Optimising the Computer Vision Module of Eonics' Autonomous Drone

Final Paper

TI3806 - Bachelor End Project

Mees Brinkhuis
Dirk den Hoedt
Mike van der Meer
Toby van Willegen
Tim Yarally

Client:	Eonics, The Hague, The Netherlands
Client Adviser:	Jan Peter Jansen and Marcus Wagenaar
University:	Delft University of Technology
Faculty:	Electrical Engineering, Mathematics, and Computer Science
University Adviser:	Thomas Overklift Vaupel Klein
Date of Document:	February 6, 2020

Abstract

Items being misplaced in warehouses easily get lost. To combat this, warehouses have to send people in scanning all barcodes in the warehouse. This is highly inefficient, which is why Eonics wants to build a drone handling this.

There are options out there to scan barcodes, but none of them match the requirements laid out by Eonics. Among these requirements are a lightweight camera, such as a GoPro, and a recording distance of 1.5-2 metres. This report will look and see if these requirements are feasible. Techniques used in this report are Mathematical Morphology, Maximally Stable Extremal Regions, Convolutional Neural Networks, Gradiental Difference and Direction Estimation with Region Extraction.

The report concludes in stating that interpreting the barcodes is not possible with mere software under these requirements. The maximal distance we were able to interpret barcodes from, based on a 4K image, was around 1 metre. Continuing the trend, we would need at least an 8K camera to detect from a distance of 1.5 metres. Detection however, is less difficult and is feasible from a distance of 1.5-2 metres. The report also derives an function to use to calculate the maximum distance a barcode can be interpreted from, based on the details of the barcode and camera. Finally, research is done regarding using hardware solutions, such as a zoom-lens, which has promising results.

I Preface

This report concludes the Bachelor End Project, TI3806, as given at the Delft University of Technology. The Bachelor End Project is a ten-week project revolving around solving a real-life problem, delivered by an external company. Our problem was provided by Eonics. Eonics is a company that specialises in Software Development and is based in The Hague, the Netherlands. Eonics' main interest lies in the development of software for customers; however, they also have in-house projects such as the one that we applied for. Eonics would like to build a drone that can fly within large warehouses and scan the inventory to check for item misplacement. Currently, every once in a while a warehouse has to hire a group of employees that manually have to scan the inventory to see if items are misplaced.

We would like to thank Eonics for providing us with the project itself, a location to work on the project, and any help they have provided in solving this problem. We would also like to thank Tom van der Drift for coming up with our team name; Airwolf.

II Introduction

The digital era has changed the world around us; information is shared more easily and faster than ever before. However, with this change, our expectations changed as well. Our shopping moved to online stores, where we expect our items to be delivered the next (or sometimes even the same) day. AliExpress, Amazon.com, Bol.com are all online stores offering a wide selection of products. These products are stored in enormous warehouses, with rows and rows of storage racks. In these warehouses, a computer keeps track of what is placed where. However, once an employee makes a mistake and places an item on the wrong rack, the item is lost. To combat these misplacements a group of employees has to check every rack and item periodically; something that inherently takes a long time.

The proceedings of the employees are fairly straightforward: pass every rack, scan every item and verify against the computer if the item belongs there. Eonics' view is that this is inefficient and prone to error. Therefore, Eonics is developing an autonomous drone to take over this task. While Eonics already has a flying prototype of their drone, it lacks the ability to scan, detect, and interpret barcodes. The goal of our project is to develop and optimise the barcode detection module of this drone.

Whilst there are options available out there, none of them fit the requirements Eonics laid down. For example, the most used barcode library as of now, ZBar, cannot handle rotated barcodes. As the drone is not likely to fly strictly parallel to the barcodes, this is unusable in its current form.

In this report we will look at several different methods for recognising, among which *Mathemathical Morphology*, *Maximally Stable Extremal Regions*, *Convolutional Neural Networks*, *Gradiental Difference* and *Direction Estimation with Region Extraction* as well as interpreting barcodes with several pre-processing steps, such as *cropping*, *rotation* and *binarisation*.

In chapter 1 we look at what Eonics' problem fully entails. In chapter 2 we explain our development methodology; how did we come up with the final solution? In chapter 3 we take a look at where the current barcode recognition technologies stand, and how we can utilise these. In chapter 4 we test our implementations of the different methods of recognising and interpreting barcodes and compare these against a pre-defined set of test images. In chapter 5 we explain how the final program is set up, with all its functionalities. In chapter 6 we look at several experiments that we conducted throughout the course of the project. In chapter 7 we look at the ethical implications of our program. In chapter 8 we explain how we handles the feedback we received from the Software Improvement Group. In chapter 9 we conclude this report with our findings. In chapter 10 we take a second look at our findings and report and discuss where we might went wrong. Finally, in chapter 11 we recommend fields of research that we deem interesting for this study.

Contents

1 Problem Definition & Analysis

The goal of this project is to develop and optimise the computer vision module of Eonics' drone which has to manage the inventory of a warehouse. In this chapter we will first discuss the problem that we need to solve in order to bring this project to a successful conclusion, after which we will analyse that problem.

1.1 Problem Definition

For companies with large warehouses it is difficult to maintain proper logistics. As a result items often go missing with no easy method to trace them. To remedy this problem once in a while some employees have to go through the warehouse. Employees have to go and pass all the racks to manually count each and every pallet, which is stated in Spider, 2018. This process could easily be automated, which is why Eonics is in the process of constructing their own drone that should eventually be deployed in such a large warehouse.
The drone will need to fly past every rack, scan all the items and accumulate data. Eventually, this data will be compared with the warehouse database. The problem that Eonics wants us to solve specifically is related to the software that processes the camera output to detect barcodes.

The problem comes down to this:

- The camera starts recording at the bottom of a column.
- An orange beam is detected with a localisation barcode on it. This barcode contains information regarding the current horizontal row and column of the drone.
- Flies up to the next level, which is determined using the orange beams.
- Detects barcodes on the products of that specific level and interprets them.
- Flies up to the next level and repeats the previous two steps until the end of the column is reached.
- Flies down and to the next column where it repeats this process.

Our task is to analyse the video feed to detect and interpret the barcodes that are in the video.

1.2 Problem Analysis

The drone is equipped with a bunch of modules that are used for localisation including a LIDAR, which is a laser scanner that creates a point cloud from the surroundings; an IMU, an unit that combines the input from accelerometers, gyroscopes and magnetometers to calculate angular velocity and rotation; a distance sensor, a flow camera and more. These components are used by a different team at Eonics and fall outside the scope of our project. We will work with the output from the camera that is positioned on the front of the drone. This camera records a continuous video feed, that should contain the various barcodes and landmarks. Once the drone has made its round through the warehouse, the video feed is passed through our program. Our program should be able to distinguish the vertical columns in which the drone flew, by detecting the barcode on a horizontal orange bar at the bottom of this column. Every level is then separated by another orange bar without any barcodes. By detecting and interpreting the products on every level, our program should output what product is located where.

To solve this problem we will look at different existing methods from papers to detect and interpret barcodes as well as test them. We will also look into methods to repair and crop barcodes from a provided image in order to increase the accuracy of interpretation. The various methods we will look at during the research phase of the project and the results we achieve with them during the research phase will be discussed in chapter 3.

1.3 Requirements

We documented our requirements in accordance with the MoSCoW format which was introduced by Clegg (1994). MoSCoW makes the distinction between the:

- Must have requirements, features that *must* be in your final solution at the end of the project
- Should haves, features that *should* be in your final solution unless you can provide a valid reason for the contrary
- Could haves, features that *could* be implemented if there is additional time and resources
- Won't haves/would likes, features that you can wish for but that will most definitely not be present in the final solution. In agreement with our client, we have decided to omit the won't haves from our requirement list.

We created a draft of our requirements first which we updated after an interview with our client in the first week of the project. This interview can be found in appendix A and the final version of our requirements can be found in appendix B.

2 Development Methodology

During this project, SCRUM (Schwaber & Beedle, 2002) will be used as our main software development methodology. We have chosen this option for multiple reasons. Firstly, this is a method we are familiar with, as we have extensively used this in courses given in the Bachelor Computer Science and Engineering at the Delft University of Technology. Secondly, the product owner wants to meet us weekly and be given a demo during this meeting. Thus, at the end of each week, we need to have the project in a working state and give a small demonstration of the work we did in that week. Lastly, SCRUM makes it relatively easy to track what everybody is doing and prevents doing the same task multiple times by different team members. For keeping track of the SCRUM tasks, we use Trello ("Trello," 2020).

We will be using the OpenCV library (Bradski & Kaehler, 2008) for the computer vision part, because this library is the best one available and most accessible for Computer Vision. It has over 2500 optimised algorithms and is used by many major companies including Google, Yahoo, Microsoft, Intel and Sony, as well as numerous startups ("OpenCV," 2019).

For our programming language we will be using Python. We have made this decision because of the accessibility and ease of usability of the OpenCV library. The OpenCV library has both interfaces for C++ and Python. All team members were inexperienced with C++, for which we decided against using that. In addition, there has been some earlier development on this product, which was done in Python. C++ tends to runs a slightly faster than Python, according to Prechelt (2000). However, since OpenCV is a library written in C++ and the Python interface is only a wrapper, this speed advantage will be negligible in the real world and outweighed by the speed of development. We have weighed these pros and cons with the team, and decided to go with Python, because we have virtually unlimited processing power available in this project.

	Python	C++
Experience	✓	×
Speed	-	-
OpenCV available	✓	✓
Existing code base	✓	×

Table 2.1: *Comparisons between C++ and Python, with results ranging from ×, through -, to ✓.*

Development Phases

(The original Project Plan created in week 1 can be found in appendix C)

Communication between team members is mostly face to face; we are present at Eonics HQ four days per week with our team. In this way we can communicate directly with each other and make sure the workload is distributed evenly. The tasks will be divided at the start of each sprint and at the end of each sprint a short evaluation is held to make sure everyone is still on track. As the first two weeks mostly consist of research and report writing, we decided to start our first sprint officially in week three of the project.

From week three onward we started with the actual development process. This process can be divided into three phases. The first phase was mainly exploratory and lasted for about three weeks. During this time, all members developed several rough image processing modules that we used for testing and verification. These scripts could only work independently and had to be started through a command line, since there was no user interface yet. We tested and tweaked our modules on a data set of images and videos that we shot ourselves during a visit to *Kuehne + Nagel* on December 10, 2019. You can find a detailed review of how we structured this visit in appendix D. The cameras that we used during this visit were carefully selected, this process is described in appendix E.

The second phase was the integration phase, this lasted for roughly two weeks. In this phase we worked together to create a working version of our program with all of the independent processing modules integrated and accessible through a graphical user interface. To achieve this we had to alter all of the algorithms so that they each operated with the same input and output. The exact workings of the program pipeline can be found in chapter 5.

The final phase was dedicated to optimisation, additional research and report. During this phase we made some alteration to the final program, but no major changes that would have a large impact. The changes were mostly focused on bug fixing and improvements to the debugging interface. In addition we made a script that can conveniently generate new image processing modules and add them to the program. We also spent some time on additional research that could provide future developers with some ideas on what to do, more on this in chapter 11.The remaining time was spent on this report and our presentation that will take place February 5, 2020.

3 Existing Solutions

As has been discussed in chapter 1, our project revolves around the detection and interpretation of barcodes. In this chapter we will first discuss the existing methods of detecting a barcode and then the existing methods of interpreting barcodes.

3.1 Detecting Barcodes

There are a multitude of techniques to detect a barcode, and each one has their own advantages and disadvantages. In this section, we will discuss a few of these techniques.

Mathematical Morphology

The first technique we will discuss is called *Mathematical Morphology*, as defined by Katona and Nyúl (2013). The algorithm is split into two phases; the *Noise-Reduction Phase*, and the *Processing Phase*.

In the *Noise-Reduction Phase* the input image is converted to grey-scale and the input noise is reduced with a *Gaussian kernel* using *Edge Enhancement*. After that *Bottom-Hat Filtering*, or *Black Top-Hat Filtering*, is used to enhance the dark bars on the light background. This filter essentially suppresses larger areas of black, whilst keeping the smaller ones. Most barcode localisation methods instead use *Gradient Filtering*, which runs faster, but Bottom-Hat Filtering favours accuracy over operation time. Afterwards the image is converted to binary by using a threshold. The threshold value is based on the size of the image and the frequency of the most occurring element.

Now the *Processing Phase* is initiated. In the pre-processed image there are still many regions that fit the criteria for a barcode but are in fact not barcodes. These regions are eliminated by setting an *Area Threshold* which is based upon the amount of pixels in the image. The algorithm uses the structure of the barcodes, densely packed parallel stripes, to find them. To find these parallel stripes, the Euclidean distance from each pixel to the nearest nonzero (non-black) pixel is calculated. Using this distance map the algorithm drops regions that are far from other regions since they do not match the criteria of being densely packed. Afterwards, occasional dense text remains. To remove that *Dilation* and *Erosion* are used. Dilation uses a *Square Structuring Element* of size $S = 3w$, where w is the width of the widest bar. Finally, the Area Threshold is re-applied.

Convolutional Neural Networks

The second technique we will discuss is a *Neural Network* approach. A camera, while it can scan virtually every barcode, generally needs certain amount of guidance from the user. They need to point and rotate the camera to scan the barcode. Hansen, Nasrollahi, Rasmussen, and Moeslund (2017) however, used *Convolutional Neural Networks* (CNNs) to make detecting and decoding barcodes as fast and accurate as possible, under all orientations.

As a neural network is a black box, not much is known about the inner workings of the algorithm. Individual neurons are responsible for recognising different parts of the image and added together, they recognise barcodes. However, barcodes are detected using this algorithm, after which the image is cropped to only contain said barcode. The algorithm then rotates it to get the barcode in the right orientation, which helps with accuracy and speed of the barcode decoding.

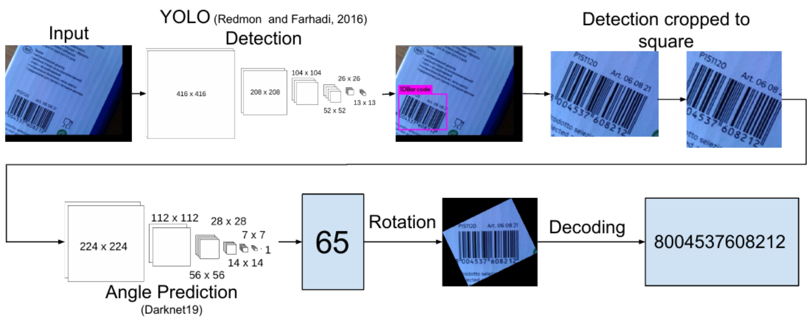


Figure 3.1: *System overview of a Convolutional Neural Network approach, from Hansen, Nasrollahi, Rasmussen, and Moeslund (2017)*

Maximally Stable Extremal Regions (MSER)

Most of the techniques that look at barcodes look at lines or textures. Creusot and Munawar (2015) make use of a different approach, based on detecting, filtering, and clustering of blobs. They make use of *MSER*, *Maximally Stable Extremal Regions*. This technique consists of several steps, as visualised in figure 3.2, and explained more into detail in chapter 5.2. It can detect blobs in images in a stable and reliable way. False positives are filtered out by looking at the width to height ratio of these blobs. Barcodes have a width to height ratio of at least $1 : 14$, this technique uses a width to height ratio of $1 : 10$.

After this step the blobs are clustered together, in a way that the stripes which belong to one barcode are grouped together in a cluster. The last step is to calculate the boundaries of these clusters, and within this boundary, one barcode is located. The big advantage with this technique is that multiple barcodes can be detected.

Gradient Difference

Yet another method was presented by Upasani, Khandate, Nikhare, Mange, and Tornekar (2016) in their paper on barcode recognition using a web-cam. The first step is again converting the

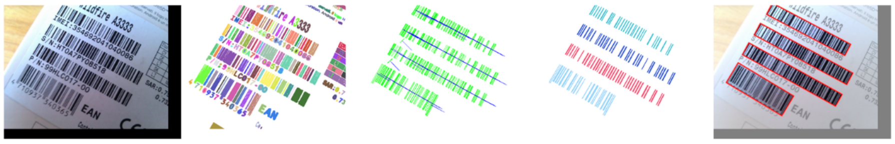


Figure 3.2: *The different steps of the MSER approach. The steps are explained in detail in chapter 5.2. From Creusot and Munawar (2015)*

image to a grey-scale image. Now the authors state that "one should expect an extended region represented by strong horizontal gradients and weak vertical gradients" (Upasani et al., 2016). This holds true provided that the camera's vertical axis is parallel to the bars of the barcode. Then for every pixel n, the horizontal derivative ($l_x(n)$) and vertical derivative ($l_y(n)$) are calculated. Then $l_e(n)$ is defined as the absolute difference of the two ($|l_x(n) - l_y(n)|$). Because of the striped design of a barcode, Upasani et al. (2016) take the assumption that many points in the barcode take on large values for $l_e(n)$. We have constructed images of the respective derivatives of a generic barcode to show that this assumption is reasonable. These images can be found in figure 3.3. Lastly, $l_s(n)$ is constructed by passing $l_e(n)$ through a 31 by 31 Gaussian block filter (also called a kernel) and then binarising the result. A disadvantage of this technique is that the barcodes must be parallel to the viewport of the camera.

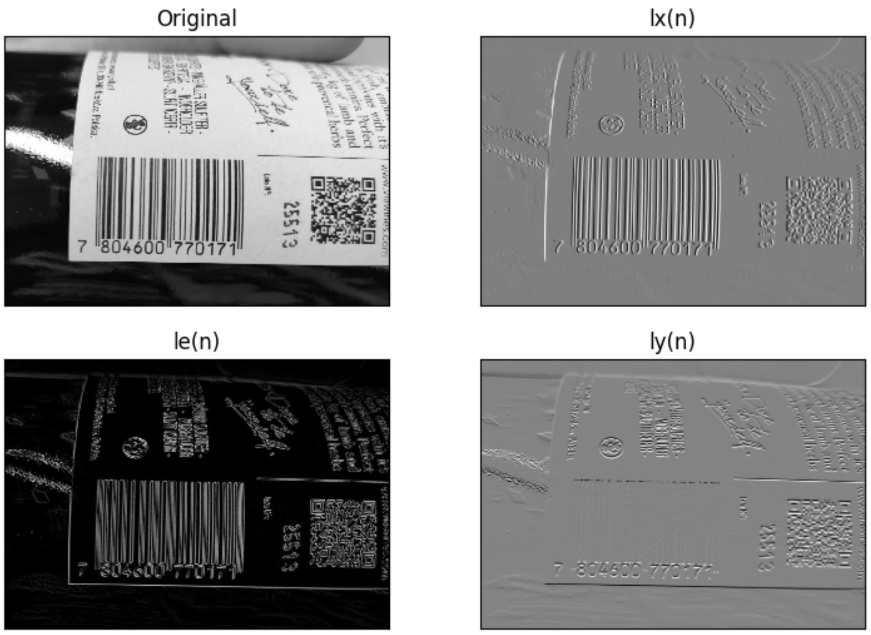


Figure 3.3: *Gradient Difference applied to an image with a barcode. Shown are the original image, the X and Y derivatives and the final difference of the two derivatives.*

Direction Estimation, Region Extraction & Key Frame Selection

Xu, Kamat, and Menassa (2017) propose a combination of three different algorithms that should improve barcode detection from a video input source. For the first algorithm the authors utilise the Hough Transformation. A Hough transformation tries to find aligned points in an image which make up a line. Solely applying a Hough transform to an image is not sufficient, because in non-perfect conditions there would remain too much noise. The key here lies in the fact that a barcode has a large density of corners on its top and bottom edge. All these corners combined form a line. Thus, firstly, the corners of the image are detected using the Harris Corner Detector, then the Hough Transformation is applied on the plot of the resulting corners. The Harris algorithm finds corners based on two edges in close proximity that are not parallel. Based on the Hough lines found through these corners the direction perpendicular to the barcode is then determined and the original image is rotated accordingly. Xu et al. (2017) state that this algorithm works robustly with a single barcode. For images that include multiple barcodes, candidate barcode regions have to be selected prior to the application of the direction adjustment algorithm.

The second algorithm uses the corrected results from the first one and isolates the barcode regions. As with all approaches, the image is first converted to grey scale. Then the contours of the image are extracted using basic edge detection. This results in a whole bunch of different regions. Now the algorithm goes through three different elimination steps. First a threshold is applied on the number of child regions per region. According to Xu et al. (2017): "The primary reason is that barcode regions usually contain more children due to the multiple bars contained within." The second threshold is derived from the knowledge that barcodes are quadrilaterals. Therefore the regions are approximated by a polygon with a limited amount of vertices. The intermediate result now only contains regions of rectangle-like shapes of varying sizes. The last elimination step removes all the regions that are smaller than some threshold.

The final algorithm is used to quickly select key frames. This reduces the load of the overall program because not every frame will be checked for barcodes. This is accomplished by looking at the histogram differences of two consecutive grey scale frames. This approach is very effective for detecting scene changes, but it also has applications for continuous shots. A sufficiently high histogram difference is often a clear indication for an unusable frame. These frames are more likely to be blurred. From the frames that are separated by these outliers, the average histogram difference is calculated. Next the frames with histogram differences that exceed this average are found, and finally, the frames prior to this selection are taken as the key frames.

3.2 Interpreting Barcodes

When the barcodes are detected, isolated and cropped; they are ready to be interpreted. At this stage the image is not ideal yet. To give some examples, the image can be victim of poor lighting conditions, noise or blurriness. Pre-processing of the image is necessary to ensure a greater accuracy during the interpretation.

Image Pre-processing

Upasani et al. (2016) go through several different steps in order to obtain an ideal barcode image. First of all the barcode is cropped to eliminate the surrounding information such as the numbers. Then the image contrast is increased, eliminating all of the blurriness. In the last step, any additional noise is removed by scanning every column and converting all the pixels to either black or white, depending on which occurred most often in that column. The final result is an ideal reconstruction of the original barcode that can now be interpreted.

Decoding

After a barcode is detected and pre-processing has been applied we should be able to use an existing library or program to decode it. There are a lot of existing implementations to do this, ranging from decades-old standalone software to very recent open-source libraries. Examples of these are `pyzbar` by Natural History Museum (2019) and `ZXing` by Oostendo (2016). To find out which one works best for our problem, we will implement and run these on the `WWU Muenster Barcode Database` by the Pattern Recognition and Image Analysis Group (2015). This dataset consists of more than a thousand barcodes.

To do this a small script was written that scans all barcodes in the database and tries to decode them all . During this process it kept track of the amount of barcodes it was able to successfully decode and the amount of time it took.

It was decided to have pyzbar and ZXing as the two libraries most suited for this project since they are the most popular barcode reading libraries for python. Pyzbar is a wrapper for the older Zbar library. It was created to add support for python3 and Windows. ZXing, on the other hand, has been in development since 2014 and is a fully collaborative java project that has been ported to a number of languages.

During testing it was quickly discovered that pyzbar performed noticeably better. Whilst it was only slightly more accurate, it took ZXing a good 7 times as long to process the whole data set.

Table 3.1: *Comparison of `pyzbar` and `ZXing`, based on their detections on the WWU Muenster Barcode Database*

	pyzbar	**ZXing**
number of image	1056	1056
successfully decoded	891	831
accuracy	84,37%	78,69%
total time	33.3s	242.5s
average time	0.032s	0.23s

Other things noticed were the fact that pyzbar is able to detect multiple barcodes within a single frame when available, whereas ZXing will only return a single barcode. In addition pyzbar is able to directly process OpenCV images whereas ZXing is fairly limited in the data types it accepts. One downside to both algorithms that we discovered later is that they are not rotation invariant. The images in the data set are all virtually straight, meaning that the bars of all the barcodes are aligned with the vertical camera axis. In an experiment that we performed using our benchmark tool (see appendix F), we learned that the accuracy of both interpreters decreased when detecting rotated barcodes.

3.3 Output Format

Once the desired information about the warehouse has been collected there needs to be a way for employees of the warehouse to use this data. There are two scenarios that need to be considered. The first one is where an employee is going to take the output and evaluate it manually, the other is the scenario where the output will be used as the input for another piece of software. In the first case the output must be easily human readable, this leaves two possible options, either output in plain text or in a format like CSV (Comma Separated Values) by the Internet Engineering Task Force (2005). For the second scenario a format like the JSON (JavaScript Object Notation) standard by the Internet Engineering Task Force (2017) or the XML (eXtensible Markup Language) standard by W3C (2008) could be used since they can easily be parsed by a computer.

4 Testing Insights and Applicable Algorithms

We have implemented our own versions of the algorithms mentioned in chapter 3. We will now elaborate on our new insights based on our own observations and results on the `WWU Muenster Barcode Database` data set.

4.1 Algorithm Implementations

Harris Corner Detection and Hough Transform

The first algorithm that we implemented corrects the rotation of the image based on Harris' Corners and Hough Lines (Xu et al., 2017). We found this algorithm to be very robust and customisable. The design of a barcode makes it very suitable to be extracted from an image using Harris Corner Detection, this can be observed in the right panel of figure 4.1. The corners are found in very close proximity to each other, basically forming a line. The angle of this line is calculated, after which the image is rotated accordingly.

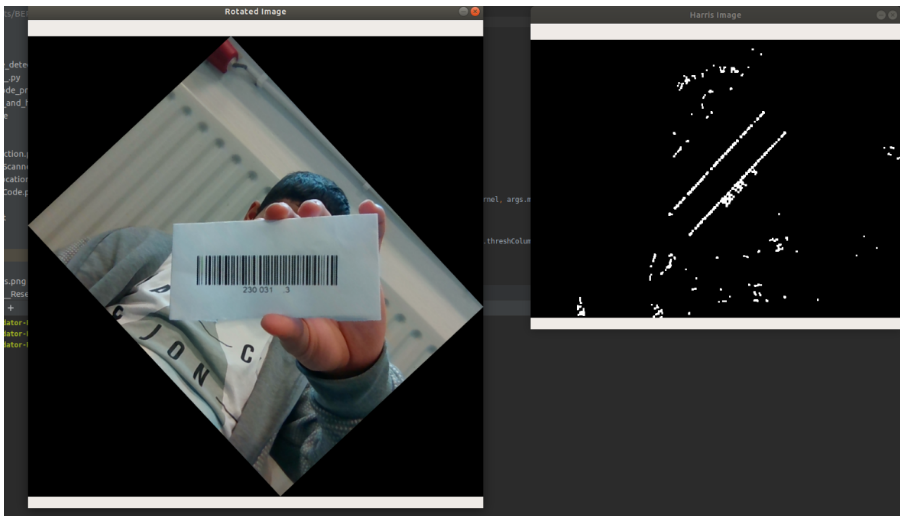


Figure 4.1: *Image correction based on Harris Corner Detection and Hough Transformation. On the right, the detected Harris' Corners can be seen, whereas on the left, the original image has been rotated accordingly.*

We will go into more detail regarding our final implementation of this algorithm in section 5.2.

Gradient Difference

The second algorithm that we implemented aimed to detect barcodes in a video stream in order to perform more optimised barcode decoding. The algorithm used for this was the Gradient Difference solution described in section 3.1. An example of a barcode detected via Gradient Difference can be found in figure 4.2. Cropping an image before running the interpreter significantly increases the speed of the program. This becomes even more apparent for a higher resolution input video. Refer to section 5.2 for a full analysis on our final version of this algorithm.

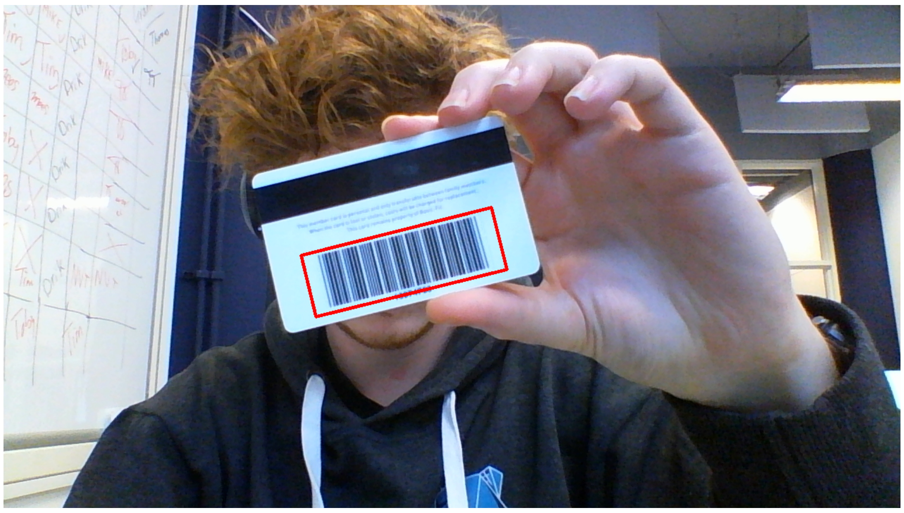


Figure 4.2: *Detection of a barcode based on Gradient Difference*

Column Thresholding

We also implemented an idea we came up with ourselves based on a technique mentioned by Upasani et al. (2016). The generic idea of this algorithm is to, instead of looking at the threshold of the whole image, look at thresholds of columns of pixels. Thus, rather than determining a threshold for the whole image, we look at a single column of several pixels wide and determine the threshold for that column alone. After the thresholding, as an optional additional step, we normalised the image by making every pixel column black or white based on the highest count in that column. The results can be seen in figure 4.3. Whilst at first this seemed to provide much better results than the adaptive thresholding techniques available in OpenCV, on closer inspection one can see that the input and output barcodes are not equal. Due to these discrepancies, this feature did not make the final product.

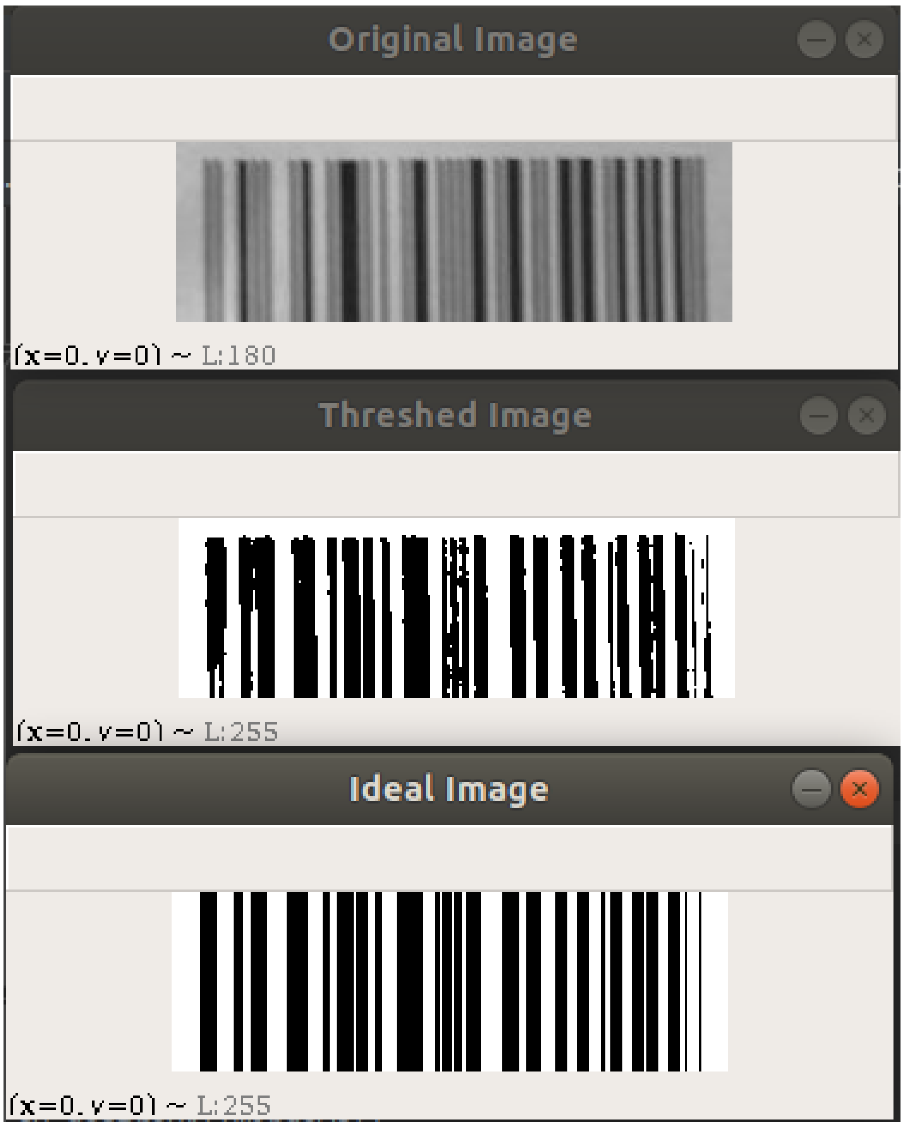


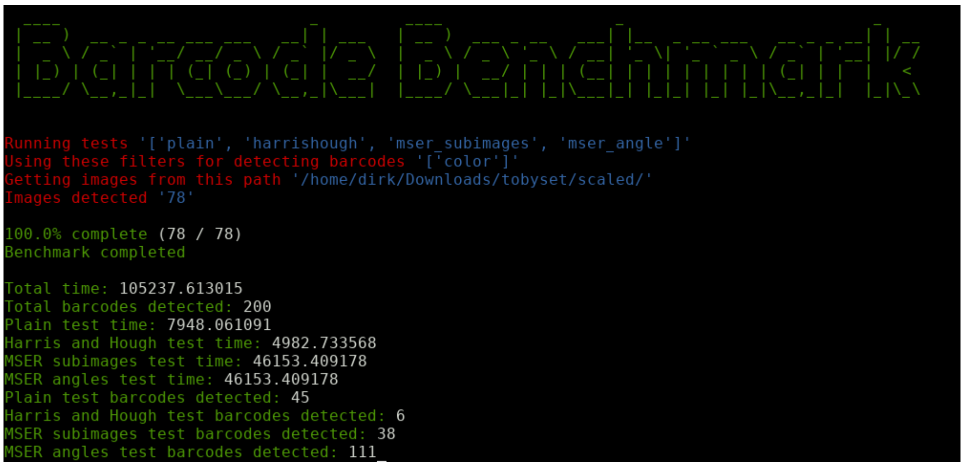
Figure 4.3: *Barcode binarisation using column thresholding. In this figure, "Threshed Image" uses a global threshold, whereas "Ideal Image" uses Column Thresholding. If you look closely to the image, you can see that the "Ideal Image" is not ideal at all compared to the Original Image.*

Maximally Stable Extremal Regions

The last algorithm that we implemented and tested was the rotational algorithm based on MSER. This algorithm is computationally rather demanding, but the results are very robust. In our final program we wrote a module that handles rotation using this concept and we adapted the algorithm in order to crop images as well. We will further elaborate on this in section 5.2.

4.2 Benchmark

As the amount of processing modules started to increase, we realised that we needed a convenient method to compare them against each other. To achieve this, a benchmark tool was developed. This tool takes a set of images as an input, and a list of algorithms that will be performed on those images. The program can also be configured to run on multiple cores. The program returns the duration of the benchmark in seconds, and how many barcodes were detected. See figure 4.4 for an example. Besides the command line output, the program also generates a more elaborate report which is saved in MarkDown format. For some extra insights, the benchmark tool was designed to run the algorithms on several different images layers: Full RGB, greyscale, the red channel, the green channel and the blue channel. Although the results were different for each channel, we found that these differences are very minor and inconsistent. We found that there was not one channel that always provided the best result. More proof on this can be found in appendix F.



```
 ____                          _        ____                  _                          _    
| __ )  __ _ _ __ ___ ___   __| | ___  | __ )  ___ _ __   ___| |__  _ __ ___   __ _ _ __| | __
|  _ \ / _` | '__/ __/ _ \ / _` |/ _ \ |  _ \ / _ \ '_ \ / __| '_ \| '_ ` _ \ / _` | '__| |/ /
| |_) | (_| | | | (_| (_) | (_| |  __/ | |_) |  __/ | | | (__| | | | | | | | | (_| | |  |   < 
|____/ \__,_|_|  \___\___/ \__,_|\___| |____/ \___|_| |_|\___|_| |_|_| |_| |_|\__,_|_|  |_|\_\

Running tests '['plain', 'harrishough', 'mser_subimages', 'mser_angle']'
Using these filters for detecting barcodes '['color']'
Getting images from this path '/home/dirk/Downloads/tobyset/scaled/'
Images detected '78'

100.0% complete (78 / 78)
Benchmark completed

Total time: 105237.613015
Total barcodes detected: 200
Plain test time: 7948.061091
Harris and Hough test time: 4982.733568
MSER subimages test time: 46153.409178
MSER angles test time: 46153.409178
Plain test barcodes detected: 45
Harris and Hough test barcodes detected: 6
MSER subimages test barcodes detected: 38
MSER angles test barcodes detected: 111_
```

Figure 4.4: *Screenshot of the Barcode Benchmark program.*

5 Final Program

Our program constitutes of an interface which requires a video or image as input. The user can then proceed to choose from a set of modules that can be added to the program pipeline. The pipeline and general flow of the program is discussed in section 5.1. The various modules that can be used in the pipeline and how new modules can be added is discussed in section 5.2. How the program goes from the input of a video or image to an output of the users choosing is discussed in section 5.3. What happens when our modules find a barcode is described in section 5.4. To conclude this chapter we will discuss the error handling performed in the program in section 5.5. A high level visualisation of the architecture of the program is shown in 5.1.

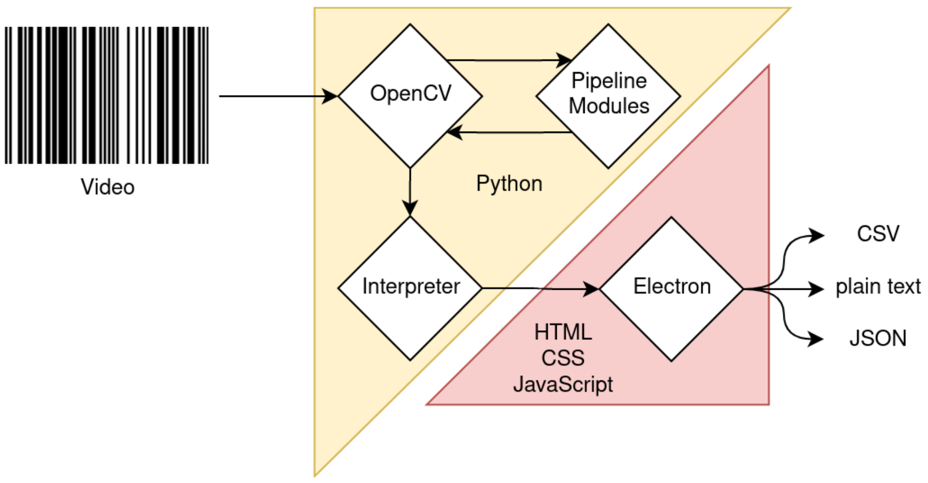


Figure 5.1: *Visualisation of the architecture of the final program. Shown are the different processes used and the programming languages in which they are implemented.*

5.1 Program Pipeline

One of the wishes of our client was that the system was modular, meaning that it should be easy to add more cropping, rotation and thresholding algorithms. To allow for this, we designed our program to be easily extendable. To add a new algorithm to the program, one only has to create a file containing the code for the algorithm and add a button to the interface. More on this in section 5.2. We call these Algorithm Modules and they simply take a single OpenCV image as input and return a list of resulting images, as visualised below.

Of course, one might want to use multiple of these modules; for example, one module can crop the image to only show the barcode and a second module can then be used to rotate the resulting images. This way the interpreter does not have to scan the entire image anymore.

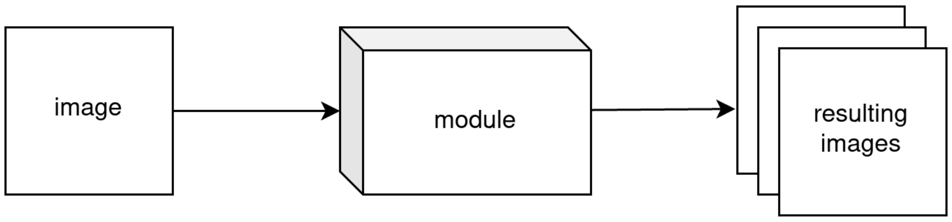


Figure 5.2: Visualisation of a module. An Algorithm Module takes one OpenCV image as input and can return several OpenCV images.

This greatly increases the speed at which we can process a video. We call a combination of modules a pipeline. A pipeline is any given set of modules in a specific order.

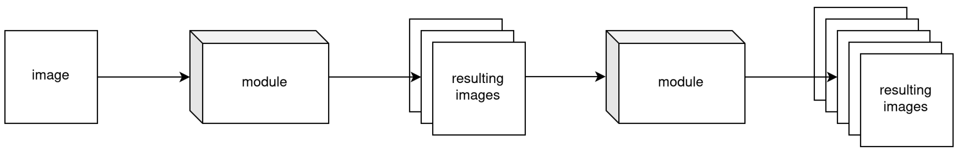


Figure 5.3: Visualisation of a pipeline. The first module retrieves a single OpenCV image, and returns several others. These then get passed into a different module one at a time, which in turn returns more images.

Figure 5.3 only shows the process on a single image, but our goal is to process an entire video feed. Since a video is a set of still images, our pipeline can run on each frame of the video feed. In the end it only has to remove duplicates from the resulting list of barcodes.

Using a pipeline system like this gives the end user near total control of which algorithms to use. This can be useful in situations where a trade-off has to be made between processing speed and accuracy. Using an efficient cropping algorithm, such as gradient cropping, can speed up the rate at which the video can be processed twofold. However, using a algorithm like this might lead to some barcodes being lost due to the fact that gradient cropping deals very poorly with rotated images. In this case one might want to use MSER rotation before feeding the video feed into this cropping algorithm, which in turn increases the processing time again.

5.2 Pipeline Modules

As mentioned above, the pipeline can run different types of algorithms. More modules can be added with the only requirement being that their algorithms take in a single image and output a list of images. We have made this even more convenient with the introduction of a module generation script. This script can be run from the command line with as input the module name, the function name, and the type of module. The modules that we currently have can be classified in three different types: rotation (0), cropping (1) and thresholding (2). The first type outputs a list of images that are rotated by different angles; the second one returns a list of cropped images and the third one returns a list of binarised results, meaning that every pixel of those images is set to either 0 or 1 based on a threshold value. In the subsections below, we will elaborate on our implementations of these modules as well as the code generation script mentioned above.

Module Generation

Modules can be created by hand and integrated into the program relatively easily, however, we have devised a more convenient method to achieve the same. Inside project folder, you can run our module generation script with the following code snippet:

```
    python module_generation.py -n module_name -f function_name -t 0
```

Listing 5.1: *Generation of a new module by using the following command.* `-n` *provides the module name,* `-f` *the function name and* `-t` *provides the type of algorithm, with 0 meaning rotation, 1 meaning cropping and 2 meaning thresholding.*

Executing this exact command will create a new file called module_name_rotating.py in the rotating package. It will also add this module to the rotating package's config file. The python file will contain a template with the function specified in the -f argument:

```
from os import path
import yaml
from debugger import show_debug_images

basepath = path.dirname(__file__)
filepath = path.abspath(path.join(basepath, "rotation_config.yaml"))
with open(filepath, 'r') as ymlfile:
    data = yaml.safe_load(ymlfile)
    cfg = data['module_name']


def function_name(image):
    if cfg['DEBUG']:
        show_debug_images([], "module_name_rotation_debug")

    return [], [], [], []
```

Listing 5.2: *Boilerplate code generated for the module by the command specified in Listing 5.1*

As you can see, there is some boilerplate code in the function already. The `cfg` variable refers to the config YAML file for the module. During generation an entry for the module is added to the config file with a boolean parameter called `DEBUG`. If `DEBUG` is set to True, the show_debug_images function takes a list of images as input and displays all those images in one fixed size window. The last thing that the generation script does is import the module and add it to the pipeline. The main HTML file also has code appended, meaning that the freshly generated module will show up in the user interface. A keen reader has noticed that in line 16 of listing 5.2, the function returns four empty lists. Only the first return value is important for the propagation in the pipeline, as this is the list of resulting images. The other three lists are used by the program to draw all the debug lines. From left to right you can return a list of contours, a list of lines, and a list of angles by which the resulting images of the first list were rotated.

Rotation Modules

Based on our prior research and extensive testing, we have developed two modules that handle the rotation step.

Harris and Hough Rotation

The first one is based a combination of *Harris Corner Detection* and *Houghs Line Detection*, we generally prefer to call this algorithm Harris and Hough Rotation. The Harris Corner algorithm works by finding all the edges of an images and determining the junction of those edges (Harris & Stephens, 1988). Barcodes have a characteristic that make them very distinct when applying this method. The top and bottom sides of a barcode have a very dense concentration of corners, while the rest of the barcode does not give any response at all. If we plot all these corners as dots on a contrasting image, it becomes apparent that that the barcode can be represented by two lines running across the top and bottom edge. The next step is to input the Harris plot into the Hough algorithm. Almost all of the noise is filtered out and ideally the function will return $2b$ lines for b barcodes in an image. The final step is to calculate the angle of these lines and rotate the image by the negative of that angle. When an image displays several barcodes in varying orientations, we first create a histogram with bin sizes of two degrees of all the found angles. For every peak in the histogram, we add a rotated image to the output list. This process is graphically represented in figure 5.4

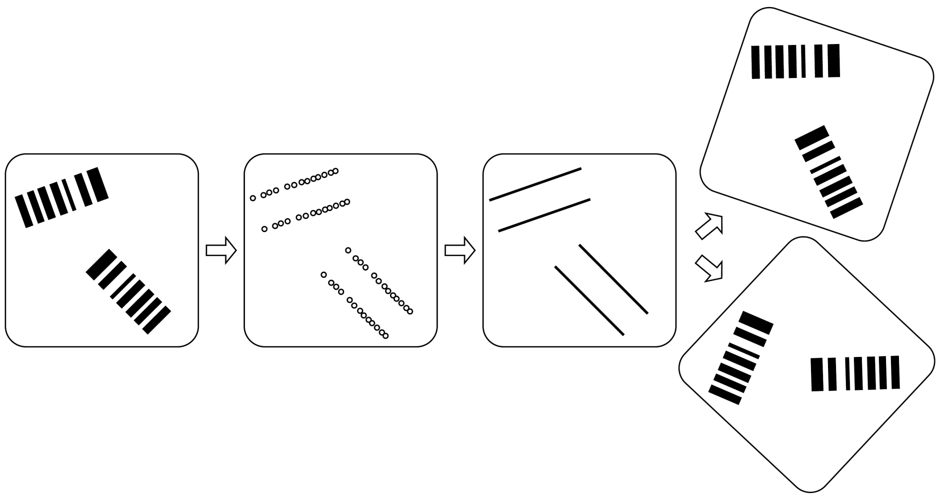


Figure 5.4: *Harris and Hough Process. The first frame shows the original image, the second shows the result of Harris Corner Detection, the third shows the result of Hough Line Detection and the final two frames show the result of the rotation of the original image.*

MSER Rotation

The second rotation module utilises the MSER region detection algorithm. MSER stands for *maximally stable extremal regions.* These regions were proposed by Matas, Chum, Urban, and Pajdla (2004) and the formal explanation is as follows: First you take every possible thresholds of a greyscale image, this would result in 255 images. This first image will be completely white and as the threshold increases, black regions will appear where the local minima are. The set of maximal regions is now defined as the set of all connected components of all the images. Because the black bars of a barcode are completely isolated from each other by a white background, they are part of the MSER set. The next step is to convert every region to a minimal area rectangle. Every rectangle has a centre, a width, a height and an angle of rotation. The rest of the process is very similar to the Harris and Hough module. We make a histogram with bin sizes of two degrees of all the found angles. Because MSER generally finds more angles than Harris and Hough, we apply a threshold for a minimum bin count. Lastly we output a list of rotated images for all the peaks in the histogram.

Cropping Modules

The second set of modules is designed to crop out parts of the input image that (probably) holds a barcode. We have three approaches for such modules.

Convolutional Neural Network

The first one is a Convolutional Neural Network (CNN) that we trained to recognise labels like the one in figure 5.7. A CNN tries to find patterns in the image space. The CNN-algorithm used in this paper is YOLO-v2 (Redmon & Farhadi, 2016). YOLO stands, in contrast to how the kids use it[1], for You Only Look Once.

YOLO is one of the fastest object detection algorithms currently available.

While a CNN would be able to detect and crop individual barcodes, we have opted to let it recognise full labels instead. While we found good algorithms to detect and crop barcodes, it was relatively difficult to detect if the found barcode belonged to a label. By using the CNN however, we can limit the other modules to just the label, and thus always find barcodes belonging to labels.

Generally, a CNN works as follows (Karn, 2016):

- The first layer is a Convolution layer. This is the layer where this type of Neural Net got its name from. The convolution layer *convolves* and input image to an output image. The image is convolved by moving a kernel map over the input image, resulting in a so called *Feature Map.* This Feature Map has a lower dimensionality than the input map. Only the most significant details are kept in this convolving step. See figure 5.5 for a schematic drawing.
- This *Feature Map* unfortunately can contain negative values, which is undesirable. Therefore, we will move this Feature Map through a ReLU layer. ReLU stands for Rectified

[1]https://www.urbandictionary.com/define.php?term=YOLO

Linear Unit, it maps every negative value to 0 and linearly scales in the positive values, effectively keeping these values intact in the Feature Map. An graph depicting the ReLu function can be found in figure 5.6.

- Finally, this Rectified Feature Map is moved through a Pooling Layer. This Pooling Layer, also called the Sub-sampling Layer, reduces the dimensionality of a feature map, but keeps the most important information. The most used version is max-pooling, where only the max value in a kernel is passed to the subsample.

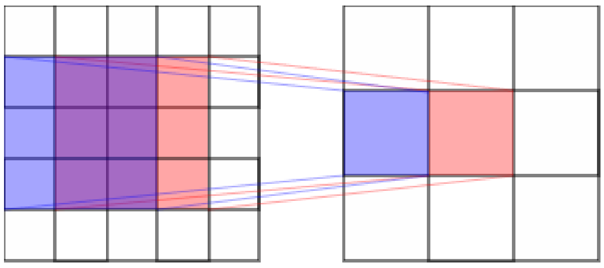


Figure 5.5: *A schematic drawing of a kernel mapping of size 3, on an 5x5 image. Resulting feature map is 3x3.*

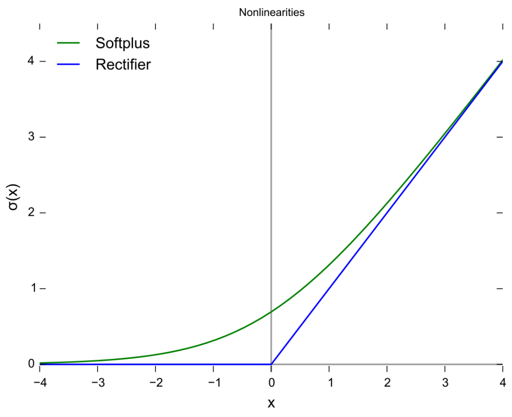


Figure 5.6: *A graph depicting both a SoftPlus function and a ReLU function. Image taken from Pccoronado (2020).*

These layers are repeated several times, taking the last output as the next input each time. After several iterations, the output is passed to a fully connected Neural Network, capable of recognising the contents of the image. This Neural Net has to be previously trained on recognising barcodes in order for it to recognise them. In our case, it was trained using the *darknet* application, with acceleration by utilising the GPU's CUDA cores (Redmon, 2013–2016).

Training was done on first on the WWU Muenster Barcode Database data set to verify the possible use of CNNs. Later, once verified that the CNN was indeed able to detect these barcodes, we continued training on the data set generated at Kuehne + Nagel. In order to make the training possible, ground truths needed to be known. This means that images had to be checked manually for any barcodes, and if they existed their corners had to be marked as such. The more accurate one does this, the more accurate the CNN will be.

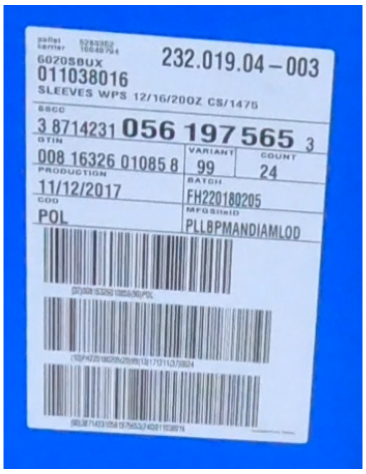

6020SBUX
011038016
232.019.04 – 003
SLEEVES WPS 12/16/2002 CS/1475
SSCC
3 8714231 056 197565 3
GTIN
008 16326 01085 8
VARIANT
99
COUNT
24
PRODUCTION
11/12/2017
BATCH
FH220180205
COO
POL
MFGSiteID
PLLBPMANDIAMLOD

Figure 5.7: *A barcode label as shot at Kuehne + Nagel. This label shows the reader the origin of the package as well as the contents.*

Sobel Operator

The next algorithm uses the Sobel derivatives to isolate barcodes in an image. This method is by far the fastest, but it also outputs the most false positives. First we take the x-derivative and the y-derivative of the input image and then we subtract these from each other. This works because the vertical derivative of a barcode basically shows zero deviation, while the horizontal derivative shows a very large deviation. The derivative of an images is approximated with *small neighbourhood gradient operators (SNGO)*. These are small 3x3 block filters that are used for convolving the image. As explained in Pratt (2007, p 471-478), the simplest form of such a SNGO calculates the difference between pixels in the rows and columns of the image. The corresponding filters look like this:

$$row = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 0 \end{pmatrix} \; column = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

We use the Sobel operator from the OpenCV library. This SNGO works according to the same basic principle as the running difference operator, but it considers more neighbouring values and weighs them differently. The Sober filters are displayed below:

$$row = (1/4) \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \; column = (1/4) * \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

You can clearly see that the values for the north, east, south and west pixels are doubled. As stated by Pratt (2007, p 467): *"The motivation for this weighting is to give equal importance to each pixel in terms of its contribution to the spatial gradient."* In terms of quality, the Sobel filter only provides a crude approximation of the actual gradient. However, for most practical applications like ours, it more than suffices.

After obtaining the gradient difference, we binarise the image. Then we perform an opening operation by eroding the image several times and then dilating that image again. This results in a black image with a lot of white regions that we can isolate with a built-in OpenCV algorithm that finds contours. The last step is to filter out some of these contours by comparing their dimensions to the known barcode dimensions. The final set of contours are then cropped and added to the output list.

MSER Clustering

The last cropping module is another implementation of MSER. As explained in 5.2, we first find all of the MSER regions. Ideally this means that we get a contour for all the black bars of a barcode. After this, we scale the widths of all these contours so that they start to overlap. In the last part of the process we loop through all these overlapping contours and find the largest cluster that contains all of them. These final clusters do not overlap with any other contour. We crop the image for every cluster and output them to the pipeline. You can refer to fig 5.8 for a overview of this process. Figure 3.2 shows a real-life implementation of the algorithm.

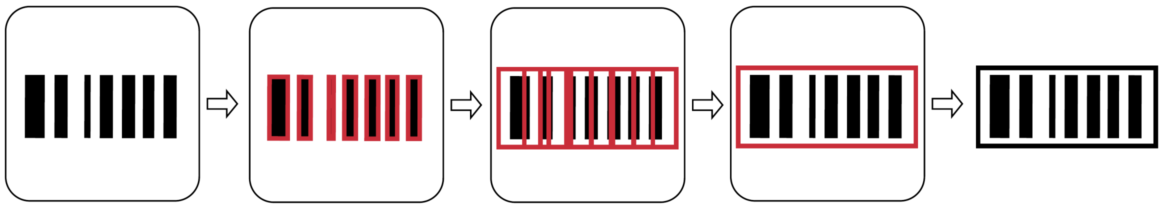


Figure 5.8: *Schematic view of MSER cropping. The first step shows the contours of all the black bars, the second step shows the overlapping contours, whereas the final step shows the largest cluster.*

Thresholding Modules

The last type of modules are designed to binarise an image, meaning that all the pixel values will be normalised to 0 or 1. Again we have two realisations for such modules. The first one is a column based approach. This algorithm determines a threshold based on the average value in a column of x pixels wide, where x is part of the input. All the columns are then binarised based on these local thresholds.

The second module is simply a wrapper for the OpenCV adaptive Gaussian thresholding algorithm. This algorithm uses a Gaussian block filter and calculates the local thresholds based on neighbouring pixel values.

For both the thresholding modules, we first convert an RBG-image to a HSV-image. The input image that we provide is the value channel of this HSV-image. We do this because of the larger contrast between black and white in this channel. Where RGB has separate intensities for red, green and blue colours; HSV has separate intensities for the hue, saturation and value. Value solely represents the brightness of a pixel, which is what we are interested in when binarising a barcode because of its black-on-white nature.

5.3 Input and Output

In order to allow warehouse personnel to use our system, a convenient and intuitive user interface needs to be available. There needs to be a way for the user to select their input file, select which modules to use, and to get a human readable output.

Input Selection

File input selection can be achieved in a number of ways. The user could provide the application with a string containing the file location on the system. This, however, is not convenient for the user, so it was decided that this approach was not suitable for our application. A more modern approach would be to have a simple button that opens a file select window, or to allow for drag on drop from the file manager. In the the end it was decided to go with a combined drag and drop area that is also clickable to open a file select window. This user experience is very much in line with modern web design, and thus likely to be familiar to the end user.

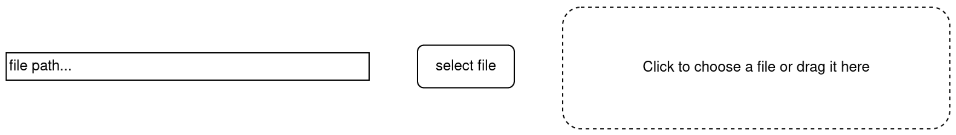


Figure 5.9: *Three possible file inputs; a file path input, a button to select a file or a drag and drop field. The latter one was chosen for its all-round availability.*

Module Selection

One of the defining features of our application is that it uses a pipeline based approach for the selection of modules. In order for us to facilitate a good user experience for selecting the modules and the order thereof a drag-and-drop based solution was designed. Using a visual representation of a pipeline where modules can be placed makes it intuitive for even an inexperienced user to figure out how to operate the application.

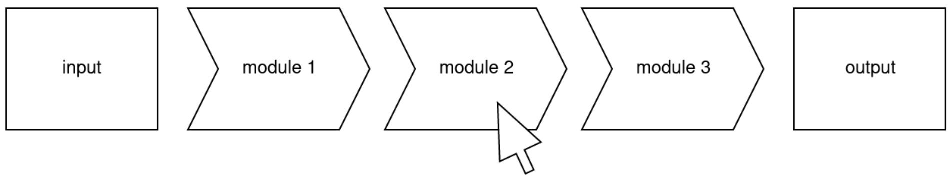


Figure 5.10: *Schematic visualisation of pipeline drag and drop. An user is able to drag modules around and place them where they want in the pipeline.*

Output Format

There are three use cases for the output of our program:

1. Real-time output. The user starts the application and the list of barcodes will be printed on the screen while the program is processing the video feed. This can be used to see if

the video feed is clear enough to give a result without having to completely process the video feed.

2. A spreadsheet based format. This is useful for long term storage and manual operations on the data later down the line. Using a format like CSV which can be opened in any spreadsheet processor (like Microsoft's Excel) make the data easy to work with. Internet Engineering Task Force, 2005

3. A computer-readable format. When the output of our application is used as the input for a different application or possible as the input for a web application, it is useful to present the user with a format that can be easily processed by a computer. We have decided to use the JSON format. Since it is widely used in modern software development. Internet Engineering Task Force, 2017.

Technology

The first draft of our graphical user interface was written using the TKinter library (Python, 2019). This allowed us to create a native user interface using Python. After experimenting with this library for a few days however, we concluded that more advanced GUI elements proved very difficult to implement. One example of this would be the drag and drop elements. The file selector that was provided by this library also did not live up to our expectations as it did not utilise the native file selector provided by the operating system. This prompted us to look for a different solution. After some consideration we decided to use Electron (Electron, 2020) to draw the graphical user interface (fig 5.11). The main reason for this was that Electron works on every platform and the interface looks identical no matter which system it is displayed on. Another advantage of Electron is that it is easy to implement drag and drop elements since it is based on existing web technologies. On top of that, one of our group members had extensive experience with Electron, making development in Electron significantly faster than other frameworks.

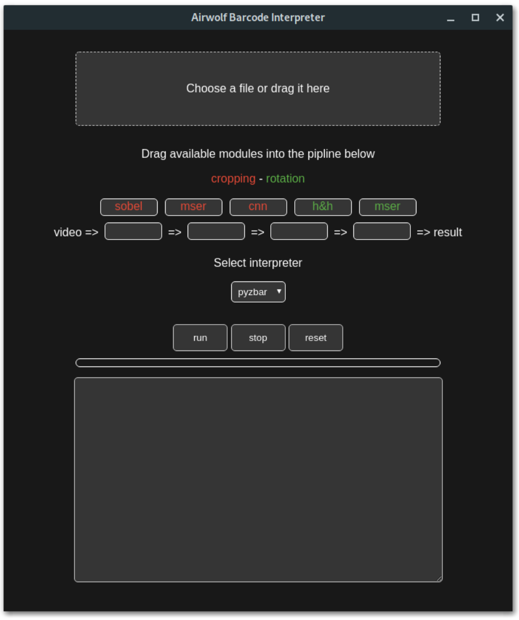

Airwolf Barcode Interpreter
Choose a file or drag it here
Drag available modules into the pipline below
cropping - rotation
sobel
mser
cnn
h&h
mser
video =>
=>
=>
=>
=> result
Select interpreter
pyzbar
run
stop
reset

Figure 5.11: *Our Electron based Graphical User Interface. The top shows the drag-and-drop field to place the video file, below that is the pipeline with available modules and below that is the output field with start, stop, and reset buttons.*

5.4 Interpreter

For our main interpreter we decided to use pyzbar (Natural History Museum, 2019). In chapter 3 we describe how we compared the two widely used interpreters pyzbar and ZXing (Oostendo, 2016), and came to the conclusion that pyzbar is superior. Nevertheless, pyzbar does not come without any problems. The main drawback that we encountered was that when a barcode could not be scanned there was no indication why it could not be scanned. The barcode could have an invalid checksum or part of the barcode could be unreadable, but pyzbar does not have the option to return partial results. We therefore decided to try to make our own interpreter with some minor error handling to see if we could interpret barcodes pyzbar could not interpret or at least return a part of the barcode that we could interpret. Both interpreters are available in the pipeline. A partial result from reading a barcode would be useful because with a camera there will be multiple frames of the same barcode, therefore there are possibly multiple partial results. Together these partial results could be used to recreate the barcode in its entirety.

The interpreter works similar to a physical scanner in the sense that it scans the barcode from side to side. The input for the algorithm has to be a cropped image that only contains the barcode. The algorithm initially counts all the widths of the bars in pixels(both black and white bars). A regular character is encoded by a combination of three black and three white bars. To indicate the end of a barcode, a stop character is used which has an additional black bar, resulting in a total of seven bars.

An example of how our custom interpreter interprets the barcode is shown in figure 5.12. Every code-128 barcode begins with a start encoding which can either be type A, B or C. All three types encode different characters in different ways (University of Washington, 2020). Therefore the encoding type can switch several times within the same barcode. This is to make sure all characters can be encoded in a way that minimises the width of the barcode. After determining the widths of all the bars, the custom interpreter combines these in groups of six. In figure 5.12, the first group encountered has the widths `2-1-1-2-1-4` which corresponds to `Start B` in all three of the encoding types. Next, the custom interpreter proceeds to iterate over these groups of six until it encounters the checksum and stop character. In this case, the characters `"H"`, `"a"`, `"l"`, `"l"` and `"o"` are found. The last 13 widths correspond to the checksum and the stop encoding. If the checksum, in this case `"d"`, is found to be correct, the algorithm has succeeded in reading the barcode.

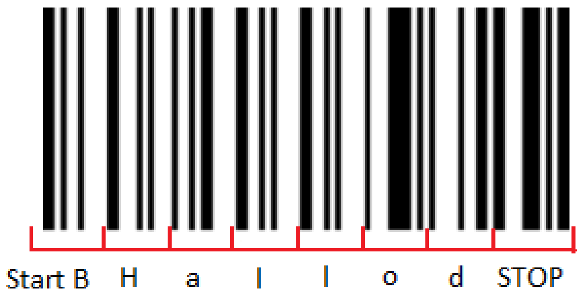


Figure 5.12: *The interpretation of a Code-128 barcode as generated by our custom interpreter.*

Merely determining the widths of the bars however proved to give very inaccurate results. In ideal conditions an image of a barcode should result in four distinct bar widths that are all multiples of each other. Real photographs, however, suffer from many flaws due to incorrect camera focus, motion blur and lens distortion. Therefore the resulting barcodes never consist solely out of four bar widths. We have devised three methods that attempt to counteract these flaws.

Bar Width Correction by Means of Clustering

The first is to use a clustering algorithm to group the bar widths into four bins and find a common denominator between these bins. After this, the widths of all the bars can be adjusted to the closest multiple of this denominator. This way all bars should be of a correct width in the context of the image. This process is graphically represented in figure 5.13.

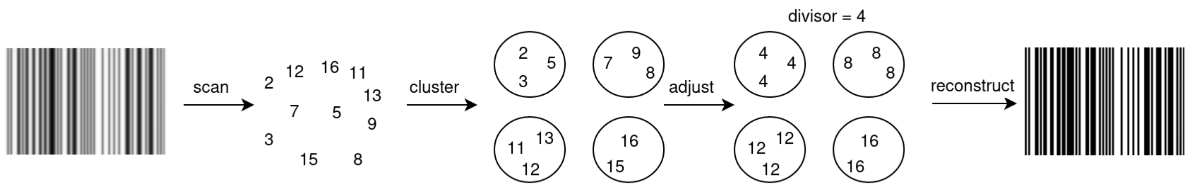


Figure 5.13: *Error correction of the bar widths by clustering similar bar widths together.*

Bar Width Correction by Normalization

For another less computationally intensive algorithm that corrects bar widths, we need to look at each character separately. The code-128 specification describes that the width of each character in the barcode should be a total of 11 units. This property can be used to determine if an error has occurred during scanning. It can also be used to adjust the widths of each bar in order to fit within these 11 units.

Take figure 5.14 as an example. The scanned image shows the widths to be: `4-2-4-4-5-4` which cannot be correct since 5 does not share a common denominator with 2 and 4. To get the correct width w, take any number n and the sum of the scanned numbers s and apply the following formula.

$$w = n \times \left(\frac{11}{s}\right)$$

This returns the following set of numbers: `1.91-0.96-1.91-1.91-2.39-1.91`, which can be rounded into the following list of integers: `2-1-2-2-2-2`. The sum of which is 11 and therefore can be interpreted.

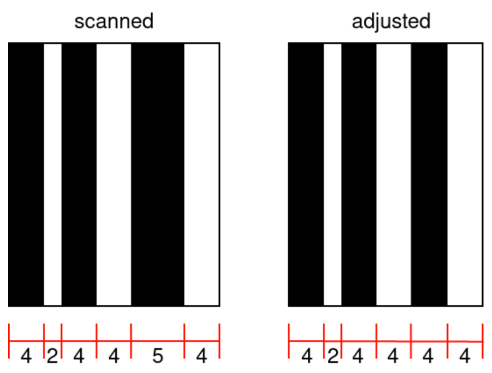


Figure 5.14: *Error correction by using the individual widths of the bars. By design, the normalised widths of the bars have to add up to 11 per encoded character. Widths shown in pxels.*

Checksum Validation

Another way to correct reading errors is to verify if the checksum of the barcode is correct. The checksum of a code-128 barcode is calculated by taking the sum of all the values and multiplying them by their position starting from the second character. Next you take the modulus 103 of this result and this value is encoded as the checksum of the barcode. Say there is a barcode with the following relevant values: $35, 51, 37, 19, 23, 16$. The checksum should have the value: $(35 + 51 + 37 * 2 + 19 * 3 + 23 * 4 + 16 * 5) \mod 103 = 80$. More formally, for any barcode B with a list of values V:

$$B_{checksum} = (V_0 + \sum_{i=1} V_i * i) \mod 103$$

If there is only a single character missing, or otherwise unreadable, in the scanned barcode, it can be reconstructed using the checksum. This approach, however, has proven to only be applicable in very select cases. If there are more missing characters, the checksum cannot be deduced and this approach will fail.

5.5 Error Handling

When a barcode is found that cannot be interpreted, the location of this barcode must be logged so the product is not lost. It is not a good solution to log every unrecognised barcode that the system throws, because this would result is many duplicate errors. In order to combat this problem, we have experimented with several different solutions.

Orange Beam Detection for Row Count

The first solution relies on a principle that we called orange beam detection. In the warehouse every level of products is separated by an orange beam. Besides that, in every level and column combination there is only one product at a time. So what this algorithm does is check if any product has been found each time it detects such an orange beam. If that is not the case then there is either no product or the barcode of the product in that spot could not be interpreted.

This solution has proven to be the most robust of all the ones discussed in this section, because the orange beams are detected without fail. A program that isolates the exact orange colour would not be a very adequate solution as changes in the lightning can alter the representation of a colour quite significantly. However, we can use the colour to enhance the contrast in the image. Most images we use are represented by three different channels: a red channel, a green channel and a blue channel (RGB). The values for each channel can range from 0 to 255, so the colour [0, 255, 0] represents green. The colour orange contains relatively high values in the red channel and relatively low values in the blue and green channel. Applying this knowledge we can create a high contrast image by subtracting one of the other two channels from the red channel and clipping all the negative values to 0. The resulting image will have high intensities for red-objects. For a better understanding of the colour ranges that get isolated when either the blue or green channel is subtracted, you can refer to figures 5.15 and 5.16.

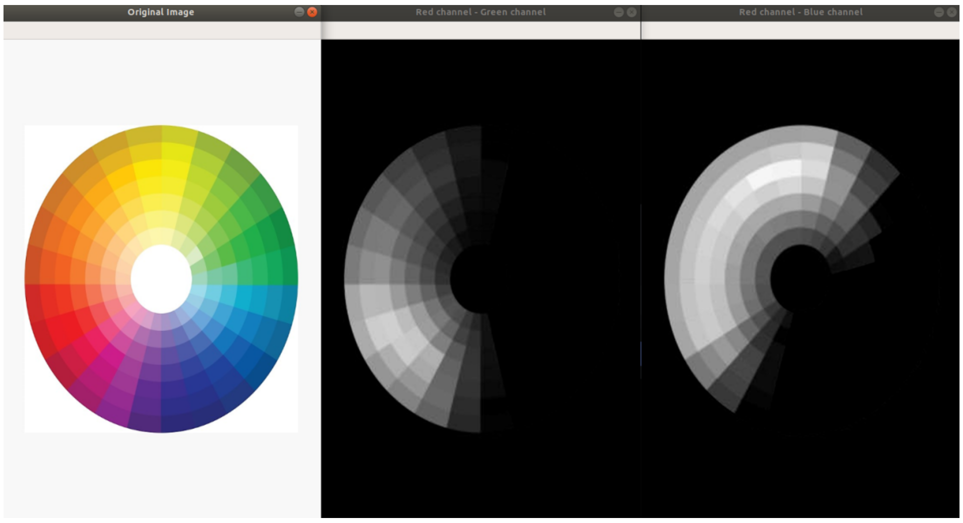


Figure 5.15: *Colours shown on a colour wheel when the green and blue channels are filtered from the red channel respectively.*

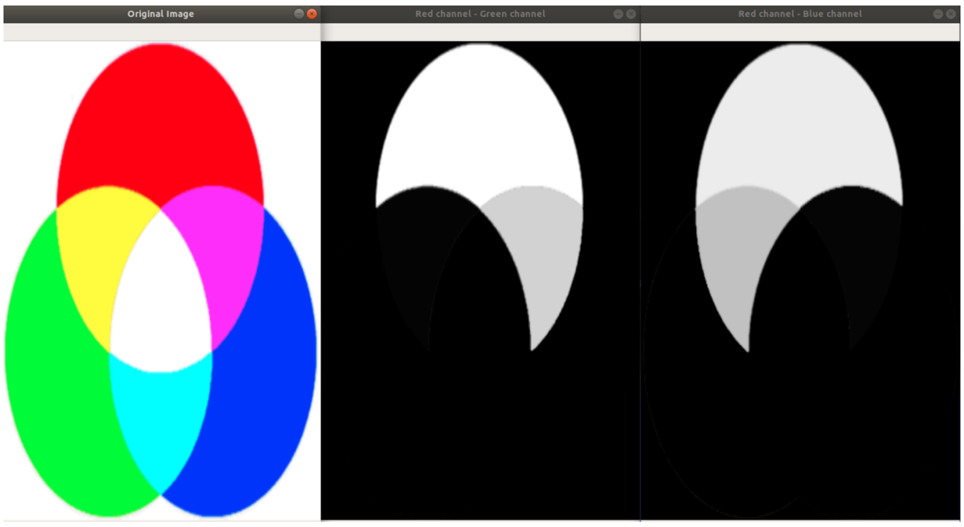


Figure 5.16: *Colours shown in a Venn diagram when the green and blue channels are filtered from the red channel respectively.*

The results from subtracting blue and green channel from the red channel can be seen in figure 5.17. Another characteristic of the beams is that they will always fill the entire screen from left to right. We can set a threshold on the average value of a row of pixels to binarise the image. This is shown in the bottom left of figure 5.17

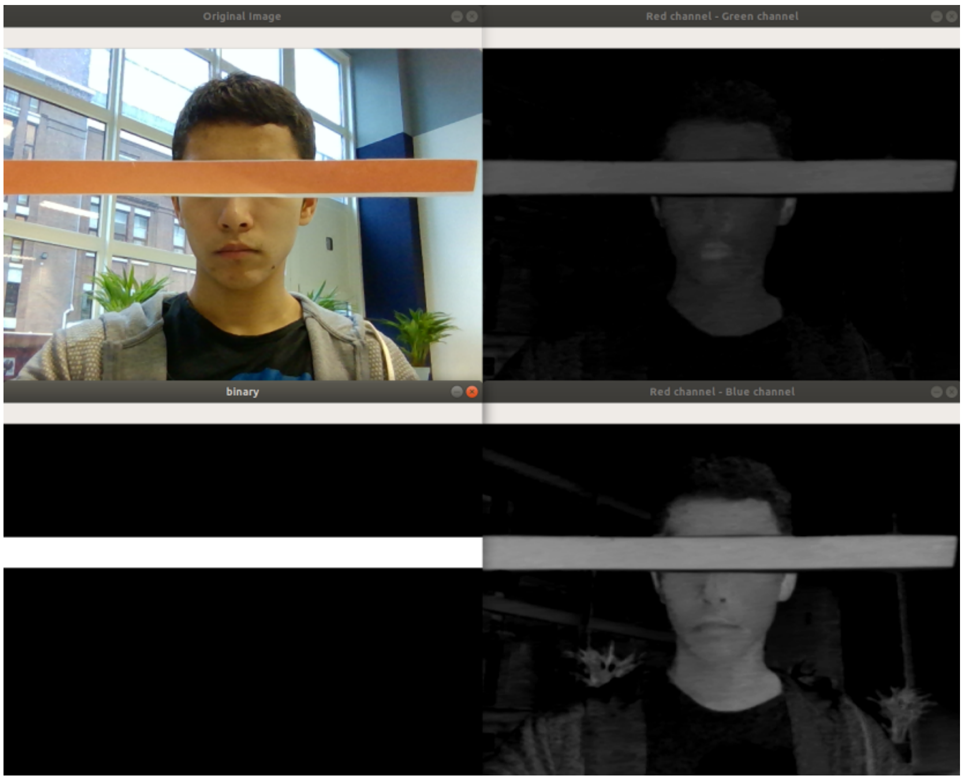


Figure 5.17: *Contrast enhancement for binarisation of orange beams. Left top shows the original image, right top shows the difference of the red and green channel of this image, the right bottom shows the difference of red and blue channel and the left bottom shows the binarised image.*

Nonetheless, the orange beam detection itself will not solve our problem for error handling. We stated that orange beams can be detected without fail, which is true, but that also implies that we detect beams that we are not interested in, such as the ones 2, 3 or 4 hallways behind the one we are filming in. The large amount of false positives makes this solution unsuitable for the error detection of the program.

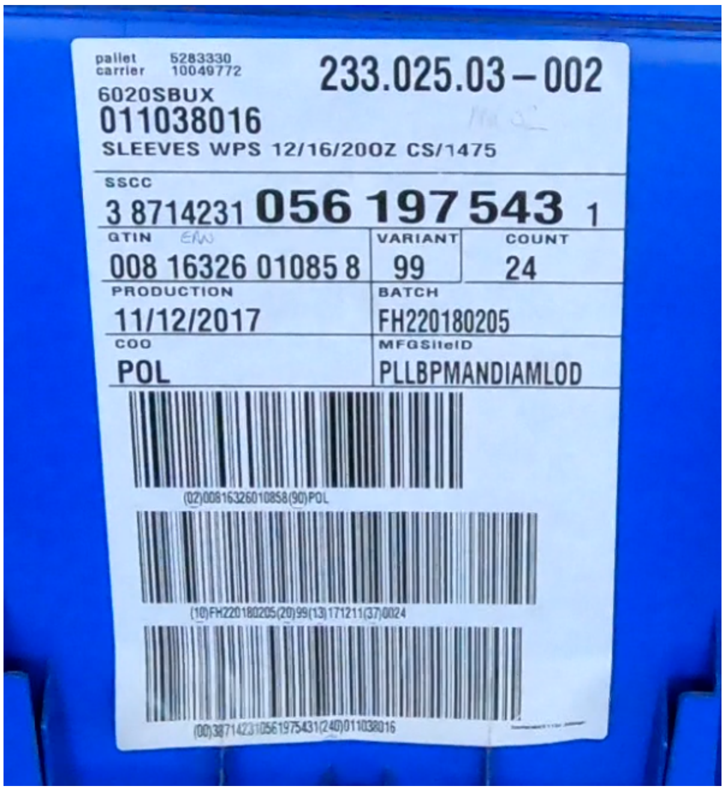
Other Solutions

We attempted a few different solutions besides the orange beam detection to combat non-interpretable barcodes. Most showed some potential, but we did not manage to solve the problem at this point in time.

Optical Character Recognition

Optical Character Recognition, Optical Character Reader or OCR converts images of text into machine readable text. Since a barcode often has the characters it encodes printed underneath itself, the idea was that if OCR could read those characters it could be used to prevent multiple errors arising from the same barcode and errors arising from a barcode that was correctly scanned in another frame.

The library we used was Tesseract OCR which has been developed by Google (2020). The problem encountered was that it was simply not robust enough. At some times, characters were mistaken for other characters, while at other times, no characters were recognised at all. In 5.18 an image is shown that was used for testing purposes and will serve as an example now.



pallet 5283330
carrier 10049772

233.025.03 – 002

6020SBUX
011038016
SLEEVES WPS 12/16/20OZ CS/1475

SSCC
3 8714231 056 197 543 1

GTIN	VARIANT	COUNT
008 16326 01085 8	99	24

PRODUCTION	BATCH
11/12/2017	FH220180205

COO	MFGSiteID
POL	PLLBPMANDIAMLOD

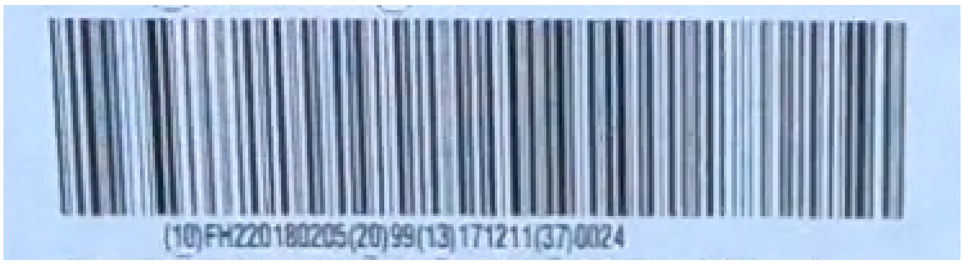
(02)00816326010858(90)POL

(10)FH220180205(20)99(13)171211(37)0024

(00)387142310561975431(240)011038016

Figure 5.18: *Image of a label, as taken in the test setup at Kuehne + Nagel. The label shows information about origin location and the product, among others.*

In figure 5.19 we used one of our algorithms to crop the image and then used Tesseract to see what it would find. It found nothing so we decided to crop it even further as can be seen in figure 5.20. In this last image the algorithm could read most of the characters correctly. This hardly fixes the problem since we have no algorithm that can crop the character string this tightly. We did not have time to further dive into this solution. It surely shows promise, but in its current state it is not robust enough for our use case.



(10)FH220180205(20)99(13)171211(37)0024

Figure 5.19: *Image of a barcode that was cropped from the label in figure 5.18 by one of our cropping algorithms.*

(10)FH220180205(20)99(13)171211(37)0024

Figure 5.20: *Manually cropped image to only show the characters of the barcode seen in figure 5.19.*

Custom Interpreter

The custom interpreter is one we made ourselves because pyzbar does not return partial results (see also chapter 5.4). Similar to OCR we attempted to use partial results to determine if we had already logged an error for a certain barcode. We could for example say that if 75% of the characters matched between 2 different frames the barcode in that frame was most likely the same one. The problem once again was that it was not robust enough. In too many cases the custom interpreter could either not start reading the barcode(no start encoding was found in the first combination of 3 white and black bars) or could not read most of the characters. Often this was because of the blurriness or resolution of the image, but these are exactly the cases that we need to perform error handling on.

LIDAR

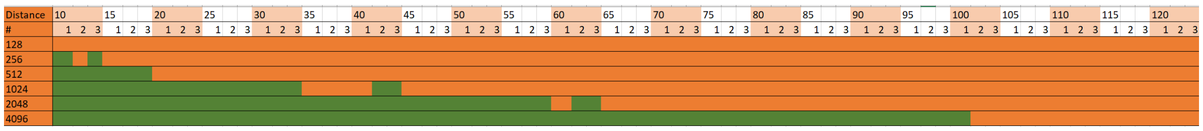
The best way to log the errors would be to use the orange beam detection in combination with LIDAR. The problem with the current error handling using orange beam detection is that there are too many false positives. For example, when a level of a column is empty the algorithm will see an orange beam in the rack behind it and increment a counter, which leads to an error being thrown. Currently there is no way of telling which orange beams are relevant and which are not, however with LIDAR the distance to a beam could be calculated. This would filter out all the irrelevant beams and lead to more robust algorithms.

6 Experiments

We performed various experiments to confirm ideas that we had during the course of this project. These experiments and the results that followed are be described below.

6.1 Resolution in Relation to Distance

We created a test set to see if there is a correlation between resolution and distance in regards to barcode recognition. This test set consists high quality images takes of a barcode from varying distances ranging from 10 centimetres to 2 metres with increments of 5 centimetres. The thinnest bar in this barcode had a width of 1 millimetre. For every distance 3 pictures were taken. Our findings are portrayed in figure 6.1. The numbers on the left side represent the width of the resolution in pixels and at the top is the distance in centimetres. As can be seen in 4k the image can still be read from 1 meter whereas pictures with a width of 2048 pixels can only read the image up to 60 centimetres.



Distance	10			15			20			25			30			35			40			45		
#	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
128	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
256	🟩	🟧	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
512	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1024	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟩	🟩	🟧	🟧	🟧
2048	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
4096	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

Distance	50			55			60			65			70			75			80			85		
#	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
128	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
256	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
512	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1024	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2048	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
4096	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩

Distance	90			95			100			105			110			115			120		
#	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
128	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
256	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
512	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
1024	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
2048	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
4096	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧

Figure 6.1: *Resolution of image in relation to distance of camera to barcode. The resolutions on the left show the amount of pixels in an image horizontally. The top shows the distance in centimetres. Green squares can be recognised, red squares cannot be recognised. Width of the thinnest bar in the barcode was 1 millimetre.*

With some more calculations we were able to derive that the virtual size of the thinnest bar has to be at least 2 pixels for the barcode to be interpreted by pyzbar. Now that we have this requirement we can describe a relation between the distance and the resolution once we take some other variables into account. Other variables that are necessary to describe this relation are the field of view of the camera and the physical width of the thinnest bar. This is brought in relation in figure 6.2.

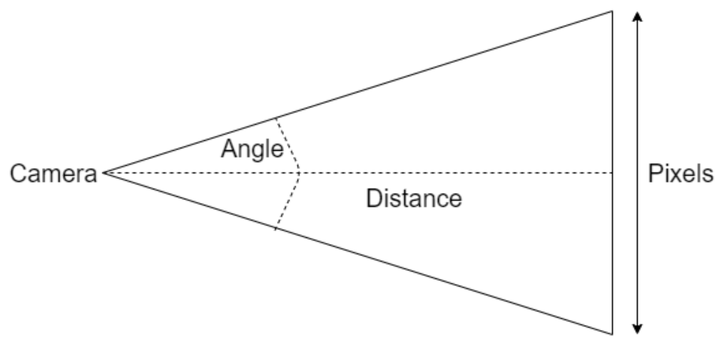


Figure 6.2: *Schematic overview of a camera shooting an image. From this image, one can derive formula 6.1*

.

From this figure we derived the following formula after simplifying:

$$d = \frac{P_{camera} \times w_{thinbar}}{4 \times \tan(\frac{\alpha}{2})} \text{ with } P_{camera} \neq 0, w_{thinbar} \neq 0 \qquad (6.1)$$

In this, d is the distance from the barcode to the camera in centimetres, P_{camera} is the horizontal pixel count of the camera, $w_{thinbar}$ is the width of the thinnest bar in centimetres and α is the viewing angle of the camera. If we utilise a zoom lens, it is evident to see that the effective viewing angle of the camera becomes smaller, thus increasing the maximum distance from camera to barcode.

Say we want to correctly interpret barcodes from a distance of 2 meters and we know the size of the thinnest bar in the barcodes; using this formula it is rather trivial to find a camera with a resolution and field of view that makes that possible.

6.2 Thresholding Modules

We experimented with some of the thresholding modules that were described in chapter 5.2 with the same setup that was described in the section above. The results are shown below in figures 6.3, 6.4 and 6.5. In figure 6.3, column-based thresholding is used. In figure 6.4, HSV-thresholding is used. In figure 6.5, we multiplied the value of the HSV image with 2 to increase contrast between black and white in the image.

Compared to the original result, which is shown in figure 6.1, it is evident that there is no increase in accuracy. Column based thresholding and contrast enhancement even show a significant decrease. We also tried a Gaussian threshold algorithm but this performed far worse. It only recognised a handful of images across all resolutions.

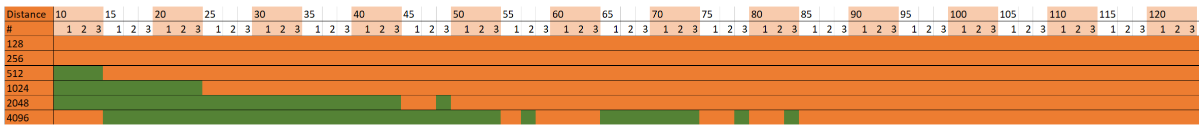


Figure 6.3: *Resolution of modified image with column thresholding in relation to distance of camera to barcode. The resolutions on the left show the amount of pixels in an image horizontally. The top shows the distance in centimetres. Green squares can be recognised, red squares cannot be recognised.*

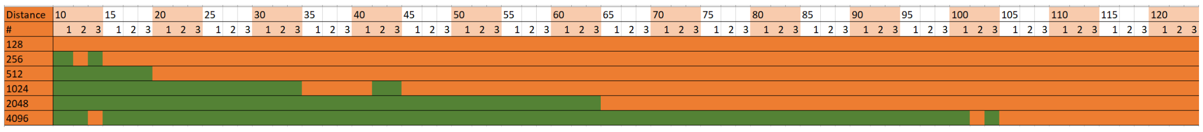


Figure 6.4: *Resolution of modified image with HSV intensity in relation to distance of camera to barcode. The resolutions on the left show the amount of pixels in an image horizontally. The top shows the distance in centimetres. Green squares can be recognised, red squares cannot be recognised.*

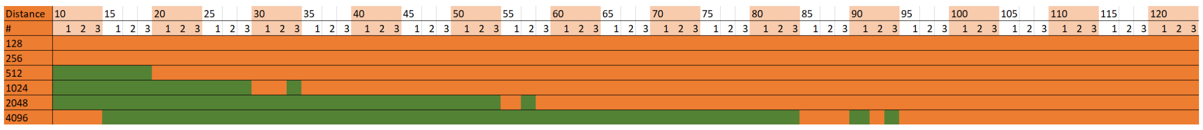


Figure 6.5: *Resolution of modified image with doubled HSV intensity in relation to distance of camera to barcode. The resolutions on the left show the amount of pixels in an image horizontally. The top shows the distance in centimetres. Green squares can be recognised, red squares cannot be recognised.*

7 Ethical Implications and Design Choices

Throughout the course of this project there were three questions we had to ask ourselves since they could have ethical implications. We will discuss these below and explain how we dealt with them.

Is flying a drone through a warehouse where people work safe?

The safety requirements tell us that the drone has to fly in the middle of the hallway between the racks. Also, humans are not allowed to be in the same hallway as the drone at the same time. Besides from that, the actual flight is out of scope for this project, and thus is not relevant for us. The drone flying in the middle of the hallway, however has implications on the distance where we record from.

Is there private information in the barcodes? If so how do we deal with that?

The barcodes in the warehouse contain information regarding where the product is from and where it is headed. This however should not be an issue, as the result from our analysis of a video feed will only be kept internally in the warehouse.

Are people being recorded while we film the racks? How do we deal with that if that is the case?

In the hallway where the drone is flying there should be no people due to safety regulations. If somehow a person is recorded through an empty rack for example, this should not result in a problem as the videos will only be kept internally at the warehouse where these were recorded.

8 Software Improvement Group

During the course of the project, there were two instances where we had to send our code base to the Software Improvement Group, or SIG for short (Software Improvement Group, 2020). The SIG provides feedback for your code based on the results of an analysis that is performed by them. The feedback we received can be found in appendix G. The main two focus points were unit complexity and unit size. The former one flags on functions that carry too many different responsibilities, and the latter checks the actual amount of lines of a function. Initially, our project had a few very long functions that ran the whole program. We processed the feedback by breaking up all of these so-called god functions into smaller chunks that carry only one or two responsibilities. We did this not only for all the files that we had in week six of the project, but we tried to keep up this standard for all the code that was introduced afterwards as well.

Other than size and complexity, we got a small bit of feedback regarding our test scores. The SIG stated that it would be good if we tried to crank up our coverage a little bit more. After the sixth week, we have attempted to unit test every applicable bit of code, nonetheless, not everything can be tested this way. In chapter 10 we will discuss why some of our modules are difficult to unit test and what other steps we took to validate these modules.

9 Conclusions

In this chapter we describe the conclusions we came to in relation to our most important requirements, as well as a general conclusion of where we currently stand with our product.

Can we detect barcodes from a distance of 0.5 meters?

Yes.

Our cropping algorithms can detect barcodes as long as the individual bars are clearly visible and the parameters are set to the right values in the YAML files. Different use cases require a different set of parameters because the relative distance between the black bars increases when the camera gets closer. To give an example: from a distance of 2 metres, the individual bars are so close together that they can easily be grouped. When the camera is only 20 centimetres away, you need to increase the size of the kernel used in the morphological operations in order to close the gap between the bars.

The CNN can also detect the barcodes with high accuracy, but it will need additional training for different use cases since the algorithm is not size invariant. This module is rotation invariant however, while our other cropping algorithms are not.

Can we interpret barcodes from a distance of 0.5 meters?

Yes, to some extend.

When using the footage from a high quality camera with one of our rotation algorithms and pyzbar, we can interpret a barcode correctly up to a distance of 1 meter. However, this is under the most ideal circumstances, meaning that the camera is placed on a steady surface, the lighting conditions are good and the background is not noisy. Our experiment with the videos from *Kuehne + Nagel* where we tried to replicate the behaviour of a flying drone resulted in a much lower accuracy. We can still detect half of the barcodes in the video, but that is at a distance of 0.3 to 0.5 meters. Theoretically we could increase the resolution of the image to 8k or higher, since our results in section 6.1 suggest that this would increase our effective distance. This is however not favourable since the price of these cameras increases exponentially.

Can we deploy our program to the warehouse in its current state?

No.

There are several reasons for this. The main ones are the the actual size of the hallways in the warehouse; the safety requirements and the lack of communication with the other components of the drone. The hallways are at least 3 or 4 metres in width and according to the safety regulations, the drone would have to fly in the middle of these hallways. This implies that the video footage will be taken from a distance of about 2 metres. To put that into perspective, figure 9.1 shows clearly how fragmented a barcode becomes from this distance. This is amplified by the fact that the barcodes in the warehouse are even more dense than the one used in this figure. Figure 9.2 shows that same barcode from a distance of 10 centimetres.

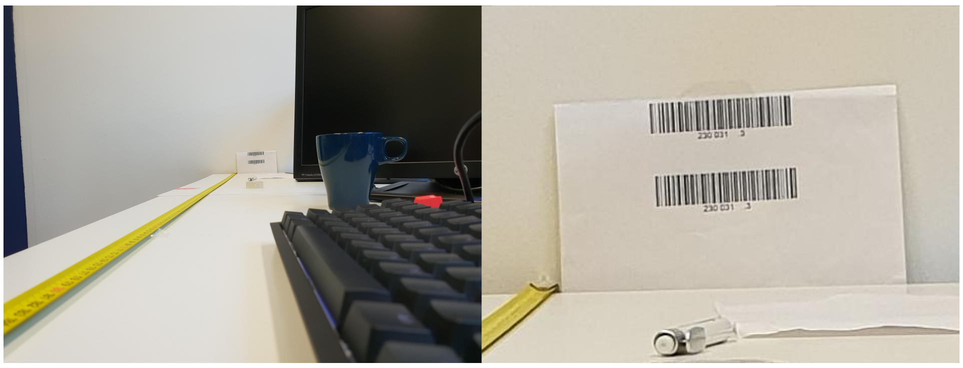


Figure 9.1: *The image on the left shows a barcode from a distance of 2 metres. The image on the right shows this same barcode, but cropped so the image only shows this barcode. One can clearly see the blurriness in the image on the right.*

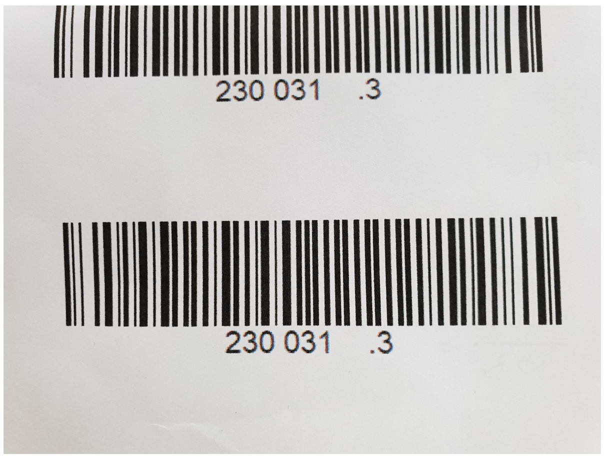


Figure 9.2: *The same barcode as in figure 9.1, but now from a distance of 10 centimetres.*

In its current state, the program solely focuses on input directly received from the camera. However, for a more accurate localisation, it will need to communicate with the other components of the drone, of which the LIDAR is the most important one. As explained in chapter 5.5, we find many false positives when relying only on 2D images for detecting certain objects, the orange beams in this example. However, we can fuse that data with the LIDAR data to add an extra sense of depth.

10 Discussion

In this section we will look back at the initial requirements that we documented at the start of the project. We will also discuss the validation of the complete system in terms of the validity of the output and that of the actual code. In the end we will reflect on the development process and how well the team functioned over the past ten weeks.

10.1 Realisation of Requirements

In order to determine if the finalisation of our project can be called successful, we need to revise our initial requirements from appendix B. Requirements 1 through 8 make up the *must haves*; these are the features that have to be in the final program in order to call the project a success. Requirements 1, 3, 4, 5 and 6 have been fully realised and do not need any further elaboration. Requirement 2 has been partially realised. It states that the computer vision algorithm must detect the horizontal orange bars of the *current* rack on a clear image. Whilst, strictly speaking, we can detect the current orange bar without any problem, we cannot make the distinction between any orange bar of this shelve and an orange bar of another shelve. As explained in chapter 5.5, this task proved to be more complicated than we had initially anticipated. Halfway through the project we discussed this issue with our client, and we came to an agreement to change this requirement to a could have, since LIDAR connection seems to be a necessity and checking for nearby bars is significantly easier using LIDAR. Requirements 7 and 8 are also partially realised, but this discrepancy stems from the same problem. The program can count the barcodes that we detect just fine, however, because of the lack of localisation that the orange beams would provide, we can't document the location of the products with high accuracy. The same applies to requirement 8. We can store frames from which the interpreter could not derive a result. However, doing so without a positional indicator would not be very helpful. We have tried to implement various different techniques for error handling (see chapter 5.5) but none have proven to be robust enough at this moment. Our current implementation of error handling is not very robust, and its use is quite limited.

Continuing on to the should haves, numbers 10 and 11 have been realised completely. The user interface can draw lines and boxes to represent the findings of the algorithms, and we even have a option to display the returned images for every algorithm. Regarding extensibility, 5.2 holds an in-depth explanation on our module generation script. Requirement 12 was not achieved. At the start of the project we had a theory that enhancing an image would increase the interpretation accuracy. However, our tests always showed that this was not the case. Zbar is state of the art technology in the field of barcode recognition. What we did achieve was make Zbar rotation invariant by the introduction of our own rotation modules. Requirement 13 states that we should have a 70% interpretation accuracy for the barcodes in the warehouse. Our actual results depend a bit on the context. For plain images like the ones that we took during our visit to *Kuehne + Nagel* (appendix D), the results are somewhat disappointing. We can correctly interpret barcodes in about 30% of all the images we took. For video footage, the results are a lot better. This is because the video footage was shot from a distance of 30cm, while the image distances ranged from 30cm to 1m. Also, in the video footage, the same barcode is in view for several seconds. This gives the program enough opportunity to detect the barcode most of the time.

Requirements 14 and 15 are the could haves. Although we did experiment a bit with error handling and OCR, these features did not make the final program for their lack of robustness.

10.2 Validation of the System

Regarding the validation of the system, there is plenty room for improvements. The barcodes that we log currently do not undergo any form of validation. We could choose to check whether the decoded barcode strings adhere to the general pattern of barcodes in the warehouse. Nevertheless, this would take away from the extensibility that we strive to achieve. The best option for validation would be to compare the decoded strings to the warehouse database. Currently we do not have access to this database, but this is a future requirement when our product would be deployed for production.

The validation of the system has been realised by both manual testing and unit testing. The core functionalities of our program have been unit tested. These include the main script with the module pipeline, the benchmark tool and the custom interpreter. The pipeline modules have mostly been tested manually. All the modules work with variable input and they have numerous parameters that can be tweaked in accordance with the use case. For this reason they are difficult to validate with unit tests. We did do some extensive work on the debug output for all the modules. During execution, the program will draw lines and contours where barcodes have been found. You can also set the DEBUG parameter for each algorithm to true, this will display all the resulting images in a separate window during run time.

For the user interface end-to-end tests have been developed. These tests will start an instance of the interface and check that it started correctly and without errors. After this it will select a test video, run the program, and check whether is program was able to correctly interpret the barcode from this video.

10.3 Reflecting on the Development Process

As with any freshly put together team, starting a project and cooperating is not a trivial task. In the first week we had some difficulty deciding on how to start and how to divide the tasks. Fortunately this phase did not take long and before we realised, we had fully integrated as a team. In the remaining nine weeks of working together, we never had a problem assigning all the tasks or falling behind schedule. Following the rules we set for ourselves in chapter 2, we met four times a week at Eonics HQ. We made sure that 80% of the group members would always be present. Most of the times we would do a brief stand-up meeting (whilst seated) in the mornings where everyone would inform the rest of the team of their current state. On the Mondays, we also used this time slot to formulate our weekly agenda. We made sure to plan a meeting with our client every week and one with our coach every other week. During these meetings we would demo our product and discuss our plans for the upcoming weeks.

In conclusion, the team worked together as a solid unit. Everyone truly had equal contributions to the final product. We could have tried to adhere to the guidelines of SCRUM a bit more strictly, but overall our workflow was agile and efficient.

11 Further Research and Recommendations

The solution that our project proposed is still a proof of concept, and it can be improved in several ways. Here are fields of our project which can still be improved.

11.1 Camera with Zoom Lens

For this project, we have used various techniques to detect barcodes in images. However, at some point, an image cannot be programmatically enhanced any further. One solution for this problem is to use an zoom lens to enlarge the barcode. You could for instance use two cameras; one camera is for detecting barcodes and the other is for zooming in on the barcode. The first camera cannot decode the barcodes because they are too far away. This is where the second camera comes into play. This camera is connected to a pan/tilt mechanism and contains a zoom lens. When the barcode is detected on the first camera, the location in the image can be send to the second camera. This camera points to the location and is able to detect the barcode with the help of the zoom lens. These cameras have to be mounted as close as possible to make the pointing of the second camera precise.

11.2 Mounting on Different Vehicles

The main idea of this project was to mount the project on an autonomously flying drone. Another idea could be to attach it to a forklift truck. A big advantage of this method is the energy efficiency. It does not need to be kept in the air all the time. It also would not need a separate battery, because it could just tap into the big battery from the forklift truck. Because the fork of the truck would constantly be going up and down, it would simulate the up and down motion of the autonomous drone. The last advantage of mounting to a forklift is that the forklift truck is driving constantly through the warehouse, a drone can only fly when there are no people around.

11.3 Detecting the Contents/Amount of Cargo

We could use the video feed for purposes other than detecting barcodes. Goods itself could be detected on the shelves, using object recognition. These objects could then also be counted. This information can then be used to compare against the Database Management System.

Because this needs a very different approach than our project, it was not possible to include this in our project.

11.4 LIDAR Connection

A LIDAR (LIght Detection And Ranging) is a tool that uses laser beams to visualise the distance to different objects (geosciences institute, 2019). A LIDAR is used on the drone to detect obstacles and to get a sense of its surroundings. A LIDAR could be used in our project to get the distance to the barcode, and adjust our cameras to be optimal at that distance. On top of that it could be used in combination with the orange beam detection to handle error handling. You can find one such LIDAR in figure 11.1.

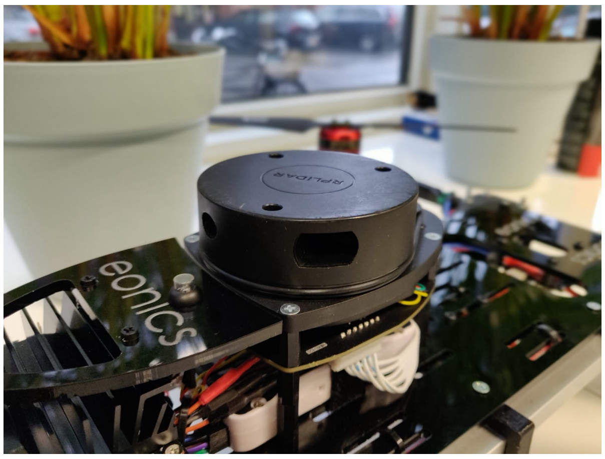


Figure 11.1: *A LIDAR, as described in section 11.4. This specific LIDAR is the RPLIDAR-A1 by SLAMTEC (2019).*

11.5 Neural Network Image Enhancements

Neural Nets have been trained to combat motion blur and other image flaws (SeungjunNah, 2019). Neural Networks also have been reliable in the past in repairing images, one could look into a neural net "repairing" a photo of a barcode into a binarised image, training it on an photograph of a barcode as input and its respective generated barcode as output. We have not actively looked into image-repairing Neural Networks during the project. Nonetheless, it may be an interesting topic for research.

Bibliography

Bradski, G., & Kaehler, A. (2008). *Learning opencv: Computer vision with the opencv library.* " O'Reilly Media, Inc."

Clegg, D. (1994). *Case method fast-track: A rad approach (computer aided system engineering).* Addison-Wesley. Retrieved from https://www.xarg.org/ref/a/020162432X/

Coleman, D. (2020). Can you zoom in on a gopro? Retrieved January 23, 2020, from https://havecamerawilltravel.com/gopro/gopro-zoom/

Creusot, C., & Munawar, A. (2015). Real-time barcode detection in the wild. In *2015 IEEE winter conference on applications of computer vision, WACV 2015, waikoloa, hi, usa, january 5-9, 2015* (pp. 239–245). doi:10.1109/WACV.2015.39

Electron. (2020). Build cross platform desktop apps with javascript, tml, and css. Retrieved January 23, 2020, from https://electronjs.org/

geosciences institute, A. (2019). What is lidar and what is it used for? Retrieved January 29, 2020, from https://www.americangeosciences.org/critical-issues/faq/what-lidar-and-what-it-used

Google. (2020). Tesseract ocr. Retrieved January 27, 2020, from https://opensource.google/projects/tesseract

GoPro. (2020). What is hypersmooth? Retrieved January 23, 2020, from https://gopro.com/help/articles/block/what-is-hypersmooth

Hansen, D. K., Nasrollahi, K., Rasmussen, C. B., & Moeslund, T. B. (2017). Real-time barcode detection and classification using deep learning. In C. Sabourin, J. J. Merelo Guerv'os, U. O'Reilly, K. Madani, & K. Warwick (Eds.), *Proceedings of the 9th international joint conference on computational intelligence, IJCCI 2017, funchal, madeira, portugal, november 1-3, 2017.* (pp. 321–327). doi:10.5220/0006508203210327

Harris, C., & Stephens, M. (1988). A combined corner and edge detector. In *In proc. of fourth alvey vision conference* (pp. 147–151).

Internet Engineering Task Force. (2005). Common format and mime type for comma-separated values (csv) files. Retrieved January 23, 2020, from https://tools.ietf.org/html/rfc4180

Internet Engineering Task Force. (2017). The javascript object notation (json) data interchange format. Retrieved January 23, 2020, from https://tools.ietf.org/html/rfc8259

Karn, U. (2016). An intuitive explanation of convolutional neural networks. Retrieved January 23, 2020, from https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Katona, M., & Nyúl, L. G. (2013). Efficient 1d and 2d barcode detection using mathematical morphology. In C. L. L. Hendriks, G. Borgefors, & R. Strand (Eds.), *Mathematical morphology and its applications to signal and image processing, 11th international symposium, ISMM 2013, uppsala, sweden, may 27-29, 2013. proceedings* (Vol. 7883, pp. 464–475). Lecture Notes in Computer Science. doi:10.1007/978-3-642-38294-9_39

Matas, J., Chum, O., Urban, M., & Pajdla, T. (2004). Robust wide baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, *22*, 761–767. doi:10.1016/j.imavis.2004.02.006

Natural History Museum. (2019). Naturalhistorymuseum/pyzbar. Retrieved January 23, 2020, from https://github.com/NaturalHistoryMuseum/pyzbar

Oostendo. (2016). Oostendo/python-zxing. Retrieved January 23, 2020, from https://github.com/oostendo/python-zxing

OpenCV. (2019). Retrieved January 29, 2020, from https://opencv.org/about/

Pattern Recognition and Image Analysis Group. (2015). Muenster BarcodeDB. Retrieved January 23, 2020, from https://www.uni-muenster.de/PRIA/forschung/index.shtml

Pccoronado. (2020). Rectifier (neural networks). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#/media/File:Rectifier_and_softplus_functions.svg

Pratt, W. K. (2007). *Digital image processing: Piks scientific inside*. Wiley-Interscience. Retrieved from https://www.xarg.org/ref/a/0471767778/

Prechelt, L. (2000). An empirical comparison of C, C++, Java, Perl, Python, Rexx and TCL. *IEEE Computer*, *33*(10), 23–29.

Python. (2019). Tkinter. Retrieved January 23, 2020, from https://wiki.python.org/moin/TkInter

Redmon, J. (2013–2016). Darknet: Open source neural networks in c. http://pjreddie.com/darknet/.

Redmon, J., & Farhadi, A. (2016). Yolo9000: Better, faster, stronger. *arXiv preprint arXiv:1612.08242*.

Rodrigues, A. (2020). H.264 vs h.265 — a technical comparison. when will h.265 dominate the market? Retrieved January 23, 2020, from https://medium.com/advanced-computer-vision/h-264-vs-h-265-a-technical-comparison-when-will-h-265-dominate-the-market-26659303171a

Schwaber, K., & Beedle, M. (2002). *Agile software development with scrum*. Prentice Hall Upper Saddle River.

SeungjunNah. (2019). Deepdeblur_release. Retrieved January 27, 2020, from https://github.com/SeungjunNah/DeepDeblur_release

SLAMTEC, (2019). Rplidar-a1 360°laser range scanner. Retrieved January 27, 2020, from https://www.slamtec.com/en/Lidar/A1

Software Improvement Group. (2020). SIG | Getting software right for a healthier digital world. Retrieved January 23, 2020, from https://www.softwareimprovementgroup.com/

Spider, C. (2018). The future of warehouse technology: Drones. Retrieved January 27, 2020, from https://clearspider.net/blog/warehouse-technology-drones/

Trello. (2020). Retrieved January 23, 2020, from https://www.trello.com/

University of Washington. (2020). How bar codes work. Retrieved January 23, 2020, from https://courses.cs.washington.edu/courses/cse370/01au/minirproject/BarcodeBattlers/barcodes.html

Upasani, S. S., Khandate, A. N., Nikhare, A. U., Mange, R. A., & Tornekar, R. (2016). Robust algorithm for developing barcode recognition system using web-cam. *International Journal of Scientific & Engineering Research*, *7*, 82–86.

W3C. (2008). Extensible markup language (xml). Retrieved January 23, 2020, from https://www.w3.org/XML/

Xu, L., Kamat, V. R., & Menassa, C. C. (2017). Automatic extraction of 1d barcodes from video scans for drone-assisted inventory management in warehousing applications. *International Journal of Logistics Research and Applications*, *21*(3), 243–258. doi:10.1080/13675567.2017.1393505

A Interview

The questions for the meeting with the project owner will be as follows:

- Can our coach and coordinator from the TU Delft have access to our repository for grading purposes?
  `Yes. It is okay to share with coordinators, students, et cetera. It is not okay to make the code available to the public.`
- When would you have time to see our final presentation? Would probably take place in last week of January.
  `I'll be there. Check my calendar for available times and plan accordingly.`
- Would the drone be flight-ready by the end of December / start of January?
  `It should be. I have an appointment where it should be able to deliver a pie somewhere. To have the drone placed on an radio controlled car would look stupid. Therefore it should be able to fly again in reasonable time.`
- Would it be possible to test in a warehouse after the Christmas Holidays? Moreover do you think we'll be allowed to gather video and photo material.
  `Yes. However, you will need to be accompanied by the Kuehne + Nagel Innovation Manager Martin Does. Once there you'll have to ask them if you are allowed to take pictures, but I think you should be fine.`
- If we require new hardware, such as a different camera, what would the budget be?
  `Eh, yeah... Around a thousand euros. Just come to me and I'll make a decision. Rather have a more robust solution than a solution where the drone needs to fly close to 30cm to the shelving.`
- Does the streaming of the video input to the off-site module have to happen live or can the data be transferred when the drone is done?
  `This is done afterwards. I think there are far better techniques for [being able to fly around autonomously], such as LIDAR and downward facing distance sensors. In the video feed you get after the flight you should be able to do inventory management based on the location of the barcodes and the orange bars. The drone however will not need this to be able to fly around. This is beyond the scope of your project.`
- Does comparing the results of the vision module with the database fall within the scope of our project?
  `Nope, that does not fall in your scope. It's boring and mainstream software development. We can get someone else to do this later.`

- Does combining the vision module with the autonomous flying drone fall within the scope of our project?
  `No. I would rather have the drone fly around without using the camera feed for navigation.`

- We're now testing on the EAN-13 (barcode type) data set. Would it be possible to generate a CODE-128 data set in the warehouse?
  `Yes. That is possible, but you will have to ask Innovation Manager Martin Does if you are allowed to record videos or take pictures.`

B Requirements

We will document our requirements according to the MoSCoW method. This is a way of displaying your requirements in order of priority. The document is split up in three parts: must haves, the requirements that need to be implemented for the project to succeed as a whole; should haves, extra requirements that should be implemented, but can be omitted with a proper explanation; could haves, bonus requirements that would be nice to have if there is enough time left near the end of the project. We will refer to the barcodes on pallets as "inventory barcodes" and to barcodes on the shelving as "localisation barcodes".

Must haves

1. The development process must be agile and follow the SCRUM methodology

The computer vision algorithm...

2. must detect the horizontal orange bars of the current shelve on a clear image
3. must detect localisation barcodes on an image where the individual bars of a barcode are all clear
4. must detect inventory barcodes on an image where the individual bars of a barcode are all clear
5. must be able to correctly interpret barcodes from photos in the warehouse taken with a phone camera from a distance of half a meter.
6. must correctly interpret barcodes from the WWUMuenster Barcode Database data set with an accuracy of 85%.
7. must count the inventory based on the data from the localisation barcodes and inventory barcodes combined
8. must log detected, unrecognised barcodes to a separate file

Should haves

The computer vision algorithm...

9. should prioritise accuracy over efficiency

10. should visualise the algorithms' findings by writing over the video feed
11. should be extensible, new object detectors should be relatively easy to add.
12. should correctly interpret barcodes from the WWUMuenster Barcode Database data set with a higher accuracy than the current pyzbar algorithm on the same set (85%), see table 3.1).
13. should correctly interpret barcodes in the warehouse with an accuracy of at least 70%. The lighting conditions of the warehouse and movement of the drone are taken into account.

Could haves

The computer vision algorithm...

14. could have an interface that displays all the unrecognised barcodes
15. could use OCR to interpret the text under a barcode as a fallback for unrecognisable barcodes.

C Project Plan

C.1 Project Description

Eonics is constructing their own drone that will eventually be deployed in a large warehouse. It is difficult to maintain proper logistics in warehouses of this scale, and as a result items often go missing with no easy method to trace said items. As the situation is now, employees have to go and pass all the racks to manually count each and every pallet. This is quite inefficient and prone to error. Eonics wants to use a drone to take over this bit of labour. Every rack is labelled with a barcode that holds three types of information: the hallway identifier, row number and column number. Every rack also holds pallets that are labelled with barcodes. These barcodes contain the item identifier used by the warehouse, country of origin, production date, batch number, etc. The goal is to analyse these barcodes and cross check the data with the inventory system of the warehouse to verify that all items are in the correct locations. This does not have to be done live. The video feed will be sent to an off-site module where the calculations can be performed.

C.2 Division of Tasks

The drone needs to be able to count the inventory, the required information can be extracted from the barcodes. The analysis of the recorded video feed can be done off-site. This allow for the usage of more powerful hardware which allows for the use of more robust algorithms that would be to draining for the drone.
We decided that the best way to divide the tasks is to have every team member working on different implementations for barcode detection and compare their robustness. After this initial phase, we will pick and combine algorithms based on our acquired experience and develop our final barcode detector. Because the off-site module will have better hardware to work with and doesn't face the concerns of a limited battery life, the team is able to introduce more expensive image processing algorithms that the drone would not be able to run. The main focus here is to reduce the error rate in order to get the most accurate inventory count possible.

C.3 Meetings with the Coach

Given that our coach does not have any prior experience with autonomous drones or computer vision, the amount of technical input we need will be limited. For this reasons we suggest bi-weekly meetings with the coach. The exact time and day will be discussed with both parties after every meeting.

C.4 Sprint Meetings with the Product Owner

The CEO of Eonics set as one of his requirements that the project should be managed following the SCRUM methodology. Together we decided to have at least one meeting each week accompanied with a small demo. The exact day of this meeting depends on the agenda of the product owner and the start or end of the week would be preferable.

C.5 Global Planning

In table C.1 a general overview per week can be found.

Table C.1: *This table includes a general overview of what we want to accomplish each week*

Week	Agenda
1 (11-11 / 17-11)	- Setup HyperDev accounts - Make project plan - Find researches - Start on research paper
2 (18-11 / 24-11)	- Finalise research paper - Start getting familiar with OpenCV - Try out different implementations for barcode detection
3 (25-11 / 1-12)	- Work on the barcode detection algorithm
4 (2-12 / 8-12)	- Finalise barcode detection algorithm - Start working on barcode interpretation
5 (9-12 / 15-12)	- Keep working on barcode interpretation modules
6 (16-12 / 22-12)	- Finalise barcode interpretation modules
7 **Christmas**	//
8 **Holidays**	//
9 (6-1 / 12-1)	- Optimisation - Additional features - *On-site testing in the warehouse*
10 (13-1 / 19-1)	- Optimisation - Additional features - *Connect with the hardware team and see if system integration and testing is feasible*
11 (20-1 / 26-1)	- Optimisation & finalisation
12 (27-1 / 2-2)	- Finish final report, presentation & demo

D Action Plan for Kuehne + Nagel visit

On Tuesday, December 10, 2019, we have arranged a visit to one of the warehouses of Kuehne + Nagel in Utrecht, The Netherlands. Apart from learning more about the environment that our software will run in, we have determined two main objectives that we want to achieve with this visit.

First of all we want to create four different data sets of around 300-600 images. Each data set will display barcodes on products or beams from varying distances and rotations. Each individual data set will be shot with a different camera. These camera's are the Pi Camera v2.1, the Nikkei Extreme X4, the Nikkei Extreme X6 and the GoPro HERO 7. For an elaborate analysis on the camera's, see chapter E.1. Because of time constraints, we will always take *six* images of the same barcode; *three* from a distance of 30-50cm and *three* from a distance of 100-150cm. Each set of three will have a straight image, one rotated to the left and one rotated to the right.

Our second objective is to create a 10 minute video with each of the camera's mentioned above. For these videos we will try to simulate the behaviour of the drone. To do this, we will mount the camera to a selfie stick and film the racks following the flight pattern of the drone.

E Choosing a Camera

To be able to read and interpret barcodes correctly having good images or a good video stream is very important. The camera needs to be able to zoom so the drone does not have fly very close to the racks. The camera also needs to deal with the movement of the drone and the vibrations of its propellers/engine because blurry images will make it that much harder to successfully interpret and detect barcodes. Which is why we will need to select a camera that can make good images or good a video while the drone if flying around, this is why image stabilisation for the camera is essential. It needs to be very light because otherwise it will slow down the drone and decrease the battery life of the drone. The frame rate needs to be good(at least 5-10) and alterable so we can experiment with what would suit our needs the most(lots of pictures but with increased accuracy and slower processing time or fewer pictures with less accuracy but faster processing time). Angle of view is something we have looked into and the wider this is the closer the drone could fly to the rack where the camera can still see all the items. Battery life of the camera also plays a big role for choosing the correct camera, the battery life needs to be at least comparable to that of the drone to ensure maximum up-time. The resolution is something we would like test various resolutions but in general the higher the better should apply because there will be more detail in the pictures from which we can extract data. The product owner told us that our budget for a camera is €1000 but that does not mean we need to spend it all, the goal is to find the camera that satisfies all our needs but is still the cheapest. File format is something we would like to experiment with if there are more multiple available, which is why we will take it into account when deciding upon a camera.

In table E.1 we compare various cameras at the hand of the above described criteria to determine whether said camera fulfils our needs. We will now elaborate on different concepts written down the table. Digital zoom is done by narrowing on the centre of a frame. It takes the existing image data and narrows down on it. Might look sharper and closer but it has the same image information and therefore is not as sharp. (Coleman, 2020). Frame rate and battery life for some of the cameras have varying values but these correspond to each other. Higher frame rate means that the battery drains faster. Frame rate also corresponds to the resolution, 4k resolution generally cannot be done with higher than 60 frames per second. The difference between MPEG-4 and MPEG-H is that a different video compression standard is used for MPEG-H which reduces the storage required by 50%. (Rodrigues, 2020) We did not compare any GoPro Hero versions before 7 because 7 came with the time-lapse option, which allows you to take a few pictures a second for a slow-motion video(which we think could be very useful for us). On top of that it came with significantly improved in-camera electronic stabilisation, this is called HyperSmooth (GoPro, 2020). Any version before GoPro Hero 6 does not have zoom options which is also a must for our camera.

Table E.1: *Comparison of 9 different cameras, based on 9 different topics.*

	GoPro HERO8	GoPro HERO7	Salora ACE900	HDR-AS50	Vizu Extreme X8S	Sony CyberShot DSC-RX0 II
Zoom options	Digital zoom x2	Digital zoom x2	?	Digital zoom x3	Digital zoom x4	Digital zoom x4
Image Stabilisation	Yes	Yes	Yes	Yes	Yes	Yes
Weight	126 g	116 g	52 g	60 g	60 g	132 g
Frame Rate(alterable?)	Yes, 30-120	Yes, 30-120	Yes, 30-120	Yes, 25-120	Yes, 30-60	Yes, 24-120
Angle of View	149°	149°	170°	170°	170°	84°
Battery Life	47-81 minutes	47-90 minutes	120 minutes	660 minutes	90 minutes	60 minutes
Resolution	4k-1080p	4k-720p	4k-720p	1080-720	4k-720p	4k-1080p
Price	€429.00	€339.00	€94.95	€149.00	€75.00	€799.00
File Format	Film: MPEG-4, MPEG-H, MP4 Picture: PNG, JPEG	Film: MPEG-4 Picture: JPEG, RAW	Film: MPEG-4, MOV Picture: JPEG	Film: MP4 Picture: JPEG	Film: MPEG-4 Picture: JPEG	Film: MPEG-4 Pictures JPEG, RAW

	Nikkei Extreme X6	Nikkei Extreme X4	PiCamera V2.1
Zoom options	?	?	Yes
Image Stabilisation	Yes	No	No
Weight	60 g	62 g	3 g
Frame Rate(alterable?)	Yes 30-120	No, 30	Yes, up to 120
Angle of View	170	170	49-62
Battery Life	90 minutes	70 minutes	-
Resolution	4k-720p	1080-720p	1080-480p
Price	49	49	30
File Format	Film: MPEG-4 Picture: JPEG	Film: MPEG-4 Picture: JPEG	Film: MPEG-4 Picture JPEG, JPEG + RAW, GIF, BMP, PNG, YUV420, RGB888

F Benchmarks

To test our algorithms, we use the `WWU Muenster Barcode Database` data set by the Pattern Recognition and Image Analysis Group (2015). This data set contains over a thousand images in which all the barcodes are oriented perpendicular to the horizontal camera axis. The algorithm we use to interpret the barcodes is pyzbar. In table F.1 we have documented the recognition accuracy of pyzbar on 4 different image channels (RGB, red, green, blue) and on the greyscale images. The first column displays the result on the regular data set, while for the other columns, a seeded random rotation between -80 and 80 degrees was applied to every image before processing. It was necessary to use seeds in order to compare the results with our own enhancement algorithms. From the table you can clearly conclude that pyzbar is not rotation invariant, in fact, applying a rotational transform to the data set decreases the overall accuracy by about 28-30%. We have adapted two different algorithms that can detect barcodes in an image, and correct that image for the rotation. The first one is based on a combination of Harris corner detection and Hough line transform. The second one uses MSER regions to detect the black bars of the barcodes. Using the same random seeds, we first ran the rotated images through these correction algorithms and then we processed them with pyzbar. The results from Harris and Hough showed an increase in accuracy of about 20-23%. Although this result is quite significant, it is not yet equal to the control. The MSER algorithm performed even better. It achieved an accuracy increase of around the 28%, basically equal to control and for some channels even slightly better. From this data we can conclude that our implementation of the MSER rotation correction algorithms increases the recognition accuracy from the pyzbar library by a compelling amount of 28% by making its results practically rotation invariant. If we also take the speed of the two algorithms into consideration, the Harris and Hough algorithm comes out on top. Processing all 1050 images takes around 122 seconds on average, while processing the same amount of images with MSER takes 212 seconds, an increase of almost 75%.

Table F.1: *Barcode recognition accuracy on (rotated) test set*

	Not Rotated	Seed 0	Seed 1	Seed 2	Seed 3	Seed 4
pyzbar Colour	83.32%	55.83%	55.83%	54.88%	55.07%	56.68%
pyzbar Red	85.5%	57.25%	58.29%	56.78%	56.97%	**60.19%**
pyzbar Green	85.31%	**57.63%**	**59.05%**	57.16%	56.97%	**60.19%**
pyzbar Blue	83.32%	55.83%	55.83%	54.88%	55.07%	56.68%
pyzbar Gray	**85.59%**	57.44%	58.39%	**57.25%**	**57.25%**	59.53%
HH Colour	N/A	77.91%	77.73%	78.29%	78.58%	79.34%
HH Red	N/A	**81.23%**	80.57%	**81.04%**	**81.33%**	**81.52%**
HH Green	N/A	80.38%	80.09%	80.76%	81.14%	81.33%
HH Blue	N/A	77.91%	77.73%	78.29%	78.58%	79.34%
HH Gray	N/A	80.47%	**80.95%**	80.85%	81.23%	**81.52%**
MSER Colour	N/A	82.56%	81.80%	83.22%	82.75%	82.27%
MSER Red	N/A	84.36%	**84.08%**	85.40%	83.98%	84.45%
MSER Green	N/A	**84.83%**	**84.08%**	**85.59%**	84.45%	**84.55%**
MSER Blue	N/A	82.56%	81.80%	83.22%	82.75%	82.27%
MSER Gray	N/A	84.36%	83.79%	**85.59%**	**84.55%**	84.17%

G SIG Feedback (NL)

De code van het systeem scoort 3.0 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code ondergemiddeld onderhoudbaar is. We zien Unit Complexity en Unit Size vanwege de lagere deelscores als aandachtspunten waar verbetering nodig is om de score te laten stijgen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Dit betekent overigens niet noodzakelijkerwijs dat de functionaliteit zelf complex is: vaak ontstaat dit soort complexiteit per ongeluk omdat de methode te veel verantwoordelijkheden bevat, of doordat de implementatie van de logica onnodig complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is, en daardoor eenvoudiger te onderhouden wordt. Door elk van de functionaliteiten onder te brengen in een aparte methode met een beschrijvende naam kan elk van de onderdelen apart getest worden, en wordt de overall flow van de methode makkelijker te begrijpen. Bij grote en complexe methodes kan dit gedaan worden door het probleem dat in de methode wordt opgelost in deelproblemen te splitsen, en elk deelprobleem in een eigen methode onder te brengen. De oorspronkelijke methode kan vervolgens deze nieuwe methodes aanroepen, en de uitkomsten combineren tot het uiteindelijke resultaat.

Voorbeelden in jullie project:

- mser_cropping.py:find_cropped_barcodes_with_mser
- mser_rotation.py:find_rotated_barcodes_with_mser
- main.py:DecodeThread.run()

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- cnn_crop.py:CNN.detect(image)
- utils.py:transform_and_crop

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen. Op lange termijn maakt de aanwezigheid van unit tests je code flexibeler, omdat aanpassingen kunnen worden doorgevoerd zonder de stabiliteit in gevaar te brengen.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

H Info Sheet

Title of the project: Optimize computer vision module of autonomous drone
Name of the Client organization: Eonics
Date of the final presentation: 3/2/2020
Final report:

Description

Problem definition: Improve the vision module for an autonomous drone that will assist with the management of the inventory of a warehouse.
Challenge: Halfway through the project we realised that the solution we were working on was not going to work to as well as we had hoped. Therefore we had to shift part of our focus to doing research for alternate solutions that the product owner could go on once our project was finished.
Research phase: During the research phase of the project we learned various ways of isolating and interpreting a barcode in an image or in a video.
Process: The requirements changed a few times during the course of the project and we adjusted regularly to bring the project to a successful conclusion. Our scrum approach assisted a lot in dealing with this.
Product: The product we created is an interface where the user can submit video files or images and use our pipeline to perform certain actions on this(Thresholding, cropping, rotation and interpretation). The output is provided in real time as well as in JSON and CSV format.
Outlook: The program we provided is completely modular which means that adding a different cropping algorithm(or any other type) to the pipeline is very simple. At the end of the project we also did some research and made some recommendations on how to improve the product from here on out.

Members of the project team

- Mees Brinkhuis
  Interests: Robotics and embedded software.
  Contributions: Front-end, Back-end and tester
- Dirk den Hoedt
  Interests: Electronics, robotics and embedded systems.
  Contributions: Benchmarking, MSER research and implementation.
- Mike van der Meer
  Interests: Artificial Intelligence, Software Testing and Engineering.
  Contributions: Back-end developer and tester.
- Toby van Willegen
  Interests: Artificial Intelligence, Databases
  Contributions: CNN trainer, Report Proofreading and Grammar.
- Tim Yarally
  Interests: Scanning Barcodes.
  Contributions: Modularisation of Algorithms, Scrum master, Group Leader

Client
Name: Jan Peter Jansen
Affiliation: Eonics

Coach
Name: Thomas Overklift Vaupel Klein
Affiliation: Department of Software Technology

Project Contact
airwolf@tvw.me

I Project Forum Description (NL)

Binnen Eonics bouwen we aan een eigen drone die op termijn in grote warenhuizen als die van Nippon en Nagel & Kuehne `[sic]` automatisch voorraadcontroles gaat uitvoeren. De doelstelling is dat de drones 's nachts zelfstandig rondvliegen in de verschillende gangen van een warenhuis, daar beeldmateriaal verzamelen, en dit beeldmateriaal vervolgens off-line verwerken. Dit laatste gebeurt door computer vision software, die vaststelt welke voorraad waar staat en een schatting doet van de resterende hoeveelheid voorraad op een locatie.

De huidige status van het project is als volgt:

- Er is een vliegend platform dat kan worden bestuurd met een of-the-shelf controller
- Er is een module beschikbaar die de drone autonoom kan laten vliegen (o.b.v. ROS), de koppeling met het vliegend platform is 80-90% gereed
- Er is een off-line computer vision module o.b.v. Python en OpenCV, die beeldmateriaal kan verwerken en hier verschillende relevante objecten uit kan destilleren, zoals de barcode van een pallet met dozen en de oranje balk met barcode waaruit de locatie is af te leiden.

Het probleem met de off-line computer vision module is dat die alleen goed werkt onder perfecte lichtcondities en o.b.v. camerabeeld van een iPhone. De drone is echter niet uitgerust met een iPhone camera, maar met verschillende Pi-camera's, en zal zeker niet onder uniforme en perfecte lichtcondities rondvliegen.

De opdracht is als volgt:

- Onderzoek waarom de module niet werkt o.b.v. de huidige hardware opstelling (met Pi-camera's)
- Onderzoek mogelijke oplossingsrichtingen (kan zowel hardware- als softwarematig zijn)
- Optimaliseer de module zodanig dat die ook onder realistische (=complexe) omstandigheden werkt. O.a. met wisselende lichtcondities, maar ook met verschillende afstanden tot de te herkennen objecten
- Eventueel: breid de module zodanig uit dat er eenvoudig nieuwe objectherkenningsalgoritmes kunnen worden toegevoegd en toon aan dat dit werkt