PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
UNIVERSITY MOHAMED BOUDIAF - M'SILA

FACULTY: Mathematics and Computer Science
DEPARTEMENT: Computer Science
N°: ……

جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

DOMAIN: Mathematics and Computer Science
FIELD: Computer Science
SUB-FIELD: Information and Communication Technologies

A Dissertation in Fulfillment for the Requirement of the Degree of MASTER

By: Noureddine BOUABDALLAH

SUBJECT:

Design and Implementation of an Object Detection and Localization system based on Camera-IMU and sensors

Defended publicly on: 13/05/2017, to the jury:

Board of Examiners

Imad Debbi	University of M'sila	Chairman
Rached Yagoubi	University of M'sila	Supervisor
Mohamed Khoudja	University of M'sila	Co-Supervisor
Salah Guesmia	University of M'sila	Examiner

Academic year: 2016/2017

Dedicated to my parents

Acknowledgements

I would like to thank my parents for their continuous support, and my supervisors for all the help and patience during hard times. I would also like to thank Andy for helping me with some hardware problems.

Contents

List of Figures

Introduction

Since the dawn of computers and computer science, scientists worked hard to give this machine the ability to perform tasks that humans do. Some of these tasks involve having an understanding of the environment and of the objects in that environment, and as such, research on computer vision started.

The limited power of computers as well as the primitive understanding of computer science didn't prevent scientists and researchers to discover and develop methods and systems to allow the machine to recognize objects in images. That new branch of computer science kept on evolving as more and more contributions were made in the field, and with that also evolved the need concerning that type of application, like OCR, face recognition, image labelling, scene recognition, frame prediction in video, and cancer cell detection in medical images.

One of the most important goals of computer vision is automation; to allow robots to act and react in their environment. For applications like search and rescue in dangerous zones, delivery, and industrial safety.

The advancement in SoC (System on a Chip) technology have allowed for really small, cheap and powerful general purpose computers, and that motivated us to implement our project on a Raspberry Pi which is open hardware and can run a full 64-bit Linux operating system, and has sufficient processing power for our purposes, with a margin for more in the future.

Python is a powerful scripting language, with a large community and a lot of modules for different hardware accessories, and the ability to just run a script without the need for compiling it saves time and allows for faster testing and modification as needed.

Our work consists of two major parts, object detection, we chose the Algerian flag as our object of interest, and localization in the form of monocular camera-based-IMU (Visual Odometry), all working on board an SoC, we chose the Raspberry Pi 3 as the hosting chip for our system. We use the Open Computer Vision library to for the detection tasks, and the H.264 encoder motion stream to estimate the velocity and absolute coordinates on the x and y axes, all the programs we create to run on the Raspberry Pi are written in Python.

The first chapter of this dissertation will provide a general overview of the three axes of our work, we will talk first about object detection and its history and the approach we used, then there's a part about localization and its techniques.

The second chapter will be about our system and its different parts, we talk about the hardware and its specifications and the reasons for choosing that configuration, and about software, the platforms and libraries we used, as well as the development environment, the development process, and the important parts of the software.

In the third chapter, we will share the tests we performed, and discuss the obtained results from Object detection and Visual Odometry tests.

Lastly we provide a conclusion and future works.

Chapter 1

Object Detection and Localization

This chapter contains an introduction to different methods and techniques for Object detection and Localization. We take a brief look at the history of Object detection, then present the method used in our work in some detail. In section 2, we share some Localization methods and introduce the interest of our work, Visual odometry.

1.1 Object Detection

Computer vision is one of the pillars of artificial intelligence, the medium with which the intelligent agent gets to see an environment and understand it. Object detection and recognition is a tool that enables the identification of objects of interest in pictures and sequences of pictures. It is used in visual search engines, security systems, robots and UAVs, autopilots in vehicles and many more applications. There are a lot of contributions in the field of object detection, due to the high demand for faster and more robust techniques.

1.1.1 History

It is important to see how computer vision research developed, and notice the change in the way of thinking about visual problems in the era of the computer. Yang writes about the major approaches in object recognition history, and mentions three major approaches[1]:

- Geometry-based approaches: This was the earliest approach, and used geometric models to account for the variation in an object's appearance due to viewpoint, illumination and occlusion changes. The principle was that the geometric description of a 3D object would allow for an accurate prediction of the projected shape in a 2D image, as shown in Figure 1.1. Much attention was made to the extraction of geometric primitives that are invariant to viewpoint changes at the time, and many algorithms were developed, for different representations of the world from the blocks world[2], to the world of generalized cylinders[3]. These approaches suffered from the difficulty of extracting primitives, the computational complexity for certain shapes, noisy geometry, missing features, etc.

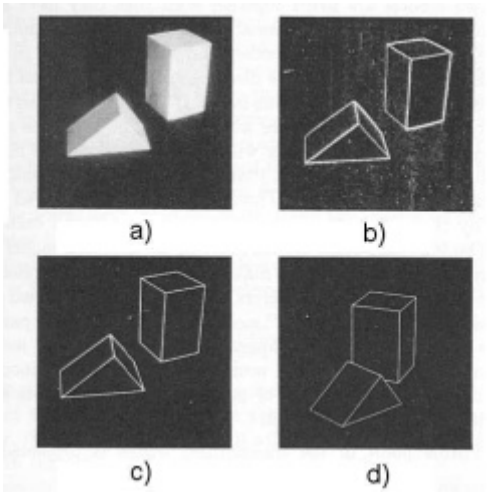

FIGURE 1.1: A representation of the Block world[2] a) A blocks world scene b) Detected edges c) A 3D polyhedral description of the scene d) The 3D scene displayed from a different viewpoint

- Appearance-based approaches: At the end of the geometric era, scientists started discovering appearance-based techniques and a lot of interest was generated around them. Most notably, the eigenface methods attracted a lot of attention as it was one of the very first face recognition systems that was relatively accurate and not so computationally complex making it more efficient, like the work of Turk $et\ al.$[4].

- Feature-based approaches: In feature-based object detection, standardization of image features and registration (alignment) of reference points are important. The images may need to be transformed to another space for handling changes in illumination, size and orientation. One or more features are extracted and the objects of interest are modelled in terms of these features. Object detection and recognition then can be transformed into a graph matching problem [5].

It can be seen that in the early days, the methods were geometrical and thinking of general shapes and their 2D and 3D transformations, and as computer science evolved so did computer vision methods, turning to finding invariant features in images. That was due to the shift in the way problems are thought about, from a human perspective to a more abstract way of thinking, because machines don't see their environment as humans do, and a better representation of the world is needed which is more optimized machine processing.

1.1.2 Feature-based approaches

Feature-based approaches are widely used in practical applications, and are still actively researched, to improve and optimize existing techniques as well as discover and develop new ones. The following are the general steps of feature-based matching, according to [6]:

1. Pre-processing of image: this step's focus is on removing unnecessary details in the image, which won't have significant information, it also allows for an increased robustness to noise. The image is converted grey-scale then to a binary or an edge image, these conversions depend on the nature of the image, and the application and its requirements. Pre-processing impacts the accuracy and time complexity of the steps that follow.

2. Feature detection: Features are structures in a scene that unique and identifiable from different viewpoints of the scene. A scene is composed of edges, corners and regions of uniform textures. A feature detection algorithm attempts to find corner points which are intersections of edges or contours of an image, because corners are the distinct point features that can be used for comparison. There are basically two types of detectors: Corner-based and Blob-based.

3. Feature description: a feature descriptor is a vector associated with a feature point of an image which will be used to find a corresponding feature point in another image. A descriptor is constructed based on the local region surrounding the point, and it can contain a global component too.

4. Point Feature correspondence: this step consists of comparing the descriptors of both images. The comparison is done for by calculating distances between two descriptors, the distance can be Euclidean or based on sign of Laplacian.

5. Region correspondence: after finding point feature correspondences, a larger comparison is done to find a whole region or scene in the image.

1.1.3 Cascade classifiers

We used a Feature-based technique called boosted cascade of weak features. And in our work, we got to know Haar[7] and LBP[8] (Local Binary Patterns) features. This technique consists of two stages: a training stage, and the detection stage.

1. Training Stage: Initially the algorithm needs a lot of positive images (images containing the object of interest) and negative images (or background images, ones that don't contain the object of interest) to train the classifier, we worked with AdaBoost learning method. The two feature types and the boosting algorithm are presented next.

- Haar Classifier: The Haar classifier uses three kinds of features. The value of a two-rectangle feature is the difference between the sum of the pixels within two rectangular regions, the regions have the same size and shape and are horizontally or vertically adjacent. A three-rectangle feature computes the sum within two outside rectangles subtracted from the sum in a center rectangle. Finally, a four-rectangle feature computes the difference between diagonal pairs of rectangles, Figure 1.2 shows a visual representation of these features.

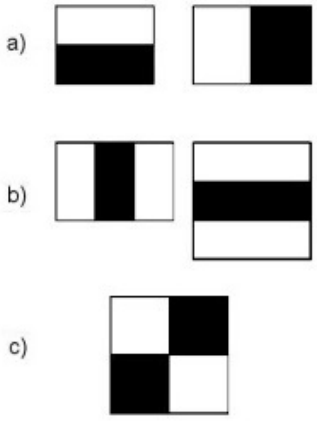

FIGURE 1.2: Haar features[7]: a) Edge features b) Line features c) Four-rectangle features

- LBP Classifier: The LBP classifier labels the pixels of an image by thresholding the-neighborhood of each pixel with the center value and considering the result as a binary string or a decimal value, usually on a 3x3 matrix, as seen in Figure 1.3. Multi-scale Block LBP[9] is an extension to the original LBP operator and differs in the matrix size, and also in MB-LBP, the computation is based on average values of sub-regions instead of individual pixels.

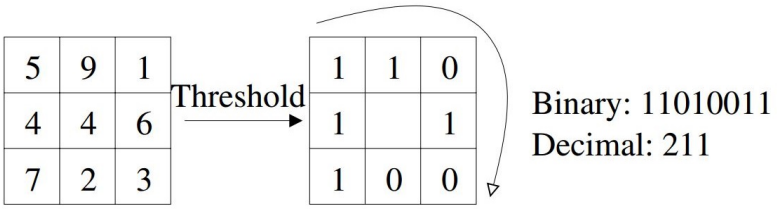

FIGURE 1.3: Example of basic LBP operator[10]

- AdaBoost: Boosting is a general method for improving the accuracy of learning algorithms, it works by combining many relatively weak and inaccurate rules[11]. AdaBoost is a boosting algorithm, introduced in 1995 by Freund

and Schapire, which was the first practical boosting algorithm, and still one of the mostly widely used and studied.

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.
Initialize: $D_1(i) = 1/m$ for $i = 1, \ldots, m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$.
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right].$$

- Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$.
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

FIGURE 1.4: AdaBoost psuedocode[11]

Pseudocode for AdaBoost is shown in Figure 1.4. Here we are given m labelled training examples $((x_1, y_1), ..., (x_m, y_m))$ where each x_i belongs in some domain X, and each label y_i is in some label set $Y = \{-1, +1\}$. On each round of the series $t = 1, ..., T$, a distribution $D_t(i)$ is computed as in the figure over the m training examples, and a given weak learner or weak learning algorithm is applied to find a weak hypothesis $h_t : X \to \{-1, +1\}$, where the aim of the weak learner is to find a weak hypothesis with low weighted error et relative to D_t. The final or combined hypothesis H is computed as a weighted majority vote of the weak hypotheses h_t where each is assigned weight at a_t.

2. Detection stage: The general architecture of multi-scale detection is as follows. An input image i is passed in, and multi-resolution images are generated (image pyramid) off of i, the classifier H then uses a cascade of weaker classifiers h_n and at each stage non-objects are rejected and the rest of the image is passed on to the next classifier, after H finishes it results in a list of tuples containing the coordinates of the boxes (rectangles) of the detected objects, along with the confidence values for the boxes.

The limitations imposed by an embedded system are still present even with the advent made in micro processor technology, and to accommodate this limitations we decided to

work with feature-based object detection because of the simpleness of its techniques, and also because of the effectiveness. Well trained classifiers can reach really high detection rates, and are not computationally taxing, which enables a real-time system to work. We also used a supervised training method because of the limited resources and time.

1.2 Localization

Localization in robotics deals with the problem of figuring out an agent's location in its environment, as well as keeping track of its trajectory. Several methods are actively used, and research is ongoing to solve the many challenges in the field. In this section we will present the major technologies and techniques.

1.2.1 Global Positioning System

GPS is a U.S.-owned technology that provides global longitude and latitude coordinates and timing, and it consists of three segments: space segment, control segment, and user segment. The space and control segments are maintained, developed, and operated by the U.S. Air Force[12].

- Space segment: it consists of a constellation of satellites transmitting radio signals to users. The United States is committed to maintaining the availability of at least 24 operational GPS satellites, 95% of the time.
- Control segment: this segment consists of a global network of ground facilities that track the GPS satellites, monitor their transmissions, perform analyses, and send commands and data to the constellation.
- User segment: consists of the GPS receiver equipment, which receives the signals from the GPS satellites and uses the transmitted information to calculate the user's three-dimensional position and time.

GPS service is available to civilian and military users. The civilian service is free continuously and around the world, while the military one is available to U.S. and their allies as well as approved government agencies.

1.2.2 Inertial Measurement Unit

The IMU is a single electronics module, it collects angular velocity and linear acceleration data which is then sent to the main processor. This unit contains two sensors, Figure 1.5 shows the two sensors, an accelerometer triad and an angular rate sensor triad[13].

- Accelerometer triad: It generates three analog signals describing the accelerations along each of its axes produced by, and acting on the vehicle.
- Angular rate triad: It also outputs three analog signals. These signals describe the vehicle angular rate about each of the sensor axes.

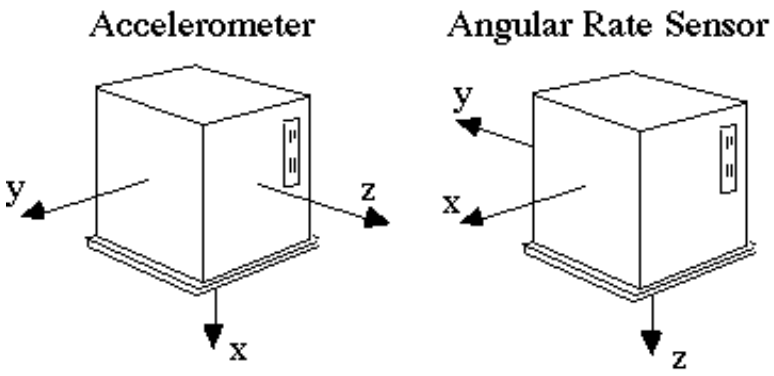

FIGURE 1.5: Initial IMU sensor coordinate axes[13]

Depending on the positioning of the IMU, some frame transformation has to be done to align the calculated angles with the angles of the vehicle in order to get correct readings, this involves converting the measurements from the IMU coordinate frame to the vehicle coordinate frame, Figure 1.6 shows the IMU orientation and the different angles.

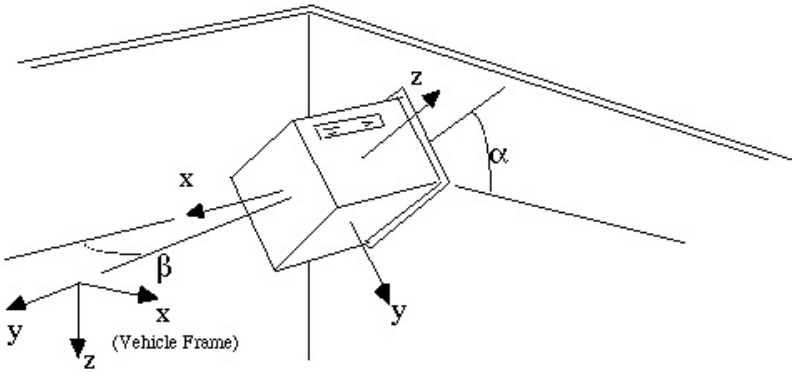

FIGURE 1.6: Transformed IMU coordinate axes compared to vehicle coordinate axes[13]

1.2.3 Visual Odometry

Visual Odometry (VO) is the process of estimating the egomotion of an agent (e.g. vehicle, human, robot) using only the input of a single or multiple cameras attached to it. The term VO was coined in 2004 by Nister in his paper[14]. The term was chosen for its similarity to wheel odometry, which incrementally estimates the motion of a vehicle by integrating the number of turns of its wheels over time. Likewise, VO operates by incrementally estimating the pose of the vehicle through examination of the changes that motion induces on the images of its onboard cameras.

In GPS-denied environments, such as underwater, aerial, indoor our underground, as well as for small movements where IMUs tend to give precise but divergent values, VO has utmost importance. And in the Visual Odometry Fundamentals book[15], it is stated that there are mainly two techniques of VO:

- Stereo: this is the most researched kind, using stereo cameras like in Figure 1.7. A lot of initial research had in common that the 3D points are triangulated for every stereo pair, and the relative motion is solved as a $3D - to - 3D$ point registration (alignment) problem. In 2004 Nister[16] proposed a completely different approach using detected features independently in all frames and only performed matches between features, and did not solve a $3D - to - 3D$ registration problem but instead did a $3D - to - 2D$ camera-pose estimation problem. Later in 2007 a different approach was proposed by Comport $et\ al.$[17], instead of using techniques similar to what we saw, they relied on a quadrifocal tensor[18], and calculated motion in $2D - to - 2D$ points in stereo pairs.

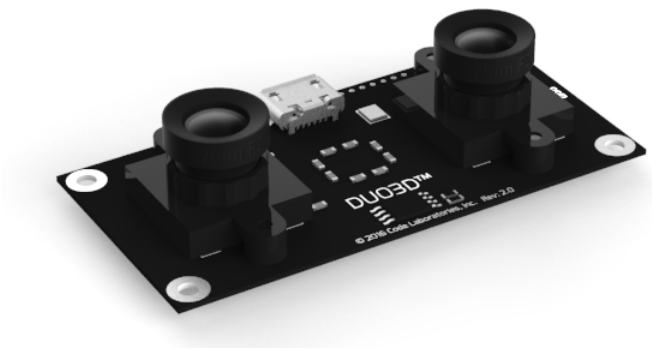

FIGURE 1.7: A stereo camera setup[19]

- Monocular: The difference from the stereo scheme is that in the monocular VO, both the relative motion and $3D$ structure must be computed from $2D$ bearing data. Since the absolute scale is unknown, the distance between the first two camera poses is usually set to one. As a new image arrives, the relative scale and camera pose with respect to the first two frames are determined using either the knowledge of $3D$ structure or the trifocal tensor. Three methods are used: feature-based, appearance-based, and hybrid methods which incorporate a combination of the previous two, a visualization of motion vectors is presented in Figure 1.8.

GPS is good for big movement and global localization of assets, but it is lacking when it comes to subtle movements and indoor applications. IMUs are widely used, and their efficiency is proven, but there are some problems that manifest as time passes, like growing error in speed estimations[15] where the divergence of values will cause instability. Our interest was to explore the visual approach using a monocular camera, a lot of interest is

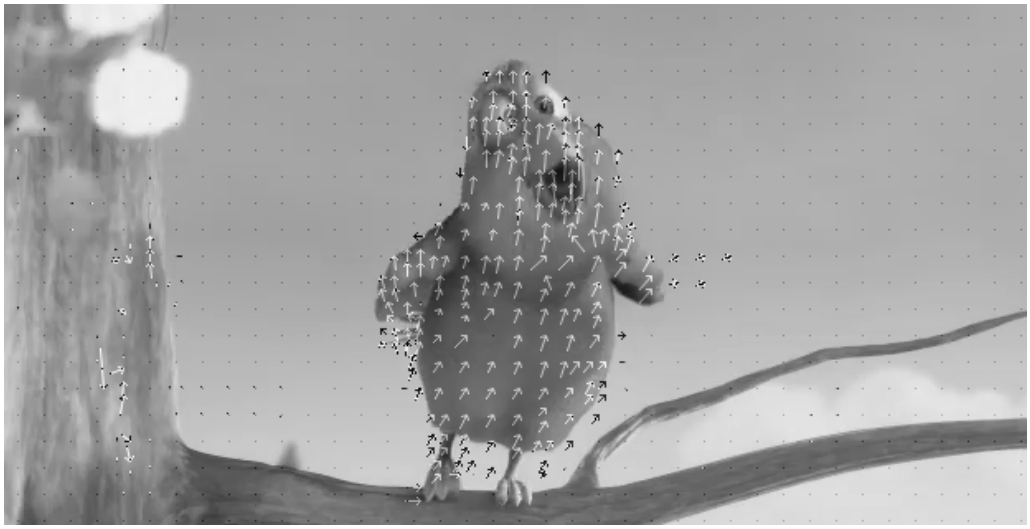

FIGURE 1.8: Visualisation of motion vectors in a frame from a H.264 encoding of the open short film Big Buck Bunny[20]

generated around this technique. And one primary use for it is to be fused with other sensor data to provide more robust values and correct the divergence from IMU estimations. Our main motivation was the PX4Flow[21] module which is an interesting open source and open hardware embedded system, and our final goal is to reach parity with its functionality and performance.

This chapter contained some required information, needed for a basic understanding of the problems this dissertation tackles, starting with some fundamental information about computer vision history, then we explore some techniques we used in our work. And lastly we have an introduction to the principle localization methods, and the some details of the method this work is based on. In the next chapter we will share our system configuration and the development tools we used throughout our work.

Chapter 2

System Specifications

After a long period of studying the basics of image processing and computer vision, we decided on the following set of software/hardware, since we wanted a simple and relatively cheap configuration while keeping the hardware strong enough for future additions. And in this chapter we will share the combination of hardware and software we used during development, and the reasons we chose each part, and how each one used.

2.1 OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms. OpenCV has a modular structure, which means that the package includes several shared or static libraries[22]. The following modules are available:

- Core functionality - a compact module defining basic data structures, including the dense multi-dimensional array Mat and basic functions used by all other modules.
- Image processing - an image processing module that includes linear and non-linear image filtering, geometrical image transformations (resize, affine and perspective warping, generic table-based remapping), colour space conversion, histograms, and so on.
- video - a video analysis module that includes motion estimation, background subtraction, and object tracking algorithms.
- calib3d - basic multiple-view geometry algorithms, single and stereo camera calibration, object pose estimation, stereo correspondence algorithms, and elements of 3D reconstruction.
- features2d - salient feature detectors, descriptors, and descriptor matchers.
- objdetect - detection of objects and instances of the predefined classes (for example, faces, eyes, mugs, people, cars, and so on).
- highgui - an easy-to-use interface to simple UI capabilities.
- Video I/O - an easy-to-use interface to video capturing and video codecs.
- gpu - GPU-accelerated algorithms from different OpenCV modules.
- some other helper modules, such as FLANN and Google test wrappers, Python bindings, and others.

We got the latest version from their online repository (on GitHub[23]) and compiled the OpenCV library on the Raspberry Pi, then created the Python links to use Python scripts to call OpenCV functions. This process took well over 4 hours in our case, compiling the library took 3 hours, and downloading and installing the required python modules and dealing with arbitrary errors took about an hour.

FIGURE 2.1: OpenCV logo[22]

We chose OpenCV because it contained a many implementations of interesting algorithms to our work, and those implementations are highly performant, and widely used in different computer vision-related projects. Also the ability to use on different platforms and with different programming languages which meant possible integration on different systems which is very important for prototyping.

The algorithms and techniques that are implemented in OpenCV can be optimized, as well as changed entirely, as needed. This flexibility allows for extensions and custom methods of work, but also allows for adapting just parts of OpenCV up or down, to accommodate system requirements. And if better solutions are found the general process can be kept intact, which makes the transition faster.

2.2 Raspberry PI

The Raspberry Pi is a single computer board (shown in Figure 2.2) — developed to encourage and aid the teaching of programming and computing. The low cost and 'plug and play' nature of Pi makes for a board that is accessible to all and has numerous connectivity options. The Pi is a good experimental tool, whether used as a desktop computer, media centre, server or monitoring/security device[24]. The Pi can run Linux-based systems as well as a special version of Windows called Windows 10 IoT core.

The Raspberry Pi 3, used in this work, has the following specifications:

- 1.2GHz 64-bit quad-core ARMv8 CPU
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 4 USB ports

- 40 GPIO pins
- Full HDMI port
- Ethernet port
- Combined 3.5mm audio jack and composite video
- Camera interface (CSI)
- Display interface (DSI)
- Micro SD card slot (now push-pull rather than push-push)
- VideoCore IV 3D graphics core

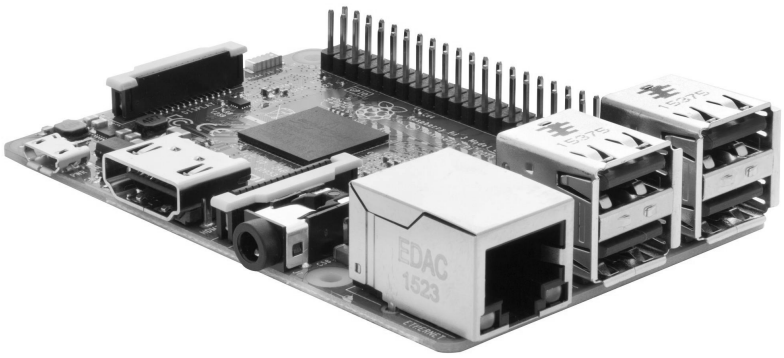

FIGURE 2.2: Raspberry Pi 3 board

We chose the Raspberry Pi because it is a powerful little computer that can run a full desktop operating system which allows for development right on the board with no need of another computer, that also enables us to use any programming language or development environment we see fit for the task. It also has an interface for interacting with a multitude of sensors, and with us running a Linux distribution, drivers for working with sensors are readily available and we can acquire the data from the sensors with relative ease.

2.3 Raspberry Pi NoIR camera

The Pi NoIR is the same as the regular Camera Module, with one difference: it does not employ an infrared filter, NoIR = No Infrared. This means that pictures taken by daylight will look different, but it gives the ability to see in the dark with infrared lighting. We acquired the NoIR v2, which has an 8MP Sony IMX219PQ image sensor, it is capable of 3280x2464 pixel static images, it supports up to 1080p30, 720p60 and 480p90 video, it weighs just over 3 grams, and its size is 25mm x 23mm x 9mm, Figure 2.3 shows the camera module.

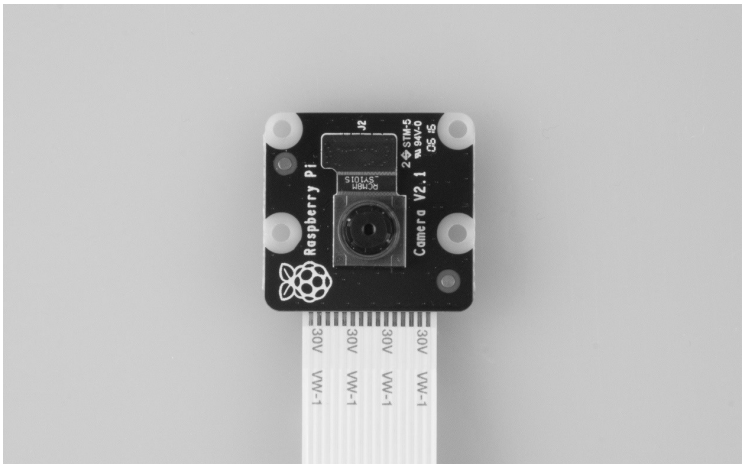

FIGURE 2.3: a picture of the NoIR camera module

In our application we don't need high quality captures, but the high definition sensor gives us a big buffer and provides better quality frames in lower resolutions, which helps decrease the chance for bad detection rates due to bad quality images, as well as high framerates when needed.

2.4 Raspbian

Raspbian is the recommended operating system for normal use on a Raspberry Pi. It is a free operating system based on Debian, optimised for the Raspberry Pi hardware. It comes with over 35,000 packages: precompiled software bundled in a nice format for easy installation on your Raspberry Pi. We used the latest stable version of Raspbian codename Jessy with PIXEL which is the official desktop for Raspbian and stands for "Pi Improved Xwindows Environment, Lightweight"[25].

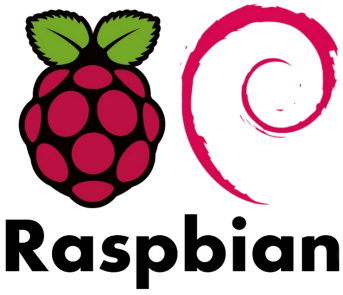

FIGURE 2.4: Raspbian Logo

For time constraints we decided to work with the official operating system of the board. While there exists other custom operating systems for the Pi, Raspbian is a well-rounded

package and it being the official OS meant that its documentation and support would be better than other custom packages, and the community is really helpful. Other operating system options include ROS (Robot Operating System), Windows 10 IoT, DietPi and Moebius.

2.5 Python

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later[26]. In our work, we used Python 3.4.0.

FIGURE 2.5: Python Logo

We used the Numpy Python library for array objects that are compatible with the OpenCV library. We also used the PiCamera library to access the Raspberry Pi camera, as well as some other useful functionality.

We decided to use Python because of the ease of execution, since the scripts don't require compiling. That meant that we could make changes on the fly and have a fast paced development process, because compile times add up really fast and hinder the process.

2.6 Development environment

We needed a few applications during the time of the development of the system. Listed below is the software we used:

- Putty: an SSH and telnet client, developed originally by Simon Tatham for the Windows platform. PuTTY is open source software that is available with source code and is developed and supported by a group of volunteers[27].

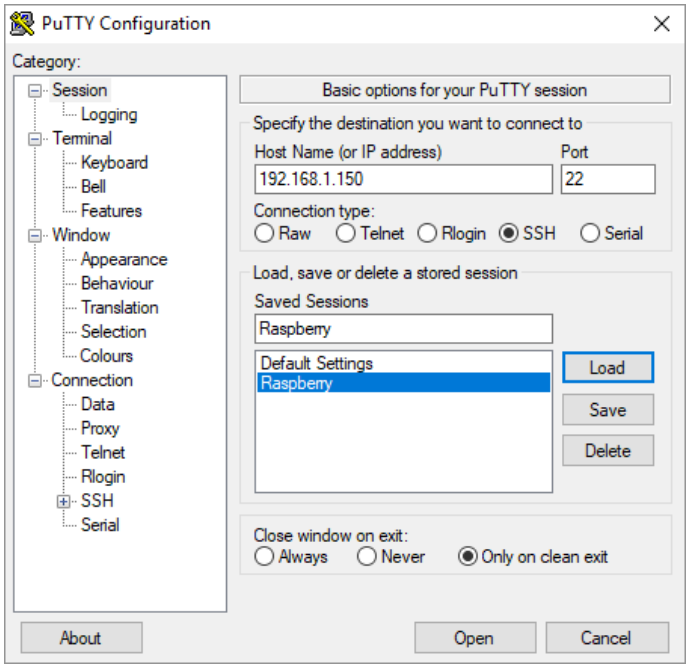

FIGURE 2.6: PuTTY interface

- FileZilla Client: a free, open source FTP client. It supports FTP, SFTP, and FTPS (FTP over SSL/TLS). The client is available under many platforms, binaries for Windows, Linux and Mac OS X are provided[28].

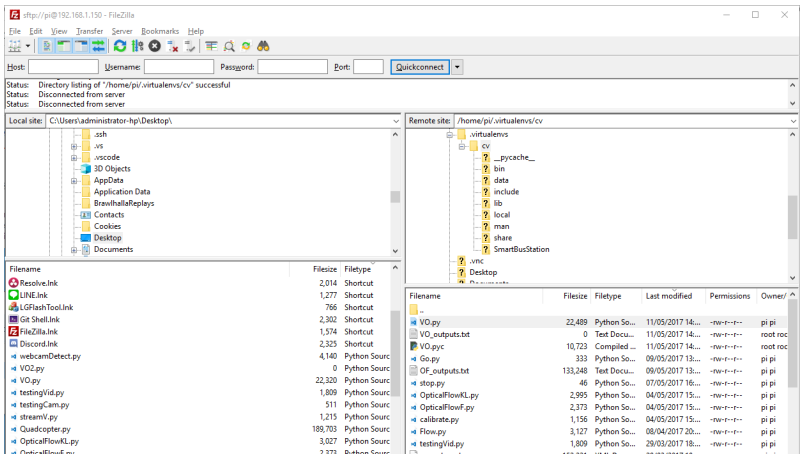

FIGURE 2.7: FileZilla Client interface

- Xming: a lightweight X Window System implementation for Microsoft Windows. It includes an X server, multiple X Window applications, and a launcher for both

the X server and for several secure shell programs. Xming is based on Cygwin/X, but is compiled to run on Windows with no additional compatibility layer [29].

FIGURE 2.8: Xming remote xeyes window

- Windows PowerShell: PowerShell (including Windows PowerShell and PowerShell Core) is a task automation and configuration management framework from Microsoft, consisting of a command-line shell and associated scripting language built on the .NET Framework. Initially a Windows component only, PowerShell was made open-source and cross-platform on 18 August 2016[30]. We used PowerShell because its commands are much like Linux, and it includes some extra functionality that made it easier to work.

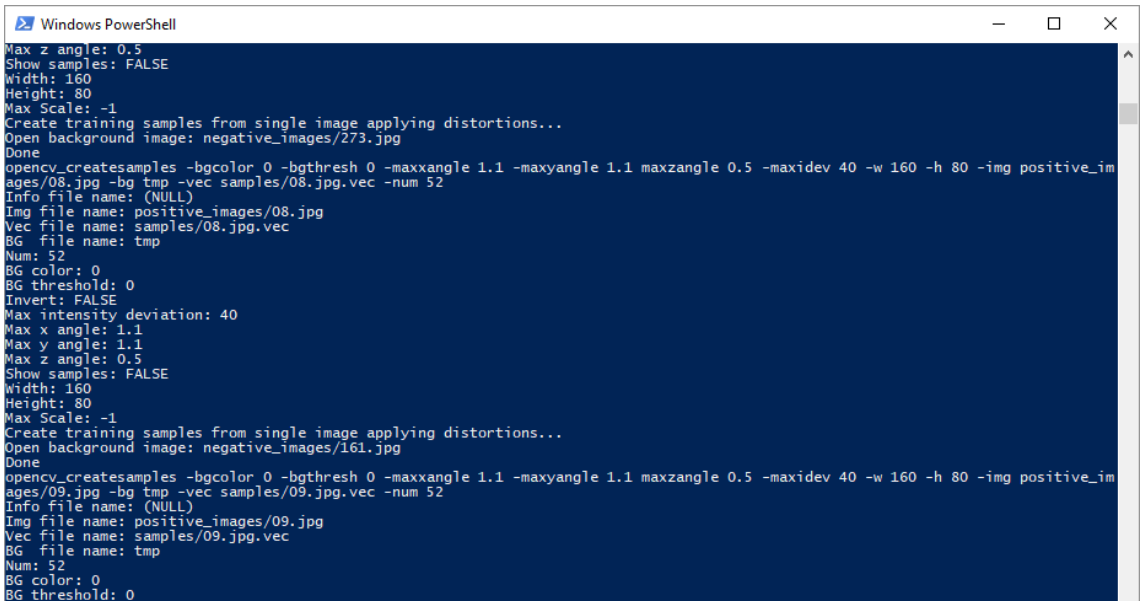

FIGURE 2.9: PowerShell terminal running sample creation script

- Visual Studio Code: a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in

support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages (such as C++, C#, Python, PHP, Go) and runtimes (such as .NET and Unity)[31], Figure 2.10 shows the interface.

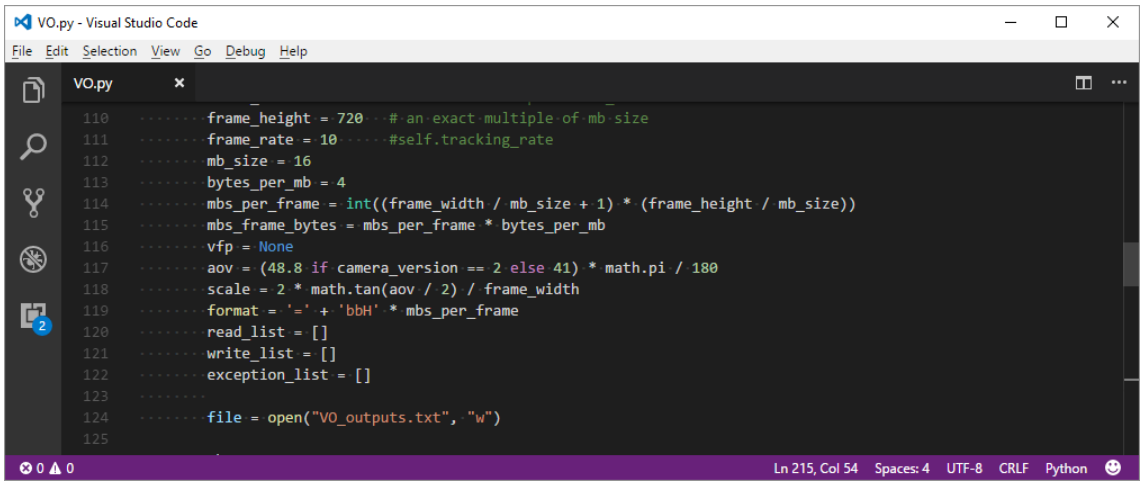

FIGURE 2.10: Visual Studio Code interface, with one of the Python scripts open

For the most part, we used a computer equipped with Windows 10, Visual Studio Code to write the Python scripts, then send the scripts over FTP with FileZilla. We connected to the Pi over SSH, and called the scripts from the command line. This allows us to not attach a keyboard and mouse to the Pi.

When one of the program needs to display windows, those are sent over to the Xming server on the Windows machine, and we can view the results of the program. This eliminates the need for a screen or any input devices to be attached to the Pi.

But sometimes we need to view visual results live, because the streamed x windows had some latency as well as lag to them, which was okay for text, but for images it was not optimal. So we attach a screen, keyboard and mouse to the Pi and work on it directly.

All in all, this collection of hardware and software is a good combination for prototyping, and allows for quick development cycles, with free or open source software and relatively cheap hardware that is powerful enough for the presented tasks. We tested a few sensors, namely depth sensors and IMUs, but we did not end up integrating them because of time constraints, we also tried using webcams instead of the NoIR but for size and quality reasons we decided on the NoIR camera sensor. The next chapter will be about the tests we performed and the attained results, a discussion of the results and some closing thoughts.

Chapter 3

Results and Analysis

In this chapter we present the way each part of our system works and share some interesting results we found, as well as some suggestions to improve on said results. There are three sections, the first two are about Object detection, which talk about the two main stages in the process, and the third is about localization, and there we share our tests of Visual odometry.

3.1 Training the Algerian Flag Classifier

We chose the Algerian flag as our object of interest to test the object detection part, because it has a moderately complex structure. We used the tool OpenCV has to train cascade classifiers, named opencv_traincascade.

We did the training on a portable computer with an Intel i7 quad core processor with 8 threads running at 3.6GHz, and 16Gb of RAM.

3.1.1 Preparing training data

Before we start the training process, training data needs to be prepared, and for that the opencv_createsamples tool is used, it takes a positive image, and some negative images and creates a vec file containing the data of the positive images mixed with a negative image as background. A positive image is an image that contains the object of interest, while a negative one must not contain it. The utility performs the following procedure to create samples. First the source image is randomly rotated across all three axes. Then the pixels with an intensity between $[bgcolor - bgthresh; bgcolor + bgthresh]$ are considered transparent (background). White noise is added to the foreground. If -inv is specified, the foreground intensities are inverted, and if -randinv is specified the algorithms randomly chooses to invert the intensities or not. Finally, the obtained image is placed into one of the background (negative) images provided at the start, resized to the specified width and height and then stored as a vec file.

We collected 30 loyalty free positive images of different sizes, but since we could not attain the recommended number of positive images, as usually thousands of positive images are needed. We used a guide[32] referenced by many tutorials and created a lot of sample images from each image we had, and appended the resulting vec files into one file

to be used in the training. The samples tool simulates different angles, lighting conditions and backgrounds, and we were able to get high detections rates even with the relatively low number of positives.

For the negative images, we used 600 images, we checked everyone of them to make sure that they contain no Algerian, preparing negative images is relatively easy, but it must be done by hand and be certain that all images don't contain the object of interest. One way to get background images is to use photos of landscapes, there are many sources online for royalty free pictures which can be used to acquire a large number of images, another method is to manually take pictures. Also, all images must not be too large in size, or the tool will not be able to use them, in our case, we scaled down all of our large images to 70% of their original size, this can be done manually for each picture, or a tool can be used and there are freeware or open source tools that can do that for a batch of files.

Before the images can be used, all the files need to be the same format, we decided to convert them all to jpg files because the output file sizes are smaller than other format, we used a function[33] for PowerShell to do that, then deleted all the original files of a different format, below is the code for the function.

```
function ConvertTo-Jpg
{
  [cmdletbinding()]
  param([Parameter(Mandatory=$true, ValueFromPipeline = $true)] $Path)

  process{
    if ($Path -is [string])
    { $Path = get-childitem $Path }

    $Path | foreach {
      $image = [System.Drawing.Image]::FromFile($($_.FullName));
      $FilePath = [IO.Path]::ChangeExtension($_.FullName, '.jpg');
      $image.Save($FilePath,
          [System.Drawing.Imaging.ImageFormat]::Jpeg);
      $image.Dispose();
    }
  }
}
```

After readying the positive and negative images, we separate them into different folders, and generate two text files that contain the directory of the images for each of the folders, we used PowerShell to generate the text files, using the following command line from the root folder containing the two folders for negative and positive images:

```
get-childitem negative_images/* | foreach {write-output
    ("negative_images/"+\$_.name)} >> negatives.txt
```

The opencv_createsamples accepts the following command line arguments as listed in the OpenCV documentation:

- -vec <vec_file_name> Name of the output file containing the positive samples for training.
- -img <image_file_name> Source object image.

- -bg <background_file_name> Background description file; contains a list of images which are used as a background for randomly distorted versions of the object.
- -num <number_of_samples> Number of positive samples to generate.
- -bgcolor <background_color> Background color (currently grayscale images are assumed).
- -bgthresh <background_color_threshold> to specify color tolerance of background color.
- -inv If specified, colors will be inverted.
- -randinv If specified, colors will be inverted randomly.
- -maxidev <max_intensity_deviation> Maximal intensity deviation of pixels in foreground samples.
- -maxxangle <max_x_rotation_angle>
- -maxyangle <max_y_rotation_angle>
- -maxzangle <max_z_rotation_angle> Maximum rotation angles must be given in radians.
- -show Useful debugging option. If specified, each sample will be shown. Pressing Esc will continue the samples creation process.
- -w <sample_width> Width (in pixels) of the output samples.
- -h <sample_height> Height (in pixels) of the output samples.

The sample creation tool is run for each image, because we needed to run the tool with the same commands, we used an open source script[32] written in Perl that does that. The script is given the names of the two text files that contain the directories and names of the positive and negative images, and what it does is that it calls the opencv_createsamples tool for each positive image with the arguments passed to it along with the text files. It results in a number of vec files, Figure 3.1 shows two sample images created using the tool. We ran the Perl script with the following parameters:

```
perl bin/createsamples.pl positives.txt negatives.txt samples 1500 –
    bgcolor 0 –bgthresh 0 –maxxangle 1.1 –maxyangle 1.1 maxzangle 0.5 –
    maxidev 40 –w 24 –h 24”
```

After the individual sample vec files are created, we call another open source script written in Python, it merges the vec files into a single one, and this last one is the one used for training. The script is fed the directory containing the vec files, and a name for the output file and it starts merging:

```
python tools/mergevec.py –v ./samples –o LBP48x48.vec
```

After multiple runs, each time changing the width and height parameters, we acquired our training data, consisting of three vec files containing samples with different sizes: $24x24$, $48x48$ and $160x80$ sizes, which are later used to train the classifiers. The process of generating sample data does not take a long time, but we noticed that it takes gradually more time as the window size is increased, and that is expected as it has to process larger images. There is another method to prepare training data for the OpenCV tool, but this method was the right one for us because of the number of positive images we managed to acquire.

FIGURE 3.1: 160x80 samples created by the sample creation tool

3.1.2 Training the cascade

After preparing the training data, we run the opencv_traincascade which takes in the data along with some parameters and starts calculating features , the cascade training tool accepts the following arguments, grouped by purpose[34]:

1. Common arguments:

   - -data <cascade_dir_name> Where the trained classifier should be stored.
   - -vec <vec_file_name> vec-file of positive images (output of opencv_createsamples).
   - -bg <background_file_name> Background description file.
   - -numPos <number_of_positive_samples> Number of positive samples used in training for every classifier stage.
   - -numNeg <number_of_negative_samples> Number of negative samples used in training for every classifier stage.
   - -numStages <number_of_stages> Number of cascade stages to be trained.
   - -precalcValBufSize <precalculated_vals_buffer_size_in_Mb> Size of buffer for precalculated feature values (in Mb).

- -precalcIdxBufSize <precalculated_idxs_buffer_size_in_Mb> Size of buffer for precalculated feature indices (in Mb). The more memory you assign the faster the training process.
- -baseFormatSave This argument is actual in case of Haar-like features. If it is specified, the cascade will be saved in the old format.
- -numThreads <max_number_of_threads> Maximum number of threads to use during training.
- -acceptanceRatioBreakValue <break_value> This argument is used to determine how precise your model should keep learning and when to stop. A good guideline is to train not further than 10e-5, to ensure the model does not overtrain on your training data. By default, this value is set to -1 to disable this feature.

2. Cascade parameters:

   - -stageType <BOOST(default)> Type of stages. Only boosted classifier is supported as a stage type at the moment.
   - -featureType <HAAR(default), LBP> Type of features: HAAR - Haar-like features, LBP - local binary patterns.
   - -w <sample_width>
   - -h <sample_height> Size of training samples (in pixels). Must have exactly the same values as used during training samples creation (opencv_createsamples utility).

3. Boosted classifier parameters:

   - -bt <DAB, RAB, LB, GAB(default)> Type of boosted classifiers: DAB - Discrete AdaBoost, RAB - Real AdaBoost, LB - LogitBoost, GAB - Gentle AdaBoost.
   - -minHitRate <min_hit_rate> Minimal desired hit rate for each stage of the classifier.
   - -maxFalseAlarmRate <max_false_alarm_rate> Maximal desired false alarm rate for each stage of the classifier.
   - -weightTrimRate <weight_trim_rate> Specifies whether trimming should be used and its weight. A decent choice is 0.95.
   - -maxDepth <max_depth_of_weak_tree> Maximal depth of a weak tree. A decent choice is 1, that is case of stumps.
   - -maxWeakCount <max_weak_tree_count> Maximal count of weak trees for every cascade stage. The boosted classifier (stage) will have as many weak trees (<=maxWeakCount) as needed to achieve the given -maxFalseAlarmRate.

4. Local Binary Patterns parameters: Local Binary Patterns don't have parameters.

We ran the tool multiple times to train Haar and LBP classifiers, to see the difference in detection rates and processing time, as well as training time. The following code snippet

is an example of the parameters we used for the classifier training tool to train a Haar classifier with samples of the width and height of 24 pixels. In total we trained 5 classifiers, 3 LBP classifiers and 2 Haar ones, with the only difference being the size of the samples, we started with a 24x24 pixels samples size, then double it to 48x48 pixels, and we ran one more time for an LBP classifier with the size of 160x80 pixels, this last one took over 31 hours, and knowing it takes several times more time to do a Haar classifier we were unable to create one.

```
opencv_traincascade -data classifier -vec samples24x24.vec -bg
    negatives.txt -numStages 20 -minHitRate 0.999 -maxFalseAlarmRate
    0.5 -numPos 1000 -numNeg 600 -w 24 -h 24 -precalcValBufSize 2048 -
    precalcIdxBufSize 2048
```

We mainly used default values for the parameters, like the number of stages and the minimum hit rate as well as the maximum false alarm rate, we found on forums and tutorials that these are the most used values. The number of stages affects the training time directly, and more stages require more time. The minimum hit rate is set to a really high value, this affects the number of stages, if it is set to a lower value, the training can end before finishing all the stages. The max false alarm rate parameter affects the number of nodes in a stage, the lower it is the higher the number of nodes, which increases the training time.

Training can take a very long time, hours, days or even weeks depending on the machine the tool is running on and the different parameters of the training, we noticed that processing the negative images takes more time in each stage, while processing tree nodes is usually the same throughout the training process.

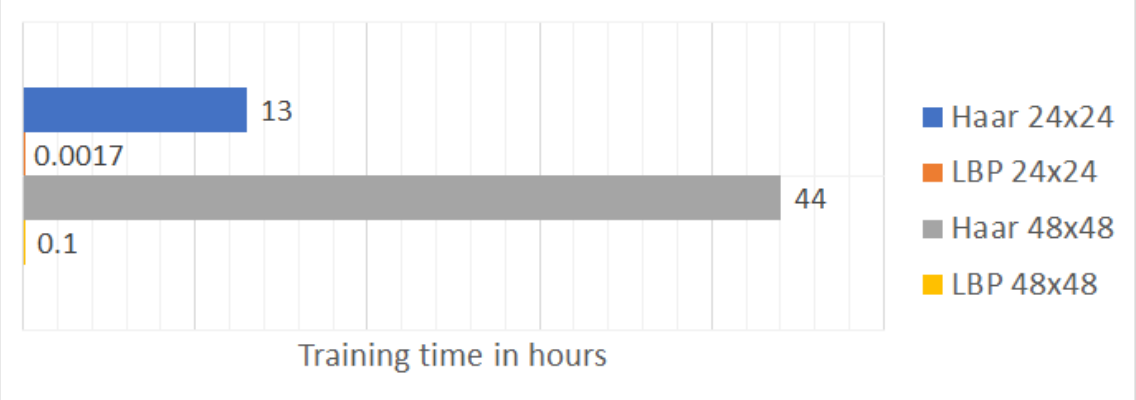

FIGURE 3.2: This graph shows the training time of the classifiers

Figure 3.2 shows the difference in training time between the two feature types, this huge different is due to the type of calculations done for each feature type. While Haar uses multiple rectangles of pixels, LBP uses a more direct pixel thresholding technique between neighbouring pixels in a matrix. This fundamental difference is the reason LBP cascades are faster to train, and that is due to the simple fact that there are less features to calculate in LBP than in Haar, and the number of is shown in Figure 3.3.

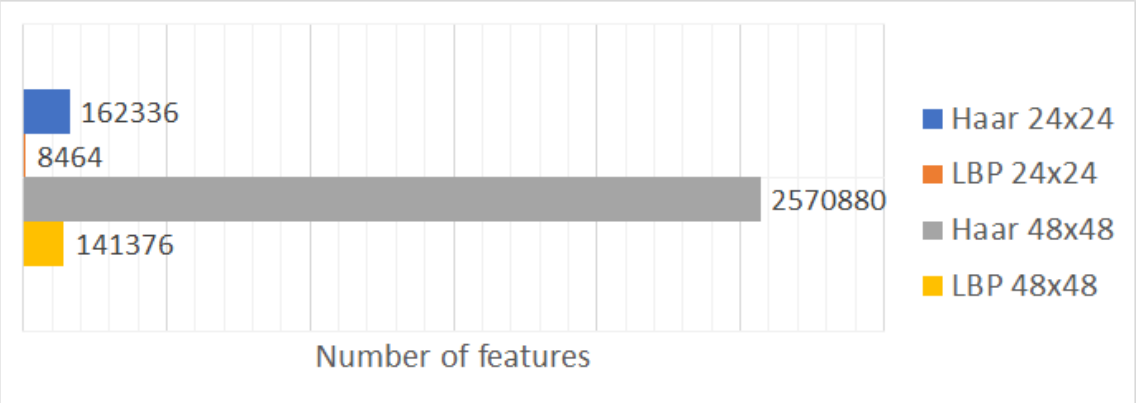

FIGURE 3.3: Graph showing the number of feature in LBP and Haar in each tested sample size

The resulting classifiers are written into xml files, the file contains a description of the parameters of the classifier such as its type, the boost type, and width and height of samples. The Haar files were about 140KB in size, while the LBP ones were about 40KB, the difference in size between Haar and LBP is significant, as the Haar classifier is more than two times the file size of the LBP one, the difference is primarily due to LBP features being a single decimal represented in the file with a single **ect** node, while Haar features are stored in a pair of **rect** nodes or more because there are multiple types of features. This size of the files could be important for embedded systems with really limited storage, but it is insignificant for our system with the amount of storage that can be added to the Pi.

We named the classifiers in the following manner: Feature WidthxHeight, for examples the Haar classifier made out of samples with width and height of 48 pixels, its name is Haar 48x48, and that is how we will be labelling our classifiers.

Due to the long time it takes to train classifiers, the lack of testing equipment; since the computer running the tool would heat up a lot and use a lot of memory making it unusable for other purposes, as well as the shortage of time we were facing, we were not able to run more tests on the cascade training tool, particularly the Haar 160x80 which is projected to take about 5-6 days to train.

3.2 Detecting the flag

For this, we set up the camera for a continuous frame capture stream using the PiCamera class, in a BGR format (Blue Green Red) and grab the frames one at a time, convert it to a grayscale image, then apply detectMultiScale function on that grayscale image. The function takes the following parameters:

- image: an array containing an image where objects are detected.
- scaleFactor: how much the image size is reduced at each image scale.

- minNeighbors: the number of neighbours each candidate rectangle should have to retain it.
- flags: we could not find any useful information about this parameter.
- minSize: minimum possible object size, objects smaller than that are ignored.
- maxSize: maximum possible object size, objects larger than that are ignored.

The detection rate for both cascades (Haar features and LBP) was high, and both were able to properly detect the flags in different angles and at different sizes (distance from the camera), as seen in Figures 3.4 and 3.5. The first thing we noticed, was that the scaleFactor and minNeighbors parameters need to be adjusted for each classifier to get the optimal detection, for example, in a single frame, the LBP classifier is over detecting, with a lot of false positives while, with the same parameters, the Haar one does not detect anything. The quality of the capture, like brightness, contrast and focus, plays a big role in the detection as well. Finally we found that our LBP classifiers tend to produce a lot of false positives more than the Haar ones, and would also fail to detect the flag while the Haar classifier would detect it successfully, and this result was in accordance with the general opinion that LBP classifiers are less accurate than Haar classifiers using the same sample set.

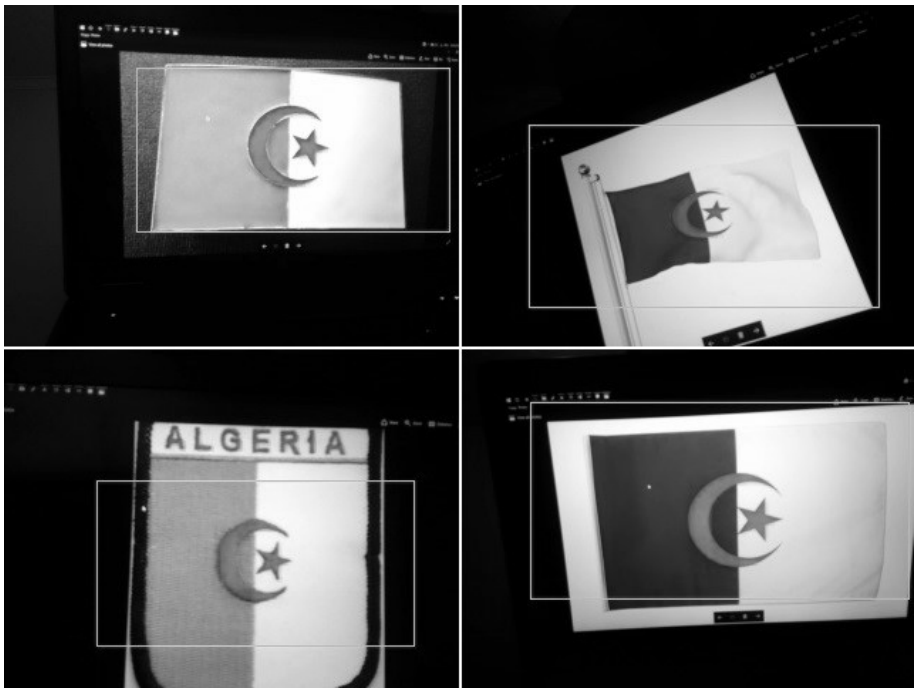

FIGURE 3.4: Screenshots of detected flags from video stream at resolution of 320x320 at 20fps, using Haar 48x48 classifier

For LBP classifiers we found that the scale factor parameter must be set higher to get better results, using values between 1.6 and 1.9, while for Haar classifiers, we had to set that value lower, using values between 1.3 and 1.7. Scale factor value increases detection

time, as there would be more scales of the image to be processed. We noticed that setting its value as less than 1.1, would result in crashing the application.

In our detection tests, the LBP 180x60 classifier was the better one in the majority of the tests, yielding better results in all resolutions, this is a surprising result, as we did not see any improvement from LBP 24x24 to 48x48, with both of them being less accurate than the Haar ones.

FIGURE 3.5: Detected from a photo of flag flailing in the wind, using the LBP 180x60 classifier

We used the Python $timeit$ function to measure the time it takes for the detectMultiScale function to process the image passed to it, we did that for each classifier on the same video stream and using the same parameters multiple times, the timings are written into a file and later graphs are generated. Figure 3.6 shows that LBP classifiers take more time to process images than Haar classifiers in all image resolutions and in all sample sizes. Then we see that the LBP 160x80 classifier's processing time, for images at the resolution of 320x320, is less than every other classifier, and for the other resolutions its speed was close to the two Haar classifiers and was better than the other LBP ones.

But the average processing time does not show the whole picture, in Figure 3.7 we see that the difference between minimum processing times for the classifiers is not as big compared to the average times, this means that processing LBP features can be comparatively fast. But that also shows that LBP classifiers, except LBP 160x80, are more unstable, meaning that it has a very wide delta of $min - max$ processing time, while the gap between Haar classifiers is not as big.

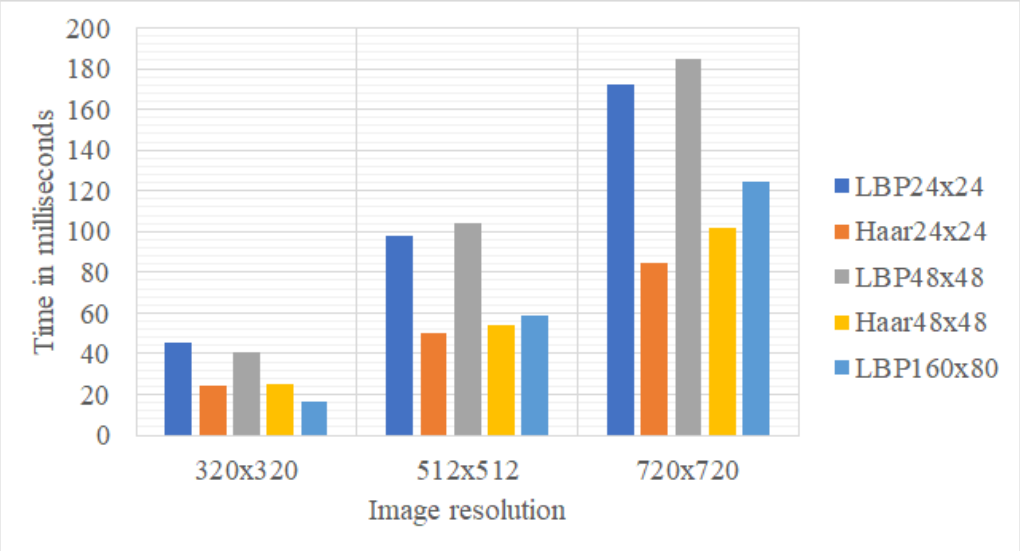

FIGURE 3.6: Average processing time of detection on the different image resolutions for the different classes of classifiers

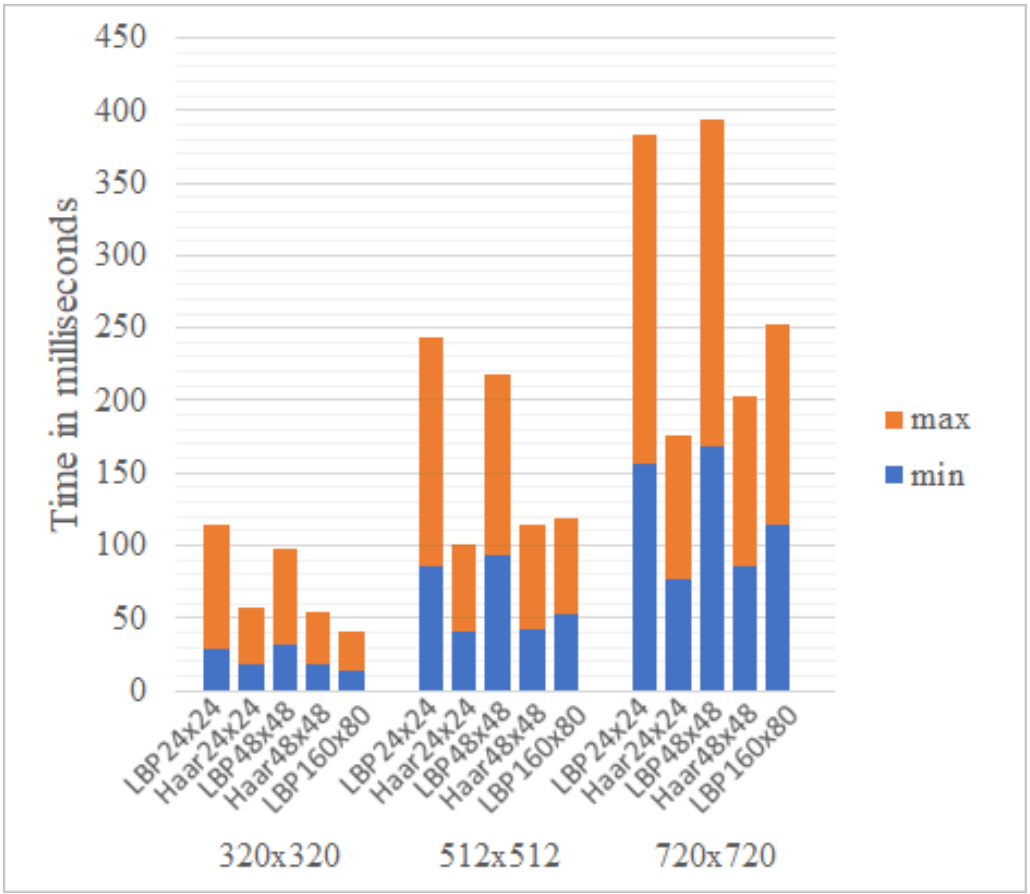

FIGURE 3.7: Min and max processing times of detection on the different tested resolutions

The results from this chapter shows us that we can run object detection in real-time at multiple resolutions and at acceptable framerates. The best compromise between framerates and resolution is 512x512 at 16fps, we can get even faster frame captures if we go down to 320x320, and get 24fps which is the rate a lot of cameras run at. With higher frame rates, the camera can capture relatively fast moving objects, but with less details, while a lower frame rate gives more clarity for stationary or slow moving objects, but would show fast moving objects as blurry or smudged, which would hinder the detection in said frames.

3.3 Visual Odometry

This is the part that took us the most time, and was the hardest to find useful technical resources for, there are many papers on this topic, but the language is advanced and mathematically complex, and the implementations we found were not sufficiently documented. Since my main focus in study was not computer vision or image processing, it was not possible to develop and implement a complete system with the limited knowledge and short time frame, and it was hard to understand the solutions and decompose them to take only the needed parts. Most of the open source projects we managed to find were big, fully featured systems with extra assistive parts for visual representations and 3D renders of the environment, and were desktop applications and incompatible with our system or were too complex and would take more time that we have available to port them.

So for our system we used the difference between blocks in a frame to estimate the direction and speed of movement. The frame is divided into blocks, dubbed macro-blocks, and the vector of movement is measured by calculating the difference between two successive frames. This functionality is provided by the PiCamera API, and this processing is done on GPU alleviating some work of off the CPU and allows for better parallelization.

First we setup a FIFO stream of the camera feed, the values we used for the camera are as follows, the resolution is set to 720 x 720 with a quality of 23, a H.264 encoding (for this encoding the quality ranges from 10 to 40, the lower the better), and the contrast to 100, and the framerate to 10 frames per second which was sufficient to capture the movement of the camera in a moderate speed, also for our tests the camera is facing either downwards or upwards.

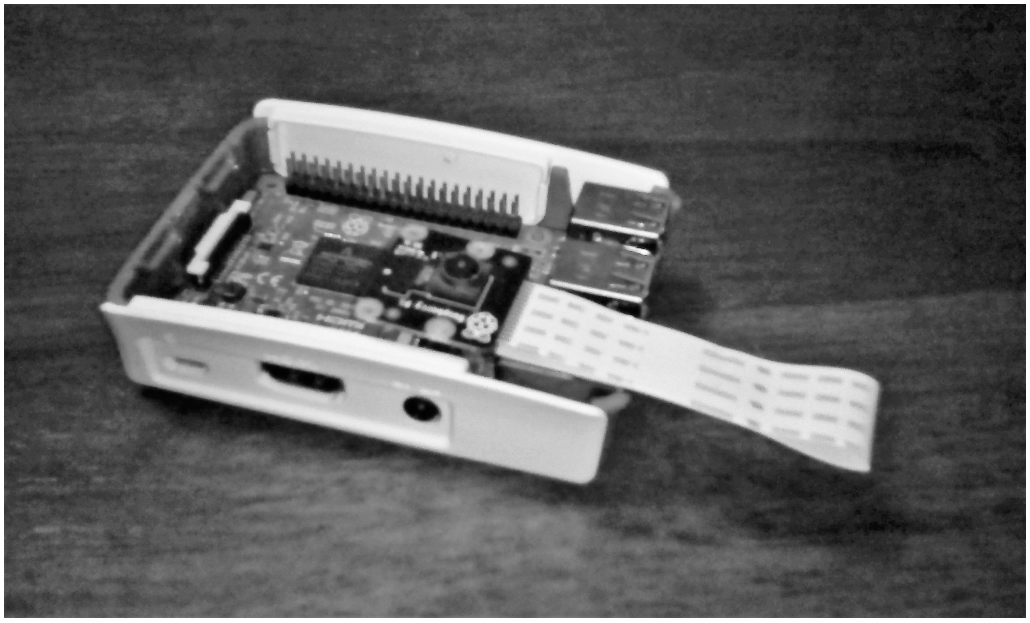

FIGURE 3.8: A photo of the camera setup, the sensor is taped to minimize vibration

Then we start grabbing frames from the FIFO stream as each finishes being captured and the vectors are calculated, each frame is in actuality a 3-dimentional array organized as frames, rows and columns, where rows and columns are the numbers of rows and columns of macro-blocks in the original frames and there is always an extra column of macro-blocks in the motion data provided by the API, no explanation is provided as to why that happens. The array is then split into a list of macro-blocks, then the vectors are grouped into clusters based on neighbors, the clusters are scored and the vectors with the best scores are chosen, then the vectors are averaged to get the final global motion vector which is the estimation of the speed in the x and y axes. This speed value is first multiplied by the scale; which is a conversion from macro-blocks to meters at a given height, then it is multiplied by the time between captured frames (which is calculated as 1/frame rate) to get a position estimate; this estimation is relative to the point where the video stream started capturing, this process take around 50% of the processing resources at max in all of our tests.

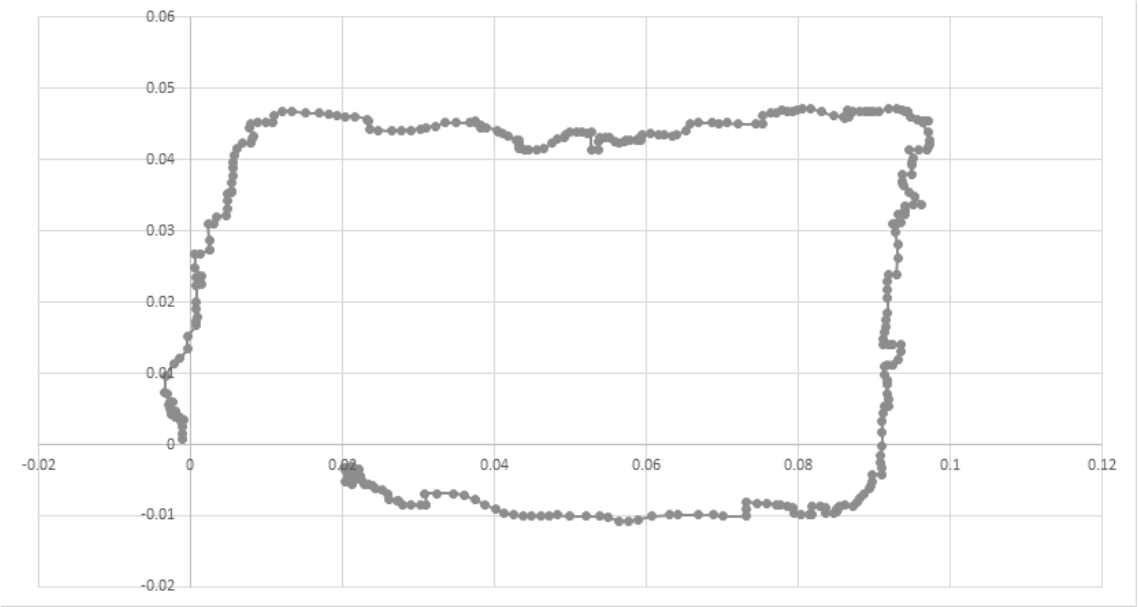

FIGURE 3.9: Trajectory generated from visual data, 720 by 720 pixels at 10 frames per second

The implementation we have is not robust against unstable camera movement, as you can see in Figure 3.9, the points that don't follow the rest and escape inwards or outwards is due to the camera shaking too much, these calculations are raw and are not being normalized. Since the Pi has a powerful CPU and GPU, we tried to increase the framerate on the same resolution, but that lead to the system not running in realtime, and it was lagging behind, so we dropped down the resolution to 512 by 512 pixels and increased the framerate to 20 frames per second.

From Figure 3.10 we see that the estimations are more stable in general, and the track is more accurately traced, but because we are using a static height value, we notice that at the top-left there are some scattered points, that is because there was a change in height which

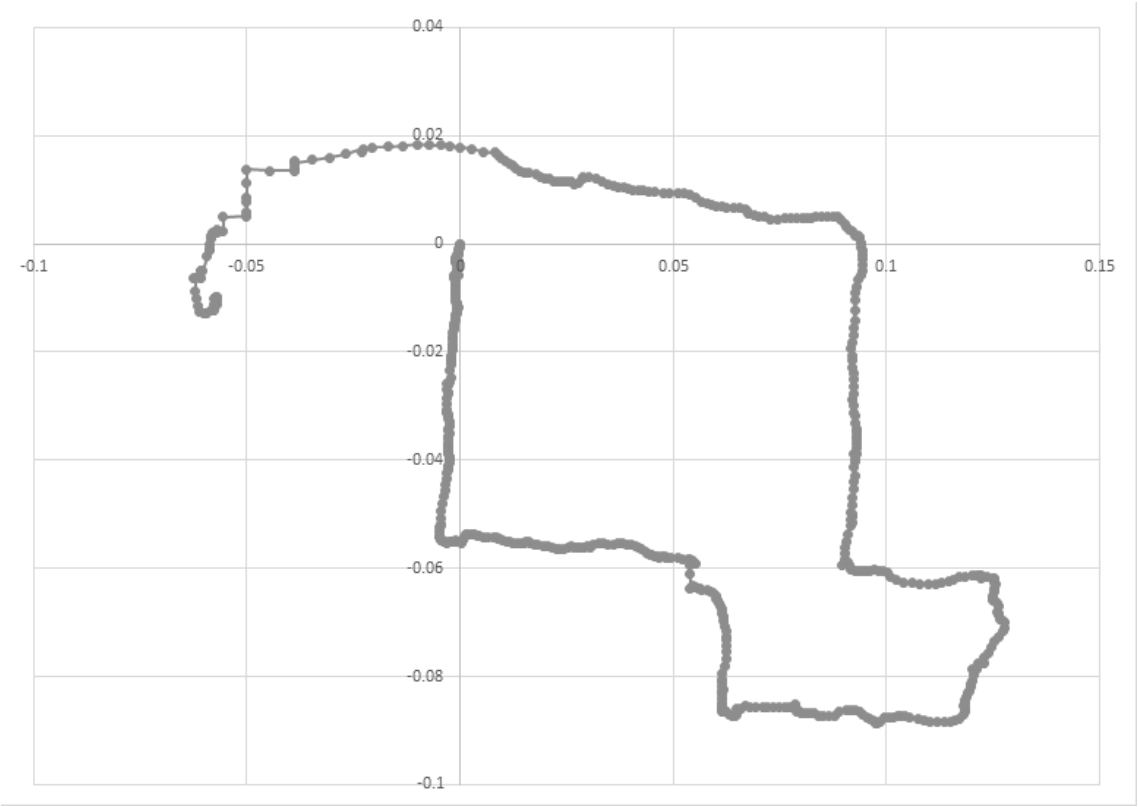

FIGURE 3.10: Trajectory generated from visual data, 512 by 512 pixels at 20 frames per second

messed the estimations, the solution to this is to get the height data, from a depth/distance sensor. We notice that the distance units are not exact, and need more calibration to get more accurate results, this requires more testing and also the use of an IMU for comparison.

In this chapter we shared some tests we did and the interesting results we acquired. We found that for Object detection using the method we employed, there are two choices to make for the two kinds of classifiers you can use with OpenCV, depending on your needs, if a fast development cycle is employed then LBP classifiers with their fast training times are the way to go, but from our findings the detection speed (the time taken to process an image) is slower than Haar classifiers.

And for Visual odometry, it is an interesting and useful technique in many scenarios, and its estimations can be fused with IMU data, to get more accurate values. In conclusion, VO can work on an SoC, with overhead in processing time as well as processing speed for sensor fusion as well as other functions for the system and still keep it running real-time or at least near real-time.

Next up we present a general conclusion, and future works.

Conclusion and Future works

The aim of our work was to realize a full navigation and control system for a UAV, using only a limited set of sensors and do all the processing on board. We made good progress on our goal, but the problem required more time, simply because there was a lot of learning to be done in order to gain the ability to understand how such a system would function, and its requirements.

Our work consists of two major parts, object detection and localization in the form of monocular camera-based-IMU (Visual Odometry), all working on board an SoC. The importance of Visual Odometry in the field of computer vision and in robotics is paramount, and Object Detection is essential for intelligent machines. We trained a fast and efficient classifiers using the OpenCV library, were able to detect objects of interest in real-time on video feed with a good detection rate, and we used two different types of features and conducted a performance comparison between the two. We also implemented a VO system that calculates velocity estimations on the x and y axes using motion vectors from the H.264 video encoding process.

What we managed to complete was finishing parts of the system and not yet fully integrate them all into one, but important progress nonetheless, as the fundamentals are now clear and the promising leads are now apparent. Both Computer Vision and Robotics are interesting fields with a lot of active research, and our work was for us, if nothing else, a good introduction into the important field, and we will be working on further developing what we have achieved.

The next step is to improve the detection rates by finding out the optimal parameters for the classifier, and also tackling the problem of detecting multiple objects of interest of different classes, as well as investigating other Computer Vision solutions, we will also be checking the problem of implementing a custom training and detection systems. As for the Visual Odometry, we plan to continue working on the implementation to get the angle values, and figure out the best parameters and also optimize the code. After VO the logical thing is exploring the SLAM (Simultaneous Localization and Mapping) problem, which pertains to maintaining a persistent visual map of a previously traversed environment, and in particular Visual-SLAM which is camera-based.

Bibliography

[1] Ming-Hsuan Yang. “Object Recognition”. In: *Encyclopedia of Database Systems*. 2009, pp. 1936–1939.

[2] L. G. Roberts. “Machine perception of three-dimensional solids”. In: *Optical and Electrooptical Information processing* (1965), pp. 159–197.

[3] T. O. Binford. “Spatial understanding: the successor system”. In: *In Proceedings of the ARPA Image Understanding Workshop* (1992), pp. 12–20.

[4] Matthew Turk and Alex Pentland. “Eigenfaces for Recognition”. In: *J. Cognitive Neuroscience* 3.1 (Jan. 1991), pp. 71–86.

[5] Zhong Guo. *Object Detection and Tracking in Video*. Apr. 24, 2017. URL: http://www.medianet.kent.edu/surveys/IAD01F-objdetection/index.html.

[6] V. Vaithiyanathan K. D. Lakshmi. “Study of Feature based Image Registration Algorithms for Navigation of Unmanned Aerial Vehicles”. In: *Indian Journal of Science and Technology* (2015).

[7] Michael J. Jones Paul Viola. “Rapid object detection using a boosted cascade of simple features”. In: *In Computer Vision and Pattern Recognition* (2001).

[8] Timo Ojala, Matti Pietikäinen, and David Harwood. “A comparative study of texture measures with classification based on featured distributions”. In: *Pattern Recognition* 29.1 (Jan. 1996), pp. 51–59.

[9] Zhen Lei Lun Zhang Shengcai Liao Xiangxin Zhu and Stan Z Li. “Learning multi-scale block local binary patterns for face recognition”. In: *In Advances in Biometrics* (2007).

[10] Timo Ahonen, Abdenour Hadid, and Matti Pietikäinen. “Face Recognition with Local Binary Patterns”. In: *Computer Vision - ECCV 2004: 8th European Conference on Computer Vision, Prague, Czech Republic*. 2004, pp. 469–481.

[11] Robert E. Schapire. “Explaining AdaBoost”. In: *Empirical Inference* (2013), pp. 37–52.

[12] U.S. government. *Global Positioning System*. Apr. 24, 2017. URL: http://www.gps.gov/systems/gps/.

[13] University of Maryland Space Systems Laboratory. *Inertial Measurement Unit*. Apr. 24, 2017. URL: http://ssl.umd.edu/projects/RangerNBV/thesis/2-4-1.htm.

[14] J. Bergen D. Nister O. Naroditsky. “Visual odometry”. In: *Proc. Int. Conf. Computer Vision and Pattern Recognition* (2004), 652–659.

[15] F. Fraundorfer D. Scaramuzza. “Visual Odometry Part I: The First 30 Years and Fundamentals”. In: *IEEE Robotics and Automation Magazine* (Dec. 2011).

[16] D. Nister. “An efficient solution to the five-point relative pose problem”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.6 (June 2004), pp. 756–770.

[17] A. I. Comport, E. Malis, and P. Rives. “Accurate Quadrifocal Tracking for Robust 3D Visual Odometry”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. Apr. 2007, pp. 40–45.

[18] Amnon Shashua and Lior Wolf. “On the Structure and Properties of the Quadrifocal Tensor”. In: *Proceedings of the 6th European Conference on Computer Vision-Part I*. 2000, pp. 710–724.

[19] duo3d. *DUO M*. Apr. 24, 2017. URL: https://duo3d.com/product/duo-mini-lv1.

[20] FFmpeg.org. *Debugging Macroblocks and Motion Vectors*. Apr. 24, 2017. URL: https://trac.ffmpeg.org/wiki/Debug/MacroblocksAndMotionVectors.

[21] D. Honegger et al. “An open source and open hardware embedded metric optical flow CMOS camera for indoor and outdoor applications”. In: *2013 IEEE International Conference on Robotics and Automation*. May 2013, pp. 1736–1741.

[22] OpenCV team. *OpenCV Library*. May 24, 2017. URL: http://opencv.org/.

[23] OpenCV team. *OpenCV Git respository*. May 20, 2017. URL: https://github.com/opencv/opencv/.

[24] Raspberry Pi Foundation. *Raspberry Pi*. May 31, 2017. URL: https://www.raspberrypi.org/.

[25] Raspberry Pi Foundation. *Raspbian*. May 31, 2017. URL: http://www.raspbian.org/.

[26] Python Software Foundation. *Python.org*. May 31, 2017. URL: https://www.python.org/.

[27] PuTTY team. *PuTTY*. May 31, 2017. URL: http://www.putty.org/.

[28] Tim Kosse. *FileZilla*. May 31, 2017. URL: https://filezilla-project.org/.

[29] Colin Harrison. *Xming X Server*. May 31, 2017. URL: http://www.straightrunning.com/XmingNotes/.

[30] Microsoft. *Microsoft Powershell*. Apr. 30, 2017. URL: https://msdn.microsoft.com/en-us/powershell/mt173057.aspx.

[31] Microsoft. *Visual Studio Code*. Apr. 20, 2017. URL: https://code.visualstudio.com/.

[32] Thorsten Ball. *Train your own OpenCV Haar Classifier*. Mar. 27, 2017. URL: https://github.com/mrnugget/opencv-haar-classifier-training.

[33] Chad Miller. *ConvertTo-Jpg PowerShell function*. Mar. 15, 2017. URL: https://stackoverflow.com/questions/6863021/convert-all-images-to-jpg.

[34] OpenCV team. *Cascade Classifier Training*. Mar. 10, 2017. URL: http://docs.opencv.org/3.2.0/dc/d88/tutorial_traincascade.html.

ملخص

رؤية الكمبيوتر هو مجال بحث هام، مستخدم على نطاق واسع في العديد من التطبيقات، من الوسم التلقائي للصور إلى الروبوتات. الروبوتات والمركبات أصبحت آلية على نحو متزايد، والحاجة إلى التعرف على الأشياء في البيئة أمر ضروري لأدائها الصحيح، ومع التقدم في تكنولوجيا النظام على رقاقة: تقلص الحجم، زيادة الأداء وتحسين استهلاك الطاقة، سمح لاستضافة نظم ذكية ذاتية الاستدامة على منصات متحركة صغيرة. استخدام آخر للبيانات البصرية هي الاودومتري البصري التي تتعامل مع تقدير الحركة، و هي مفيدة في البيئات التي لا تغطيها خدمات تحديد الموقع، لان بيانات وحداة قياس القصور الذاتي تتباعد مع الوقت لذلك يستخدم الاودومتري البصري لتحقيق الاستقرار في تلك القيم. في عملنا، صممنا نظام تموقع و تعرف على الأشياء على لوح مدمج. استكشفنا تدريب مصنف خاص للكشف عن الاشياء، و قمنا بإنجاز طريقة الاودومتي البصري و أيضا طبقنا بعض التجارب وجمعنا نتائج مثيرة للاهتمام.

رؤية الكمبيوتر, التعرف على الأشياء, الاودومتري البصري

Abstract

Computer vision is an important research field, widely used in a multiple of applications, from automatic image labelling to robotics. Robot and vehicles are increasingly being automated, and the need to recognize objects in the environment is essential for their correct functioning, and with the advancement in System-on-Chip technology, shrinking the size, increasing the performance and improving power consumption, allowing for whole self-sustaining intelligent systems on small mobile platforms. Another use of visual data is visual odometry which deals with movement estimation, it is helpful in GPS-denied environment, since IMU data diverges in time so VO is used to stabilize the values. In our work, we implemented an object detection and localization system on an embedded board. We explored training our own classifier for object detection, and implemented and did some testing on a VO method and collected some interesting results.

Computer vision, Object detection, Visual odometry, Raspberry Pi, Python

Résumé

La vision par ordinateur est un domaine de recherche important, largement utilisé dans un multiple d'applications, du marquage automatique de l'image à la robotique. Le robot et les véhicules sont de plus en plus automatisés et la nécessité de reconnaître les objets dans l'environnement est essentielle pour leur fonctionnement correct, et avec l'avancement de la technologie System-on-Chip, en réduisant la taille, en augmentant les performances et en améliorant la consommation d'énergie, en permettant l'implémentation de tous un systèmes intelligents autonome sur des petites plates-formes mobiles. Une autre utilisation des données visuelles est l'odométrie visuelle qui traite de l'estimation du mouvement, elle est utile dans l'environnement où le service GPS n'est pas disponible, car les données de l'IMU divergent dans le temps afin que VO soit utilisé pour stabiliser les valeurs. Dans notre travail, nous avons mis en œuvre un système de détection d'objet et de localisation sur une carte embarqué. Nous avons aussi exploré la formation de notre propre classificateur pour la détection d'objets, et nous avons fait des tests sur une méthode VO et nous avons recueilli des résultats intéressants.

Vision par ordinateur, Détection d'objet, Odométrie visuelle. Raspberry Pi, Python