FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Android application for determination of sulfonamides in water using digital image colorimetry

Pedro Daniel dos Santos Reis

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hélder Filipe Pinto Oliveira

Second Supervisor: Pedro Henrique Moreira Queirós Carvalho

July 28, 2020

Android application for determination of sulfonamides in water using digital image colorimetry

Pedro Daniel dos Santos Reis

Mestrado Integrado em Engenharia Informática e Computação

July 28, 2020

Abstract

Antibiotics are widely applied for the treatment of humans and animals. These compounds can be found in the different water compartments, including wastewater and drinking water. The presence of antibiotics in the aquatic environment causes the development antibiotic-resistant bacteria, which is related to the emerging of untreatable infectious diseases.

Sulfonamides are an important antibiotic group, and have been frequently found in the aquatic medium. One of the most common methods for determination of sulfonamides in water consists in high-performance liquid chromatography coupled with mass spectrometry (HPLC-MS/MS). Despite the high sensitivity and selectivity of this methodology, it is an expensive, reagent consuming process and not suitable for an in situ analysis strategy.

One important property of sulfonamides is how the compound reacts when added the colorimetric reagent *p*-dimethylaminocinnamaldehyde. When these two compounds mix, a color is obtained, and it's intensity is related to the concentration of sulfonamides in the sample. This opens the possibility of using colorimetry to measure the concentration of this group of antibiotics in a sample. To allow an analysis on the field, the solution needs to be fully mobile and practical, so as to avoid carrying heavy, potentially unnecessary equipment. In this context, a new screening method was developed that utilizes a picture and a computer for the analysis. However, despite this approach improving the analysis process when compared to traditional methods, it is still not fully mobile, since it requires cumbersome equipment to be transported, as the processing is done on a laptop or PC due to software restrictions of the language used for the algorithm.

Smartphones' computational capabilities are increasing, and they are more powerful than many laptops of older generations. Taking this into account, we developed a mobile analysis application that leverages the computing power and ease of use of a smartphone. In the app, an existing image of the reaction can be loaded or a new one can be taken. This input will pass through a color correction algorithm to normalize the capture considering the environmental lighting. When the algorithm finishes processing the image, the app will return the estimated concentration of the sample. This approach enables in situ analysis, without requiring an Internet connection nor specific analysis equipment, and the ability to have a rather precise guess of the level of contamination of any water. It is also capable of using manual input to assist the algorithm should the automatic version fail.

Resumo

Antibióticos são utilizados para o tratamento de humanos e animais. Estes compostos podem ser encontrados em diferentes compartimentos aquáticos, incluindo águas residuais e de consumo. A presença de antibióticos nestes ambientes leva ao desenvolvimento de bactérias com resistência a antibióticos, que está relacionada com o aumento de doenças infeciosas sem tratamento.

Sulfonamidas são um grupo importante de antibióticos, e têm sido frequentemente encontradas no meio aquático. Um dos métodos mais comuns para a determinação de sulfonamides em água consiste na parceria de cromatografia líquida de alta eficiência (HPLC-MS/MS). Apesar da sensibilidade e seletividade alta deste método, é um processo caro, que consome muitos reagentes e não apropriada para análise in situ.

Uma propriedade importante de sulfonamidas é como estes compostos reagem quando é adicionado o reagente colorimétrico *p*-dimethylaminocinnamaldehyde. Quando estes dois compostos se juntam, obtém-se uma cor cuja intensidade está relacionada com a concentração de sulfonamidas na amostra. Isto abre a possibilidade de usar colorimetria para medir a concentração deste grupo de antibióticos numa amostra. Para permitir uma análise no local, a solução tem de ser móvel e prática, para evitar levar equipamento pesado, potencialmente desnecessário. Neste contexto, um novo método foi desenvolvido que utiliza uma imagem e um PC para efetuar a análise. No entanto, apesar de esta metodologia ser uma boa melhoria comparativamente aos processos tradicionais, ainda não é completamente móvel, visto que requer o transporte de equipamento pouco prático, já que o processamento será feito num portátil ou PC devido a restrições de software.

A capacidade computacional de smartphones tem aumentado, e são mais poderosos que muitos portáteis de gerações anteriores. Tendo isto em conta, desenvolvemos uma aplicação para análise mobile que aproveita o poder computacional e facilidade de uso de um smartphone. Na aplicação, uma imagem da reação pode ser carregada da galeria ou uma nova pode ser capturada. Esta fotografia irá ser passar por um algoritmo de correção de cores para normalizar a imagem considerando a iluminação ambiente. Quando o algoritmo acaba de processar a imagem, a aplicação irá providenciar a concentração estimada da amostra. Esta abordagem possilibita a análise in situ, sem necessitar de uma conexão à Internet ou de equipamento específico para o processamento, dando uma estimativa relativamente precisa do nível de contaminação de qualquer água.

Acknowledgements

First things first, this work financed by the ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e Tecnologia within project POCI-01-0145-FEDER-031756. I would also like to thank all the people involved in the project, namely professor Marcela Segundo, investigator Patrícia Peixoto, Henrique Carvalho and professor Hélder Oliveira.

It's important to note that this dissertation would not be finished if not without the support of my loved ones. Well, to put it more accurately, it would have been finished, I would just be a lot less sane by the end.

I would like to thank my father, mother and brother for their valuable suggestions, support and occasional "you spend too much time looking at computers" reminders. A big shout-out to my second family, which is an amazing gang of friends. I'll cherish all the countless moments we spent on Discord (including the one writing this very sentence) talking, bonding, listening to music, and even venting our frustrations, followed quickly by funny memes. Lastly, I would like to thank my girlfriend Margarida Silva, for the unbelievable patience and support, only enhanced by a sporadic screech that perfectly encapsulated her own problems.

I am truly, truly grateful for having such a fantastic group of people.

Pedro Reis

"These things never make sense. They just happen and we get swept up in the storm."

Commander Shepard

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Contributions . . . 3
1.4 Document Structure . . . 3

2 Literature Revision **5**
2.1 Sulfonamides . . . 5
2.2 Image Capture . . . 6
2.2.1 Colorimetry . . . 6
2.2.2 Android and Mobile Cameras . . . 7
2.2.3 Device and Lighting Variation . . . 8
2.3 Colorimetry as an Analysis Method . . . 10
2.3.1 Colorimetry Using Smartphones . . . 10
2.3.2 Colorimetry in Chemistry . . . 11

3 Architecture **15**
3.1 Requirements . . . 15
3.2 Key Architecture Decision Decisions . . . 17
3.3 Information Flow . . . 19

4 Analysis **21**
4.1 Color Checker Detection . . . 21
4.1.1 Automatic Color Checker Detection . . . 21
4.1.2 Manual Color Checker Detection . . . 23
4.2 Color Patches Identification . . . 25
4.2.1 Automatic Color Patches Identification . . . 25
4.2.2 Manual Color Patches Identification . . . 26
4.3 Disk Detection . . . 27
4.3.1 Automatic Disk Detection . . . 28
4.3.2 Manual Disk Detection . . . 28
4.4 Sample Detection . . . 29
4.4.1 Automatic Sample Detection . . . 30
4.4.2 Manual Sample Detection . . . 32
4.4.3 Extracting the Result . . . 33

5 Performance **35**
5.1 Memory Constraints 35
5.1.1 Algorithm Optimizations 36
5.2 Bitmap Size Constraints 36
5.2.1 Image Resolution Optimizations 37

6 Tests **39**
6.1 Precision Comparisons 39
6.2 Automatic vs Manual Methods Comparisons 41
6.3 Mobile Performance Metrics 41
6.3.1 Algorithm Stage Comparisons 42
6.3.2 Overall Quality Slider Comparisons 43
6.3.3 Color Checker Finding Precision Slider Comparisons 44
6.3.4 Optimized Results 44

7 Conclusions and Future Work **47**

References **49**

List of Figures

2.1 Example of color correction algorithm for Tongue Imaging. 8
2.2 ColorChecker segmentation process. 13
2.3 Patches Center Segmentation. 13
2.4 Patches bounding box. 13
2.5 Disk Identification Process 14

3.1 Sample Menu Screen. 16
3.2 Settings Menu Screen. 17
3.3 Example image. 18
3.4 Flowchart of the application. 20

4.1 OpenCv Fill Holes Operation. 23
4.2 Automatic Color Checker Detection. 24
4.3 Manual Color Checker Detection. 24
4.4 Automatic Patches Detection. 26
4.5 Manual Patches Detection. 27
4.6 Warping with Manual Patches Detection. 27
4.7 Automatic Disk Detection. 29
4.8 Manual Disk Detection. 29
4.9 Color corrected image. 30
4.10 Automatic Sample Detection. 31
4.11 Manual Sample Detection. 32
4.12 Results Screen. 33

List of Tables

6.1 Hue differences between the application and *MATLAB*. 40
6.2 Concentration precision tests for 24 and 13 patches. 40
6.3 Difference between automatic and manual version. 41
6.4 Average time for each stage. 42
6.5 Impact of overall quality slider. 43
6.6 Impact of color checker precision. 44
6.7 Tweaked settings impact. 44
6.8 Comparison with low-end smartphone and desktop PC. 45

Abbreviations

AgNP	Silver **N**ano**p**article
CNN	**C**onvolutional **N**eural **N**etwork
DNN	**D**eep **N**eural **N**etwork
GPU	**G**raphics **P**rocessing **U**nit
HPLC	**H**igh-**P**erformance **L**iquid **C**romatography
HPLC-MS	**H**igh-**P**erformance **L**iquid **C**romatography tandem **M**ass **S**pectrometry
LCC	**L**inear **C**olor **C**orrection
MP	**M**ega**p**ixel
MS	**M**ass **S**pectrometry
OpenCV	Open **S**ource **C**omputer **V**ision
OpenGL	Open **G**raphics **L**ibrary
PPG	**P**hoto**p**lethysmo**g**ram
RAM	**R**andom **A**ccess **M**emory
RGB	**R**ed-**G**reen-**B**lue
SA	**S**ulfon**a**mides
1μPAD	**M**icrofluidic **P**aper-based **A**nalytical **D**evice

Chapter 1

Introduction

As modern medicine advances, more forms of combatting diseases arise. Sulfonamides are an important group of antibiotics, and they are widely used not only to treat humans, but also farm animals [17]. Sulfonamides and their metabolites are frequently found in environmental water. They can reach this medium through different pathways, such as wastewater discharges, contaminated manure and slurry, and aquaculture. Their presence accelerates natural selection on bacteria colonies in the water, since sulfonamides kill the bacteria that cannot resist antibiotics, allowing the others to proliferate. It was also found that the aquatic medium is highly favorable to the transfer of genes [3], which lead to more bacteria acquiring the antibiotic resistant gene. This escalated into the global threat of antimicrobial resistance, hence the quantification of sulfonamides concentration in water is crucial to assess environmental risk and to establish health and environmental policies.

Currently there are several strategies being used to quantify sulfonamides, but for most traditional methods the researcher has to collect a water sample to be tested and take it to a laboratory capable of doing this procedure. This approach is impractical, expensive, and not mobile.

1.1 Motivation

Recently, a new methodology was developed to detect sulfonamides in water by Carvalho et al. [5]. This approach is based on the colorimetric reaction between sulfonamides and the reagent *p*-dimethylaminocinnamaldehyde, after retention of these antibiotics in a solid support. It was observed that there exists a relation between the intensity of the color product to the concentration of sulfonamides in the sample. Knowing this, research was done to conclude whether an image of the color was enough to quantify sulfonamides in μg per liter using digital colorimetry. The investigation concluded that this approach was viable, and provided a more streamlined process to get an estimation of the degree of contamination in waters. However, despite being more practical and much less expensive, it still has its own drawbacks, namely requiring a computer to run the analysis algorithm.

Traditional methodologies for analyzing contaminated water for sulfonamides have limitations. They require the investigator to collect a sample in situ, transport it to a lab, and expensive and resource intensive processes. On top of that, in some remote regions of the world, getting the sample from the location to the lab can also be a costly and time consuming endeavour. An algorithm using digital image colorimetry can mitigate some of these limitations. For example, after collecting a sample, the investigator could later use software to process the image of the chemical reaction between the sulfonamides and the reagent. This would be less expensive, and generate less overhead to get an estimation of the contamination. This approach was researched by Carvalho et al. [5], where an algorithm was created that could accurately extract the value of sulfonamides concentrations in a sample. However, requiring an extra piece of equipment to perform the processing hinders the solution's mobility, as it would need to be either carried to the field, or the sample be transported and later processed in a lab computer.

Mobile devices are not only becoming more prevalent [15], but also more computationally capable. A smartphone could handle taking the picture of the reaction, and after this there were two possible solutions:

- Upload the captured photograph to a web server, and run the already finished algorithm on that capture. A simple approach, with potentially the best results in terms of performance, since the performance could be improved by upgrading hardware on the servers. It also enabled the algorithm to be used without needing any porting effort. However, this has two major disadvantages, which are the reliance on an Internet connection and the need for servers capable of processing the algorithms. The former is of critical importance, as there are use cases of researchers in Africa analyzing waters in remote regions with no Internet access.

- Use the photograph and run a new, mobile-friendly version of algorithm to detect sulfonamides concentration locally. This would be the most complex task, as it requires the porting of the algorithm, coupled with the need for performance optimizations, since smartphone hardware isn't as powerful as a computer. Despite this, it has none of the major shortcomings of the already mentioned alternative. This makes all the difference if an area or device has connectivity issues.

Given both options, we decided to tackle running a new algorithm locally, as its advantages far outweigh the disadvantages, it would be a fully mobile solution, and is overall a more challenging approach.

1.2 Objectives

There were several key objectives to this research:

- **Fully migrate the algorithm to a mobile-friendly language**: The algorithm done by Carvalho et al. [5] is reliant on *MATLAB*, which is a proprietary language that mobile devices

cannot run. There is a version of *MATLAB* for Android (*MATLABMobile*), but it is reliant on an Internet connection for any processing. This would not be ideal, since some of the regions could be remote, and no Internet connection would be available; In order for the processing to take place in the device and take advantage of the work already done, the algorithm would have to be ported to a new language. This removes the limitation of working with proprietary software, allowing the team to be more in control of the entirety of the code;

- **Use manual input to assist the algorithm**: Some steps of the algorithm, like cropping the image to just contain the sample, can be done manually. Using the touch capabilities of smartphones, we can give the user direct control on specific parts of the algorithm, if the automated version fails;
- **A high-performance application**: Despite the advances in mobile computing, it will never be as good at performing complex operations as a laptop or a desktop. For this reason, performance is a critical priority, so as to allow the quick analysis of multiple pictures. Ideally, the user could also tweak the level of precision to allow for faster analysis.

1.3 Contributions

At the end of the dissertation we have:

- A fully functional standalone application;
- An adapted version of the algorithm done Carvalho et al. [5] running on an Android device;
- An interface that the user can interact with to assist the analysis process.

1.4 Document Structure

This dissertation will have the following structure:

- Chapter 2 - Literature Revision: In this section we analyze in-depth the current state of affairs of sulfonamide analysis, as well as approach any area that might be related to this dissertation. It is subdivided in three sections, exploring sulfonamides, the image capture process, and the usage of colorimetry as an analysis method;
- Chapter 3 - Architecture: An analysis of the key user requirements for the application, followed by the structural components and key design decisions to allow all the requirements;
- Chapter 4 - Analysis: An in-depth exploration into every stage of the algorithm, divided by it's automatic and manual counterparts, with images of the interface;

- Chapter 5 - Performance: A list of all the necessary optimizations that were necessary to increase stability and reduce analysis times;
- Chapter 6 - Tests: A summary of all the tests done to validate all the aspects that relate to the application's execution.

In this chapter we addressed what motivated this dissertation, the contributions we have made at the end of development, and how this document is layed out.

Chapter 2

Literature Revision

To organize the literature revision, we split the research into three different components:

- Sulfonamides: What the compounds are and what are the traditional methods to detect them in aquatic solutions;
- Image capture: Images are the input that is going to be used to obtain a result, so we will delve into every aspect of the image capture and processing issues;
- Colorimetry as an Analysis Method: An exploration into how colorimetry can be used for analysis, first by viewing how smartphones can help in this function, followed by colorimetry in the chemistry and physics fields.

2.1 Sulfonamides

Appearing in 1968, sulfonamides are an important functional group of antibiotics that are very widely used. Some of it's common use cases revolve around the treatment of both urinary and upper respiratory tract infections [7]. The widespread consumption of this compound is understandable, since it is low cost, low toxicity (i.e. well tolerated by patients), and very effective against bacteria. However, the appeal of the consumption of antibiotics is promoting an overprescription of these compounds [16]. To further complicate matters, these drugs have also been used for the treatment of livestock. These factors cause an increase in the appearance of sulfonamides in environmental water.

The presence of sulfonamides in water promotes the proliferation of bacteria colonies with antibiotic-resistant genes. To measure the concentration of these compounds, there are several methods, with the ones listed requiring a laboratory capable of performing these procedures:

- High-performance liquid chromatography tandem mass spectrometry (HPLC-MS): This method essentially combines two different processes, where liquid chromatography separates the individual components of the sample, and mass spectrometry analyses the mass structure of each element [11]. This synergistic approach is extensively used, not only for

the detection of sulfonamides, but also other classes of organic compounds composed by multiple molecules;

- Capillary electrophoresis: The previous method can sometimes be limited by low efficiency in the separation process, which is critical in residues analysis [14]. This method can be an alternative to that approach, by separating the components of a solution according to their charge and size. This is done by the use of a very small tube (capillary), in which the sample (injector) travels until reaching a destination vial. Smaller molecules are faster than larger ones, and in a point of the cable there is a detector (integrator or computer) that analyzes what is passing through the capillary;

- Immunoassay: Antibodies are highly specific molecules capable of binding to a particular target structure. In the context of the research done by Li et al. [20], an antibody was created that could bind to the structure of sulfonamides. This area of investigation started in 2000, when Muldoon et al. [25] produced a monoclonal antibody that detected eight sulfonamides. More recently, Korpimäki et al. [19] produced antibodies capable of detecting different sulfonamides under various levels of concentration;

- Gas chromatography coupled with atomic emission detection: Developed by Chiavarino et al. [6], it can analyse nine sulfonamides by initially separating the compounds using gas chromatography, and then using atomic emission detection for the determination of the sulfonamides.

2.2 Image Capture

One of the major components in this dissertation is how images are used for the estimation of sulfonamides. This chapter will delve into the image capture process and we will approach three key factors for the analysis, namely an explanation of what colorimetry consists in, followed by how images can be captured on smartphones, and the problems that this platform has when it comes to color constancy in photos.

2.2.1 Colorimetry

Colorimetry is a methodology that quantifies physically the human color perception [26]. Similar in nature to spectrophotometry, it is solely focused on the visible region of the electromagnetic spectrum.

To perform colorimetry, a device capable of capturing the wavelengths of visible light is required. Traditional tools include:

- Tristimulus colorimeter: Also called a colorimeter, it is frequently used for display calibration. It works by analyzing the colors the screen emits using photodetectors, to establish a display profile that can be used for the calibration [27].

- Spectrophotometer: a device that can quantitatively measure the spectral reflectance, transmittance or irradiance of a color sample [29].

Although these devices are specialized in obtaining raw wavelength data, they are geared only for experienced users, like photographers or researchers, not the average consumer. For this reason, any function that might require an average user (like taking a picture to analyze a wound) becomes out of reach for most. A smartphone is a more practical, common tool, that has most of the features required for this type of analysis, albeit with some issues. We will address the topic of Android more thoroughly in the next subsection.

2.2.2 Android and Mobile Cameras

It's important to mention why Android was the target operating system. This decision was made in 2016, and iOS was not part of the scope of the project. However, we could also use frameworks such as React Native, Ionic or others to create an application for both operating systems. This presented a problem, as we used *OpenCV* for the processing, and the SDK is either made for iOS or Android. Since we needed to create a very high-performance application using this framework, we decided to focus exclusively on native Android development, as developing for iOS would require essentially doubling the amount of work, which could compromise the robustness of the final application.

Android is one of the largest current operating systems in the world, with over 70% market share as of June 2020. Developed by Android Inc. and later acquired by Google in 2005, it has grown massively since it's debut in 2007. It now amasses over 85% of the smartphone market share, totalling over 329 million devices as of the second quarter of 2018 [22].

Current Android devices are computationally capable and equipped with a slew of features, such as connectivity, Bluetooth, database storage, and others. For the aspects of this thesis, the most important capability of these devices is the camera.

The first mobile camera appeared in 2000 with the J-Phone, featuring 0.11 megapixels. 6 years later, Sony released the Sony Ericsson, with a 3.2MP camera. Not only was this a lot more powerful in terms of raw pixel count, it also featured more technologies such as image stabilization, auto-focus and a flash.

Tech giants entered a race for megapixels, with again Sony releasing in 2009 the Sony Ericsson S006, featuring a 16MP camera. Around that time, the smartphone industry started to boom, and devices with large cameras were becoming more inconvenient. Size and form factor started to carry more importance, and the previous model's cameras were quite bulky. For these reasons, camera development reached a standstill, since it was becoming more apparent that it was not only raw megapixel count that determined the quality of a picture [13].

Software was coming to the front stage, with companies starting to developing more and more features, such as HDR, panoramas, and more intelligent ways of capturing detail. Images were being processed with more advanced methods, some even featuring AI to enhance a photograph. This advancement in software features is a double-edged sword however, as two cameras with

similar hardware specifications might produce completely different results. This presents a large problem with using colorimetry, since the accuracy of the color value is of the utmost importance for the final result.

2.2.3 Device and Lighting Variation

When the color of the image is critical to the result of the processing, it becomes vital to ensure that different devices under the same conditions can capture a very similar photograph. Unfortunately, this does not happen, as the vast majority of smartphones operates in the RGB color space, which is device dependent, i.e. changes in every equipment. To address this, before using the image, it needs to be processed from a device dependent color space like RGB, to a device independent color space like CIE L*a*b*, CIE XYZ or sRGB.

Color constancy is the perception of the same color appearing constant even in varying lighting scenarios [10]. Color correcting images to a device independent color space helps achieve constancy by approximating the hue of the colors to the real hue of the objects. There are multiple algorithms that can transform color spaces, such as:

- Root-Polynomial Regression: Mapping from a device dependent RGB color space to a device independent XYZ space using linear color correction (LCC) can often have a high amount of error. Polynomial color correction can have higher precision, but it is susceptible to changes in exposure. Root-polynomial regression was created to solve this problem. It is an extension of LCC with low complexity that enhances color correction performance [9]. An example of the color correction method can be seen in figure 2.1

- Simulated Annealing: Standard camera calibrations use a color checker, which is a professional grade equipment featuring color patches, where the values are known beforehand by the manufacturer. This allows the application of a transformation matrix from each patch to its corresponding known value in the XYZ color space. A common methodology to do this is by using a least-squares regression, but there is an error associated caused by algorithm limitations and CCD dark currents (when unwanted free electrons are generated in a charged coupled device of a camera due to thermal energy) [30].

Figure 2.1: a) Photo captured with sRGB sensors, b) Photo captured without sRGB sensors, but color corrected to sRGB using polynomial model , c) Same color correction applied after intensity of light increase [9].

Machine learning was an avenue of exploration for color constancy. Bianco et al. [4] used a Convolutional Neural Network (CNN) to estimate the scene illumination using as input a RAW image. It uses a color checker to establish the ground truth, and utilizing a CNN combining feature learning and regression the results obtained were better than every state-of-the-art technique at the time. Around the same time, Lou et al. [23] was also exploring AI in the same field, more specifically deep learning neural networks. By approaching color constancy as a DNN-based regression and using trained datasets with over a million images, they managed to outperform state-of-the-art implementations by 9%, while maintaining good performance throughout.

Dang et al. [8] also highlighted the importance of color correcting images when used for health purposes, more specifically a photoplethysmogram (PPG). This operation is used for detecting blood volume changes through illuminating the skin and measuring changes in light absorption [2]. Using different devices for capturing the photo results in errors that reduce the precision of the measurements. For this they developed a color correction algorithm using a least square estimation based method, and with this reduced the mean and standard deviation differences between devices.

In the context of sulfonamide analysis using digital image colorimetry, Carvalho et al. [5] compared the efficiency of different color correction methods. For all tests, they used a color checker as mentioned in the simulated annealing method. The current implementation of their algorithm tested out 5 different methodologies:

- Weighted Gray Edge: Different edge types exist in images: shadows, highlights, materials, etc. These different edges contribute differently to illuminant estimation. This algorithm classifies edges based on photometric properties, and then evaluates the edge-based color constancy according to the edge types. The research showed that using an iterative weighted Gray-Edge based on based on highlights reduces the median error by about 25%. In an uncontrolled scenario, it can offer improvements of up to 11% against regular edge-based color constancy [12].

- Illuminant Estimation Using White Patch (White): Utilizes a white patch, and estimates the illuminant based on the difference of the value of the the patch to absolute white, since any deviation is caused by illumination.

- Illuminant Estimation Using Achromatic Patches (Neutral): Similar to the previous method, but uses the average of a 6 patch color set.

- Color Correction Matrix RGB to RGB: Used the 24 colors provided in a color checker, and compares the colors to the reference values provided by the manufacturer. Uses a least square regression between the two sets of values.

- Color Correction Matrix RGB to XYZ: Similar to the previous method, but maps to the XYZ color space.

Comparisons were done between the standard deviations of all the previous methodologies in each color space, and in over 90% of them, the color correction matrix from RGB to XYZ yielded the best results.

2.3 Colorimetry as an Analysis Method

We will divide the usage of colorimetry in analysis in two components, namely how smartphones can be used for this task, followed by colorimetry in chemistry.

2.3.1 Colorimetry Using Smartphones

One of the fields that can be most improved with the introduction of colorimetry using smartphones is Medicine. Traditional medical equipment, such as X-Ray machines or spirometers are expensive and complex, and because of this, are only used by trained professionals in clinical environments [1]. These limitations can increase the waiting time for patients, which could be critical in an early diagnosis.

Modern smartphones have many sensors that can be useful in assisting a medical diagnosis, such as microphones to record user input, heart beat monitors, or cameras. Due to the context of this dissertation, we will focus only on the latter.

One example of the camera being used in medicine is for the analysis of diabetes wound healing. In the US, 7.8% of the population have diabetes, and over 5 million patients have chronic wounds that require frequent visits to the doctor. Agu et al. [1] designed an application called Sugar with the intention of helping the user track the progress of their wounds more easily. To achieve this, the application had the following execution flow:

1. The patient captures a photograph of the wound;
2. The photograph is decompressed;
3. Algorithm calculates wound boundaries through set segmentation;
4. Relative size of red, yellow and black tissues within the wound are measured, and these areas outline the current progress of the wound healing.

Another example of a smartphone camera using colorimetry for a medical situation was the analysis of tongue images. Wang et al. [28] researched in the area of a traditional chinese medicine, where tongue images can indicate physiological differences on the body. The researchers faced the same adversities that we mentioned previously, namely the device variation arising from the RGB color space. Photographs captured in one device might contain different color values from another photograph taken in the same environment with another smartphone. Traditional methods to ensure color constancy like polynomial regression were not very successful due to the restricted range of colors of the tongue, so they needed to be further optimized. Firstly, they chose the target device independent color space. The decision process obeyed the following criteria:

- It had to be well endorsed by companies, since it needed to be easily stored and exchanged;
- Should be RGB-like to display images on monitors without any extra processing;
- Needs to have a constant relation with CIE $L*a*b*$ to allow calculation of color differences.

The color space that best fit these criteria was sRGB. The color correction algorithm also utilizes a color checker to calibrate the device for the scenario of the tongue images. After the algorithm was developed and applied, the researchers reached a difference of under 0.085% compared to the real value, with a high adaptability to different tongues and varying lighting conditions.

Another example can is the work of Jarujamrus et al. [18] for the analysis of mercury in water. A small amount of this element in water can produce massive damages to the entire ecosystem that depends on it. For this reason, they developed a way to use smartphones and colorimetry to assess the level of mercury contamination in water.

Mercury does not have a visible representation, so in order to analyze a sample using colorimetry, there had to be a way to visually check the concentration. For this they created a reaction that produced a color that changed intensity according to the amount of mercury in the sample. To achieve this they used a microfluidic paper-based analytical device (μPAD) that was coated with synthetic silver nanoparticles (AgNPs). These particles disappeared from a deep yellow to a fainter yellow as the concentration of mercury increased.

Previous iterations of the prototype built were not very practical, as the methodology required a digital camera, with an optimized control light box. After the photo was taken, an external software needed to be run, and since the camera could not perform any calculations, this operation was done in a computer. Another limitation was the fact that it was not easy for an unskilled user to correctly perform the operation. All of this meant that this methodology was not available for on-site usage.

To improve upon the previous version, they opted to use a smartphone and the optimized light box. A smartphone solved easily two of the previous problems: it had a built-in camera with several features, including flash, (one of the method's requirements), and the ability to process calculations locally. Before this newer prototype, determining the concentration of mercury was not an easy task. It required complex laboratory analysis, and it required the sample to be transported, a problem similar to sulfonamide analysis.

The researchers concluded that this prototype could be used as a viable alternative for the task. It was fast, simple, with an instant report on the screen of the device, and most importantly, offered comparable accuracy to a laboratory analysis. Another advantage was that it was easy enough for an unskilled user to get reliable results.

2.3.2 Colorimetry in Chemistry

In this subsection we will delve into how colorimetry can be used for analysis in chemistry. For this, the most relevant work in the context of this dissertation was the algorithm done by Carvalho et al. [5]. The goal of their work was to research a more inexpensive way to assess concentration

of sulfonamides in water samples, as traditional methodologies were impractical and resource expensive since as they required a sample to be sent to the laboratory to be analyzed. This increased overhead to obtain results, diminishing efficiency. To solve this problem, they explored using digital image colorimetry to extract the result from a sample.

For their new method, a chemical reagent was required (*p*-dimethylaminocinnamaldehyde), but the process was streamlined, and did not require any complex laboratory equipment for the analysis process. To analyze the reaction, all that was necessary was a smartphone camera to take a picture of the reaction, and a computer to run the algorithm.

The algorithm needed to be fully automatable, with the full sequence of events of the implementation bwing as follows:

1. **Acquire photo**: To test the algorithm, many pictures were taken, varying in sulfonamide concentration, and device that captured them. To make the concentration vary, samples were prepared in a laboratory to ensure the values of the concentration were known beforehand. This allowed the results of the analysis to be directly compared to the real values. Photographs were taken with different smartphones to determine the impact that device variation had on the final results;

2. **Segmentation**: Since the algorithm was completely automatic, it needed to detect where the color checker and the sample were in the picture. The segmentation of each object is done individually. The objects that need to be identified are as follows:

   (a) **Color checker:** Before we identify the individual patches, it is necessary to first locate the color checker. For this research, an x-*rite ColorChecker Passport* was used, that contained 24 patches, with a black frame. This last characteristic is important, since after grey-scaling the image, a multi-level Otsu thresholding is used that will identify the darkest contours in the image. The largest found contour is the color checker. After that, knowing the orientation of the color checker, the selection is split in two and the side of the object containing the color patches is selected. An overview of the process can be seen in figure 2.2.

   (b) **Patches:** After having selected the region of the image containing the patches, it is necessary to select the center of all the individual squares, so a new Otsu thresholding operation is performed. However, it is difficult to binarise the image in a way that selects all the patches. For this reason, since we know that the color checker contains a grid of 6 by 4 patches, the position of the missing pieces in the grid are estimated. A visual representation of this segmentation process can be seen in figure 2.3.

   To mitigate the possibility of the selected pixel being an outlier due to some artifact, a bounding box is created for each patch, with the identified point as its center. From this bounding box, the median value is selected. The visual representation of the results can be seen in figure 2.4

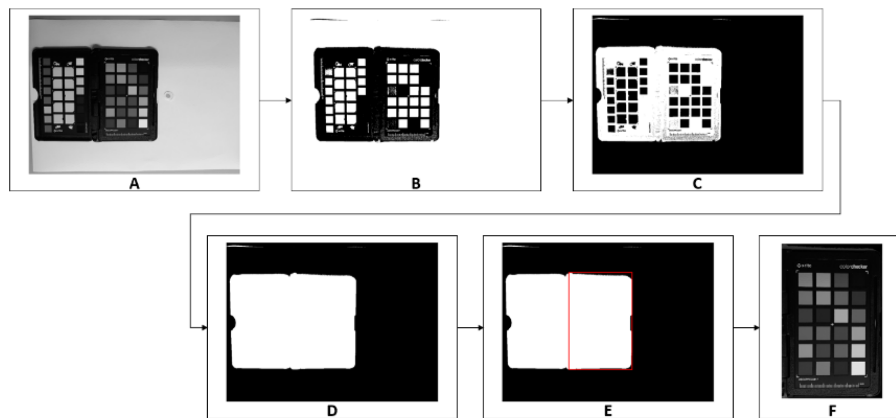

Figure 2.2: ColorChecker process [5].

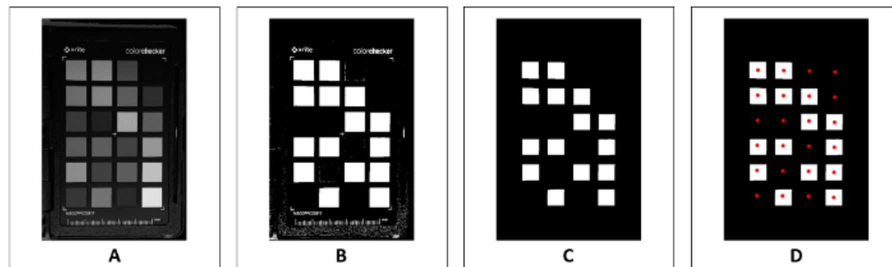

Figure 2.3: Patches Center Identification [5].

(c) **Disk:** Having identified the color checker, the only element remaining is the disk. This is a more complex object to binarise, because in the case of low concentration the color is not intense enough to be easily observed in a histogram. To solve this, the image is grey-scaled, normalized between 0 and 1, and then equalised using the histogram. After this process, another Otsu thresholding is done, and this results in the outline of the selection. With this, the median of all values is obtained, and the color is extracted. The process can be seen in figure 2.5.

3. **Color Correction**: With the information of the color patches, they tested various color

Figure 2.4: Patches Bounding box [5].

Figure 2.5: Disk identification process [5].

correction algorithms as mentioned before. The results indicated that the best precision was achieved when using the color checker to map from the RGB color space to the XYZ color space, which is device independent.

An important factor for the calculus of the concentration is the relation between the concentration of sulfonamides and the color of the reaction in all color spaces and color components. The results showed that using the Hue color component provides the best mapping between color property to the concentration.

In this section we analysed the current literature on sulfonamide detection, colorimetry and setbacks of using smartphone cameras for colorimetry tasks. We also explored the basis for this project, which is the work done by Carvalho et al. [5]. In the following chapter we will analyse the overall architecture and key design decisions for the application.

Chapter 3

Architecture

Before delving deep into the design decisions, it's important to know what is expected of the application in terms of use cases and requirements. By always adhering to what was required, the application maximizes it's usability for the end user.

3.1 Requirements

The key advantage of this application is that it enables the analysis of a sample on the field, without requiring any specific equipment for the task. With this use case in mind, several requirements arise. Several functional requirements were identified in the application planning, namely:

- The user needs to be able to load previously taken pictures, or take pictures from within the application to launch an analysis;
- The user needs to be able to store samples, so that they could analyse them later;
- The user can intervene directly in the algorithm, with a manual implementation for each step, so a valid estimation could be achieved, even if the automatic version fails;
- The user can select a specific number of patches for the color checker, as well as utilizing a revised 13 patch-set done by Carvalho et al., in order to have full control over the patches used;
- The user can have select the level of precision of the algorithm, to allow control over the time it takes to get an analysis.

For the first requirement, in order to begin analyzing a sample, the user would have to input some image, either by loading an already taken picture, or by launching the phone's camera application and capturing a photo. After that, the flow of information converges to the analysis setup screen. The interface for the analysis setup screen is in figure 3.1.

In the picture, the analysis setup interface is shown, where the user views the image that needs to be stored, and is required to provide a name to later help identify the sample. If the user either

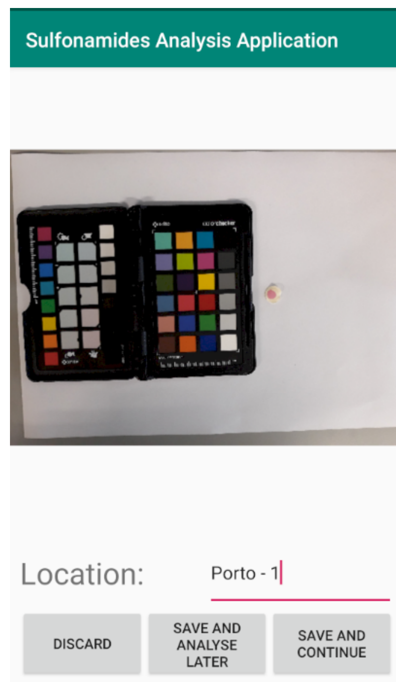

Figure 3.1: Sample Menu Screen.

selects to continue the analysis, or analyze later, the image is stored in the application's external memory, saving the date of the upload as metadata. This storage is required to not only allow the user to continue with the tests without immediately analyzing, but also to re-analyse a picture if a mistake was made. With this system in place, the second requirement was handled.

Another requirement that arose in the middle of development was the possibility of utilizing a specific set of color patches of the checker for the color correction matrix. The color patches are all the individual squares on the right side of the color checker. The original algorithm used exclusively all 24 patches for the color correction matrix, but more recently Carvalho et al. [5] concluded that using a specific subset of 13 patches obtained better results. As this requirement appeared mid-development, the color patch stage had to be reworked to enable this feature, while demonstrating clearly to the user which were the patches that were being used. There are three different patch sets to choose from: the full 24 color patch set, the revised 13 patch set, and a custom setting, in which the user could select whichever patches they wanted. This configuration is stored persistently.

The final requirement was related to performance, as we always wanted to go above and beyond the minimum requirements, and one idea that was established to improve the usability of the application was a customizable level of precision. For this, we developed two main optimizations that can be controlled, namely the color checker finding precision and the overall quality. We will go in-depth into what these two options do in the performance chapter. However, in order for the user to tweak all the mentioned options, we needed to have all of them in a settings menu, which can be seen in the figure 3.2.

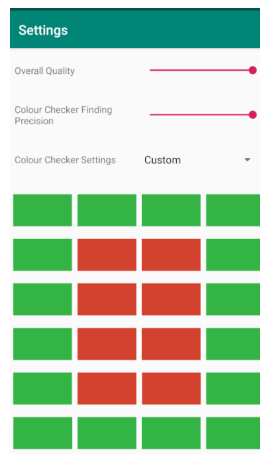

Figure 3.2: Settings Menu Screen.

3.2 Key Architecture Decision Decisions

The first major decision on creating this mobile application was the selection of the target devices. Our goal was to support the lowest possible Android version that didn't compromise the requirements for the app, which ended up being version 4.0.3. At the time of writing, it is estimated that all Android devices on the market run a version of Android equal or higher than 4.0.3.

The second major component regarding the architecture is the framework that was used to port the algorithm itself. Since *MATLAB* cannot run on Android devices, a capable alternative had to be found. For this purpose we investigated what image processing frameworks were available for Android that also allowed for an extremely high level of performance. We selected *OpenCV* for the following reasons:

- It is well supported, with a specific build for Android;
- It is overall decently documented, expediting the development process;
- It is the most comprehensive alternative to *MATLAB* that could run on Android device;
- Allows the code to be run without requiring any Internet connection;
- It is open-source and does not require any type of licenses;

After deciding on the framework to use, we began designing the structure of the application. From the start of the project, one of the major avenues for improvement was the utilization of user input as a backup plan in case the automatic detection fails in any step. This would boost the robustness of the application, allowing the extraction of results from a photo, even if it couldn't be done automatically. The automatic process could fail for various reasons, but the most reliably reproducible error is when the background of the photo has high contrast elements that influence the thresholding results.

Figure 3.3: Example image.

In order to fully realize the feature of using manual intervention, the architecture of the application had to take into account the possibility of incorrect automatic results, remaining flexible enough to allow obtaining the expected results manually. To achieve this, a chain of inputs and expected outputs for each stage was designed. This enables the user to intervene, regardless of which step the algorithm is in, and then seamlessly transition to the next stage, which will be performed automatically again. This shift back from manual to automatic increases accuracy, while prioritizing speed by removing the human input bottleneck whenever possible.

Before going into the details of the architecture, it's important to have a visual example of the types of images the algorithm will work on. Figure 3.3 showcases a typical photo, in which the color checker is located left of the disk, with the color patches on the right side. With this information in mind, it is easier to understand the steps to obtain a concentration result.

One of the aspects of the work done by Carvalho et al. [5] that was very advantageous to the architecture design was that it clearly outlined the process in five key steps:

1. Detection of the Color Checker: Identifies the corners of the part of the color checker that contains the color patches;
2. Detection of the Patches: Identifies all the color patches on the color checker;
3. Color Correction: Creates a color correction matrix that will be used in fixing the lighting variations on the images;
4. Detection of the Disk: Detects the disk in which the sample is located;
5. Detection of the Sample: Identifies the parts of the disk that contain the sample after the reaction.

Since these steps are clearly separated, it allowed the entire process to be done sequentially, and with a clear visual feedback to the user if the step has gone wrong. Knowing if a step went

wrong, the user can intervene and manually correct the results, and progress to the next step. This can be visually represented by the flowchart in figure 3.4.

After outlining the key stages of the algorithm, we now need to approach the specific information and data structures flow that make using manual input possible.

3.3 Information Flow

The inputs and outputs of each stage are chained to allow the user to potentially intervene in only one step, without breaking the next automated steps. To allow this we needed to establish a clear flow of information and data structures, which goes as follows:

1. Detection of the Color Checker:
   - Input: Base image;
   - Output: An array of points for each corner;
2. Detection of the Patches:
   - Input: Cropped image of the right side of the color Checker using previous step's points;
   - Output: All the color patches in the grid;
3. Color Correction:
   - Input: All the color patches found in the previous step;
   - Output: Color correction matrix;
4. Detection of the Disk:
   - Input: Base image and corners of color checker;
   - Output: Bounding box for disk;
5. Detection of the sample:
   - Input: Color corrected image and disk bounding box;
   - Output: Median color of the sample;

In this chapter we explained the key design decisions in terms of data structure and flow of information to allow meeting the established requirements. With a strong knowledge of the overall architecture and design process of the application, we will explore comprehensively into how the algorithm performs each step, and what alterations had to be done from the original *MATLAB* code in the following chapter.

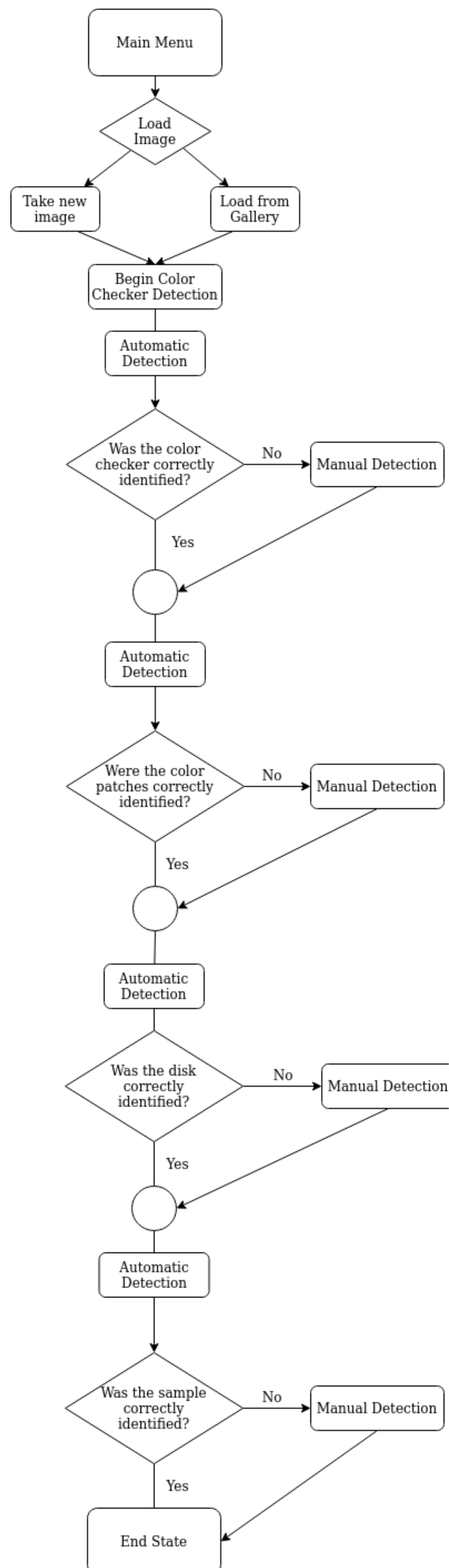

Figure 3.4: Flowchart of the application.

Chapter 4

Analysis

In this chapter we will go through the process of porting each of the stages of the algorithm, as well as the challenges we had to overcome on each of them. We will first explore the automatic version of each step, followed by the manual version, since the latter was far simpler to implement, and is easier to explain.

Throughout this chapter, to visually demonstrate what the user sees on each stage there is an accompanying figure, and they all follow the same logic:

- On the top, there is the visual result of each step;
- Below the image, there is a short description to guide the user that varies from stage to stage, and whether it is in automatic or manual mode;
- Below that, there are button prompts to control whether to move forward or backward in the analysis process.

Let's start with the first step, the color checker detection.

4.1 Color Checker Detection

The first step to be done is the identification of the portion of the color checker containing the patches. In terms of raw complexity, this phase was the most challenging to complete, since many helper functions that were used later in the the algorithm had to be created in this stage.

4.1.1 Automatic Color Checker Detection

The first operation to do is to remove possible fireflies or small noise in the raw image. For this effect, the original *MATLAB* code used a function that *OpenCV* does not have, namely `medfilt2`. Situations of missing functions ended up being somewhat common, and in those cases we looked at what was the expected outcome of the desired operation, and either manually implemented a function that served a similar purpose, or used a close alternative. We will mention whenever we

hit these obstacles throughout this chapter. In this case, a very slight blur did the intended purpose of removing fireflies well (and fast) enough to be acceptable.

After blurring the image came one of the biggest hurdles. As of writing this thesis, *OpenCV* does not support multi-level Otsu thresholds. In this function, an Otsu threshold is performed n-times through histogram analysis. There are versions of this algorithm for *OpenCV*, but these are done for the C++ version of *OpenCV*, which is not what we are using. The Java version of *OpenCV* for Android came with a prohibitively high impact to performance when iterating through matrices with thousands of elements, so a new solution had to be devised. Throughout the original *MATLAB* code, there are several usages of this type of Otsu threshold, but all of them shared two characteristics: they were all three leveled thresholds, and the extracted result was always the lowest values. Taking this into account, we created a several step system to achieve a similar result, while always using the native libraries for maximum performance:

1. Perform a regular Otsu threshold, generating a binary image for the center threshold;
2. Store the values in two masks, one being the regular binary image (for the values above the threshold) and an inverted binary image (so that the values below the threshold appear as white);
3. Get the maximum value of the image below the threshold, which was an easy way to acquire the threshold value;
4. Create a new image containing a filled image with the values above the threshold with the value acquired in the previous step;
5. Create a new image consisting of the multiplication the original image with the inverted mask. This image will be filled with black above the threshold and the original image outside of those zones;
6. Do a bitwise `OR` operation with the two images. This generates the final image, containing the original image below the threshold, with the rest filled with the threshold value;
7. Do a new Otsu threshold with the new image. This will be the second tier of the multi-level Otsu, and the values below the new threshold will be the results we are looking after.

This multi-level Otsu, despite being more taxing to performance compared to a single Otsu threshold, is still fast enough to be used with requiring too much time, while gaining robustness to different images under different lighting conditions. The increase in reliability is due to the fact that the color checker frame is very close to black, which makes it easy to identify under normal thresholds. A multi-level Otsu is beneficial when there are other dark elements in the image that may be lighter, but are caught in the threshold, which lead to incorrect results.

After performing the threshold, the patches inside the color checker will be excluded from the selection due to their wide range of values. *MATLAB* has a function to fill the holes in an image,

Figure 4.1: OpenCV Fill Holes Operation [24].

but *OpenCV* does not have a standalone function reserved solely for this purpose. Fortunately, an alternative was not hard to derive, while still maintaining a very high level of performance.

Initially, a simple flood fill from the first pixel, followed by an inversion would generate the mask correctly and when performing a bitwise OR with the initial threshold, the holes were filled correctly. However, there is an edge case in which the first pixel is already white, which would render the whole operation useless. To combat this, we tweaked the algorithm, so that the flood fill would be performed by every pixel in the borders of the image, if the pixel in question was set to black. This ensures the best results, and despite having a small performance penalty associated with it, it's negligible since the flood fill operation is not performed too many times. A visual representation of the process can be seen in the figure 4.1, where the left image is the base image, followed by the original threshold. After that, the a new image is created based on the threshold, using a flood fill on the corners, is later inverted, and then added with a bitwise `OR` with the original threshold. This produces the same results as *MATLAB*'s implementantion [24].

At this point the thresholds are identified, so what remains is to find the largest contour in the image. Although some tweaks had to be done, the overall process is quite straightforward, since *OpenCV* natively supports a function to find contours. We iterate through all the found contours, and return the bounding rectangle for the one with largest area. From this rectangle, the points are easily found, and we have the output for this step completed. The interface can be seen in the figure 4.2. All the prompts for the automatic stage are questions to check whether the detection went smoothly. If the user confirms that everything is correct, the algorithm moves on to the next stage, otherwise the application switches to the manual version of the current step.

4.1.2 Manual Color Checker Detection

The manual implementation of the color checker detection is not complex, since the only inputs the user needs to provide is the four corners. This is done via touch input, in which the user picks in order the four corners. After four points have been selected, the user can grab each one of the four points, and drag the position until all four points align with the corners. Although this technique worked well when testing with an emulator, when using an actual device, it was clear that dragging the point to where the user was touching was not practical, since the precise location was being occluded by the finger. For this reason we altered slightly the input scheme, and when the user clicks anywhere on the screen, the nearest point will be selected, and the difference from the current input is used to alter the point's position. This allows for a much higher degree of precision, while simultaneously increasing the user comfort when interacting in this mode. The

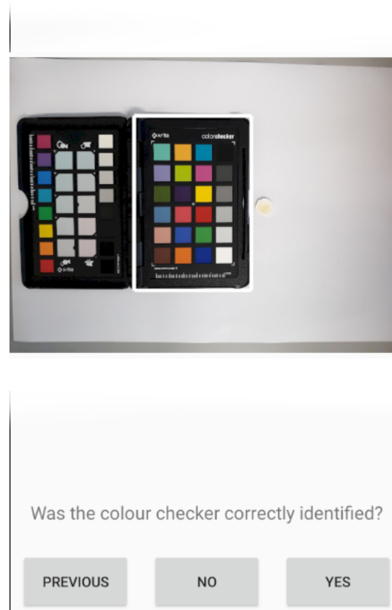

Figure 4.2: Automatic Color Checker Detection.

interface of this step can be seen in figure 4.3. The green handles represent the ways the user can interact with the selection, and these remain consistent throughout the entire application.

Since the crop for the next stage requires the points to be ordered, instead of forcing the user

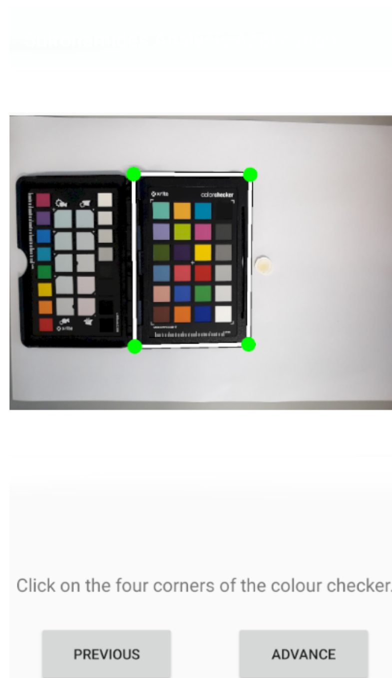

Figure 4.3: Manual Color Checker Detection.

to comply to a certain order to the selection, we give total freedom, and later sort the points to conform to the order the input required in the next step. For this we created a helper function that will be reused throughout the project whenever an array of points is generated so that the all points are ordered the same way, no matter the step nor type of interaction. To further assist the user, even if the points are selected rather loosely, without closely adhering to the color checker shape, the following stages are resilient enough to handle high levels of warping, as we will see later.

Another caveat that needs to be addressed is the fact the points that the user has selected in the device space, meaning that the points that appear on the screen will not match the points when used on the image. To work around this, we created a function that translates the points from any device to the image that is represented on the screen, regardless of smartphone, screen orientation or image characteristics. This function is reused for all the functions that require user input as well.

After translating and sorting the points, the output for this step is in the correct format for the color patches detection.

4.2 Color Patches Identification

In this step, the points from the color checker are used to perform a crop, and the cropped image is the input that both the automatic and manual process require. Since the crop may not necessarily be a perfect rectangular shape, warping is done to make sure that the entire inside crop results in a new image with a fixed width and height. The latter values are calculated by measuring the distance between the top left and top right corners, and top left and bottom left corners respectively.

4.2.1 Automatic Color Patches Identification

After cropping the image, we need to identify all the patches inside. An Otsu threshold is applied to mask out the color checker itself, but it also masks some patches due to their low values. Despite this, it will have picked the necessary amount to allow us to mathematically interpolate the rest.

Before discovering the hidden patches we first store the ones we did find after performing the threshold. For this, we check to see if the width or height of each contour is of an appropriate size taking into consideration the overall size of the cropped image. If they fall within acceptable range, we save that contour. After that we need to check the real position in the grid of patches for each found patch. In the original *MATLAB* code, this was done using complex array manipulation similar to what Python allows. Since this was not an option for Java, we devised a new way of performing this operation. Firstly, we create an array filled with empty rectangles representing the grid of patches, then we iterate through each patch we discovered in the previous stage, and check what would we be their appropriate position in the grid in terms of X and Y positions. After that, the patch is added to the array of rectangles in the correct position. After completing this process, we now have a more complete grid with a few known elements. The empty rectangles are interpolated based on average X value per column and average Y value per row. A finished result can be seen in figure 4.4.

Figure 4.4: Automatic Patches Detection.

With a grid of rectangles, we now extract the mean value for each rectangle, which differs from the original algorithm that uses the median. This is done since our custom median implementation we will explain in the sample subsection requires iterating through every value in a matrix, and the matrices in this step are too large to do that efficiently. After obtaining the mean we then store the colors in a list, which is the necessary output for the next step.

As we stated before, this stage can handle some, though not too much warping, which can happen if the previous stage was manually and loosely done. To correct this, a new selection could be done in the previous step, or the user could manually intervene, as we will see in the next subsection.

4.2.2 Manual Color Patches Identification

If the automatic detection fails, and the user wants to manually intervene, the user interface will change, showing to the user a grid of rectangles with a bounding green rectangle. The user then drags the nearest selected point until the grid aligns well with the patches grid. When that happens, the user can select advance, and the application will calculate the averages of each grid rectangle, and store them in the required list of colors. The method of interacting with the corner handles is similar to the previous step, to maintain consistency in user input while also providing a forgiving experience, with a precise alignment being easy to perform.

After selecting the grid, when the user attempts to advance to the next stage, some further processing is done. Firstly, the four corners are translated to the image. Next, unlike the automatic version, no complex interpolation is required, since the user selected the location of all the patches.

Figure 4.5: Manual Patches Detection.

We used the position of each patch, and obtained the mean value for each patch. We stored the results in the list of colors that will be used later for the color correction matrix.

An example of the finished user interface design can be seen in figure 4.5.

One advantage of this method, is that even in very extreme warping cases, the user can still correctly get a decent result, since the user can still alter the position of off-screen handles. A compilation of several cases can be seen in figure 4.6.

4.3 Disk Detection

One of the requirements of the original *MATLAB* code is that the disk containing the sample be placed right besides the color checker. With this assumption in mind, the algorithm takes into account the bounding box of the color checker detection, and crops the region right next to it.

Figure 4.6: Warping with Manual Patches Detection.

This image crop is the basis for the disk detection step, as it is inside this new image where the algorithm will attempt to discover the disk.

4.3.1 Automatic Disk Detection

After cropping the image to the correct region, the first step is to perform a Canny edge detection to discover all the possible contours within the crop. The original code performs this edge detection in all three channels, and then combines the results of each Canny into a single image. To allow this behavior, a new function was created to extract individual channels from an array, as well as creating a new function to later merge them back into another image. The threshold values used for the Canny edge detection are the same as in the original *MATLAB* code, which obtained good results at identifying the correct regions. The resulting contours are simple lines, which may not be connected. To discover the edge surrounding the disk, it is first necessary to dilate the image, expanding the edges and thus connecting shapes. After multiple dilations, the main shape is assumed to be connected. However, the shape is now too large, but before eroding away the shape, it's important first to make sure that newly connected shapes do not lose their connection, so we first fill the image's holes utilizing the function that we previously mentioned in the color checker detection. After filling the empty spaces, we can safely erode, to get a closer approximation of the actual shape of the disk.

After all the preprocessing of edges, the original *MATLAB* code chose the contour with the largest area, which worked well since the process was fully automated. However, when testing it was revealed that if the user manually selected the color checker contours, they normally did not find the exact edges of the object, selecting only an inner region. This imprecision is harmless in that stage, but the first crop in this stage is dependent on the positioning found for the color checker. Since the color checker frame generates such a high contrast with the background, and is bigger than the disk, the algorithm usually selected the object's contour. To work around this limitation, extra heuristics were created to determine more accurately what is the correct contour. This included checking whether the bounds of the contour obeyed a specific set of restrictions, which resulted in a more stable algorithm, without requiring extra care from the user in previous stages.

With the correct contour identified, a bounding box was created, outlining the position of the disk in the image. This set of points is the output of the disk detection, that will feed into the final stage of the algorithm. An example of the interface can be seen in figure 4.7.

4.3.2 Manual Disk Detection

The manual implementation uses the same image as the automatic version, but the user interaction is slightly different compared to other steps. After testing, precisely selecting four corners in such a small portion of the image was too much to be done in a comfortable way. On top of that, the output for this stage was a perfect rectangle, so it was much simpler to finely control just two diagonally opposed points.

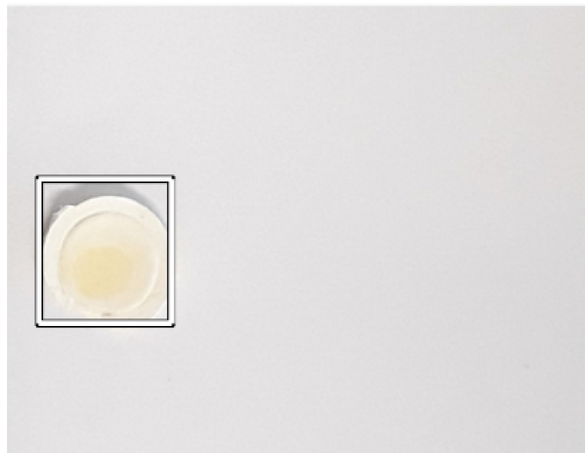

Figure 4.7: Automatic Disk Detection.

After multiple iterations, we ended with the user first drag selecting the overall bounding box. If the selected region is not perfect, the user can then fine-tune the selection by dragging either the top-left or bottom-right handle. As before, the user does not need to touch the point itself, since the closest point will be automatically selected, and only the motion is used to change the point's position. When performing this, the rectangle will be resized, without warping the shape. When tested, this proved to be an efficient solution, capable of delivering fast and precise results.

The selected bounding box will the input to the final stage of the algorithm, and an example of the finished user interface can be seen in figure 4.8.

4.4 Sample Detection

For the final stage of the analysis process, the algorithm will use the color patches discovered in the second stage and the bounding box found in the previous stage to crop a new image. This will

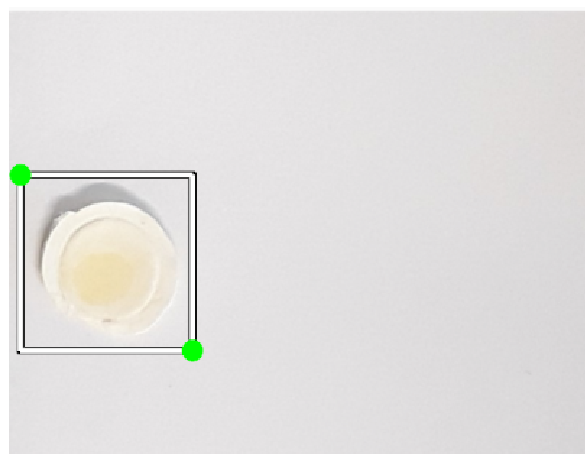

Figure 4.8: Manual Disk Detection.

be the starter input for only the automatic version, as the manual mode requires a special input that will be explained in that subsection.

4.4.1 Automatic Sample Detection

After cropping, the newly created image contains the disk, with the sample being a smaller region inside. To discover the latter area, we do the same steps as the original *MATLAB* code, we extract the blue channel of the image, and perform a simple Otsu threshold on that channel. We then identify the largest contour available, and create a new temporary image that is cropped to that contour.

To showcase the results, we needed to finally correct the colors of the image, and store it in a new variable. This is what will be used to show the results. Unfortunately, the color correction process was not straightforward. Despite the original algorithm taking only about 20 lines of code, one key assumption differs from *MATLAB* to *OpenCV*. In the original algorithm, a color matrix is converted from one color space to the XYZ color space using the function `rgb2xyz`. It would be straightforward to assume that the original color space was RGB, when it is in fact sRGB. In *OpenCV*, the RGB to XYZ conversion is done as the name suggests, and no sRGB variant exists. Due to this, we had to create our own conversion function that was able to convert from sRGB to XYZ and back. Despite not being complex operations by themselves since they are just matrices multiplications [21], there was one major caveat. The XYZ to sRGB operation was different depending on the values of the individual pixel. As we mentioned before, iterating through every value of a larger matrix is prohibitively expensive. For this reason, we needed to come up with a new solution, that still respected the conversion properties while also prioritizing efficiency. In the XYZ to sRGB conversion process, the operation changed whether the pixel channel value was higher than a certain threshold. This provided the necessary clue to solving the puzzle, as two images were created through standard binary thresholds, one for the values above the threshold, and the other for the values below. We then applied the different calculations to each image, and then combined the result taking into consideration the masks. The result of the color correction can be seen in figure 4.9.

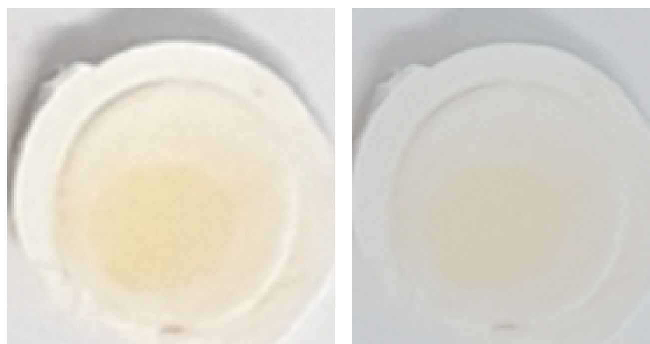

Figure 4.9: Color corrected image.

After correcting the colors, we intended to diverge from the original algorithm, as it further cropped the image to center the selection even more. This lead to the selection not fitting the shape of the sample, as it selected a smaller subset of pixels. Since we wanted to show clearly to the user the contour that was used to extract the color value, initially we did not crop the image further. In initial tests, the results were promising, as the full contour was being correctly identified. However, the results were only positive when there was a high sulfonamides concentration, which increased the contrast between the sample and the disk. When testing with a sample without any antibiotics presence, the contrast was so low that the threshold was unable to detect the sample region correctly. For this reason, we ultimately had to opt for the same workaround as Carvalho et al. [5], and crop the image further. This presented a larger problem related to the presentation of the results. Since the original algorithm was never meant to require any user input, the data did not need to be presented in a comprehensible format. This was not the case for this application, and if a clear contour was not presented, the user might be left confused as to what was actually used for the value. Initially we simply showed a final masked image, with black and color values being the colors that were used for obtain the median. This proved to be too unclear, as the user could not tell what part of the picture was used for the crop, and from where did those colors come from. We ended up with a solution that does not show the user any contour information, as it would only cause confusion, but instead shows the color value that was extracted, side-by-side with the original corrected image. If the two values are extremely similar, then the user knows the value that was selected is right, and if not, direct user intervention is necessary. Should the color be well selected, this is the last output required before presenting the results.

The representation of the final screen can be seen in figure 4.10.

Despite the presentation side being handled, the algorithm was not yet complete, as we still

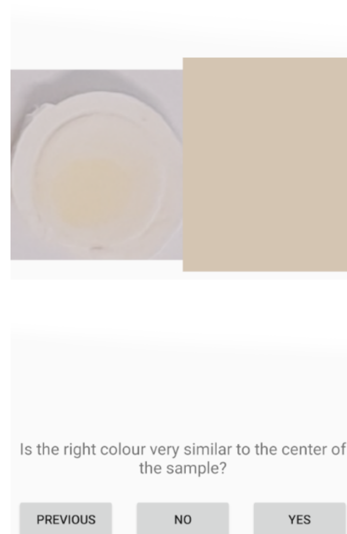

Figure 4.10: Automatic Sample Detection.

need to extract the color value. Continuing the algorithm with the more aggressively cropped image, we began to identify the pixels that might belong the reaction and not the disk. For this, we followed the original code by Carvalho et al. [5] and performed two steps to spread the values of the image. Firstly, we identified the minimum and maximum values in the image, and shifted the minimum values to 0, and the maximum values to 1. This spreads the values evenly throughout the entire range. After that, a histogram equalization was done to enhance the contrast.

After altering the contrast of the image, we performed the custom multi-level Otsu threshold we developed for the color checker identification to extract a threshold containing pixels inside the sample region. This threshold, after inversion, is the region of the image that contains the sample pixels, and will be multiplied with the color corrected image.

To extract the concentration result, we had to calculate the median of the values in the mask. Unfortunately, the *OpenCV* build we used did not natively contain this function, so we had to create a custom median function that returned the center value after ordering, while discarding all the values in the array that were pure black, as these represented the mask. With the median value obtained, we have the input for the final part of algorithm.

4.4.2 Manual Sample Detection

Unlike the other manual interventions, this step requires an input obtained directly from the automatic version. It would be unnecessary, more taxing and time-consuming to color correct the cropped image again, so the image that was created in the automatic version is used. This is what the user will see on the left side of the screen as the base image, similarly to before. On the right screen, there will be another panel, showing the currently found color. A new crosshair will appear on the center of the left side, which will be the user's way of input. By dragging around the screen, the crosshair will move according to the motion, and the new color will be the average of a small square around the currently selected point. The user will then hover the crosshair to the center of the sample, and advance. An image of the activity is seen in figure 4.11.

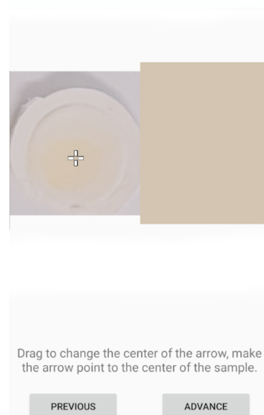

Figure 4.11: Manual Sample Detection.

4.4.3 Extracting the Result

Although the algorithm itself is complete, we still did not extrapolate the concentration value from the metrics we obtained. For this, Carvalho et al. [5] determined that the color property that could be most accurately used to extract the concentration was the Hue. Similarly to before, we had to develop our own function that could obtain the Hue value from an sRGB color.

After this, we needed to use an equation that mapped out the Hue values value to a concentration. This presented a problem, as each phone had a different curve that more accurately fit the tests. This meant that we could potentially get worse results if we used different phones to capture the images. For this dissertation, we created curves tailored for only one phone, since validating the results of different photos captured with various phones was outside of the scope of this dissertation.

Figure 4.12 shows an image of the final activity screen containing the measured result that was achieved by inserting the extracted median value into the created curve.

In this chapter we talked about the automatic and manual implementation of every stage of the algorithm. We will address next the performance optimization pass made to ensure that even low-end phones could get approximate results, without sacrificing too much time or responsiveness.

Figure 4.12: Results Screen.

Chapter 5

Performance

From the beginning of the development, performance was a clear priority, as our goal was to have the application running smoothly without requiring too much time from the user to obtain a result. Another nice-to-have was the possibility of having a quality slider to have some manual control if the maximum precision mode is too taxing. In this chapter, we will address what limitations we had to overcome, and compromises that had to be established to ensure a great user experience.

5.1 Memory Constraints

To stress test the performance of this application, we tested on a two phones with very different specifications, the weakest of them being a Motorola G3 of 2015 with 1GB of *RAM* memory. In 2017, two of the most common *RAM* configurations were 1 and 2GB , and since then this number keeps on increasing, with some phones even going to 12GB of *RAM*. For this dissertation, we believed that 1GB was the correct lowest possible target. With this small budget, we set out to optimize the application.

The major limiting factor when it comes to memory management was the size of the images. The pictures that were taken that we used for testing had a very high resolution, of up to 4128 by 3096 pixels, which needed to be stored in matrices that contained all three channels. This means that in a worst case scenario, a single matrix containing an image had to store over 38 million individual elements. On top of that, we frequently had to use multiple temporary matrices after applying some sort of processing. Despite *OpenCV* managing to do surprisingly well on Android, constant care and attention needed to be had, or otherwise crashes due to out of memory exceptions were soon to follow.

To mitigate some of the issues of large matrices, we employed a strategy of aggressive memory flushing, where we would manually release each matrix as soon as it stopped being necessary. This allowed us to not only keep memory leaks in check since *OpenCV for Android* is a Java abstraction for C++ code, but also to decrease the memory load to the necessary minimum at all times. However, even these measures were not enough on their own, as we still had occasional crashing, so more redesign had to be done.

5.1.1 Algorithm Optimizations

In the original *MATLAB* code, the color correction stage occurs right after detecting the color checker patches. In this step, the entire image is color corrected. However, this matrix was not only as big as the entire image, it could not be flushed until the very end of the algorithm, as this was the image that was cropped after the disk outlines are discovered. This increased massively the amount of required memory, without even taking into account that the process of correcting the colors was very taxing when large matrices were involved. To work around this, we changed the moment in which this stage happens, and only used the original image for all the detections. Only when cropping the original image to fit the disk do we decide to color correct this image. This resulted in a huge memory decrease, from around 1GB to under 400MB. This optimization alone made this application run on lower end devices, and in much better times, as this is the part of the algorithm that had the highest cost in memory and time. This high impact is due to the need of using the custom XYZ to sRGB function that, despite being better than iterating pixel by pixel, was still very demanding, and did not scale well with higher resolution images. Despite this optimization, there was still more room for improvement.

The second most demanding stage when it comes to memory consumption is the color checker identification. In this step, there are two matrices that encompass the entirety of the image, namely the matrix that stores the baseline image, and a processing matrix on which we perform the threshold operations and other processing. After testing, it was clear that the high contrast of the color checker with the contrast made it easily discoverable when using the custom multi-level Otsu threshold. A hypothesis arose that the thresholding matrix did not need to be the size of the entire image, but instead could be a scaled down version. This presented a problem, because a new, scaled down image needed to be stored as well. However, the extra memory allocated to that image still justifies the optimization, as the operations done to matrices require more memory when dealing with larger resolutions. To make this tweak work, the points discovered on the scaled matrix needed to be rescaled back according to scaling factor used. This tweak also had some processing time improvements, as we will mention in the next section's tests.

After performing all the mentioned optimizations, another pass at testing for memory leaks was required, as profiling the application revealed that the memory allocated to native C++ code did not fully reset if the user decides to abort the process mid-analysis. However, after further testing, it appears that the base memory usage fluctuates between 28 and 40 MB, and after using the application for a while, the base memory when idling is consistent, so no memory leaks were ultimately identified.

5.2 Bitmap Size Constraints

Throughout development, most of the tests were done in an emulator out of commodity, which despite not being an accurate measurement for raw performance, still gave a good indication of relative performance across stages. However, one aspect in which it could not replace live testing

was a bitmap drawing. On older hardware with more limited *GPU*s, large bitmaps such as the ones used for testing could not be drawn at all, due to them exceeding maximum *OpenGL* texture sizes. To circumvent this, we needed to develop an algorithm that could create a scaled bitmap based on the properties of the device it was running on. This fixed the issue, and presented an interesting avenue for improvements.

5.2.1 Image Resolution Optimizations

One of the optional requirements presented by the product owners was the ability to manually select the precision, should the user require a faster, but less precise version, either due to time or hardware constraints. Tweaking the algorithm itself was tricky, as alterations might bear unexpected side effects, decreasing consistency in results and lowering user trust. However, the impact image resolution had extended beyond memory, as calculations become much quicker as the size decreased.

We already mentioned that in the color checker stage we opted with using a scaled version for the thresholds. This was reworked to make this parameter be customizable by the user in the settings menu. This activity also included another slider for the overall quality of the image. Decreasing this value decreases the resolution of the base image for the calculations, and can improve times substantially. However, this decrease in resolution can negatively impact other areas of the algorithm, most notably the disk detection. In this stage, the Canny edge detection can be affected, since some finer lines might become less clear. Changing the resolution also required some tweaks in the algorithm, as the values used in the dilation and subsequent erosion of edges needed to scale according to the quality slider setting.

Regarding the color checker precision slider option, we capped the maximum resolution to half the original image's size, as this option offered excellent results, even with a lower overall quality setting selected. We took this decision since the user could think that using full precision mode might be necessary, when in fact half resolution is more than adequate, with great memory and performance improvements. What matters by the end of the fist stage is whether the corners are correctly identified, so if the algorithm takes 2 seconds or half a second, as long as those points are correct, precision itself does not matter.

In this chapter we approached all the aspects regarding performance and optimizations that we had to do to get the responsiveness to the level we wanted. We will explore how we tested the application in the next chapter.

Chapter 6

Tests

Throughout development, we kept on iterating on the algorithm, matching it's stability to the original *MATLAB* version, and improving it's performance. After finishing the application, it was critical to check if these two aspects were to the standard of quality we intended, so we performed extensive testing. In this chapter we tested three different topics:

1. The precision of the application's results;

2. The divergence in output from using the manual method;

3. The performance metrics using live phones.

For the first two components, all tests were run on an emulator, as time was not a factor to be taken into consideration. In the performance section, we will perform a deep-dive into every facet of the efficiency aspect.

6.1 Precision Comparisons

For the precision aspect of the application, we only used one run for each test, as there are no elements of randomness in these purely automatic tests. With the exact same image and settings, we will always get the exact same result, regardless of device. For each level of concentration we used the average of four different photos, both for *MATLAB* as well as the application. For the comparisons we will use the median Hue value obtained, since that removes from the equation imprecisions associated with the calculation of the concentration. The main purpose of this test is to check the differences between the final results of the *MATLAB* code and the newly developed application, not the precision of the concentration value itself. In table 6.1 we compared the two values in terms of raw Hue difference, using the 24 patch set for both methods. The *Conc.* column refers to the concentration of sulfonamides in the sample, the Mob. column to the results obtained in the application, and the PC results are extracted from the original *MATLAB* version of the algorithm.

Table 6.1: Hue differences between the application and *MATLAB*.

Conc. (μg/L)	**PC Avg.**	**PC σ**	**Mob.Avg.**	**Mob. σ**	**(PC Avg.-Mob. Avg.)/Avg. PC (%)**
0	0.5908	0.008	0.6028	0.004	2%
5	0.5740	0.009	0.5833	0.010	1.6%
10	0.5379	0.006	0.5462	0.024	1.5%
15	0.5221	0.011	0.5217	0.010	>-0.1%
20	0.5085	0.017	0.5079	0.015	-0.1%
25	0.5065	0.009	0.5062	0.012	>-0.1%
40	0.4541	0.002	0.458	0.005	0.9%
50	0.4426	0.004	0.44525	0.009	0.6%
100	0.4202	>0.001	0.416725	>0.001	-0.8%
150	0.4117	0.002	0.40795	0.002	-0.9%

For the comparison, we use a weighted difference, to demonstrate that even in low values the results are still very comparable. We registered a maximum difference of about 2%, which is well within the accepted ranges for precision. This validates that the results obtained using the application are comparable to running the *MATLAB* version of the algorithm.

For the second precision test, we compared the automatic results to the real concentration value. For this we used the average of 4 runs for each concentration value, using the 13 patch set. This was done as the most recent revisions of the work done by Carvalho et al. [5] concluded that tests were more accurate using that specific set of patches. If the user uses a custom set of patches, it defaults to using the 13 patch set curve. The final precision results using the generated curves are in table 6.2.

The precision of the results themselves vary significantly, as the curve that is used can only provide an approximation. However, despite the occasional stray of individual estimations, overall it remains close enough to get an approximation of the actual concentration in the sample. It's important to note that improvements to this approximation fall outside of the scope of the dissertation, as it's main challenge was the port of the original *MATLAB* algorithm to a mobile Android smartphone.

Table 6.2: Concentration precision tests for 24 and 13 patches.

Concentration (μg/L)	**Avg (μg/L)**	σ
0	3.16	0.22
5	4.56	1.22
10	8.86	3.43
15	14.34	2.56
20	17.99	4.11
25	19.06	4.16
40	42.28	4.65
50	55.32	6.29
100	97.03	3.92
150	123.25	2.06

6.2 Automatic vs Manual Methods Comparisons

In these tests we wanted to verify whether there is a discrepancy from the results that are obtained manually from the ones performed automatically. This helps ensure that manually performing the entire operation will not cause worry that the final estimated concentration value would stray far from the automatic approach. Since this test involves a certain degree of randomness tied to the manual input, the average of 5 manual runs is used for each concentration, and the results refer to the measured Hue. All tests used the 13 patch set, and all the manual runs include all the stages done manually. The results are found in table 6.3.

Similarly to the first tests, we used a weighted difference to measure the deviation between the manual and automatic results, divided by the latter. This once again shows it's impact more dramatically on values that are lower than 1, which is the case, and even still, the maximum recorded difference was 2.8%. The standard deviation is also well within acceptable ranges. To make matters even better, this difference is mostly due to the square of pixels in the sample that were selected in the final stage differing from the automatic approach. If the user only intervenes in the bounding box detection, such as the color checker, patches and disk identification, there is a chance that there is no difference at all between the two versions. Because of all of these factors, this test proves that the manual version is a reliable alternative that, if used carefully, can be just as good as the complex *OpenCV* algorithm.

6.3 Mobile Performance Metrics

For performance we used real phones with the latest version of the application. For the tests we wanted to compare the time differences between a mid-range phone (which is one of the most common device ranges), to a relatively low-end phone by the current standards. The two phones we used were:

- Xiaomi Mi A1: The mid-range smartphone, featuring a Qualcomm MSM8953 Snapdragon 625 for the chipset, running an Octa-core 2.0 GHz Cortex-A53 CPU, an Adreno 506 GPU,

Table 6.3: Difference between automatic and manual version.

Concentration (μg/L)	**Auto.**	**Manual**	**σ Manual**	**(Auto.-Manual)/Auto. (%)**
0	0.6306	0.6219	0.0033	-1.3&
5	0.5992	0.5822	0.0014	-2.8%
10	0.5502	0.5544	0.0052	0.7%
15	0.5132	0.5143	0.0007	0.2%
20	0.5	0.5	0	0
25	0.4961	0.4941	0.0009	-0.4%
40	0.4637	0.4592	0.0019	>-0.1%
50	0.4402	0.4416	0.0012	0.3%
100	0.4191	0.4243	0.0009	0.1%
150	0.407	0.4082	0.0002	0.3%

Table 6.4: Average time for each stage.

	Checker (ms)		Patches (ms)		Disk (ms)		Sample (ms)	
	Avg	σ	**Avg**	σ	**Avg**	σ	**Avg**	σ
Mi A1	1306.2	24.98	957.4	14.74	886.6	11.39	758.8	15.18
Moto G	2435.4	293.2	1691	208.7	1691	39.58	2148	186.1

and 4GB of RAM memory [1].

- Motorola Moto G (2015): Although not unusable, very low-end compared to modern mid-range phones. Features a Qualcomm MSM8916 Snapdragon 410 chipset, a Quad-core 1.4 GHz CPU, an Adreno 306 GPU with the 1GB RAM variant [2].

To have a better understanding of the performance of the application, we did 4 types of tests evaluating different aspects of the application, namely:

1. Algorithm Stage comparison: Assessing which stages of the algorithm execution take the longest;
2. Overall Quality Comparisons: Measuring the differences in time and final result when tweaking the overall quality slider in the Settings activity;
3. Color Checker Finding Precision Comparisons: Similar to the previous test, but this time comparing the color checker finding precision option;
4. Optimized Results: Tweaks to the options in the Settings activity that yield the best performance without sacrificing too much accuracy in the results.

6.3.1 Algorithm Stage Comparisons

Despite solely using the automatic version of the algorithm, there is an inherent randomness to the timings, as loads vary throughout execution of the application. For this reason, we ran each test 5 times and then averaged out the results.

For the first test, we measured the time each stage took to complete, to check which step is the most taxing to responsiveness, using an average of 5 runs on both smartphones, and the results are in table 6.4. This was done to estimate the relative time cost between each stage, and how evenly spread it is throughout an entire run.

The results indicate that the most taxing part of the process is the color checker, and this can be easily explained, since that is the stage that uses the largest matrices in the entire process. As the stages progress, the processing of the respective image may be more taxing, but the size of matrices is lower since they all deal with cropped versions of the base image. It's also important

[1] Xiaomi Mi A1 - Full Specifications, GSM Arena, https://www.gsmarena.com/xiaomi_mi_a1_(mi_5x)-8776.php

[2] Motorola Moto G (3rd Gen) - Full Specifications, GSM Arena, www.gsmarena.com/motorola_moto_g_(3rd_gen)-7247.php

to note that the concentration of the sample has no impact on time, as it is always the same code that is run. Regarding the standard deviation increase on the low-end phone, this is explained by the fact that the first run might need some additional time to create activities and bitmaps, as this run was the worst in almost all stages, and in some cases by a large margin (600 ms).

6.3.2 Overall Quality Slider Comparisons

For the second and third tests, we wanted to measure the impact that changing precision had on time and results, for both the overall quality and the color checker detection. As usage always showed a linear decrease in time, we performed only one run for each configuration using the mid-range smartphone.

For the overall quality test, we used maximum color checker finding precision, with the comparisons being in table 6.5. As resolution of the image decreases, the performance improves massively, but the results start to vary, albeit mildly, since the hue is altered by just 1%, but achieving the outcome nearly 10 times faster.

It might be hard to believe that the times are so much better, while achieving almost the same exact results. This happens due to two reasons:

1. Decreasing resolution improves performance significantly, but it might compromise the detection of certain elements. For example, if the image loses too much resolution, the disk borders might be impossible to threshold, because that information is lost in the downscale. In this example, all the detections worked correctly, despite the reduced amount of information;

2. Even if the thresholds work correctly, the sample pixels are altered, so the extracted Hue should be different in theory. This does not happen because we use the median of the found pixels, and not the average. This implies that even if a large number of pixels is changed, what matters is the pixel exactly in the center of the ordered array. If there are many pixels in the center region with the same value, that increases the chance of the same value being selected. For these two reasons, it's common to see exactly the same result, even while altering the quality of the base image.

Table 6.5: Impact of overall quality slider.

Quality (%)	Time (ms)	Hue
100%	3985	0.602
80%	2566	0.6
60%	1372	0.6
40%	785	0.596
20%	571	0.593
10%	467	0.593

Table 6.6: Impact of color checker precision.

Color Checker Precision (%)	Time (ms)	Hue
100%	1250	0.602
80%	860	0.602
60%	662	0.6
40%	554	0.6
20%	491	0.602
10%	462	0.602

6.3.3 Color Checker Finding Precision Slider Comparisons

For the color checker precision test, we used the maximum overall quality, and only measured the time of the first stage, as this is the only part that is altered in the entire algorithm. The results are seen in table 6.6.

The small deviation in results is explained by the way the color is obtained from each of the patches. We use an average of all the colors inside the identified patches, and small differences in the detection of corners might alter very slightly the position of the detected patches. This can include colors that end up offsetting the average, resulting in a different color correction matrix. Despite this, the results are very consistent overall, with the required processing time diminishing considerably as the resolution of the color checker matrix goes down.

One aspect to keep in mind is that the overall quality slider and color checker resolution precision settings are somewhat coupled. If the overall quality of the image decreases, the base image loses resolution, and so the color checker stage will go faster as a result. If the color checker precision is also set, these two settings are multiplied, and it's important to keep in note that low quality and low precision may lead to a potentially faulty color checker detection. This can be easily solved either by using manual mode, or even sometimes continuing regardless, as most likely the patches identification step will still function properly.

6.3.4 Optimized Results

For the following test we wanted to see how well the algorithm performed if we tweaked a few settings based on the power of the hardware. This gave us our "best case scenario" when dealing with lower-end phones. The results are in table 6.7.

Table 6.7: Tweaked settings impact.

	Time (ms)	Hue
Mi A1 Baseline	3909	.600
Mi A1 Optimized (0.8 Overall Quality + 0.4 Color Checker Precision)	1720	.600
Moto G Baseline	8037	.600
Moto G Optimized (0.5 Overall Quality + 0.3 Color Checker Precision)	1995	.600

Table 6.8: Comparison with low-end smartphone and desktop PC.

	Time (ms)	**Hue**
Moto G with Optimized Settings	1995	0.600
MATLAB Code	4204	0.594

Looking at the results, it becomes apparent that, even though the baseline time is more than two times larger on the low-end phone compared to the mid-range phone, it can still provide fast results, and in this case without losing any precision. This might not happen in all cases, as thresholds may not be as accurate, but it is still viable for images that follow a similar structure to the example image in figure 3.3.

Our target for performance was high, as we wanted to get results from a low-end to be faster than the *MATLAB* code running on a desktop. For the comparison, we used the tweaked settings for the Motorola Moto G and ran them against a PC equipped with an Intel i7-4790k running at 4.5 GHz paired with an AMD RX 480 at stock frequencies running the original algorithm. It's important to note that the *MATLAB* results could still be improved, since the priority of the work done by Carvalho et al. [5] was precision, not performance.

The results presented in table 6.8 speak for themselves, with only a .006 difference in hue in half the time. With our ambitious performance target reached, and comparable precision to the original algorithm, the development of the application was complete.

The final stage of tests was going to be done live, using the application installed on a smartphone on the laboratory where we could control the concentration of the samples. Unfortunately, due to the pandemic that occurred during the time of writing the dissertation, we were not able to collect that information, as the investigator that helped us throughout development did not return return to the laboratory.

In this chapter we approached all the testing we did to measure stability, performance and precision compared to the original algorithm. All that remains is the conclusions for the dissertation and future work.

Chapter 7

Conclusions and Future Work

Sulfonamides are an important antibiotic group to combat bacteria. It is low cost and low toxicity, which lead to it's widespread usage. This usage has lead to an increase in the presence of antibiotics in water compartments, which proliferates bacteria capable of resisting antibiotics. The impact of these bacteria is becoming a global threat, and analysis needs to be performed on waters to assess their level of contamination.

Current methods for determining sulfonamides concentration are expensive and impractical. They require resources and a laboratory setup. Colorimetry appears to be an interesting way forward, increasing practicality, but so far is still not fully mobile. Smartphones have already been used in the field for digital image colorimetry, with positive results. This dissertation further proved the viability of smartphones for colorimetry tasks, providing adequate estimations of concentration of sulfonamides in a sample.

Some of the limitations of this dissertation are somewhat similar to those found in the work by Carvalho et al. [5], namely:

1. Despite being limited after color correcting the image, lighting variations can still play a big role in the final concentration estimation;

2. Different phones may capture different photographs, with some camera applications processing the image irregularly to achieve more realistic or pleasing results, such as when using HDR. This presents a problem, since color correcting the image will be of reduced efficiency, if not worthless. This could be mitigated by either creating an entire new custom camera application that followed a specific set of parameters, or by creating a new camera view within this application that handled that part. That way we could limit some features to ensure that the result is captured raw, without extra image processing.

References

[1] Emmanuel Agu, Peder Pedersen, Diane Strong, Bengisu Tulu, Qian He, Lei Wang, and Yejin Li. The smartphone as a medical device: Assessing enablers, benefits and challenges. *2013 IEEE International Conference on Sensing, Communications and Networking (SECON)*, 2013.

[2] Aymen Alian and Kirk Shelley. *Photoplethysmography: Analysis of the Pulse Oximeter Waveform*, pages 165–178. 01 2014.

[3] Sevcan Aydin, Bahar Ince, and Orhan Ince. Assessment of anaerobic bacterial diversity and its effects on anaerobic system stability and the occurrence of antibiotic resistance genes. *Bioresource Technology*, 207:332 – 338, 2016.

[4] Simone Bianco, Claudio Cusano, and Raimondo Schettini. Color constancy using cnns. *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2015.

[5] Pedro H. Carvalho, Sílvia Bessa, Ana Rosa M. Silva, Patrícia S. Peixoto, Marcela A. Segundo, and Hélder P. Oliveira. Estimation of sulfonamides concentration in water based on digital colourimetry. In Aythami Morales, Julian Fierrez, José Salvador Sánchez, and Bernardete Ribeiro, editors, *Pattern Recognition and Image Analysis*, pages 355–366, Cham, 2019. Springer International Publishing.

[6] Barbara Chiavarino, Maria [Elisa Crestoni], Annito [Di Marzio], and Simonetta Fornarini. Determination of sulfonamide antibiotics by gas chromatography coupled with atomic emission detection. *Journal of Chromatography B: Biomedical Sciences and Applications*, 706(2):269 – 277, 1998.

[7] Erin E. Connor. Sulfonamide antibiotics. *Primary Care Update for OB/GYNS*, 5(1):32 – 35, 1998.

[8] D. Dang, C. H. Cho, D. Kim, O. S. Kwon, and J. W. Chong. Efficient color correction method for smartphone camera-based health monitoring application. In *2017 39th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 799–802, 2017.

[9] G. D. Finlayson, M. Mackiewicz, and A. Hurlbert. Color correction using root-polynomial regression. *IEEE Transactions on Image Processing*, 24(5):1460–1470, May 2015.

[10] David H. Foster. Color constancy. *Vision Research*, 51(7):674 – 700, 2011. Vision Research 50th Anniversary Issue: Part 1.

[11] Malgorzata Gbylik-Sikorska, Andrzej Posyniak, Tomasz Sniegocki, and Jan Zmudzki. Liquid chromatography–tandem mass spectrometry multiclass method for the determination of

antibiotics residues in water samples from water supply systems in food-producing animal farms. *Chemosphere*, 119:8 – 15, 2015.

[12] Arjan Gijsenij, T. Gevers, and Joost Weijer. Improving color constancy by photometric edge weighting. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34:1 – 1, 05 2012.

[13] Simon Hill. A complete history of the camera phone. https://www.digitaltrends.com/mobile/camera-phone-history/, Aug 2013.

[14] Rodrigo Hoff and Tarso Kist. Analysis of sulfonamides by capillary electrophoresis. *Journal of separation science*, 32:854–66, 02 2009.

[15] Arne Holst. Number of smartphone users in the u.s. 2010-2024. https://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/, Aug 2019.

[16] Sam Hostettler. Many antibiotics overprescribed for outpatients. https://today.uic.edu/many-antibiotics-overprescribed-for-outpatients, Oct 2016.

[17] Karel Hruska and Milan Fránek. Sulfonamides in the environment: A review and a case report. *Veterinarni Medicina*, 57, 01 2012.

[18] Purim Jarujamrus, Rattapol Meelapsom, Somkid Pencharee, Apinya Obma, Maliwan Amatatongchai, Nadh Ditcharoen, Sanoe Chairam, and Suparb Tamuang. Use of a smartphone as a colorimetric analyzer in paper-based devices for sensitive and selective determination of mercury in water samples. *Analytical Sciences*, 34(1):75–81, 2018.

[19] Teemu KorpimÃ¤ki, Eeva-Christine Brockmann, Outi Kuronen, Maija Saraste, Urpo LamminmÃ¤ki, and Mika Tuomola. Engineering of a broad specificity antibody for simultaneous detection of 13 sulfonamides at the maximum residue level. *Journal of Agricultural and Food Chemistry*, 52(1):40–47, 2004. PMID: 14709011.

[20] Chenglong Li, Xiangshu Luo, Yonghan Li, Huijuan Yang, Xiao Liang, Kai Wen, Yanxin Cao, Chao Li, Weiyu Wang, Weimin Shi, and et al. A class-selective immunoassay for sulfonamides residue detection in milk using a superior polyclonal antibody with broad specificity and highly uniform affinity. *Molecules*, 24(3):443, Jan 2019.

[21] Bruce Lindbloom. Rgb/xyz matrices. www.brucelindbloom.com/Eqn_RGB_XYZ_Matrix.html, Apr 2017.

[22] Shanhong Liu. Android - statistics & facts. https://www.statista.com/topics/876/android/, Jun 2020.

[23] Zhongyu Lou, Theo Gevers, Ninghang Hu, and Marcel P. Lucassen. Color constancy by deep learning. *Procedings of the British Machine Vision Conference 2015*, 2015.

[24] Satya Mallick. Filling holes in an image using opencv (python / c++), Nov 2015.

[25] Mark T. Muldoon, Carol K. Holtzapple, Sudhir S. Deshpande, Ross C. Beier, and Larry H. Stanker. Development of a monoclonal antibody-based celisa for the analysis of sulfadimethoxine. 1. development and characterization of monoclonal antibodies and molecular modeling studies of antibody recognition. *Journal of Agricultural and Food Chemistry*, 48(2):537–544, 2000.

[26] The Editors of Encyclopaedia Britannica. Colorimetry, Feb 2014.

[27] Jnos Schanda, George Eppeldauer, and Georg Sauter. Tristimulus color measurement of self-luminous sources. *Colorimetry*, page 135–157.

[28] Xingzheng Wang and D Zhang. An optimized tongue image color correction scheme. *IEEE Transactions on Information Technology in Biomedicine*, 14(6):1355–1364, 2010.

[29] X-Rite. What is a spectrophotometer. https://www.xrite.com/learning-color-education/other-resources/what-is-a-spectrophotometer.

[30] Haifeng Yu, Tian Cao, Bo Li, Rong Dong, and Huiyu Zhou. A method for color calibration based on simulated annealing optimization. pages 54–58, 07 2016.