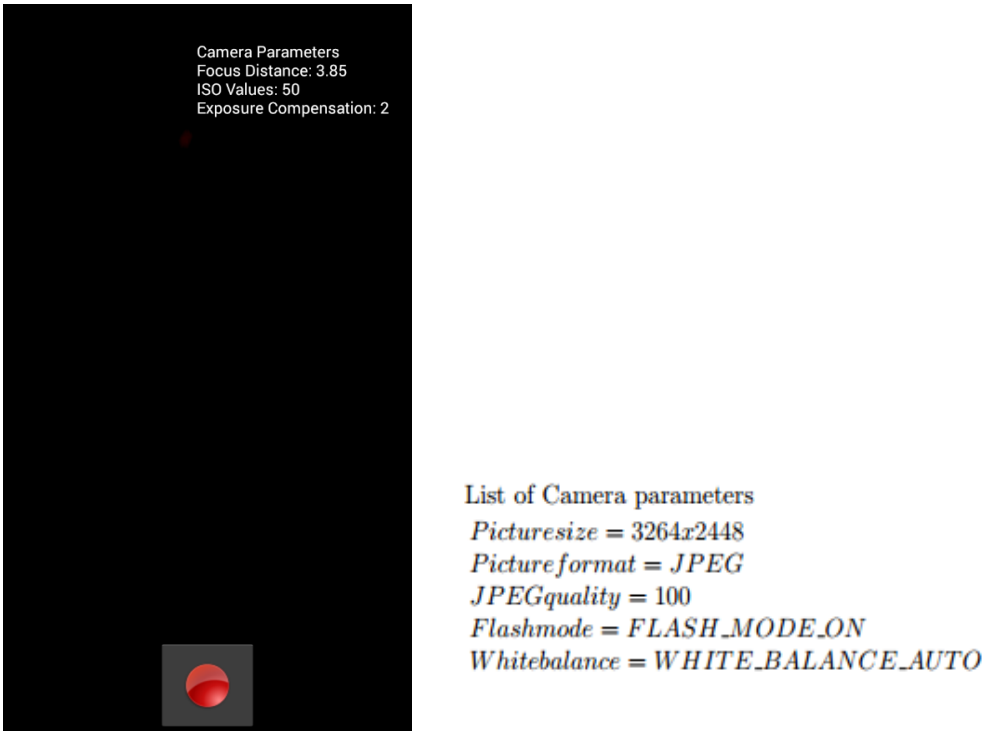

Figure 7. Camera surface holder and parameters

3.3 Color Space

A color space, by definition, is an organization of colors. Also, you can treat it as a system to reproducibly represent the colors both in analog and digitally. Along with the color space, we often refer to the abstract mathematical model describing how the colors can be represented as tuples of numbers as the color model. In this section, it will discuss the most widely used two major color spaces we are interested in based on the analysis of its linearity of change in different concentration.

The first color space is RGB. It is any additive color space based on the RGB color model. In this color model, each color can be represented by assigning one numerical value in each of the three chromaticities of the red, green and blue additive primaries. Throughout this thesis, we call them red, green and blue channels. In each channel, the range of tuple is from 0 to 255. Thus we can represent total 256x256x256 = 16777216 different colors. For example, (0, 0, 0) represent pure black and (255, 255, 255) represent pure white. RGB color space is the most commonly used color space in colorimetric analysis. In this project, we notice that the change in concentration of the glucose has a significant effect on the green channel.

The second color space we are interested in is the HSB (also called HSV), shown in Figure 8. It is the most common cylindrical-coordinate representations of points in an RGB color model. In HSB, H stands for hue ranging from 0 to 360 which represents the angle about the color wheel. S stands for saturation which describes the distance from full color to the gray. B is the brightness, telling you how dark the color is.

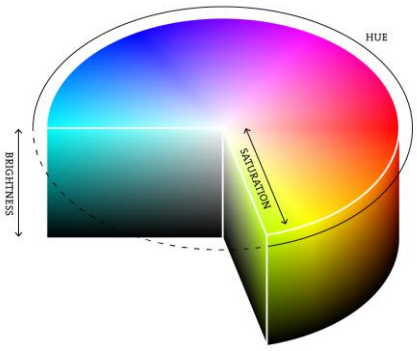

Figure 8. HSB color space

In Android, we normally first read out the RGB pixel information from the input images, then convert it into HSB color space for further analysis. To convert the color space, we usually adopt Algorithm 1.

Algorithm 1 Convert RGB into HSB color space

procedure RGB2HSB(A)
  $maxRGB = MAX(R, G, B)$
  $minRGB = MIN(R, G, B)$
  $B = maxRGB$
  $S = (maxRGB - minRGB)/maxRGB$
  $\triangle = maxRGB - minRGB$
  if $maxRGB = R$ **then**
    $H = (G - B)/\triangle$
  end if
  if $maxRGB = G$ **then**
    $H = 2 + (B - R)/\triangle$
  end if
  if $maxRGB = B$ **then**
    $H = 4 + (R - G)/\triangle$
  end if
  $H = (60 * H) mod 360$
  Return H, S, B
end procedure

For our ovulation analysis, the color change of the input image does not have a reasonable linearity on any single color channel. Therefore, we choose to use the value of the red, green and brightness channels together to threshold the image. More detail about how we perform linear regression in the green channel for glucose estimation and how we threshold the red, green and brightness channels for ovulation estimation will be discussed in section 3.7.

3.4 OpenCV in Android OS

OpenCV is a powerful library which implements most ready-to-use computer vision algorithms for image processing. It is also a cross-platform library, which supports C++, C, Python and Java interfaces on desktop and mobile devices. In this thesis, we leverage the convenience of this powerful tool to help us analyze and detect the features in the image to locate areas that we are interested in.

To use OpenCV for Android, the fastest way is to use OpenCV4Android SDK. You may use the Tegra Android Development Pack (TADP) released by NVIDIA for Android development environment setup. This setup includes an Eclipse IDE (Version Luna) with all necessary dependencies integrated. All you

need to do before coding is to initialize the OpenCV libraries either asynchronously or synchronously. This is also the setup we adopted in this thesis.

For the library initialization, we should mention that although it is recommended to asynchronously communicate with OpenCV manager to dynamically initialize the library when the application launches, this method may not be applicable in mainland China due to the blocking of Google service. Thus, in the MoboFoto application, we adopted the synchronous initialization method, which requires copying the corresponding OpenCV native libs from the source to libs folder within Android project. This essentially builds up the size of the application in distribution.

The major task for detecting glucose in our application is to correctly locate the coordinates of the glucose reagent area. The method we performed is to first locate the white region of the strip holder in the image. Since the strip will always be bounded on the strip holder and the relative position of one specific reagent area is fixed, we can use offset to calculate the approximate region of the glucose reagent. In the future, even if the position of the strip holder changes in the setup (and so in the image), we can still correctly detect it.

Another reason we choose to first detect the strip holder is because it has strong color contrast to its neighborhood (white and black as listed in Figure 9). This is very helpful when we apply canny edge detection to sort out all apparent edges in the images. Thus, in this section, we will first discuss how and why we use this algorithm.

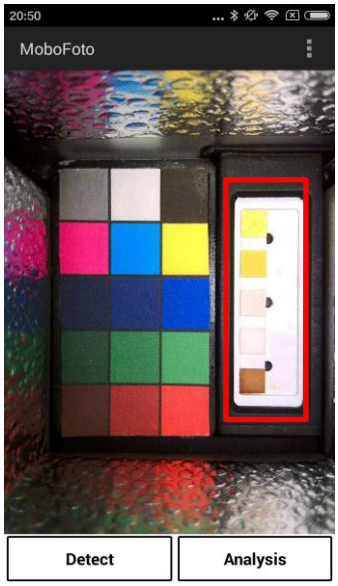

Figure 9. Original image before processing

3.4.1 Canny Edge Detection

Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in the image. It usually is performed starting with noise reduction using a 5x5 Gaussian filter. Then it tries to find the intensity gradient of the image. Finally, it performs non-maximum suppression and hysteresis thresholding to find the best result as edges. Detailed explanation about these terms will be excluded from this thesis, as it is a well-established algorithm. In Figure 10, there is a comparison of our original image and its edge-detected version. As you can see, although we had filtered and thresholded the image, there is still much noise that can confuse our detection. Therefore, the best way to exclude this post-detection noise is to find the largest contour over the entire image.

Carefully inspecting the canny edge detected image, you can see that most edges are open-looped, which means it cannot connect back to itself to form a closed loop. The find contour algorithm will eliminate these edges and only preserve the contours of closed loops. Then we select the contour with the largest area and display it. This draw-out is the contour of the strip holder. Rarely, the algorithm may falsely detect the maximum contour. This is because when capturing the picture, some part of the strip holder edge has a strong reflection of the smartphone flash light, which causes the canny edge detector to fail to detect the strip holder contour as a closed loop. This problem can be resolved by re-taking the photo.

In the real code, we always need to first convert the image into mat type. Mat class is a primitive class in OpenCV, which represents an n-dimensional dense numerical single-channel or multi-channel array. This helps us to more efficiently work on selected color channels and data from the image. In order to use the canny edge function in OpenCV we need to first perform a meanshift segmentation of the image. This function essentially outputs the filtered “posterized” image with color gradients and fine-gain texture flattened. For every pixel (X, Y) of the input image, the function performs following operation:

$$(x, y) : X - sp \leq x \leq X + sp, Y - sp \leq y \leq Y + sp, \|(R, G, B) - (r, g, b)\| \leq sr$$

where sp is the spatial window radius and sr is the color window radius. Empirically, in our case, the sp of 10 and sr of 20 yield the best output result. Then we convert the source into gray image and perform a thresholding function for all the pixels in the image using the following operation:

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxval} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

It is called binary thresholding because based on whether the data is larger than the threshold or not, we either set it to the maximum value we declared or set it to zero. In practice, we set the maximum value to be 255 and the threshold to be 50. Since we are only interested in the white strip holder area, anything below the threshold will be set to 0 and be eliminated from the canny edge detection in the next step.

For the canny edge detector, we defined the lower threshold to be 60 and upper threshold to be 100. Under this setup, we retrieved the best output image. After canny detection and finding the contours, all the contours were stored in a list as Mat points. We iterate through the list and find the contour with maximum area for display.

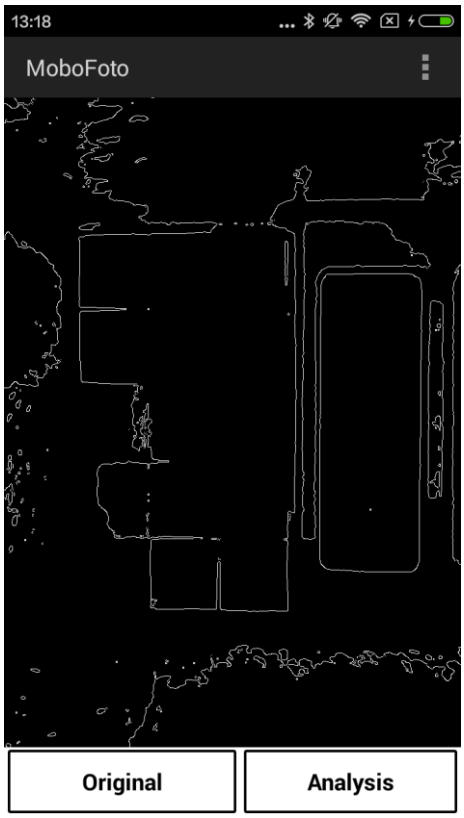

Figure 10. Canny edge detection

3.4.2 FAST Feature Detection

So far we have found the contour of the strip holder and also plotted it using white on a black background. Then we need to perform feature detection to detect the corners for this contour. There are many available algorithms to detect corners in computer vision such as SIFT and SURF; however, since our image almost has no noise (just white and black color), we adopted a feature detector called Features from Accelerated Segment Test (FAST) algorithm. It is a high-speed corner detection algorithm but not robust to high levels of noise, which exactly fits our circumstances. The basic idea of FAST is to examine a circle of 16 pixels around the pixel under test. Then it detects a set of contiguous pixels in the circle, which are all brighter or darker than a threshold. More details about FAST will not be given in this thesis since it is a well-documented algorithm and out of our scope here.

In reality, the rectangular contour of the strip holder detected does not have four sharp corners. Instead, each corner is actually a non-smooth curve. It is obvious that the FAST corner detection will detect multiple corners on each curve and our task is to locate the coordinates that can best represent the shape of the contour. Thus, we adopted following procedure to select these points:

Select Corner Points
procedure SELECT($Points$)
    $Sort(Points.X)$
    $Sort(Points.Y)$
    $TopLeft = min(Points.X), min(Points.Y)$
    $TopRight = max(Points.X), min(Points.Y)$
    $BotLeft = min(Points.X), max(Points.Y)$
    $BotRight = max(Points.X), max(Points.Y)$
    Return $TopLeft, TopRight, BotLeft, BotRight$
end procedure

Basically, we sorted all the points by X and Y coordinates and selected the extreme value for each dimension. As soon as we have selected the four corner points, we can measure the offset of the glucose reagent area to extract pixel information. An example of the largest contour detected with four corners marked as red circle is shown in Figure 11.

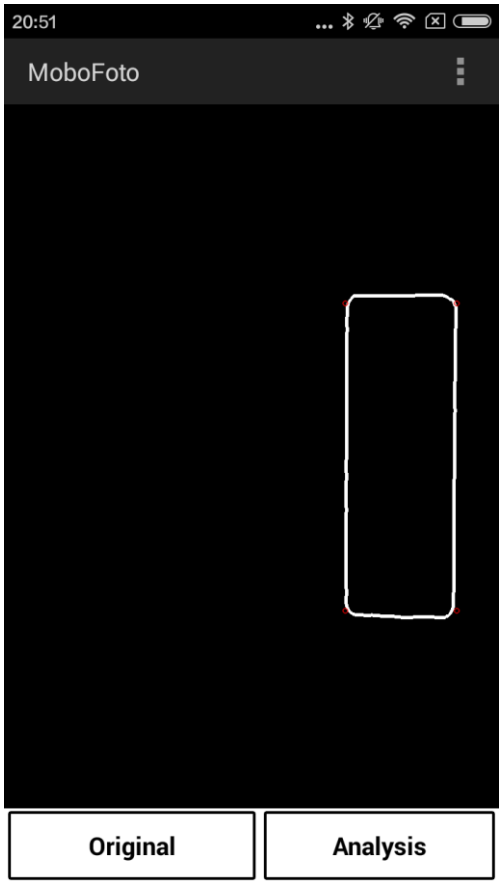

Figure 11. Corner detection (red circles)

3.4.3 Rectangle Detection

We have used FAST feature detection to locate the contour of the strip holder. For glucose estimation, this is already sufficient. However, to estimate the ovulation concentration we need to apply another step of rectangle detection. Compared to the glucose sensing strip, there is no obvious reagent area on the ovulation strip. And our goal is to locate the left and right boundaries of the strip to perform a vertical pixel scan from top to the bottom along the strip. This requires us to correctly locate the coordinates of the boundary first.

Luckily, there is a black rectangle on the ovulation strip that we can easily locate using the rectangle detection algorithm and based on that we can gather the leftmost and rightmost coordinates to perform a vertical pixel scan.

The first few steps to locate the strip holder contour are exactly the same as we did for the glucose strip. Then we crop the original image and only keep the strip holder area. Again, we perform canny edge detection on the image with low threshold to be 0 and high threshold to be 255 since the black rectangle we are trying to detect has a strong contrast with the neighboring color. The next step involves using a Gaussian blur filter to reduce image noise and find contours on the smoothed image. The last step is to examine all the contours found to make a judgment on whether the contour is a square or not. Within all square contours, we are only interested in the one that is more than 30 pixels wide and for which the ratio of the width to height is greater than 1, essentially, a rectangle.

As long as we detected the right rectangle, we can use the coordinates of the left top corner and right top corner as the boundary to start vertical scanning. An example of the detected rectangle is shown in Figure 12.

Figure 12. Rectangle detection

The code snippets of canny edge detection, FAST feature detection and rectangle detection in OpenCV Java can be found in Appendix B.

3.5 Thread

Thread programming is important in MoboFoto application development because the image processing using OpenCV libraries requires much time to process. We want these executions to actually happen in the background. In Android OS, the MoboFoto application will spawn the main thread as a single UI thread that corresponds to any UI interaction on the phone screen. Launching the camera or loading images only works as a separate activity and thus does not require another thread. When applying a

computer vision algorithm such as canny edge detection and FAST feature detection, it usually takes 5 to 10 seconds to process depending on the image size. If we launch these heavy computations on the main thread, then the UI becomes unresponsive. Thus, the best way to resolve this problem is to launch another work thread, which processes the computational work in the background and merges the result back to the main thread when it is done. In this case, the user interface will remain responsive all the time. This technique is called multi-threading and typically is implemented using AsyncTask or Thread class in Android interfaces. In the MoboFoto application, we used AsyncTask for all image processing related computation.

3.6 Data Extraction

This section discusses how we can extract and threshold the reasonable data from the processed image data for linear regression estimation. Since glucose and ovulation work differently, we need to approach them separately.

For glucose measurement, the main task is to extract the pixel information from the reagent area using the offsets. We first use the height/width ratio to find a point located within the reagent area and define a neighborhood size as the area of interest. Empirically, we set this pad size to be 30x30, which contains 900 pixels. As mentioned earlier, the green channel of these pixels has a linear relation with the glucose concentration due to the enzyme reaction. Therefore, we access each pixel's RGB information and store every green channel value.

Usually we can simply take the average of these green values and assign a representative reading for this test to feed into our linear regression model. However, this assumes that the solution had been evenly distributed over the reagent area. It also assumes that the enzymes are equally distributed in the reagent area during manufacturing and there is no over-reaction or under-reaction during the test. In reality, this is not practical. Therefore we need to filter out the non-dominant values. In Figure 13, there is a 30x30 image cropped from the glucose reagent area that we are interested in. Obviously, there is a non-negligible number of pixels that are not consistent with their neighbors. Some of them are light in color, close to white, because of the reflection of the flash light or because there is no reaction happening at this point. Some of them are dark in color, close to black, because of over-reaction or accumulation of enzymes. They are biased in calculating the green channel value for representing the

average value of this region. On the right side, there is a distribution of green channel value for the 30x30 area, which reports the number of instances of certain pixel values. It is obvious that most counts exist in the middle range and fall on both high and low green values like a Gaussian distribution. Thus, we decided to set the existence threshold, based on the experiments, to be 15 times which means we treat any green value existing less than 15 times to be the minority and eliminate it from average computation.

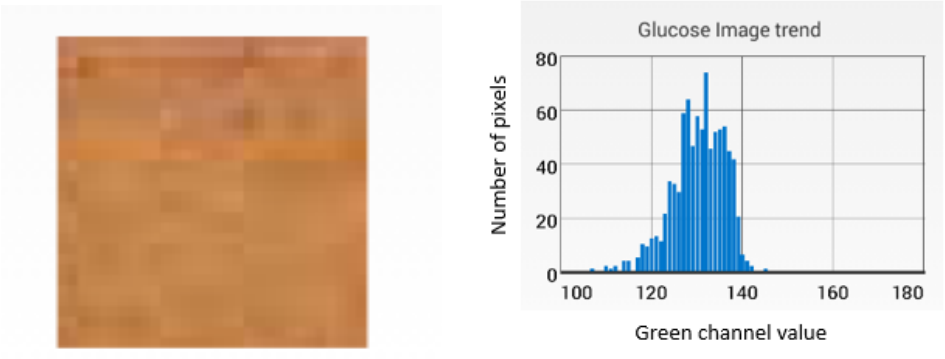

Figure 13. Glucose data extraction

For ovulation measurement, our task is to correctly locate the red lines and count the number of pixels which fall into our criteria. Based on the rectangle we located in the last section, we vertically scan all the pixels within the boundary from top to bottom. For each pixel, we compare the red channel value, green channel value and the converted brightness value with the threshold. In this part, there is no trick in selecting the correct corresponding pixels, only experiments. First we exclude the pixels with high brightness values from the data set because the major part of the ovulation strip is white and tends to be more reflective compared to the red lines area. We set this threshold to be 0.9 for the brightness channel. Then, we noticed that there are some shadows in the image that might become errors in computation if we only use the red channel for thresholding; thus we set the threshold of the red channel to be 220 and that of the green channel to be 180. Again, we stored and classified the number of the pixels that pass the threshold along the path of vertical scanning as represented in Figure 14, which is taken using the ovulation sample solution (8500 mIU/ml). On the left side is the ovulation strip with the boundary detected. The graph on the right side shows the pixel response from top to bottom. The X coordinates represent the row number of the pixels starting with row 0 corresponding to the top-most pixels. The Y coordinates represent how many pixels fall into our criteria for each row. You can

clearly see that there are two spike-like responses in each image between row 0 and row 80. They are actually the control line and test line on the sensing strip. Then, the last thing we need to do is to count the total number of qualified pixels and subtract it with the 'common background'. The 'common background' means we know a certain area does not have a response but still has few qualified pixels. These qualified pixels exist even if we do not have a control line or test line; thus, they are eliminated from the count.

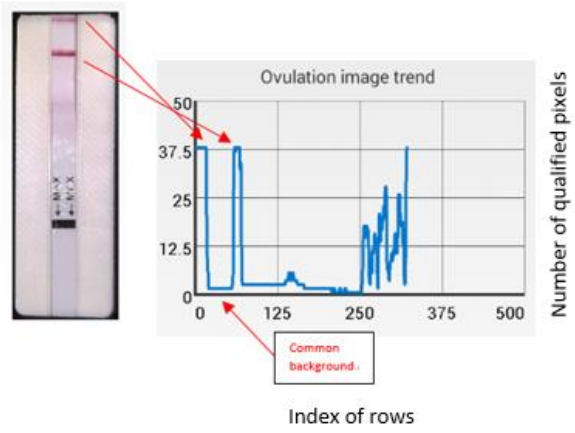

Figure 14. Ovulation data extraction

3.7 Colorimetric Linear Regression

Colorimetric linear regression means using color information to fit a linear mathematical model for the prediction of new data. The essential part is to find the most reliable linear relationship between two data sets. In MoboFoto, we first use solutions with known concentration to generate the baseline of the regression. Then we use Apache common math library to fit a linear regression on the data. For glucose estimation, the green channel value exposed a solid linearity with the concentration; thus, we use green channel value as the Y data and concentration as the X data to get the linear regression Table 1 and the plot in Figure 15.

Table 1. Glucose experiment data

Avg G Channel	235	215	175	156	135	115	110	97	91	70	54
Concen(mmol/L)	0	10	20	30	40	50	60	70	80	90	100
Std for G	1.21	4.73	3.53	4.73	3.53	1.41	3.53	9.89	16.97	9.19	3.54

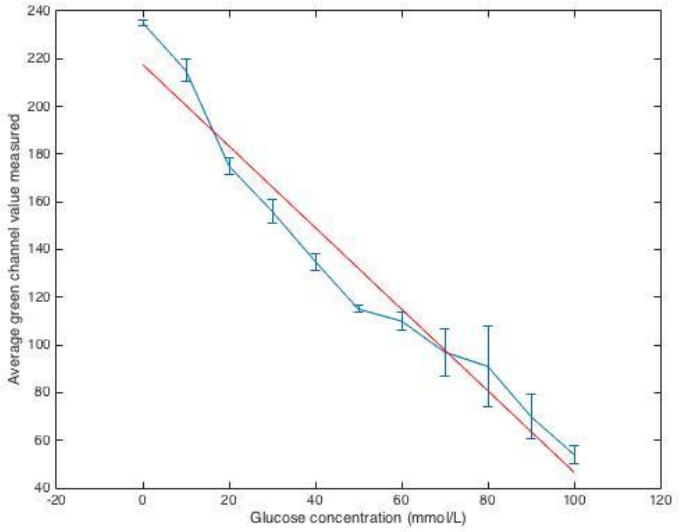

Figure 15. Glucose linear regression

We obtain the green channel value for each concentration by repetitive tests using the same setup with a different glucose sensing strip. To ensure a stable value, we computed the average and standard deviation for each test as listed in Table 1. A standard deviation around 5 would be a relatively stable reading. For those concentrations with high standard deviation, it is necessary to conduct more tests to obtain generalized experiment data. However, it still produces a pretty good result.

For ovulation estimation, rather than using any color channel, we use the ratio between the number of qualified pixels in the test line and the number of qualified pixels in the control line. We fit the data using this ratio as the Y data and logarithmic concentration (base 10) as X data to get the linear regression Table 2 and the plot in Figure 16.

Table 2. Ovulation experiment data

N_test/N_control	0	0.001	0.94	2.78	6.0
Log(Concen)(mIU/ml)	-0.07	0.929	1.929	2.929	3.929

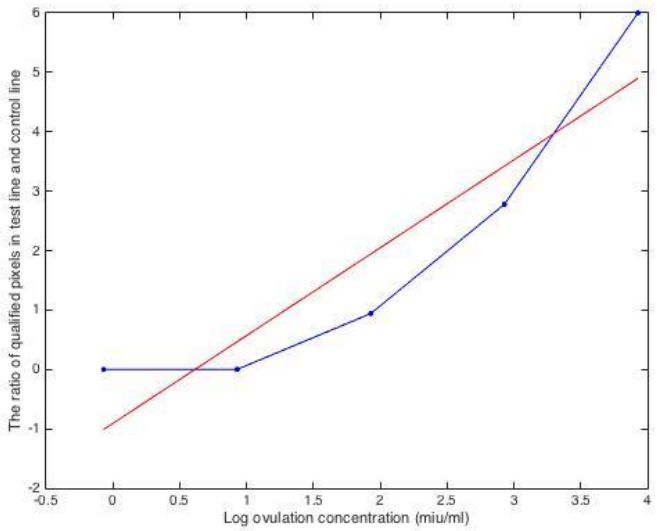

Figure 16. Ovulation linear regression

The concentrations of the ovulation solution we tested are 0.85, 8.5, 85, 850, 8500 mIU/ml. These solutions are preserved at a temperature of 40 F° and tested within a week after preparation. The outcome of linear regression is reasonable since for concentration 0.85 and 8.5, the reaction is so weak that you can barely see the red test line using the naked eye. You can see the sample test on the ovulation strip in Figure 6.

We are limited by the experiment conditions and do not have a more diverse set of concentrations for glucose and ovulation calibration, and we also observed that the linearity is more convinced in segments; thus, we adopted the nearest neighbor methodology to make the estimation. In the glucose test, we first compute a range on linear fitting for the input. Then we picked four data, which are closest to the input on the fitting line. Based on these points, we construct a new linear regression model and compute the final estimation from it. The estimation result for glucose and ovulation will be presented in Chapter 4.

3.8 Data Visualization

For data visualization, we used the Fragment interface in Android OS and GraphView open source library. The Fragment interface provides many convenient features to format a scrollable report page as a tab. The GraphView library supports various graph layouts and it is very easy to customize.

For each run, the user is required to provide a name for the current test, which usually should be the owner of the solution under inspection. This name works as a unique key in the storage system implemented using the SharePreference interface in Android and associates with a list of data containing concentration value and test date pairs. When visualizing the data, the program will iterate the corresponding HashMap to retrieve all the values sorted by the test date. Then, they will be visualized as fragments through GraphView pipeline. We provide two graph visualizations for the users. One is the trend graph showing the tested concentration by date, on which user can click the points to get more detail. Another is the debug graph that visualizes either the pixel value in glucose or pixel response in ovulation test for debugging. Sample data visualization can be found in Figure 17.

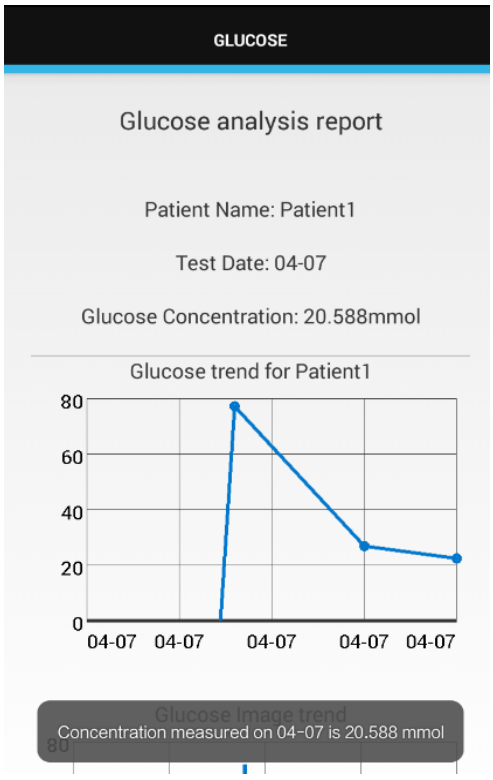

Figure 17. Data visualization

4. Result

4.1 Glucose Result

To test the calibration model from linear regression, we selected a set of solutions with various concentrations and fed them into the application. Each concentration had been tested three times and we computed the mean value and standard deviation for estimation. Table 3 shows the result.

Table 3. Glucose result (Conce for concentration)

Sample Conce	Sample Green	Mean Conce	Std Conce	Mean Green	Std Green
30 mmol/L	156	26.78 mmol/L	1.8 mmol/L	163	5.13
50 mmol/L	120	50.32 mmol/L	4.72 mmol/L	117.6	5.5
80 mmol/L	91	79.34 mmol/L	7.69 mmol/L	88	12.16

Using mean concentration as the result, the error rates for each sample concentration are 10%, 0.6% and 0.8%, which are within the reasonable error range. However, we also observed that the error rates will increase in both high and low concentration and are relatively low in normal range.

4.2 Ovulation Result

To test the ovulation strip, we first feed in the solution sample used for model calibration and then mix 8500 mIU/ml solution with the distilled water to create 4250 mIU/ml solution, which later is fed into the model as well. Table 4 shows the result.

Table 4. Ovulation result

Sample Concentration	Measured Concentration	Error rate
0.85 mIU/ml	0 mIU/ml	N/A
8.5 mIU/ml	0.1 mIU/ml	N/A
85 mIU/ml	86 mIU/ml	1.2%
850 mIU/ml	849 mIU/ml	0.12%
8500 mIU/ml	8499 mIU/ml	0.012%
4250 mIU/ml	4500 mIU/ml	5.9%

Based on the result in Table 4, the model is not sensitive for concentrations below 10 mIU/ml. In reality, you can hardly see any red line using the naked eye for this concentration. The performance of the repeated data is very good such that the error rate is around or below 1%. For new data, the error rate is around 5.9%, which still in a reasonable range.

5. Conclusion and Future Work

The first stage of testing MoboFoto with mixed glucose and ovulation solution demonstrated that using colorimetric information in relatively isolated lighting conditions can correctly measure the unknown solution concentration. The MoboFoto platform can provide a cheap, fast and accurate way to measure the common health index's concentration. It also provides a friendly user interface to help either common people or doctors to keep track of tested data by dates. The setup is easy and convenient. You only need a device case, a strip holder and some sensing strips to start analyzing either the glucose concentration in your urine or your luteinizing hormone level.

In the future, there are more problems related to accuracy and extensibility we need to solve. First of all, our dataset is small for ovulation. The small dataset can only produce a rough linear regression, thus introducing a lot of error in the real measurement. Thus, we need to expand our dataset either with more pre-made ovulation solution or with clinical experiments. For glucose measurement, the calibration dataset is comprehensive; however, the test dataset is relatively small too. We need to produce more estimation results using random glucose concentration. Moreover, the glucose sensing strip is actually capable of measuring many other chemical solutions. There is an opportunity to extend the current setup with a different calibration dataset to let it measure other chemical concentrations. And we are already planning to measure pH value and protein.

References

[1] A. K. Yetisen and J. L. Martinez-Hurtado, "A smartphone algorithm with inter-phone repeatability for the analysis of colorimetric tests," *Sensors and Actuators B: Chemical*, vol. 196, pp. 156-160, June 2014.

[2] J. I. Hong, "Development of the smartphone-based colorimetry for multi-analyte sensing arrays," *Lab on a Chip*, vol. 14, no. 10, pp. 1725-1732, 2014.

[3] S. Sumriddetchkajorn, "Mobile device-based self-referencing colorimeter for monitoring chlorine concentration in water," *Sensors and Actuators B: Chemical*, vol. 182, pp. 592-597, June 2013.

[4] S. Sumriddetchkajorn, "Mobile-platform based colorimeter for monitoring chlorine concentration in water," *Sensors and Actuators B: Chemical*, vol. 191, pp. 561-566, February 2014.

[5] N. Lopez-Ruiz, "Smartphone-based simultaneous pH and nitrite colorimetric determination for paper microfluidic devices," *Anal. Chem.*, 86 (19), pp. 9554-9562, 2014.

[6] N. Lopez-Ruiz, "Determination of O_2 using colour sensing from image processing with mobile devices," *Sensors and Actuators B: Chemical*, vol. 171-172, pp. 938-945, August-September 2012.

[7] L. Shen, "Point-of-care colorimetric detection with a smartphone," *Lab on a Chip,* vol. 12, no. 21, pp. 4240-4243, 2012.

[8] T. Smith and J. Gild, "The C.I.E. colorimetric standards and their use," *Transactions of the Optical Society,* vol. 33, no. 3, p. 73, 1931.

Appendix A MoboFoto Application Class Workflow

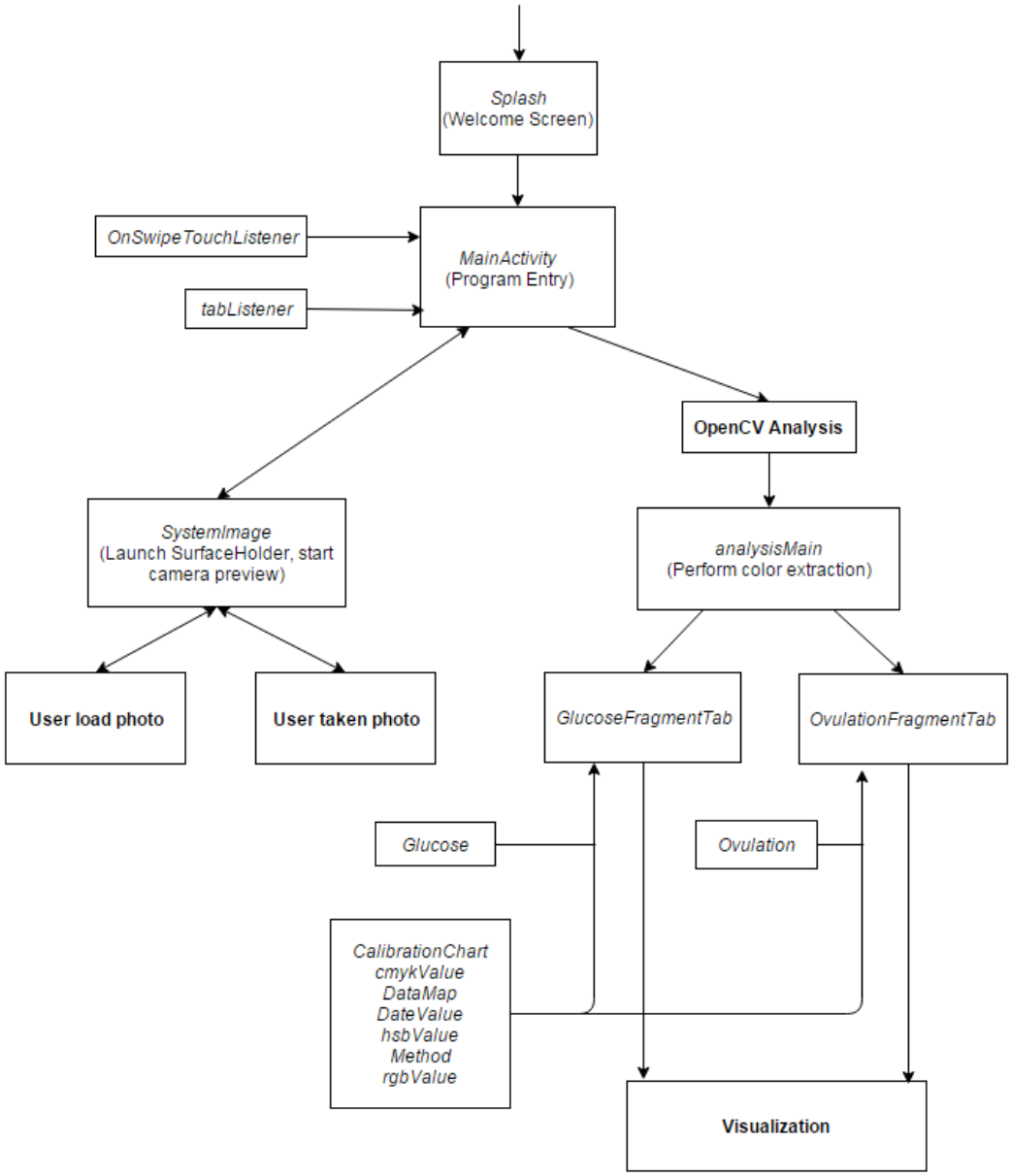

Figure 18. MoboFoto application class workflow

Following is a key to read Figure 18:

Italic: Name of Java class.

Bold: User actions, no individual Java class implementation.

Arrow: Unidirectional class calling flow.

Double Arrow: Bidirectional class calling flow.

Appendix B MoboFoto Application OpenCV Code Snippets

Canny Edge Detection

```
Bitmap bmp = ((BitmapDrawable)image.getDrawable()).getBitmap();

if(bicolor==null){
    Mat mImg = new Mat();
    Utils.bitmapToMat(bmp,mImg);
    Imgproc.cvtColor(mImg, mImg, Imgproc.COLOR_RGBA2RGB);
    Imgproc.pyrMeanShiftFiltering(mImg,mImg,10,20);
    bicolor = Bitmap.createBitmap(mImg.cols(), mImg.rows(), Bitmap.Config.ARGB_8888);

    Utils.matToBitmap(mImg, bicolor);
    //convert to mat:
    final Bitmap bitmap = bicolor;

    int iCannyLowerThreshold = 60, iCannyUpperThreshold = 100;
    Mat m = new Mat(bitmap.getWidth(), bitmap.getHeight(), CvType.CV_8UC1);
    Utils.bitmapToMat(bitmap, m);
    Mat thr = new Mat(m.rows(),m.cols(),CvType.CV_8UC1);
    Mat dst = new Mat(m.rows(), m.cols(), CvType.CV_8UC1, Scalar.all(0));
    Imgproc.cvtColor(m, thr, Imgproc.COLOR_BGR2GRAY);
    Imgproc.threshold(thr, thr, 50, 255, Imgproc.THRESH_BINARY);
    Imgproc.Canny(thr, thr, iCannyLowerThreshold, iCannyUpperThreshold);
    //Imgproc.findContours(m, contours, new Mat(), 0, 1);
    List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
    Imgproc.findContours( thr, contours, new Mat(),Imgproc.RETR_EXTERNAL, Imgproc.CHAIN_APPROX_SIMPLE, new Point(0,0) );
    Scalar color = new Scalar( 255,255,255);
    //find the max contour
    double maxArea = -1;
    int maxAreaIdx = -1;
    for (int idx = 0; idx < contours.size(); idx++) {
        Mat contour = contours.get(idx);
        double contourarea = Imgproc.contourArea(contour);
        if (contourarea > maxArea) {
            maxArea = contourarea;
            maxAreaIdx = idx;
        }
}
```

Figure 19. Code for canny edge detection and find the largest contour in OpenCV Java

FAST Feature Detection

```
//corner detection
MatOfKeyPoint points = new MatOfKeyPoint();

Mat mat = new Mat();
Imgproc.cvtColor(dst, mat, Imgproc.COLOR_GRAY2RGBA);
FeatureDetector fast = FeatureDetector.create(FeatureDetector.FAST);
fast.detect(mat, points);
Scalar redcolor = new Scalar(255,0,0);
Mat mRgba= mat.clone();
Imgproc.cvtColor(mat, mRgba, Imgproc.COLOR_RGBA2RGB);
KeyPoint[] point = points.toArray();

//Calculate the max min X and Y.
double[] thres_x = new double[point.length];
double[] thres_y = new double[point.length];
for(int i =0; i<point.length; i++){
    thres_x[i]= point[i].pt.x;
    thres_y[i]= point[i].pt.y;
}
Arrays.sort(thres_x);
Arrays.sort(thres_y);
lefttop[0] = thres_x[0];
lefttop[1] = thres_y[0];
righttop[0] = thres_x[point.length-1];
righttop[1] = thres_y[0];
leftbot[0] = thres_x[0];
leftbot[1] = thres_y[point.length-1];
rightbot[0] = thres_x[point.length-1];
rightbot[1] = thres_y[point.length-1];
Point leftTop = new Point();
Point rightTop = new Point();
Point leftBot = new Point();
Point rightBot = new Point();
leftTop.set(lefttop);
rightTop.set(righttop);
leftBot.set(leftbot);
rightBot.set(rightbot);
KeyPoint KleftTop = new KeyPoint();
KeyPoint Krighttop = new KeyPoint();
KeyPoint Kleftbot = new KeyPoint();
KeyPoint Krightbot = new KeyPoint();
KleftTop.pt = leftTop;
Krighttop.pt =rightTop;
Kleftbot.pt = leftBot;
Krightbot.pt = rightBot;
KleftTop.size = 7f;
Krighttop.size =7f;
Kleftbot.size = 7f;
Krightbot.size = 7f;
KeyPoint[] corner_point = {KleftTop, Krighttop, Kleftbot, Krightbot};
Log.d("test", "The new length is " + corner_point.length);
MatOfKeyPoint corner_points = new MatOfKeyPoint(corner_point);
//corner_points.fromArray(corner_point);
Features2d.drawKeypoints(mRgba, corner_points, mRgba, redcolor, 4);
Imgproc.cvtColor(mRgba, mat, Imgproc.COLOR_RGB2RGBA);
```

Figure 20. Code for FAST corner detection in OpenCV Java

Rectangle Detection

```
private Bitmap detect_rect(Bitmap originalPhoto){
        Mat imgMat=new Mat();
        Utils.bitmapToMat(originalPhoto,imgMat);

        Mat imgSource=imgMat.clone();
        Imgproc.cvtColor( imgMat, imgMat, Imgproc.COLOR_BGR2GRAY);
        Bitmap grayscale=Bitmap.createBitmap(imgMat.cols(),imgMat.rows(),Bitmap.Config.ARGB_8888);
        Utils.matToBitmap(imgMat,grayscale);
        Imgproc.Canny(imgMat,imgMat,0,255);
        Bitmap canny=Bitmap.createBitmap(imgMat.cols(),imgMat.rows(),Bitmap.Config.ARGB_8888);
        Utils.matToBitmap(imgMat,canny);
        Imgproc.GaussianBlur(imgMat, imgMat, new  org.opencv.core.Size(1, 1), 2, 2);
        Bitmap blur=Bitmap.createBitmap(imgMat.cols(),imgMat.rows(),Bitmap.Config.ARGB_8888);
        Utils.matToBitmap(imgMat,blur);

        //find the contours

        List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
        Imgproc.findContours(imgMat, contours, new Mat(), Imgproc.RETR_LIST, Imgproc.CHAIN_APPROX_NONE);
        MatOfPoint temp_contour = contours.get(0); //the largest is at the index 0 for starting point

        for (int idx = 0; idx < contours.size(); idx++) {
            temp_contour = contours.get(idx);

                //check if this contour is a square
                MatOfPoint2f new_mat = new MatOfPoint2f( temp_contour.toArray() );
                int contourSize = (int)temp_contour.total();
                MatOfPoint2f approxCurve_temp = new MatOfPoint2f();
                Imgproc.approxPolyDP(new_mat, approxCurve_temp, contourSize*0.05, true);

                if (approxCurve_temp.total() == 4) {
                    MatOfPoint points = new MatOfPoint( approxCurve_temp.toArray() );
                    Rect rect = Imgproc.boundingRect(points);
                    if(rect.width > 30 && rect.width/rect.height>1){
                    left_x = rect.x;
                    left_y = rect.y;
                    right_x = rect.x+rect.width;
                    right_y = rect.y;
                    Core.rectangle(imgSource, new Point(rect.x,rect.y)
                    , new Point(rect.x+rect.width,rect.y+rect.height), new Scalar(255, 0, 0, 255), 3);
                    break;
                    }
                }
        }
        Bitmap analyzed=Bitmap.createBitmap(imgSource.cols(),imgSource.rows(),Bitmap.Config.ARGB_8888);
        Utils.matToBitmap(imgSource,analyzed);
        return analyzed;
}
```

Figure 21. Code for rectangle detection in OpenCV Java