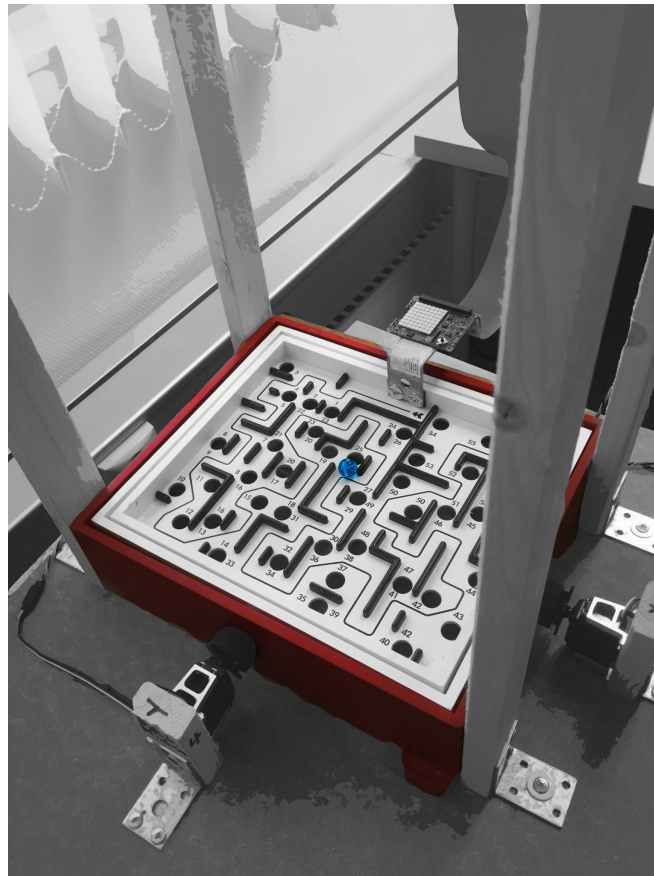




CHALMERS

UNIVERSITY OF TECHNOLOGY



Sensor and vision aided labyrinth game

The implementation and controlling of a labyrinth game with the aid of vision technology and sensor-fusion

Bachelor thesis in mechatronics

Alfons Massey Mattias Gustafsson

BACHELOR THESIS 2016

Sensor and vision aided labyrinth game

The implementation and controlling of a labyrinth game with the aid
of vision technology and sensor-fusion

Alfons Massey Mattias Gustafsson



Department of Signals and Systems
Automatic control, Automation and Mechatronics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Sensor and vision aided labyrinth game
The implementation and controlling of a labyrinth game with the aid of vision
technology and sensor-fusion
Alfons Massey, Mattias Gustafsson

© Alfons Massey, Mattias Gustafsson, 2016.

Supervisor: Göran Hult, Department of Signals and Systems
Supervisor: Björn Bergholm, Broccoli
Examiner: Manne Stenberg, Department of Signals and Systems

Bachelor Thesis 2016
Department of Signals and Systems
Automatic control, Automation and Mechatronics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46(0)31 772 25 60

Cover: Sensor and vision aided labyrinth game.

Typeset in L^AT_EX
Gothenburg, Sweden 2016

Sensor and vision aided labyrinth game

The implementation and controlling of a labyrinth game with the aid of vision technology and sensor-fusion

Alfons Massey, Mattias Gustafsson

Department of Signals and Systems

Chalmers University of Technology

Acknowledgements

This thesis has been carried out by Alfons Massey and Mattias Gustafsson, both reading our third year mechatronical engineering(180hp) at Chalmers University of Technology.

The work has been done at Broccoli Engineering, which is an engineering consulting firm that works mainly with hardware and software development in embedded systems.

We would like to thank Björn Bergholm, our supervisor at Broccoli Engineering, and the rest of the Broccoli team for the opportunity to do this with them.

We would also like to thank our supervisor, Göran Hult, and examiner, Manne Stenberg, for their support throughout the project.

Alfons Massey, Mattias Gustafsson, Gothenburg, June 2016

Keywords: Labyrinth game, Raspberry Pi, Controlling, Gyroscope, Python, openCV

Abstract

This thesis describes the process of constructing a platform with all the requirements for electrical controlling with the aid of sensor-fusion. The goal is to have a final product where a control algorithm for the system can be implemented.

The system is made up of a Raspberry pi 3 model B, 2 Parallax standard servo motors, an Adafruit servohat, a picamera and a Raspberry sensehat. By interlinking these different components a suitable foundation for controlling the game has been constructed.

The programming language used is python version 2.7 and it's combined with the vision detecting program openCV. In order for the controlling of the system to work the ball position as well as the path needs to be acquired.

The finished result is a functioning platform where a basic control algorithm has been implemented to show the possibilities of the system as well as the advantages of a gyroscope.

Sammanfattning

Den här rapporten beskriver processen för konstruktionen av ett elektroniskt styrt labyrintspel, detta med hjälp av ett gyroskop och en visionkamera. Målet är att slutprodukten ska vara en färdig plattform där en regleralgoritm ska kunna implementeras.

Systemet består av en Raspberry pi model 3 B, 2 Parallax standard servo motorer, en adafruit servoHat, en picamera samt en Raspberry senseHat. Genom att länka samman dessa komponenter så ska ett lämpligt fundament för reglering av processen konstrueras.

Programmeringspråket som används är Python version 2.7 vilket kompletteras med openCV för möjligheten till bildbehandling. För att regleringen av processen ska vara möjlig så krävs det att bollens position samt väg är känd.

Resultatet är en fungerande plattform där en grundläggande regleralgoritm har blivit implementerad för att visa systemets möjligheter samt gyroskopets fördelar.

Contents

List of Figures	xv
1 Introduction	1
1.1 Background	1
1.2 Purpose	1
1.3 Questions to research	2
1.4 Limitations	2
2 Background Material	3
2.1 The labyrinth	3
2.2 Actuators; Servomotors	3
2.3 An overview of the Raspberry Pi	4
2.3.1 HAT's	4
2.3.1.1 The Sensor HAT	5
2.3.1.2 The Servo HAT	5
2.3.2 The PiCamera	6
2.4 The RPI Software	6
2.4.1 Raspbian (Linux)	6
2.4.2 OpenCV	6
2.4.3 Python	7
3 Methodology	9
3.1 Pilot study	9
3.2 Documentation	9
3.3 Coding	9
3.4 Material	9
4 The pilot study	11
4.1 Platform	11
4.2 Hardware	11
4.2.1 Requirements for the actuators	12
4.2.2 Requirements for the computer	12
4.3 Software	12
5 Hardware system implementation	13
5.1 Getting the servos to work	13
5.1.1 Installing the servo Hat	13
5.1.2 The servo control logic	13

5.1.3	A code example	14
5.2	The SenseHat	15
5.2.1	Accessing the gyroscope	15
5.2.2	Implementing the gyroscope	16
5.2.3	Problems with gyroscope inaccuracy	17
5.3	Stacking HATs	17
6	Vision System	21
6.1	Requirements of the Vision System	21
6.2	Vision System setup	21
6.2.1	Installing the camera	21
6.2.2	Installing openCV; first try	21
6.2.3	Installing OpenCV; second try	22
6.3	Importing the RPI Camera into OpenCV	22
6.4	Fixing the frame-rate	23
6.5	Image processing	24
6.5.1	Circular pattern recognition	24
6.5.2	Color recognition	25
6.5.2.1	Colour recognition with openCV in python	25
6.6	GUI features and user inputs	28
7	Coding and controlling	31
7.1	The main code	31
7.1.1	Importing libraries	31
7.1.2	Declaring variables	31
7.1.3	Creating functions/subroutines	31
7.1.4	Main loop part 1; Getting a frame and finding the ball	31
7.1.5	Main loop part 2; Controller 1	32
7.1.6	Main loop part 3; Controller 2	32
7.1.7	Main loop part 4; set the servos and showing the current frame	32
7.2	The logic behind the controlling	32
7.2.1	Proportional controlling; method 1	33
7.2.2	Proportional controlling; method 2	34
7.2.3	Adding Derivative controlling; PD-controller	36
7.2.4	Adding Integral controlling; PI-Controller	37
7.2.5	PID-controller	38
7.2.6	The Gyroscope-Servomotor Controller	39
8	Results	41
8.1	Research results	41
8.1.1	Possibilities for controlling the system with the help of the RPI and all its modules	41
8.1.2	The accuracy of the actuators that are needed	41
8.1.3	OpenCV as a suitable software for vision programming	42
8.1.4	The implementation of a gyroscope in combination with a vision camera	42

9	Conclusion	43
9.1	Strengths	43
9.2	Weaknesses	43
9.3	Development opportunities	44
	Bibliography	45
A	Appendix 1	I
B	Appendix 2	XIII
C	Appendix 3	XV
D	Appendix 4	XVII

List of Figures

2.1	The labyrinth game (Authors image)	3
2.2	The servo (Authors image)	3
2.3	The RPI (Authors image)	4
2.4	The sense HAT (Authors image)	5
2.5	The servo HAT (Authors image)	5
2.6	The picamera (Authors image)	6
4.1	The platform (Authors image)	11
5.1	Positioning times (Credit to Parallax for the rights to use this image) . . .	13
5.2	The servo logic with the servo Hat (Authors image)	14
5.3	Example code (Authors image)	14
5.4	The PWM (Credit to Parallax for the rights to use this image)	15
5.5	The pitch, yaw and roll logic. (Credit to Raspberrypi.org for giving out the rights to use this illustration)	16
5.6	The mounting of the SenseHat on the labyrinth (Authors image)	17
5.7	The flat cable connection between the servo HAT and the Sense HAT(Authors image)	19
6.1	the Hue value scale (Authors image)	25
6.2	Color recognition code (Authors image)	25
6.3	Color recognition code (Authors image)	26
6.4	First mask (Authors image)	26
6.5	Second mask (Authors image)	27
6.6	Third mask (Authors image)	27
6.7	The final image with the marked ball (Authors image)	28
6.8	The trackbar (Authors image)	29
6.9	The circle and line commands in action (Authors image)	29
7.1	How the distances are calculated (Authors image)	33
7.2	Plotting of the y-axis-actual-value change over time (Authors image) . . .	34
7.3	How the two controllers interact while using method 2 (Authors image) . .	35
7.4	Plot of the P-controller using method 2. Green is the set value and blue is the actual value. (Authors image)	36
7.5	Plot of the PD-controller using method 2 (Authors image)	37
7.6	Plot of the PD-controller using method 2 (Authors image)	38
7.7	Plot of the PID-controller using method 2 (Authors image)	38
7.8	Plot of the P-controller, green is set value and blue is actual value (Authors image)	39

7.9 Plot of how the 2nd controller works when the 1st controller is using PID-
controlling. (Authors image) 39

Glossary

HAT Hardware Attached on Top. 4, 5, 13, 15, 17–19

I2C Inter-integrated circuit. 5, 13, 17, 18

P Proportional. 37, 39

PI Proportional and Integrated. 37

PID Proportional, integrating and derivating. 32, 38, 39

PWM Pulse width modulation. 3, 5, 13, 14, 31

RPI Raspberry Pi. 4–7, 12, 13, 15, 17–19, 21, 22, 24, 41

1

Introduction

1.1 Background

Broccoli is a consulting firm that works mainly in the hardware- and software development industry. In addition to working with consultants they also focus a lot of their time on education and development of new techniques and products.

If you should speak about an technology that is under constant development, vision technology is the one. Vision technology is definitely here to stay which makes it more important to find new innovating ways of implementing it. Broccoli is interested in developing their own version of a so called “labyrinth game”. They want to investigate the possibilities of solving the controlling of the system with different parameters than the ones that are most commonly used in similar setups.

1.2 Purpose

The purpose of this project is to investigate if the controlling of a labyrinth game can be solved with different parameters than the ones that are already used in similar setups today. By researching what different gyroscope setups, as well as actuators, can be implemented for the system a study will be made on the most suitable setup for the specific project.

First a suitable gyroscope, as well as actuators, will be chosen. Then a Raspberry pi micro computer will be used for the implementation of these components. Lastly a control algorithm will be implemented to show the possibilities of the system.

This project also serves as an update to two earlier thesis projects done at Chalmers that both revolved around building a electronically controlled labyrinth game. These projects were done with technology that is now quite outdated and this project will look into what is possible with the help of modern computing and vision systems. It might seem overconfident to both build a platform and controlling it in the same project but much of the newer equipments available today cuts down on both the cost and time needed to set up the hardware and software required. It should also be mentioned that inspiration has been taken from a project found on instructables.com [20].

1.3 Questions to research

- Is it possible/suitable to control a system of this magnitude with a raspberry pi?
- How accurate does the actuators need to be for controlling the system?
- Is a gyroscope a suitable complement to a vision setup?
- Is openCV a suitable software for object tracking?

1.4 Limitations

The project will be limited to constructing a functioning electrically controlled labyrinth game with the implementation of a simpler controlling algorithm. A goal is to construct demo program that will use controlling algorithms.

The controlling algorithm will focus on getting the ball from one position to another. This will be done on a flat surface without the maze.

2

Background Material

2.1 The labyrinth

The labyrinth game used in this project is a brio labyrinth game with the measurement 350x300(mm)[9].

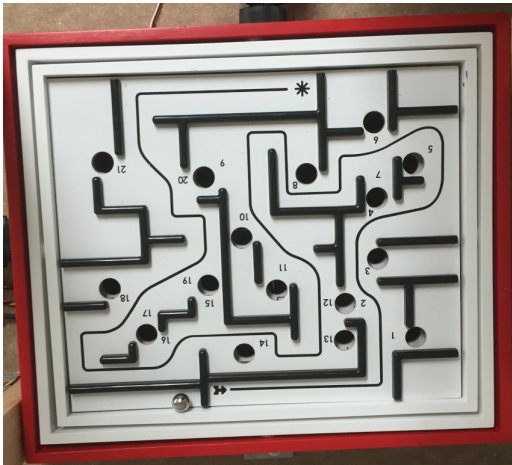


Figure 2.1: The labyrinth game (Authors image)

2.2 Actuators; Servomotors

The actuators used in this project are two Parallax standard servomotors.[5] They are servos that are commonly used in educational programming, such as basic stamp. The Parallax standard servos require a power supply of 4-6 volt and communicate through PWM at a 50Hz update frequency. They are very fast and precise which meets the requirements for the actuators.

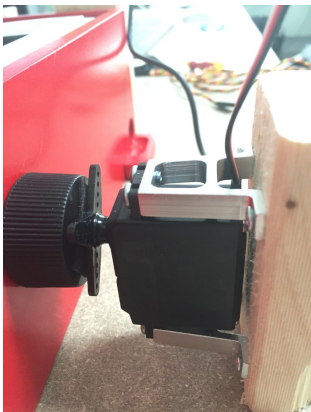


Figure 2.2: The servo (Authors image)

2.3 An overview of the Raspberry Pi

The RPI is a single board computer the size of a credit card, and the version of the RPI that is used in the project is the RPI 3 model B. It's a perfect micro computer for teaching the basics of computers science but also an easy to use platform for all kinds of projects. The main programming language that is used for the RPI is Python, and has a syntax which allows users to express different kinds of codes in less lines than one would do in similar programming languages like C++ or Java[1].



Figure 2.3: The RPI (Authors image)

The RPI 3 model B can run several different OS, all from Windows 10 to Ubuntu. The OS used in this project is Raspbian which is a custom version of Linux for the RPI and is recommended by the Raspberry foundation itself. Raspbian has a high reputation within the Linux community for having high quality, this being the reason it was chosen.

The RPI 3 model B is equipped with a 1,2 GHz 64-bit quad core processor, wireless Local Area Network, Bluetooth, full High Definition Media Input (HDMI), 40 General-purpose input/output (GPIO) pins, 1 GigaByte Random Access Memory, 4 USB ports and several other handy components. [11]

2.3.1 HAT's

HATs, also known as “hardware attached on top”, is one of the many different accessories that can be equipped to the RPI. There are different kinds of HATs that all have different intended purposes.

They are mounted on top of the RPI, connected to the 2x20 signal pins located on the board. [1]

In this project a servo HAT as well as a sense HAT are the ones that has been used.

2.3.1.1 The Sensor HAT

The sensor HAT is a board card that can be added onto the RPI. For this project its main task is to sense the angle of the plane, the roll and pitch, which then is represented in the code as the actual position of the plane.

It is equipped with a 8x8 Light Emitting Diode (LED)-display, a 5 button joystick as well as several other useful functions[2]:

- Humidity sensor
- Accelerometer
- Magnetometer
- Temperature sensor
- Barometric pressure
- Gyroscope

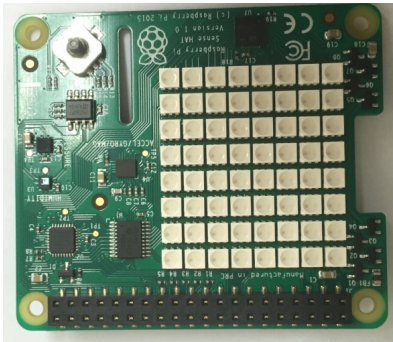


Figure 2.4: The sense HAT (Authors image)

2.3.1.2 The Servo HAT

The servo HAT is an accessory for the RPI that is used to control dc servomotors with very high precision, something that the RPI struggles with otherwise. It can control up to 16 servos with 12 bit precision and only uses two of outputs on the I2C[3].

The servo HAT has two separate power supplies. The VCC, which is the 3.3V that comes from the RPI pins, powers the PWM chip as well as determines the I2C logic[8]. The second is the 5V which powers the servos. The reason for this separate power supply is that servomotors can spike a lot and having a separate power supply protects the RPI from these spikes.

If the current peak from the servos is high, there is a spot on the HAT which a capacitor can be soldered on, this was done for this project.

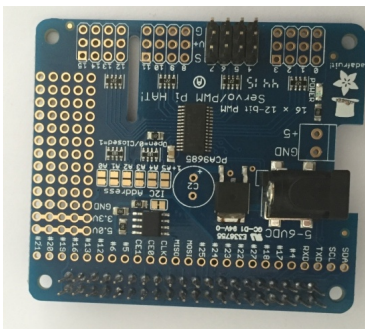


Figure 2.5: The servo HAT (Authors image)

2.3.2 The PiCamera

The picamera is an easy to use, high definition camera built for the RPI and has several useful functions that can be accessed through the picamera python library. It is attached to the RPI with a 15cm ribbon cable to the Camera Serial Interface (CSI) port.[4]



Figure 2.6: The picamera (Authors image)

2.4 The RPI Software

This section will introduce the software tools that was used in the project. All of them are supported and well integrated by the RPI system.

2.4.1 Raspbian (Linux)

The RPI is commonly delivered without a pre-installed OS so choosing an OS is up to the user. It is although recommended to buy an Secure Digital (SD) card with either NOOBS or Raspbian on it. Raspbian is a version of Linux that is optimized for usage on the RPI and is recommended by the RPI itself. Raspbian comes with many useful programming tools pre-installed like Python, Scratch and Java. This makes it so that all the needed programming languages are available already at first startup of the RPI. NOOBS is an operating system installer that contains Raspbian as well as other operating systems that can be downloaded and installed. It is for example possible to use Windows or os-x on the RPI although it is not really optimized and as easy to use for programming on the RPI.

2.4.2 OpenCV

OpenCv is an open source free-to-use library of programming functions that is available for several different programming languages such as C++ and Python as well as for many different platforms like Windows, Linux and Android.

OpenCv stands for Open Source Computer Vision and it has several appropriate applications for picture and video analysis. The library was developed by Intel's russian software team led by Gary Bradsky and Vadim Pisarevsky and was first released in 2000.[10] The library is still under constant development and the version used in the project was OpenCv 3.0.0 which was released in June 2015. Although

OpenCV is often used on the RPI platform it is not really straight forward to install right away. There are a lot of tutorials available on the web on how to do it but it still takes a lot of trial and error before it works. Most installations also take up to 3 hours to make. Some things to look into is which version of Rasbian that the tutorial is made for and which RPI it is done on. Once a functioning version of OpenCV is installed on the RPI all the useful functions for vision-programming are available in the Python code after importing the libraries. One thing to add about OpenCV is that it requires some other libraries in order to use all the functions available. One of them is Numpy which is a tool for scientific calculations and numerical operations. All OpenCV array structures is done in Numpy.

2.4.3 Python

As mentioned before the Rasbian build of RPI comes with two versions of Python already installed. Python version 2 and 3 are very similar and for the most part indistinguishable. There are a few libraries that are not yet supported for version 3 so that is the only reason that this project was mainly done in version 2. Programming anything that makes use of the pins as input and outputs on the RPI is almost always done through Python. But that doesn't mean python is restrictive. It is a widely-used high-level programming language that can mostly serve the same purpose as similar languages like C/C++. The main difference between Python and C/C++ is that python is very emphasized on efficiency and more easily readable code that can be expressed in fewer lines. An example of this is how you can construct a simple for-loop in C/C++ and in Python. This is in C/C++:

```
#include <iostream>
using namespace std;

int main ()
{
    for( int b = 10; b < 16; b = b + 1 )
    {
        cout << "value of b: " << b << endl;
    }

    return 0;
}
```

And this is the same thing in Python:

```
for b in range(10, 16):
    print "Value of b: %d" % (a)
```

Both these codes will print out:

```
value of b: 10
value of b: 11
value of b: 12
value of b: 13
```

2. Background Material

```
value of b: 14  
value of b: 15
```

It is easy to see that the python demands much fewer lines to execute the same thing as the C/C++ code. Python does not use curly braces { } to control the flow of the code but relies simply on white-space indentations. This is already an implied way for people to write code especially in the C/C++-languages so that you get a code that is easier to read and understand.

One downside to Python is that it is slower^[10] than C/C++. But it is also possible to combine Python code with C/C++. That way computationally intensive codes can be written in C/C++ and then made into python modules through wrapping and imported into python code. OpenCV makes use of this feature to make the complicated C/C++ algorithms run faster in the background while the main code is written in simple, easy to understand, but slower Python-code.

3

Methodology

This chapter explains the approach that has been taken for this project.

3.1 Pilot study

In order to be able to take on the task of creating a sensor as well as a vision aided labyrinth game some research had to be made. This to ensure that the task was possible and could be done within the time frame.

Before the project could get started some research had to be done to choose a suitable platform, hardware and software. This was done by setting up requirements and afterwards doing a study to decide the best option for the project.

3.2 Documentation

The documentation during the project has been done in a Google drive map which can easily be accessed by both members of the group. Here everything from the daily blog to the excel document that describes a plan for every week of the project can be found.

3.3 Coding

The coding during the project has been done "step by step". By not focusing on coding the final product from the start instead the coding has been done by focusing on each required task and later on combining them. This with the intention of saving time as well as getting a better understanding.

3.4 Material

- L^AT_EX
- Google Drive
- Python
- OpenCV
- Simulink
- Photoshop

4

The pilot study

4.1 Platform

The requirements for the construction platform was the following:

- An easily built platform but not on the cost of the end result
- The possibility to implement a camera above the labyrinth
- The possibility to implement a gyroscope
- The possibility to implement two actuators on the steering knobs

By taking these requirements into consideration, a study was made about how the platform was to be built. A lot of the more time consuming constructions were discarded because they were unnecessarily complex. Instead the decision was made to build a wooden platform that had the necessities needed for the project.

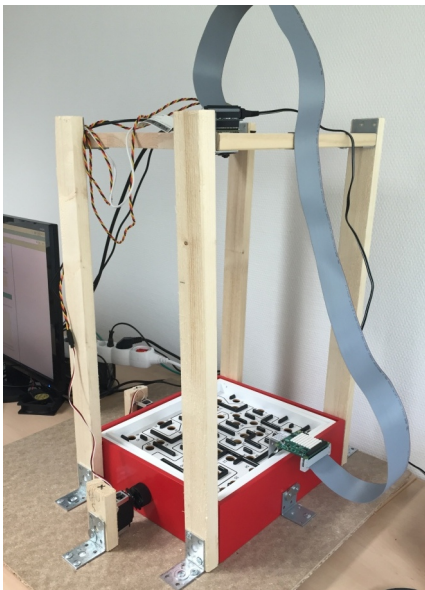


Figure 4.1: The platform (Authors image)

4.2 Hardware

The different hardware component used in the project is probably the most vital decision that was to be made initially. This decision sets a lot of the limitations as well as the possibilities.

In order to make proper decisions, a study had to be made, where the different requirements varied depending on the component.

4.2.1 Requirements for the actuators

- High precision
- Affordable
- High speed

After these requirements were set the Parallax standard servo motors were chosen.

4.2.2 Requirements for the computer

- Powerful enough for image processing
- Affordable
- Useful accessories
- Easily installed

After these requirements were set the RPI 3 model B was the micro computer that was chosen.

4.3 Software

When the decision was made to use the the RPI 3 a study about what software programs that were going to be used in the project was made. The study was made based on two different requirements:

- The software needs to be able to process images
- It must be a widely used software with a good reputation from the general community

After taking these requirements into consideration in our research the decision was made to use a coding language called Python as well as combining this with a image processing program called openCV.

5

Hardware system implementation

5.1 Getting the servos to work

5.1.1 Installing the servo Hat

Once the servo HAT is attached to the RPI there is some configuring that needs to be done. The first step is to type `sudo apt-get install python-smbus` in the terminal, this adds the I2C support for python. When that is done the python library needs to be imported. This is done by writing `git clone https://github.com/adafruit/Adafruit-RaspberryPi-PythonCode.git`, `cd AdafruitRaspberryPi-PythonCode` and `cd Adafruit_PWM_Servo_Driver` in the terminal. How this library is imported in the Python code is explained in the subsection 5.1.3.

5.1.2 The servo control logic

This section explains how the servo motors is controlled with the help of the imported Python library for PWM-controlling. Note that this section assumes that the reader has some basic knowledge about PWM.

It's known that the servo HAT communicates with a 12-bit protocol[3] and the servos work out of a 50Hz frequency[5]. So the servomotor needs a new signal every 20 milliseconds and the length of this signal determines which position it will be set to. The Adafruit PWM code library can set the length of this signal as a 12-bit number with 0 being no signal and 2^{12} (4096 bits) being 20 milliseconds if a frequency of 50 Hz has been set. The length of this signal that the servomotor requires can be found in the data-sheet 5.1. The servomotors goes to the minimum-position when given a signal of 0.75 milliseconds and maximum-position when given a signal of 2.25 milliseconds.

BASIC Stamp Module	0.75 ms	1.5 ms (center)	2.25 ms
BS1	75	150	225

Figure 5.1: Positioning times (Credit to Parallax for the rights to use this image)

To set the length of the signal, the number of bits then needs to be calculated for setting a minimum, center and maximum position. So the equation 5.1 shows how many bits that is required to get a signal of one millisecond:

$$\frac{2^{12}[\text{bits}]}{20[\text{ms}]} = 204.8[\text{bits}/\text{ms}] \quad (5.1)$$

It's now possible to calculate the amount of bits that needs to be set to get the servos in different positions, as shown in equation 5.2, 5.3 and 5.4:

$$\text{Minimumposition} : 0.75 * 204.8 = 153.6[\text{bits}] \quad (5.2)$$

$$\text{Centerposition} : 1.5 * 204.8 = 307.2[\text{bits}] \quad (5.3)$$

$$\text{Maximumposition} : 2.25 * 204.8 = 460.8[\text{bits}] \quad (5.4)$$

The figure below 5.2 further illustrates how this applies to the positioning of the servomotor.

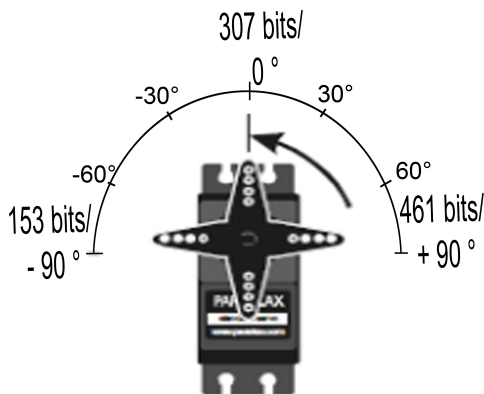


Figure 5.2: The servo logic with the servo Hat (Authors image)

5.1.3 A code example

A code that moves the servo to the center position could look like figure 5.3:

```
from Adafruit_PWM_Servo_Driver import PWM #Imports the servohat PWM library
pwm = PWM(0x40) #Sets the adress
pwm.setPWMFreq(50) #Sets the freq to 50Hz
centerposition = 307 #Sets the servo in center position

pwm.setPWM(4, 0, centerposition) #Sets servo 4 at centerposition
```

Figure 5.3: Example code (Authors image)

In the first line of the code the **PWM** function from the **Adafruit_PWM_Servo_driver** library is imported. The third to fifth line then sets which address is going to be written to, what frequency is going to be used as well as the desired position. Then, on the seventh line, the **pwm.setPWM** function is used to choose the desired position of the servo. The function takes three arguments, the channel for which servomotor will be set, the time (number of bits) for when the signal will go high and then time for when the signal will go low again. Argument two is for the sake of simplicity set to zero so that the signal always starts at a high flank. This way the third argument can be the number of bits that directly sets the length of the signal.

The PWM signal would then look something like figure 5.4.

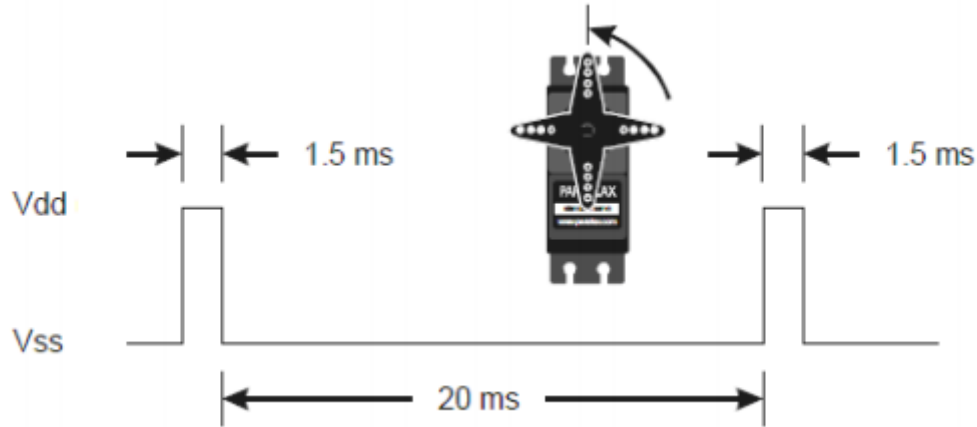


Figure 5.4: The PWM (Credit to Parallax for the rights to use this image)

5.2 The SenseHat

The SenseHat is all about sensing the environment. It is actually especially made for the AstroPi-mission to make measurements of the environment inside the international space-station (ISS)[15]. This section will explain the logic and how to access the gyroscope in the Sense HAT.

5.2.1 Accessing the gyroscope

There is a library of functions available on the RPI for accessing each sensor and the display on the SenseHAT. This library can be installed on the RPI just by typing `sudo apt-get install sense-hat` into the command window. With the help of this library the orientation values from the gyroscope can easily be accessed. The command is called `get_orientation_degrees` and it will return a dictionary data type with the gyroscopes pitch, yaw and roll-values. A dictionary is a data type commonly used in python that works like a list but with a name for each element as well as a value. To get the pitch value from the dictionary, just type `get_orientation_degrees["pitch"]`. Pitch, Yaw and Roll are values that are used to describe the orientation of an object in space. Pitch and Roll are the values that are being used in this project.

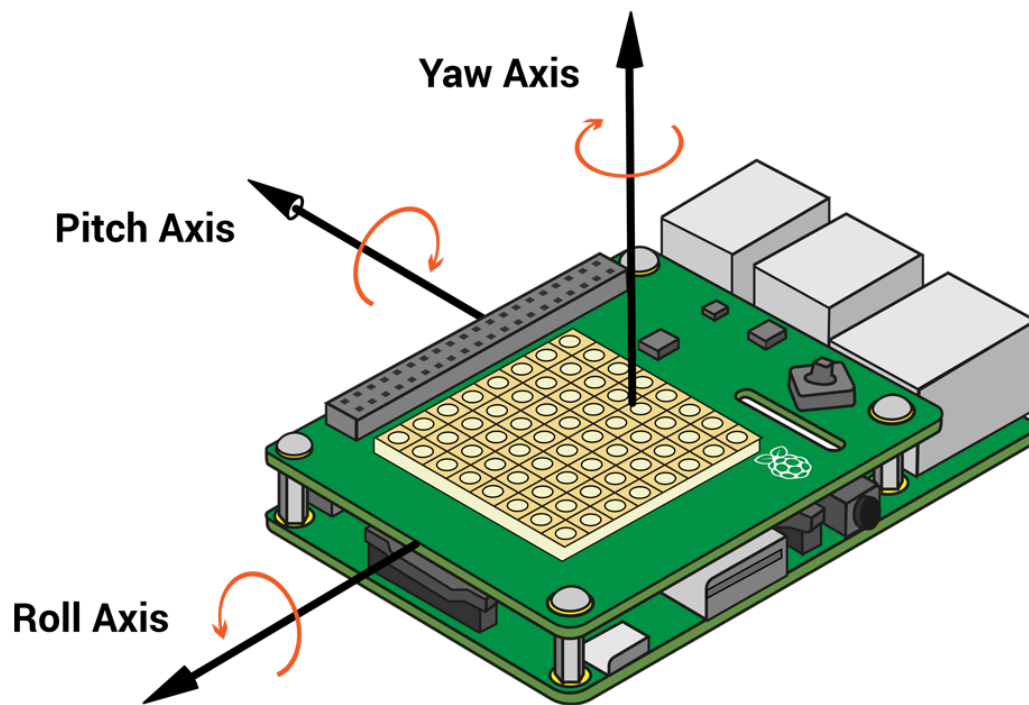


Figure 5.5: The pitch, yaw and roll logic. (Credit to Raspberrypi.org for giving out the rights to use this illustration)

5.2.2 Implementing the gyroscope

The senseHAT was attached on the long side of the labyrinth directly to the center plane so that the gyroscope has the same inclination as the plane itself. So if the gyroscope works as intended it could be implemented for a better controlling of the labyrinth. Some possible applications are the following:

- It could record the path of the ball by inclinations over time. Therefore if the ball gets through the labyrinth successfully, it could possibly be recorded and played out with the same results each time.
- The gyroscope makes more accurate equations possible on how a ball behaves in inclined planes.
- A fault in the attachment of the servomotors on the labyrinth knobs makes it so that the plane and the servomotors becomes oblique if enough pressure is put on the planes. Sometimes the servomotors have to be re-calibrated and reattached so that the starting position correspond to the plane being a flat surface with no inclination. This can be worked around by using the gyroscope to calibrate the servomotors instead each time the code starts to start out from a flat surface.
- The servomotors can further be controlled by the gyroscope so that an inclination can be requested by the code and the gyroscope can guide the motors to actualize the plane into this position.

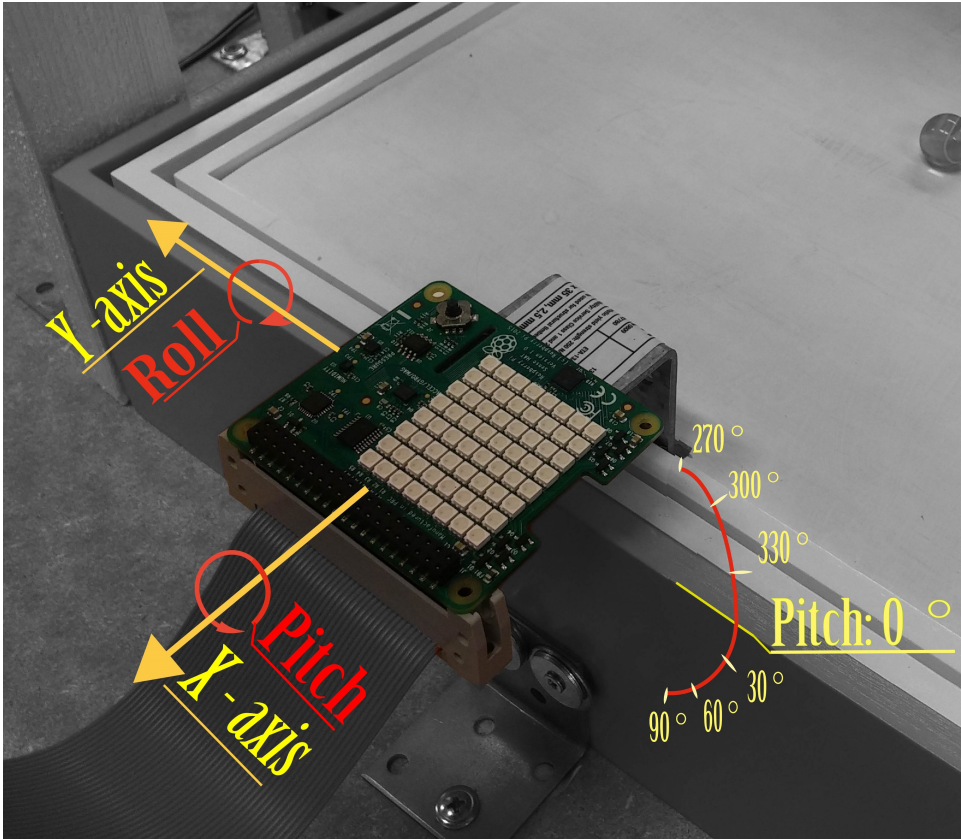


Figure 5.6: The mounting of the SenseHat on the labyrinth (Authors image)

5.2.3 Problems with gyroscope inaccuracy

To get an accurate value from the gyroscope the `get_orientation_degrees` needs to be addressed/updated frequently. The sensor HATs library-code needs to do a lot of measurements in order to get the right value for the environmental sensors. So if the rest of the main code takes up a long time and the gyroscope is only addressed once in the main loop the values becomes unreliable. In most of the codes used in this project the loop takes up at least 30 milliseconds which is too slow to accurately measure orientation. The solution to this issue is solved by creating a separate class and a technique called threading[12]. This creates a separate loop which pulls values from the gyroscope in the background while running the main code. When using this method, orientation-values becomes reliable and accurate. See appendix D for the code that was created to solve this problem.

5.3 Stacking HATs

One big reason to use RPI in a project like this is the possibility it has for using multiple modules together and controlling them by a method called "stacking". The two main modules that are being stacked in this project are the servo HAT and the sense HAT. The camera module is also being used in conjunction with the HAT-modules but this module will basically work no matter what you stack on top of the RPI. The RPI and the different modules communicate with the help of a previously mentioned protocol called I2C which makes "stacking" possible. Once I2C-Kernel

support is activated the RPI can automatically recognize what modules are being stacked on top of it. The servo HAT takes up two I2C addresses when attached. You can look up which these addresses are by typing "sudo i2cdetect -y 1" into the command window. Something like this will then show up:

```

    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- -- -- -- -- -- -- -- --

```

These are the I2C-addresses and, as you can see, the addresses 0x40 and 0x70 are being used up by the servo HAT. If the only sense HAT is mounted on the RPI and the same command is run something like this will show up:

```

    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- UU -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- 5c -- -- 5f
60: -- -- -- -- -- -- -- -- -- -- -- -- 6a -- -- --

```

The addresses of interest is 0x70 and 0x40, and that they aren't occupied when the sense HAT is mounted. This means that the two modules could be stacked on top of each other without signals colliding and disturbing each other. So the servo HAT was mounted directly on top of the RPI and the Sense HAT was then mounted on top of that. When then starting up the RPI and running the I2C-detect command this showed up:

```

    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- 46 -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- 5c -- -- 5f
60: -- -- -- -- -- -- -- -- -- -- -- -- 6a -- -- --
70: 70 -- -- -- -- -- -- -- -- -- -- -- -- -- --

```

The servo HAT worked as intended but the Sense HAT did not function properly. This is because the position 0x46 should be a so called UU-index position. So the RPI is not running the correct Sense HAT initiation upon start-up because it does not recognize the module when stacked on top of the servo HAT. With the help of a forum-thread on the Raspberry website [16] it was discovered that you can force the

RPI to recognize and initialise the Sense HAT even when being stacked. There is a text-document called **config.txt** in the boot-folder of the RPI that can be edited to change the start-up sequence. If the line **toverlay=rpi-sense** is added here the RPI will recognize the Sense HAT again. After rebooting the RPI and running the I2C-detect command it will now show this:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00:				—	—	—	—	—	—	—	—	—	—	—	—	—
10:	—	—	—	—	—	—	—	—	—	—	—	—	1c	—	—	—
20:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
30:	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
40:	40	—	—	—	—	—	UU	—	—	—	—	—	—	—	—	—
50:	—	—	—	—	—	—	—	—	—	—	—	—	5c	—	—	5f
60:	—	—	—	—	—	—	—	—	—	6a	—	—	—	—	—	—
70:	70	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

The functions of both hats worked identically as when they where mounted separately on top of the RPI after doing this.

Because the Sense HAT needed to be attached directly to the labyrinth-game it could not be stuck on top of the RPI. Therefore a flat cable was connected between the servo HAT and the Sense HAT so that the Sense HAT could be on a separate location.



Figure 5.7: The flat cable connection between the servo HAT and the Sense HAT(Authors image)

6

Vision System

6.1 Requirements of the Vision System

With the help of tools like openCV, setting up a vision system on any device that has C/C++, Python or Java and a camera attached to it has become much more straightforward in recent years. There are several OpenCV functions that can be implemented into this project. The most important thing is to find the position of the ball inside the labyrinth-game. Preferably in a frame-rate that makes precise controlling possible. Other interesting inquiries that will be explored if time allows:

- Is it possible to find the black lines indicating the path that the ball is supposed to take and use it as a guideline in a program to steer the ball through the maze?
- Is it possible to map out the different holes and use their coordinates to avoid that the ball falls into them?
- Is it possible to map out the perimeters of the game such as the outer frame and the position of the walls inside the maze? Could that information be used in a meaningful way?
- Is it possible to map out earlier positions of the ball both with position and time in order to figure out velocity and acceleration?
- Could the inclination of the planes be calculated with the help of the gyroscope so that a predicted path of the ball could be mapped out in openCV?

6.2 Vision System setup

6.2.1 Installing the camera

In order for a vision system to be set up on the RPI one first needs to have some sort of camera attached. Either a USB-Webcam or a camera module that is designed especially for the RPI system. The RPI camera is quite straightforward. First open a Linux terminal window, install the camera through the **apt-get install** command and then enable camera module inside the RPI-config. A more detailed explanation for this can be found online at raspberry Pis own website.

6.2.2 Installing openCV; first try

Once a functioning camera was installed and tested, OpenCV needed to be installed. This was done by following a tutorial on a blog about vision technology[14]. All the

tutorials on how to install OpenCV involves downloading a zip-file from the OpenCV website, unpacking it, installing and building the Libraries on the RPI. The first tutorials way to use OpenCV in Python involved opening Python through a virtual environment inside the terminal in order to have access to all the OpenCV functions. The main goal when one knows that OpenCV works with Python is to inside a python prompt type **import cv2** without getting any error messages. That only worked if the Python-program was first accessed through the virtual environment. A big problem was though that this method of running openCV on Python took up almost all the RPIs processing power when running just an OpenCV test program. A program that where to run all the modules required at the same time and OpenCV could possibly be too strenuous on the RPI. So the decision was made to start over with another way of installing OpenCV on the RPI.

6.2.3 Installing OpenCV; second try

After removing every trace of the previous installation another tutorial was followed[13]. One major difference between this method and the first was that the installation was done by controlling the RPI remotely through a separate computer using an ssh server. Using the graphic interface on the RPI takes up unnecessary processing power which is needed since during the build-phase of the installation almost all processing power available is being used on the RPI and overheating is a possibility. After this installation was done an OpenCV code was tested in Python. No virtual enviroment was necessary this time. The program ran much more smoothly with just about 20 percent of the RPIs processing power. This method of installing OpenCV is therefore recommended over the first.

6.3 Importing the RPI Camera into OpenCV

The OpenCV developers have made it very straightforward to import video and pictures from a camera that is connected to the device, as long as that camera is a web-cam connected via USB. The RPI camera is not a web-cam so all the useful functions for importing video in OpenCV cannot unfortunately be used, at least not directly. The RPI Camera does although have a set of its own functions which can be addressed in a similar way for recording or streaming video that can then be used in OpenCV. In order to use the RPI camera in the code one needs to first import the modules at the top of the code. These are called **PiCamera** and **PiRGBArray**. PiCamera and PiRGBArray are both classes that have multiple functions within them. The functions used in the code are the following:

- **PiCamera.resolution:** modifies the resolution of the camera. The piCamera can support a resolution of maximum 1920x1080 and can then use a framerate of 1 to 30 frames/second. The values are assigned as a tuple of 2 values like for example: `PiCamera.resolution = (640,480)`.
- **PiCamera.framerate:** modifies the frame-rate that the camera will capture video at.
- **PiCamera.hflip/vflip:** Flips the picture recorded horizontally or vertically.

- **PiRGBArray(Picamera(), size=(320, 240))**: This is a way to get a faster access to the stream by using the usual JPEG-type data that the Picamera class uses. OpenCV uses bgr-type data anyway for processing images. Usually stored in a variable called `rawCapture`.
- **piCamera.capture_continuous(rawCapture, format="bgr", use_video_port=True)**: (BGR = Blue Green Red) This starts a camera-stream which can later be accessed in the code.

To get a frame from the stream for each repetition in the program, the code should look something like this:

```
... initiation ...

camera = PiCamera()
camera.resolution = (640, 480)
camera.framerate = 33
camera.vflip = True
rawCapture = PiRGBArray(camera, size=(640, 480))

# Camera warm-up time
time.sleep(0.1)

for frame in camera.capture_continuous(rawCapture,
format="bgr", use_video_port=True):

    img = frame.array

... rest of the code ...
```

The for-loop that started at the end of the code becomes the main-loop for using the camera stream and the `img`-variable is the latest frame. This frame is where the OpenCV functions can be applied. To show the captured frame on the screen just use the function **cv2.imshow("windowname",img)** to create a window that shows the current frame. This will show the stream created in the for-loop in that window frame-by-frame.

6.4 Fixing the frame-rate

One problem with using the `Picamera()`-method though is that the frame-rate on the stream that is processed in the for-loop is dependant on the speed of the code and not the frame-rate set for the Pi camera. If the code is for example slower than the frame-rate of the stream, there will be a blocking of frames gathered from the stream that the code cannot keep up with. This will create a slow and inaccurate stream and might also create untrustworthy results from the OpenCV calculations. The issue comes because the stream-capture and the image-processing-code are both handled in a single main thread of the program. If these two are different threads that don't interfere with each other the frame-rate of the stream becomes radically faster. The camera stream is handled in its own thread from which the image-

processing gathers the latest frame from whenever it needs.

The frame-rate issue could easily be spotted by the naked eye when running a code for identifying the balls position and steering it toward a chosen point on the labyrinth. The frame-rate could sometimes be so bad that the ball would be on other side on the board from where it where in the latest frame shown. To find what caused this problem, the speed of the code-execution was examined for each lap of the for-loop. The code-execution varied between 80 and 100 milliseconds. So the frame-rate used by the code was therefore between 10 and 12,5 frames per second. In order to get a functional vision system this issue needed to be addressed.

There is a package of python-tools available called Imutils that is created by Adrian Rosebrock of the pyimagesearch website. He has a tutorial on how to improve the frame-rate of the RPI camera for usage in openCV[12]. In the Imutils package there is a python-class called PiVideoStream() that can be used instead of the pi-Camera() class. The command PiVideoStream().start() starts a stream from the RPI camera in a separate thread from the main programs thread. The command PiVideoStream().read() will now get the latest frame gathered from the stream.

This method of streaming from the RPI camera greatly improves the frame-rate. The codes execution cycle time did go down to about 30 milliseconds or less which means that it shows a much more desired frame-rate of about 33 frames per second.

6.5 Image proccessing

Object detection is a task that is very important, with the downside of being very difficult, when it comes to vision technology. OpenCV has several useful ways of detecting different objects in an image, two of them explained in the subsections 6.5.1 and 6.5.2.

6.5.1 Circular pattern recognition

One of these ways of detecting an object in an image is to do this by detecting certain patterns, in this case a circular pattern. This is done by using a extraction technique called the **Hough Circle Transform**[17], which is a product of the original **Hough Transform**[18].

The basics of hough transform is to, by given a set of edge points, find aligned points in images that create, in this case, a circle. It uses an algorithm that looks for these edge points in an image and how many of them are aligned for certain objects.

This method of detecting a circle is very complicated and requires time consuming algorithms, which slows down the code. The method was therefore not used for this project.

6.5.2 Color recognition

6.5.2.1 Colour recognition with openCV in python

Color recognition in openCV is really about filtering out the unwanted colors and making them black while the colors that are interesting will be white. The recognition-functions needs stark contrasts for it to work effectively. The key function for filtering out color is called **cv2.inRange()** and it filters out colors by working with the so called HSV-scale.

The HSV-scale is a three dimensional way to represent colors with. HSV stands for Hue, Saturation and Value. To put it simply the variable of interest here is the Hue, since it represents the entire color-scheme while the Saturation is the purity of the color and Value is the brightness. This is more thoroughly explained on Wikipedia.[21]

The **Cv2.inRange()**-function takes three arguments, the picture that the mask will be applied on, the lower HSV-scale limit and the upper HSV-scale limit. The HSV-scale is represented as a cylinder with Value being its height, saturation its radius and Hue is the circle around represented as a degree. So the argument for Hue in the **Cv2.inRange()**-function is in this case a number between 0 and 180 degrees as can be seen in the figure below 6.1. While the arguments for Saturation and Value are both numbers between 0 and 255.



Figure 6.1: the Hue value scale (Authors image)

To explain how this would work in python a example code has been constructed, see figure 6.2

```
colorlower = (50, 86, 6)
colorupper = (100, 255, 255)
while(1):
    image = vs.read()                #Gets image from camera
    image2 = image                   #Creates a copy of the shown image

    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)    #Shows the color in the chosen spectrum
    mask1 = cv2.inRange(hsv, colorlower, colorupper) #Threshold the HSV image
    mask2 = cv2.erode(mask1, None, iterations=2)    #Erodes 'mask' 2 times
    mask3 = cv2.dilate(mask2, None, iterations=2)   #Dilates 'mask' 2 times
    findball(mask3, image2)

    cv2.imshow('image', image)        #shows 'image'
    cv2.imshow('mask1', mask1)        #shows 'mask1'
    cv2.imshow('mask2', mask2)        #shows 'mask2'
    cv2.imshow('mask3', mask3)        #shows 'mask3'
    cv2.imshow('image2', image2)      #shows 'image2'
```

Figure 6.2: Color recognition code (Authors image)

The first step of the code is to set the lower and upper color limit, **colorlower** and **colorupper**. In the code example 6.2 the Hue limit is set between 50 and 100 so the **Cv2.inRange()**-function will therefore filter out all the color except green and bright blue, as seen in figure 6.1. The Saturation is set between 86 and 255(maximum) so it filters out any color with Saturation below 86 which is basically just white. Similarly, the function only filters out Value that is below 6 which is basically just black.

Lets take an image as an example, see figure 6.3

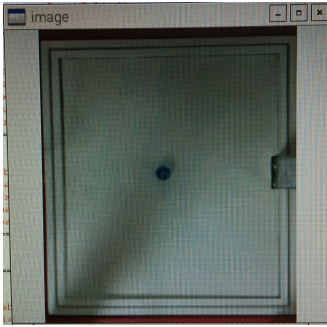


Figure 6.3: Color recognition code (Authors image)

When image 6.3 is read it is then converted to be within the modified HSV scale with the openCV function **cv2.cvtColor**. This function basically allows the user to work with the HSV color scale.

To then be able to mask out the color of interest the function **Cv2.inRange()**, as previously mentioned, uses the lower and upper color limit to then get a image where the color of interest is marked with a white color and the rest is black, as seen in figure 6.4.

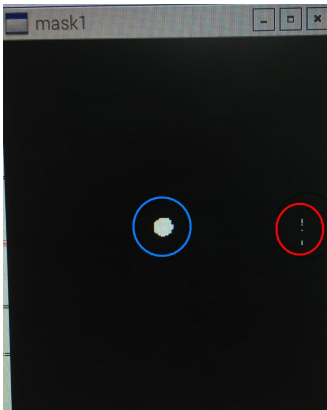


Figure 6.4: First mask (Authors image)

In the image there is now a white round object, in the blue circle which represents the detected ball, and a bit of white 'noise', in the red circle. The white noise is removed by using the openCV function **cv2.erode** and the result of it is shown in figure 6.5.

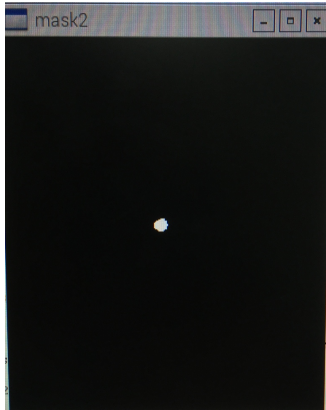


Figure 6.5: Second mask (Authors image)

This image is then enhanced with the openCV function **cv2.dilate** and the result of it shown in figure 6.6. This is now the final mask that is used for finding the position of the ball in the game.

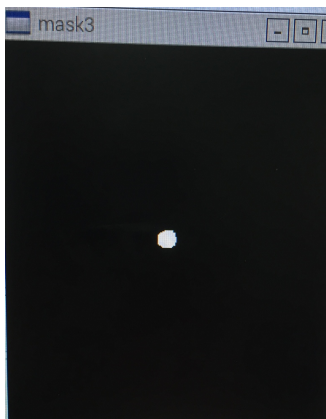


Figure 6.6: Third mask (Authors image)

It is now possible to use the OpenCV-function **cv2.findContours()** with the mask that has been created. This function is called **findball()** and can be found in the appendix C C. OpenCV uses the contrast between the black and the white and maps out the cohesive white areas of the picture and returns a list of pointers to these. The code now checks if at least one white area has been found and then it gets the coordinates to the largest one. It then draws an enclosing circle around the white area and maps out the exact coordinates to the center of this circle. These center coordinates are now used in the rest of the code as the balls position. The result of this can be shown in figure 6.7



Figure 6.7: The final image with the marked ball (Authors image)

Because you mask out the object of interest color recognition works a lot faster than circular pattern recognition, this being the reason it was used in this project. A code was created which only is to find and map out the ball and this code was able to project a stream that showed the balls position accurately while still running at about 30 frames per second. The speed of the calculations are quick and maybe for importantly consistent in speed.

In the final code, the picture taken from the camera has a resolution of 300x350 and the position coordinates that is calculated are then one coordinate in x and y of these 300x350 pixels. The function that creates a circle around the contours of the ball and then takes the circles center-position as the balls position are pretty accurate as to being the actual balls center-position. This would be more of an issue if the object for color-recognition wasn't a round shape.

6.6 GUI features and user inputs

OpenCv has a wide range of tools for gathering user input. Many of these were used in the project in order for creating more user-friendly and interactive programs. It is possible for example to mark a point in the picture and to draw out circles, lines and paths from where the mouse is clicked. The functions that were used in the project were these:

- **cv2.setMouseCallback()** This function tracks the mouse's movements and actions inside a window that has been created. It calls to an interrupt-function whenever an event inside the window takes place. From inside this function three parameters can be read; the x-and-y position of the mouse cursor and what type of event that triggered the interrupt. These events can be that the mouse moved or that a mouse-button was pushed. So the function is very useful to be able to mark a position in the frame and get its exact coordinates for future calculations.
- **cv2.getTrackbarPos()**, **cv2.createTrackbar()** These functions are used to set up and to read values from a Trackbar that is created in a separate window. It is possible to create multiple trackbars inside the same window,

all with different names and value-ranges. The function can be used in a very similar way to `cv2.setMouseCallback()` in that it can trigger an interrupt function whenever a value is changed. `cv2.getTrackbarPos()` can also be used directly in the code just to read the current value set on the trackbar, see figure 6.8.

- **`cv2.circle()`, `cv2.line()` and `cv2.rectangle()`** These are straightforward functions that can draw something on a frame or a picture. Basically uni-

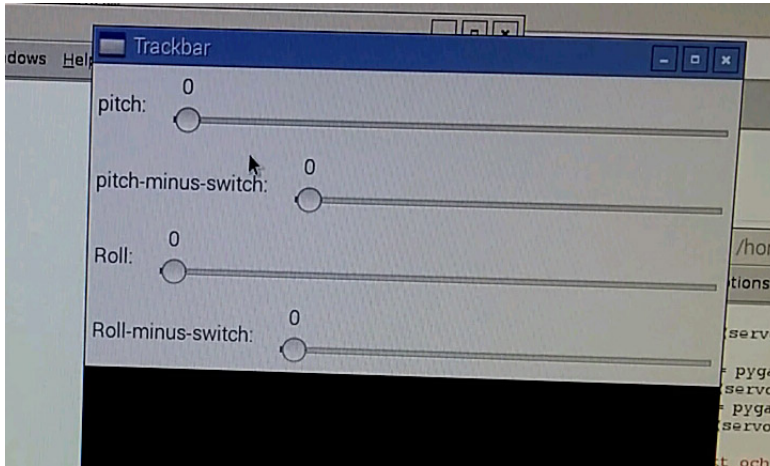


Figure 6.8: The trackbar (Authors image)

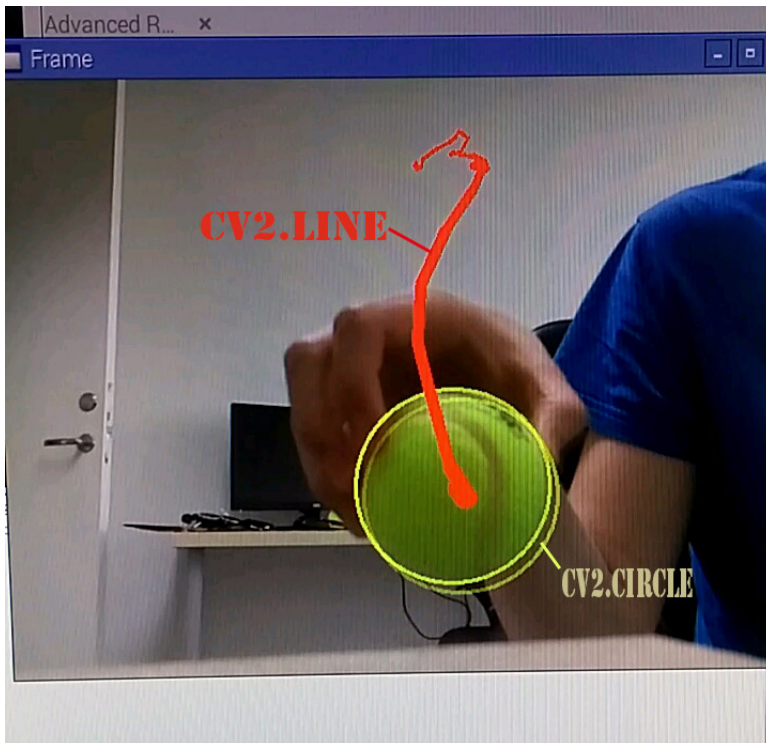


Figure 6.9: The circle and line commands in action (Authors image)

7

Coding and controlling

7.1 The main code

In this section the code will be explained, please refer to appendix A for the code and appendix B for a flowchart of the code.

7.1.1 Importing libraries

Many functions that are used in the code needs to be imported into the file with the help of libraries. There are also classes that are imported into the project like the PWM-class, PiVideoStream-class and the orientationstream-class that was created for this project.

7.1.2 Declaring variables

Many variables are used as globals in the code like the variable for the balls position, the chosen position and the length between those two. There are also other constant values that are being declared like the Servos maximum and minimum position and the upper and lower limits for the HSV-scale color mask. It is also here in the code where the camerastream and gyroscopestream is initialized.

7.1.3 Creating functions/subroutines

To make the main code shorter some parts of it was instead put into subroutines.

7.1.4 Main loop part 1; Getting a frame and finding the ball

Since the camera-stream is running in the background it is now possible to grab the latest frame from it and put it into a variable called **image**. This image will be converted into an HSV-image since the OpenCV-functions always uses HSV. After applying two more filters to the image, the balls shape can be spotted with the help of **cv2.findContours()** and its center-position can be found with the help of **cv2.moments()**. This center-position is now the actual value of the position of the ball in the labyrinthgame. This is called **ballposition** in the code. This variable along with the set-value **chosenpoint**-variable is converted from pixel-form into metric-form in the function **pixeltometric()**.

7.1.5 Main loop part 2; Controller 1

If a chosen position has been set and the ball has been found, the first controller will calculate a distance between the ball and the chosen position which then will act as the error-value for the first controller. This will be explained further in the controlling section 7.2. Other variables like for example `old_distx`, `sumx` and `h` will also be explained further in the controlling section since they are needed for creating a functioning PID-controller. The controller function itself is the function called `get_ux` and `get_uy`. The resulting variable that is returned from the controller is the set value for the inclination of the planes which is called `bor_roll` and `bor_pitch`. The list of the set and actual values is also updated here so that they can be plotted out.

7.1.6 Main loop part 3; Controller 2

This controller is explained in greater detail in the gyro-servo-control section 7.2.6. The actual value for the inclination of the planes are gathered from the gyroscope-stream and translated into a number between -6 and 6 degrees. Then the plot-list of this controller is updated and the error-value `distp` and `distr` for the controller is calculated. The output-control-value `Servosetpitch` `Servosetroll` to the servo-motors are then ramped up proportionally to the calculated error-value.

7.1.7 Main loop part 4; set the servos and showing the current frame

The last thing that happens in the main loop is that the servo motors are updated with the control value from the second controller. The window is also being updated with the latest frame gathered at the start of the loop with both the balls and the set-position marked out.

7.2 The logic behind the controlling

Even though this projects main focus has not been to analyze and create the best system for controlling possible, this chapter will go through what possibilities there is for controlling and what has been tried. The different methods for controlling was taken from the course book in dynamic systems[19] the main purpose for the controlling in an automated labyrinth-game is to simply get the ball rolling in the direction and to the destination that is requested. In the logic that was used in the testing programs a two dimensional coordination plane was used to represent the board of the game. The y-axis is the long side of the board the x-axis is the short side. This way the servomotor that manipulated the pitch-value of the plane could move the ball two-dimensionally along the y-plane and the roll-manipulating servo could move the ball along the x-axis. So if the ball was in a position (x_1, y_1) and it is requested to go to a position (x_2, y_2) it should need to travel a distance of Δx along the x-axis and Δy along the y-axis.

This simple logic is the cornerstone for the controlling that has been done in this project. Each servo that controls the inclination (pitch and roll) for the plane are

viewed as separate controllers that handle the balls position along the x and the y- axis. To make it simple this text will now on focus on one of the axis since the logic for the other is the same but with different parameters. So the calculated distance Δx will become the error-value $e(k)$ which is the difference between the measured position-value for the ball and the set position-value. This is the basis for the controlling algorithm.

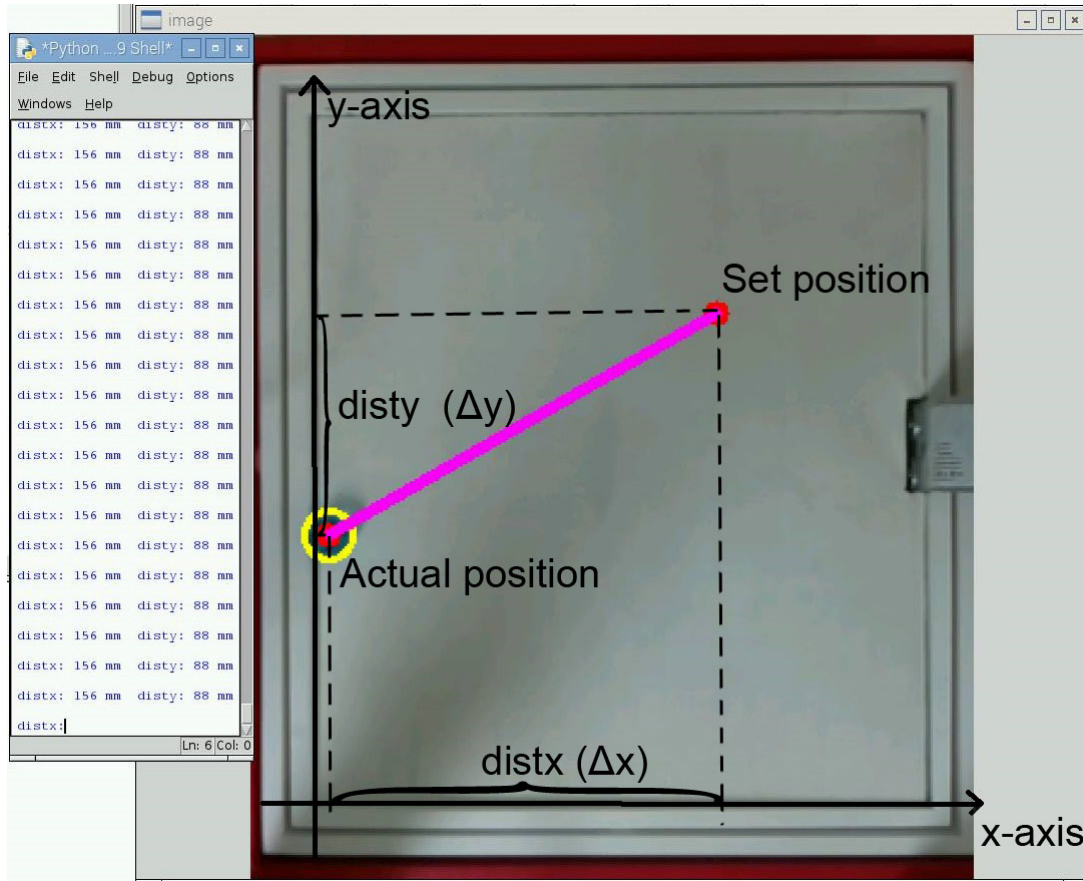


Figure 7.1: How the distances are calculated (Authors image)

7.2.1 Proportional controlling; method 1

The first controlling algorithm that was tested was a simple proportional controller that takes the error calculated ($e(k)$) and multiplies that by a proportional coefficient K and puts it out to the servomotors. This way, if the error is large, the servomotors will be given a large output from the controller and vice versa. The first value that was tried for the controller was calculated by taking the maximum output to the servomotors and dividing that by the maximum length of the error. The reasoning behind this is that the maximum inclination of the plane will only happen if the ball is at the opposite end of the labyrinth. As mention in the hardware implementation chapter, the output value to the motors is a number between 153 and 512 with 307 approximately being the neutral position. This interval was cropped of to be a number between 157 and 457 to secure the labyrinth own mechanisms from the servo motors. The length of the labyrinth-platform is 240x220 mm. So putting these numbers together the control algorithms output to the servomotors looked

something like this:

$$\text{ServoMotor} - x = 307 + 150 * e(k)/220$$

$$\text{ServoMotor} - y = 307 + 150 * e(k)/240$$

This controlling-method was tested and recorded one axis at the time and the results can be seen in the figure 7.2. Note that the time axis in these plots are the time that it takes to run through the code, so in figure 7.2 for example the time axis shows the 250 first laps in the code. These laps are about 30-40 milliseconds each. It can be seen that the controlling the very fast and inaccurate. From this test it was concluded that a more complicated controlling method was required. Note that the time value for the x-axis on these plots are the time that it takes to loop through the code. These are unfortunately not consistent and fluctuates between 50 and 30 milliseconds a lap. The y-axis represents the position of the ball in the image.

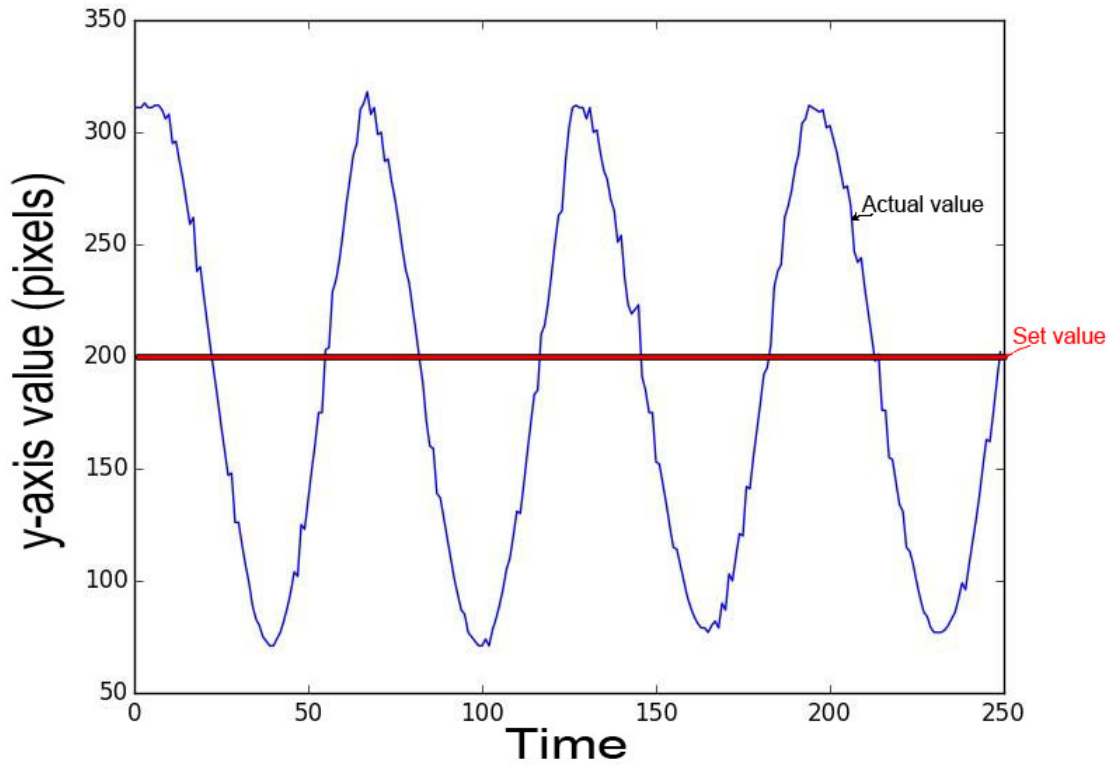


Figure 7.2: Plotting of the y-axis-actual-value change over time (Authors image)

7.2.2 Proportional controlling; method 2

Another code was created in this project when finding implementations for the gyroscope. In this code the user requests a specific inclination of the planes which then will be actualized with the help of the gyroscope guiding the servomotors to the correct position. This code was combined together with the proportional controller explained in the section above to make a code with two different controllers working together. The first controller works in a similar manner as before but it calculates an inclination between -6 and 6 degrees instead of a motor-output between 157 and

457. This then becomes the set-inclination-value for the second controller that uses the gyroscope to guide the servomotors to the correct position. How the second controller works by it self is explained in more detail in section 7.2.6.

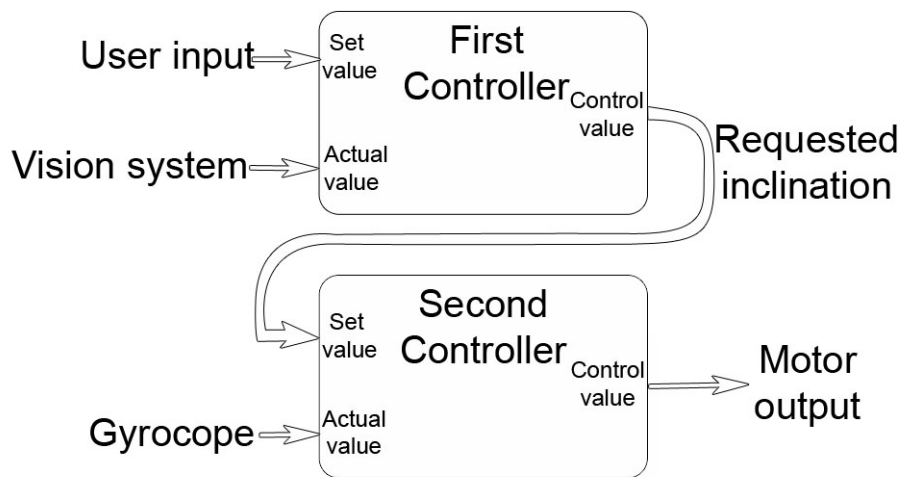


Figure 7.3: How the two controllers interact while using method 2 (Authors image)

The second method ramps up the value of the servomotors output while the first method always put the servomotors to an exact value in each loop of the code. The second method results in a much less "jerky" controlling of the servo motors. Since the second method gave a more favorable controlling behaviour, all the controllers that are discussed from here on out in this chapter will be using the second method by default.

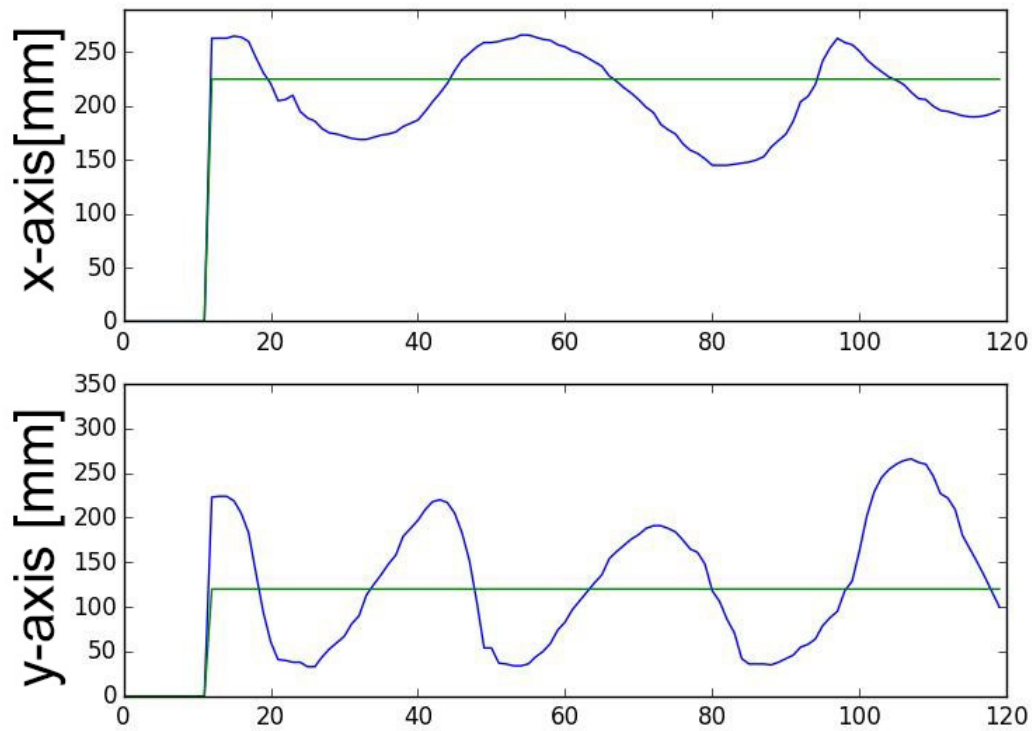


Figure 7.4: Plot of the P-controller using method 2. Green is the set value and blue is the actual value. (Authors image)

7.2.3 Adding Derivative controlling; PD-controller

To make the controller more stable a derivative term was added. the function for a PD-regulator looks like this:

$$u(k) = K[e(k) + T_D \frac{e(k) - e(k-1)}{h}] \quad (7.1)$$

Where $u(k)$ is the control value, K is the proportional term, $e(k)$ is the latest error value calculated, T_D is the derivative term, h is the timesample length and $e(k-1)$ is the previous error calculated. Below is how this controller worked. Take note that the system is more slow but stable and that there is a steady-state error at the end. For Roll the values where $K = 0.5$ and $T_D = 1.3$ and Pitch $K = 0.6$ and $T_D = 1.5$. These values where not calculated or taken from any scientific method but where simply tested many times over until a favorable result was reached.

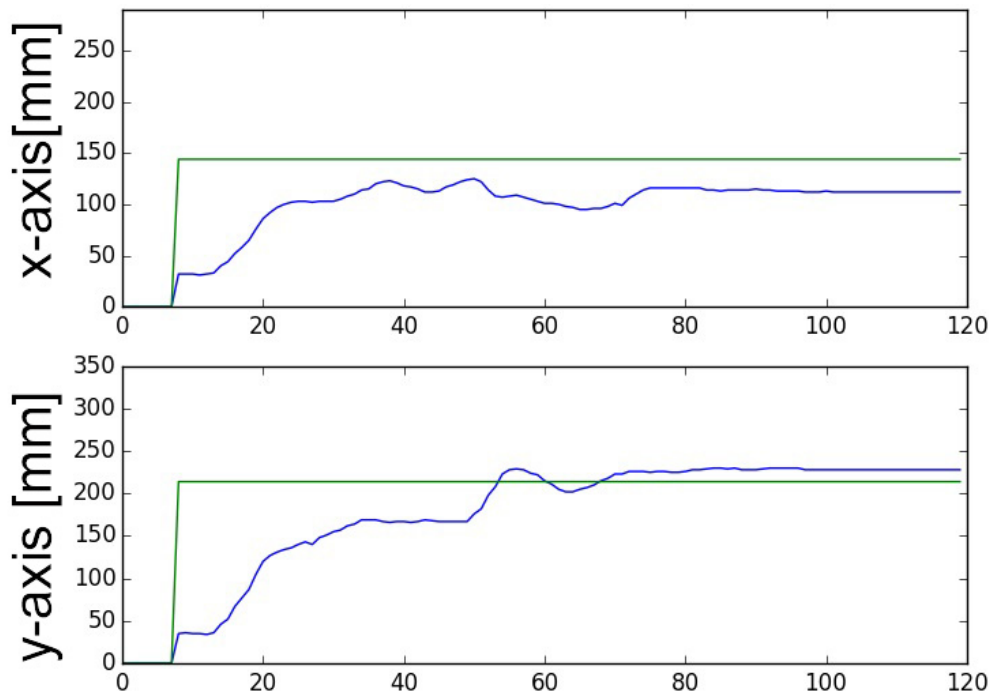


Figure 7.5: Plot of the PD-controller using method 2 (Authors image)

7.2.4 Adding Integral controlling; PI-Controller

An integral term in the algorithm will be proportional both to the magnitude of the error and the duration of the error. It is a part of the equation that will stack up over time if the error is consistent for a long time to eliminate any steady-state error. This is the way a PI-regulator function looks:

$$u(k) = K[e(k) + \frac{h}{T_I} \sum_{i=1}^k e(i)] \quad (7.2)$$

To get the sum of all the previous errors into the code, they were saved in a list and then added together. The list was capped to be a maximum of 50 values and the T_I was for both Roll and Pitch 1.4. The K value was unchanged from the PD-regulator. A PI-controller was tested and the results can be seen below. They are very similar to the results of a P-controller so they are not very efficient for detecting whether the integrated factor solved the steady-state error. The controlling is very irregular and unstable. The abrupt spikes at the tops of the curves in the plot are simply the ball hitting the edge of the game.

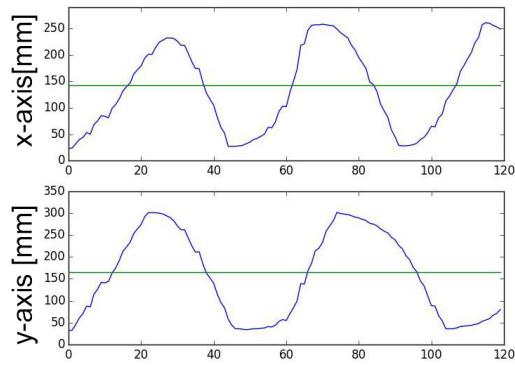


Figure 7.6: Plot of the PD-controller using method 2 (Authors image)

7.2.5 PID-controller

The function for a PID-controller looks like this:

$$u(k) = K[e(k) + T_D \frac{e(k) - e(k-1)}{h} + \frac{h}{T_I} \sum_{i=1}^k e(i)] \quad (7.3)$$

This is the controller where the Integrative term could be put to the test. It is also the controller where the most tests were done. The list length of errors for the integrative part was changed to 300 and T_I was enlarged to 7. This way the Integrative part of the equation would not have much significance at first but add up over time to counteract steady-state error. The rest of the values for Roll the values where $K = 0.5$ and $T_D = 1.3$ and for Pitch $K = 0.6$ and $T_D = 1.3$. The time-axis was also extended since the controller had a very long settling time.

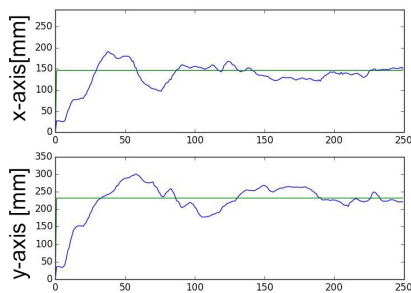


Figure 7.7: Plot of the PID-controller using method 2 (Authors image)

7.2.6 The Gyroscope-Servomotor Controller

The controller that communicates between the gyroscope and the servomotors is a proportional controller where the actual value is given from the gyroscope and the servomotor gets the control value. The controller was first created in a separate program where the user can set an inclination for both pitch and yaw between -6 and 6 degrees. The interval between -6 and 6 degrees is the maximum and minimum inclination value when running the servomotors between their maximum and minimum position. The plot of the set and actual value in this code can be seen in figure 7.8.

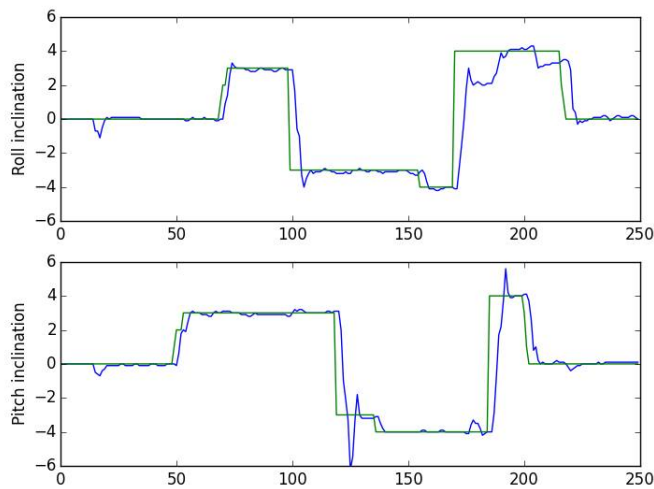


Figure 7.8: Plot of the P-controller, green is set value and blue is actual value (Authors image)

This controller is, unlike the other controllers that was created in this project, very fast and accurate. It is though unfair to compare the controllers since the the balls position is very hard to make accurate while the servomotors are relatively simple to get to a set position. In figure 7.9 is the plot of the second P-controller and how it works with the PID-controller.

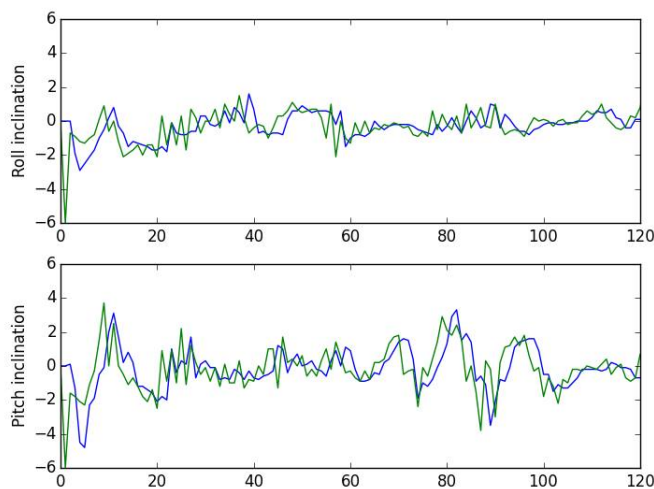


Figure 7.9: Plot of how the 2nd controller works when the 1st controller is using PID-controlling. (Authors image)

8

Results

8.1 Research results

At the start of the project there were some questions put down to do some research on. Here are the results and conclusions that were made during the procedure of this project.

8.1.1 Possibilities for controlling the system with the help of the RPI and all its modules

The RPI is certainly a powerful little computer. Even when running the most computing heavy program that takes use of all the modules and the complicated algorithms that runs in the background when using numpy and OpenCv the RPI uses about 30 -40% of its processing power. There were however some issues with the speed at which the code ran sometimes. When the code needed to do multiple things like updating the frame, calculating ball position, setting the position on the motors, gathering the gyroscope values etc. in the same loop of the main program, there were definitely some issues with speed. The frame-rate for example needs to be consistent for the controlling to work properly. This can be solved by working with multi-threaded programming, so it is not necessarily the hardware's fault that the code runs too slow. Though the algorithms used for finding objects in the image were fast enough to be sufficient for this system.

8.1.2 The accuracy of the actuators that are needed

The actuators that were used in the project are servomotors. The choice was inspired by a project that was done on the website [instructables.com](https://www.instructables.com)[20]. The direct benefit with using these servomotors is that they have an internal logic that makes them simple to set a required position in a range of 180-degrees with. They are also quick and accurate enough for controlling the labyrinth game. The tact of the servomotors can be illustrated when running the program which sets a specific inclination for the plane. When running this program the entire platform can be tilted and the servomotors and the gyroscope will still keep the plane itself as a flat surface. The issue is sometimes though that the servomotors are too quick and react to the slight change in the code. This can be worked around by creating a more moderate controller.

8.1.3 OpenCV as a suitable software for vision programming

OpenCV is possibly one of the most vital parts of the entire project. It takes a little while to install and figure out how it works but once you get the hang of it it is very effective and straightforward. It quickly finds the ball with the help of the color-recognition tools when working in a room with adequate lighting. When running in a room with different lighting, the code did need some calibration though. Not only does OpenCV have the tools necessary for processing and finding objects in image but it packs a lot of useful functions on top of that. It can set up the camera-stream itself, gather user inputs with a bunch of different methods and showing the frames in a window with the help of a single line of code.

8.1.4 The implementation of a gyroscope in combination with a vision camera

The gyroscope does not help with the same part of the controlling process that the vision camera has. But it definitely can be used to control the inclination of the planes as well as record the inclinations over time during the controlling process. So it makes for a more complete system over all. The code that directly sets the inclination of the planes really shows how accurate and useful the gyroscope is. It can, as mentioned before, keep the plane flat even when the entire platform is being tilted.

9

Conclusion

9.1 Strengths

A functioning platform has been constructed that can control the ball with some precision.

At the start of the project we had two previous bachelor thesis's as our inspiration. One where a platform for controlling of the system and one for developing the control-algorithm to get the ball through the maze. These were used with quite outdated technology by today's standard and we wanted to see what was possible with more modern tools.

We managed to both build the platform and implement a control-algorithm in one thesis-project. We also managed to successfully implement two sensors instead of one, the vision sensor and the gyroscope. This opens up a lot of opportunities when it comes to testing and constructing a more precise controlling of the system since both the planes inclination and balls position can be recorded and calculated.

Even though the platforms construction is very basic this is something we consider a strength. The simplicity of the way it is constructed has made it a very cheap project to build as well as time saving without adventuring on the performance of the final product. This is a testament to how much technology has improved while the cost for it has been reduced.

9.2 Weaknesses

During the project quite few weaknesses has shown themselves but the ones that would be considered the biggest are:

- Servo motors limitations of accurately controlling the labyrinth game
- The issue that the code runs too slow and creates lag, creating an irregular controller and frame rate drops.
- The irregular surface of the platform
- The standard IDLE-program for running Python code on the RPI has no function for monitoring values for variables in the code, making troubleshooting difficult.

These are all issues that can be fixed but haven't because of time constraints.

9.3 Development opportunities

For the labyrinth game to be able to navigate itself through the maze there is a couple of things that needs to be done:

- The system needs to be tested in order for a more accurate control algorithm to be implemented
- The issue with frame rate drops needs to be fixed

There is also room for other kinds of development opportunities such as the implementation of yet another sensor.

Bibliography

- [1] A summary of the Raspberry Pi
<https://en.wikipedia.org/wiki/RaspberryPi>
(2016/04/27)
- [2] Introduction to the sensor hat
<https://www.raspberrypi.org/products/sense-hat/>
(2016/04/27)
- [3] Introduction to the servo hat
<https://www.adafruit.com/products/2327>
(2016/04/27)
- [4] Introduction to the picamera
<https://www.raspberrypi.org/products/camera-module/>
(2016/04/27)
- [5] Introduction to the servomotors
https://www.elfa.se/Web/Downloads/_t/ds/900-00005_eng_t.ds.pdf?mime=application2Fpdf
(2016/04/27)
- [6] Calculating the time for each bit <https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/library-reference>
(2016/05/18)
- [7] Configuring the servohat
<https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/attach-and-test-the-hat>
(2016/05/18)
- [8] Powering the servos <https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/powering-servos>
(2016/05/18)
- [9] Labyrinth game speccs
<http://www.brio.se/Products/Games/games/labyrinthgameboards>
(2016/04/27)
- [10] Introduction to OpenCV
http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_setup/py_intro/py_intro.html#intro
(2016/05/05)
- [11] RPI 3 components
<http://www.raspberrypi.org/products/raspberry-pi-3-model-b>
(2016/05/09)
- [12] Increasing Raspberry Pi FPS with Python and OpenCV

- http : //www.pyimagesearch.com/2015/12/28/increasing - raspberry - pi - fps - with - python - and - opencv/*
(2016/05/12)
- [13] Second tutorial used for openCV in python
https : //www.raspberrypi.org/products/raspberry - pi - 3 - model - b/
(2016/05/09)
- [14] First tutorial used for openCV in python
http : //www.pyimagesearch.com/2015/02/23/install - opencv - and - python - on - your - raspberry - pi - 2 - and - b/
(2016/05/09)
- [15] About the Mission to ISS with Raspberry and the SenseHat
https : //www.raspberrypi.org/blog/astro - pi/
(2016/05/18)
- [16] How to make the RPI recognize both the sense and the Servo- Hat.
https : //www.raspberrypi.org/forums/viewtopic.php?t = 123590&p = 830744
(2016 - 05 - 19)
- [17] Hough circle transform
https : //en.wikipedia.org/wiki/Circle_Hough_Transform
(2016 - 05 - 20)
- [18] Hough Transform *https : //en.wikipedia.org/wiki/Hough_transform*
(2016 - 05 - 20)
- [19] Bertil Thomas. Modern Reglerteknik. Forth edition. Stockholm: Liber AB; 2008.
- [20] Servo Controlled Labyrinth project *http : //www.instructables.com/id/Servo - Controlled - Labyrinth/*
(2016 - 05 - 26)
- [21] The HSV scale *https : //en.wikipedia.org/wiki/HSL_and_HSV*
(2016 - 06 - 01)

A

Appendix 1

The resulting code that can control the balls position on a flat surface. User puts in a destination for the ball and the code calculates a control value for the servomotors with the help of the vision-camera and the gyroscope.

```
from OrientStream import orientationstream
#Imports the gyroscope stream class
from imutils.video.pivideostream import PiVideoStream
#Imports the video stream code
import matplotlib.pyplot as plt
#Imports plot tools
from collections import deque
#Imports deque
import numpy as np
#Imports math tools
from picamera.array import PiRGBArray
#Imports the camera array
from Adafruit_PWM_Servo_Driver import PWM
#Imports the adafruit PWM class
from picamera import PiCamera
#Imports the PiCamera from the picamera library
from sense_hat import SenseHat
#Imports the SenseHat variable from the sense_hat library
import argparse
#Imports the argparse
import imutils
#Imports the pyimagesearch blog library
import time
#Imports time functions
import cv2
#Imports openCV
```

```
bor_pitch=0
```

```

bor_roll=0
minus_pitch = False
minus_roll = False
listlength = 600
#Sets the list length for the integrated value
plotlength = 120
#Sets the list length for the plot
pwm = PWM(0x40)
#Chooses the adress for the servohat communication
servo_channel_pitch = 4
#Chooses pitch to be on pin 4 on the servohat
servo_channel_roll = 5
#Chooses pitch to be on pin 5 on the servohat
pwm.setPWMFreq(50)
#Chooses the freq to 50Hz
o_s = orientationstream().start()
#Starts a stream for the gyroscope
time.sleep(0.1)
colorLower = (50,86, 6)
#Chooses the lower color limit (HSV scale)
colorUpper = (100, 255, 255)
#Chooses the upper color limit (HSV scale)
chosenpoint=None
chosenmetricpos = None
metricballpos = None
ballposition = None
ServoMin = 157 # Min pulse length out of 4096
ServoMid = 307
ServoMax = 457 # Max pulse length out of 4096
ServoSetpitch= ServoMid
#Starts the servos in center position
ServoSetroll= ServoMid
#Starts the servos in center position
#kamerainitiering
cv2.namedWindow('image', cv2.WINDOW_NORMAL)
#Opens up the window "image"
res = (300,350)
#Sets the resolution of the frame
framerate = 33
#Sets the fram rate
Pistream = PiVideoStream(res ,framerate)
#Videostream initiation
Pistream.camera.vflip = True
#Verticly flips the camera image
Pistream.camera.hflip = True
#Horisontally flips the camera image

```

```
vs = Pistream.start()
#Stats the pistream
time.sleep(1.0)
cx = -1
cy = -1
t = 0
y_axis = []
y_borvarde = []
listx = []
listy = []
list_ar_x = []
list_ar_y = []
list_bor_x = []
list_bor_y = []
list_roll = []
list_pitch = []
list_bor_roll = []
list_bor_pitch = []
for i in range(0,listlength):
    #Creates a "listlength" long list for the integrated effecd
    listx.append(0)
    listy.append(0)
for i in range(0,plotlength):
    #Creates a "plotlength" long list for the plotfunction
    list_ar_x.append(0)
    list_ar_y.append(0)
    list_bor_x.append(0)
    list_bor_y.append(0)
    list_roll.append(0)
    list_pitch.append(0)
    list_bor_pitch.append(0)
    list_bor_roll.append(0)
ar_pitch = 0
ar_roll = 0
old_distx = 0
old_disty = 0
e1 = 0
```



```
def event_pitch(x):#Event pitch is used for polarity
```

```

global bor_pitch , minus_pitch

if minus_pitch == True:
    bor_pitch = -1*x
else:
    bor_pitch = x

def event_pitch_minus(x):#Event pitch is used for polarity
global minus_pitch , bor_pitch
if x == 1:
    minus_pitch = True
    if bor_pitch > 0:
        bor_pitch = -1*bor_pitch
else:
    minus_pitch = False
    if bor_pitch < 0:
        bor_pitch = -1*bor_pitch

def event_roll(x):#Event roll is used for polarity
global bor_roll , minus_roll
if minus_roll == True:
    bor_roll = -1*x
else:
    bor_roll = x

def event_roll_minus(x):#Event roll is used for polarity
global minus_roll , bor_roll
if x == 1:
    minus_roll = True
    if bor_roll > 0:
        bor_roll = -1*bor_roll
else:
    minus_roll = False
    if bor_roll < 0:
        bor_roll = -1*bor_roll

def findball( mask, image):#Finds the biggest contour
                                #in the image and marks it
    global ballposition
    cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
                           cv2.CHAIN_APPROX_SIMPLE)[-2]
    center = ballposition

    # only proceed if at least one contour was found
    if len(cnts) > 0:
        # find the largest contour in the mask, then use

```



```
# it to compute the minimum enclosing circle and
# centroid
c = max(cnts , key=cv2.contourArea)
((x, y), radius) = cv2.minEnclosingCircle(c)
M = cv2.moments(c)
center = (int(M["m10"] / M["m00"]),
          int(M["m01"] / M["m00"]))
cx = int(M["m10"] / M["m00"])
cy = int(M["m01"] / M["m00"])

# only proceed if the radius meets a minimum size
if radius > 10:
    # draw the circle and centroid on the frame,
    # then update the list of tracked points
    cv2.circle(image, (int(x), int(y)), int(radius),
                (0, 255, 255), 2)
    cv2.circle(image, center, 5, (0, 0, 255), -1)
return center

def mousechoice(event,x,y,flags,param):
    #Grabs the user input from the image
    global chosenpoint, newpoint

    if event == cv2.EVENT_LBUTTONDOWNCLK:
        #Chooses a point in the image
        chosenpoint = (int (x),int (y))
        for i in range(0,listlength):
            listx[i] = 0
            listy[i] = 0
    if event == cv2.EVENT_RBUTTONDOWNCLK and chosenpoint != None:
        #Nothing will happen if the right button is clicked

    plt.figure(1)
    #Plots the cordinate of the ball in the "x" direction
    plt.subplot(211)
    plt.plot(list_ar_x)
    plt.plot(list_bor_x)
    plt.axis([0, 120, 0, 290])
    plt.ylabel('Roll')

    plt.subplot(212)
    #Plots the cordinate of the ball in the "y" direction
    plt.plot(list_ar_y)
    plt.plot(list_bor_y)
    plt.axis([0, 120, 0, 350])
    plt.ylabel('Pitch')
```

```
plt.savefig("arvardepos.png")

plt.figure(2)
#Plots the value of the inclination of the roll
plt.subplot(211)
plt.plot(list_roll)
plt.plot(list_bor_roll)
plt.axis([0,120,-6,6])
plt.ylabel('Roll_inclination')

plt.subplot(212)
#Plots the value of the inclination of the pitch
plt.plot(list_pitch)
plt.plot(list_bor_pitch)
plt.axis([0,120,-6,6])
plt.ylabel('Pitch_inclination')

plt.savefig("arvardevinkel.png")
plt.show()
cv2.setMouseCallback('image', mousechoice)

def pixeltometric():
    #Converts from pixels to metric [mm]
    global chosenmetricpos, metricballpos, chosenpoint, ballposition
    if chosenpoint != None:
        chosenmetricpos = (chosenpoint[0] * 290 / res[0],
                           chosenpoint[1] * 335 / res[1])

    if ballposition != None:
        metricballpos = (ballposition[0] * 290 / res[0],
                        ballposition[1] * 335 / res[1])

def getdistx(bor, ar):
    #Gets the e[k] value in the x-axis
    return (bor[0] - ar[0])

def getdisty(bor, ar):
    #Gets the e[k] value in the y-axis
    return -(bor[1] - ar[1])

def sumdistx(distx):
    #Summarize the list of old e[k] (x-axis)
    #values for the integrated effect
    global listx
    listx.pop(0)
    listx.append(distx)
    return sum(listx)
```

```
def sumdisty(disty):  
    #Summarize the list of old e[k] (y-axis)  
    #values for the integrated effect  
    global listy  
    listy.pop(0)  
    listy.append(disty)  
    return sum(listy)  
  
def get_ux(distx, old_distx, h, sumx):  
    styr = 0.5*(distx + 1.3*((distx - old_distx)/h)+((h/10)*sumx))  
    #The controller which calculates the control value  
    if styr > 6:  
        return 6  
    if styr < -6:  
        return -6  
    else:  
        return styr  
  
def get_uy(disty, old_disty, h, sumy):  
    styr = 0.6*(disty + 1.3*(disty - old_disty)/h+((h/10)*sumy))  
    #The controller which calculates the control value  
    if styr > 6:  
        return 6  
    if styr < -6:  
        return -6  
    else:  
        return styr  
  
def control_function_pitch( distp):  
    #Ramps up the control value of pitch  
    global ar_pitch, bor_pitch, Servosetpitch  
    if ar_pitch <= bor_pitch+0.1 and ar_pitch >= bor_pitch-0.1:  
        Servosetpitch=Servosetpitch  
    elif ar_pitch > bor_pitch :  
  
        Servosetpitch += int(15*distp)  
        if Servosetpitch >= ServoMax:  
            Servosetpitch = ServoMax  
    elif ar_pitch < bor_pitch :  
  
        Servosetpitch -= int(15*distp)  
  
        if Servosetpitch<=ServoMin:  
            Servosetpitch = ServoMin
```

```

def control_function_roll(distr):
    #Ramps up the control value of roll
    global ar_roll, bor_roll, Servosetroll
    if ar_roll <= bor_roll+0.1 and ar_roll >= bor_roll-0.1:
        Servosetroll=Servosetroll
    elif ar_roll > bor_roll :

        Servosetroll += int(15*distr)

        if Servosetroll >= ServoMax + 50:
            Servosetroll = ServoMax + 50
    elif ar_roll < bor_roll :

        Servosetroll -= int(15*distr)

        if Servosetroll<=ServoMin - 70:
            Servosetroll = ServoMin - 70


while(1):
    #Start of mainloop

    image = vs.read()
    #Read an image
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    #Converts the image to a modified HSV scale
    mask = cv2.inRange(hsv, colorLower, colorUpper)
    #Masks out the colors within the lower and upper limit
    mask = cv2.erode(mask, None, iterations=2)
    #Errodes the first mask
    mask = cv2.dilate(mask, None, iterations=2)
    #Dilates the second mask
    ballposition = findball(mask, image)
    #Finds the ball in the mask and then shows it in image
    pixeltometric()
    #Converts from pixel to metric [mm]

```

```

if chosenpoint != None:
    #Doesnt start the controlling until a point is chosen
    cv2.circle(image,chosenpoint,20,(255, 255, 255),2)
    #Creates a white circle around the chosen point
    cv2.line(image,chosenpoint, ballposition, (255, 255, 255), 3)
    #Creates a white line between the chosen point and the ball
    distx = float((getdistx(chosenmetricpos,metricballpos)
                    * 0.027273))
    #Calculates the distance for the x-axis
    sumx = sumdistx(distx)
    #Summarises the list of x-axis distances
    disty = float((getdisty(chosenmetricpos,metricballpos)
                    * 0.025))
    #Calculates the distance for the y-axis
    sumy = sumdisty(disty)
    #Summarises the list of x-axis distances
    e2 = cv2.getTickCount()
    #Starts the first clock
    h = (e2 - e1)/cv2.getTickFrequency()
    #Calculates the sample value
    bor_roll = - round(get_ux(distx,old_distx, h, sumx),1)
    #Rounds up the controll/Set value (x-axis) to 1 decimal
    bor_pitch = round(get_uy(disty, old_disty, h, sumy),1)
    #Rounds up the controll/set value (y-axis) to 1 decimal
    e1 = cv2.getTickCount()
    #Starts the second clock

    old_distx = distx
    #Collect the old distance value(x-axis)
    old_disty = disty
    #Collect the old distance value(y-axis)

    list_ar_x.pop(0)
    #Deletes actual x value in the first position of the list
    list_ar_y.pop(0)
    #Deletes actual y value in the first position of the list
    list_bor_x.pop(0)
    #Deletes set x value in the first position of the list
    list_bor_y.pop(0)
    #Deletes set y value in the first position of the list
    list_ar_x.append(metricballpos[0])
    #Inserts a new actual x value in the first position of the list
    list_ar_y.append(metricballpos[1])
    #Inserts a new actual y value in the first position of the list
    list_bor_x.append(chosenmetricpos[0])

```

```

    #Inserts a new set x value in the first position of the list
    list_bor_y.append(chosenmetricpos[1])
    #Inserts a new set y value in the first position of the list

ar_pitch = o_s.read_pitch()
#Collects the pitch value from the gyroscope
ar_roll = o_s.read_roll()
#Collects the roll value from the gyroscope

if ar_pitch > 350:
    ar_pitch = ar_pitch - 360
    #Converts the pitch actual value to be a number
    #between 0 -> -6 degrees
    if ar_roll > 350:
        ar_roll = ar_roll - 360
        #Converts the roll actual value to be a number
        #between 0 -> -6 degrees

list_bor_roll.pop(0)
#Deletes set roll value in the first position of the list
list_bor_pitch.pop(0)
#Deletes set pitch value in the first position of the list
list_roll.pop(0)
#Deletes actual roll value in the first position of the list
list_pitch.pop(0)
#Deletes actual pitch value in the first position of the list
list_roll.append(ar_roll)
#Inserts a new actual roll value in the first position of the list
list_pitch.append(ar_pitch)
#Inserts a new actual pitch value in the first position of the list
list_bor_roll.append(bor_roll)
#Inserts a new set roll value in the first position of the list
list_bor_pitch.append(bor_pitch)
#Inserts a new set pitch value in the first position of the list

distp = abs(ar_pitch - bor_pitch)
#Gets the distance between the choosen point and the ball
distr = abs(ar_roll - bor_roll)
#Gets the distance between the choosen point and the ball

control_function_pitch(distp)
#Ramps up the controll value for pitch
control_function_roll(distr)
#Ramps up the controll value for roll

```

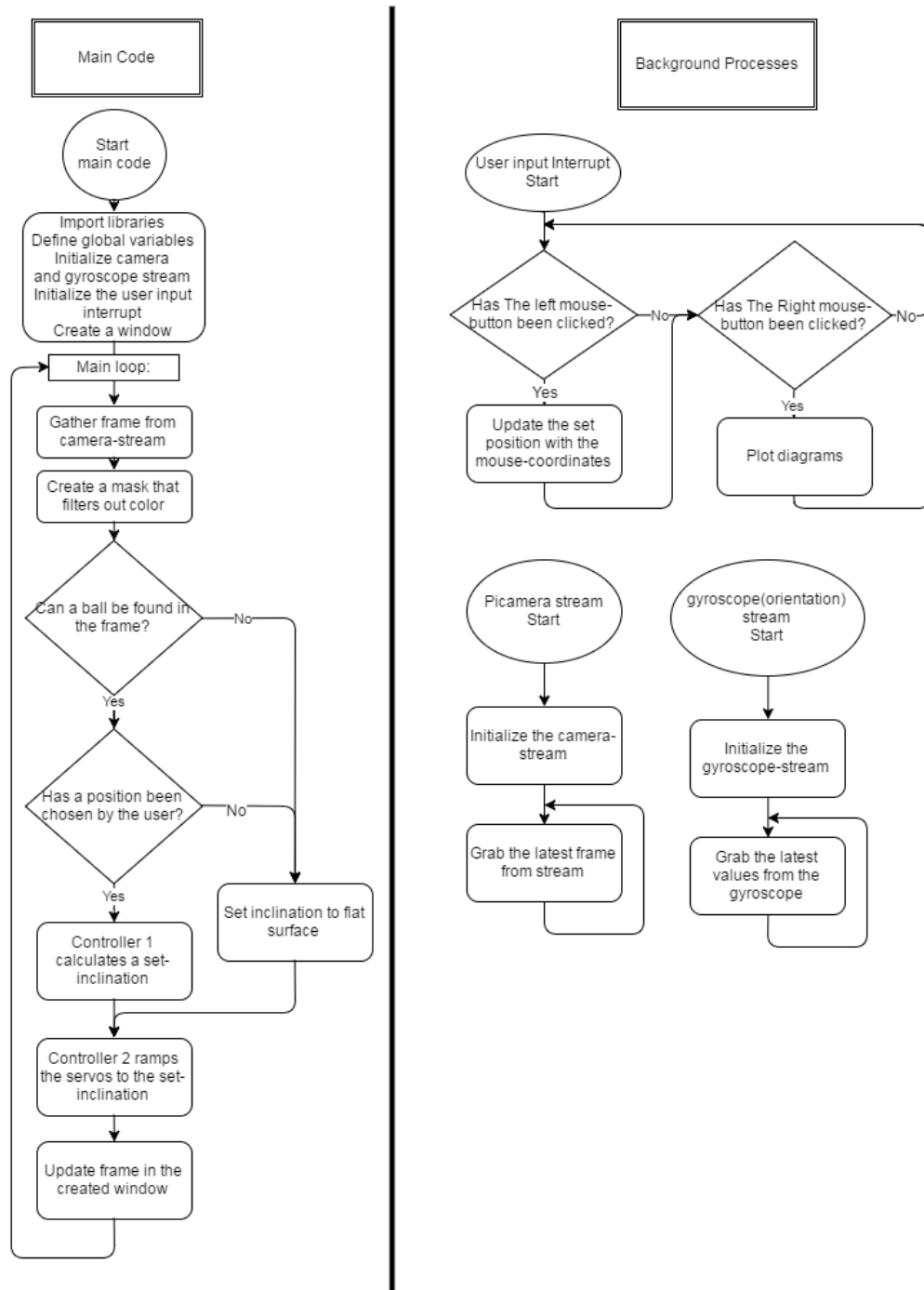
```
pwm.setPWM(servo_channel_pitch, 0, Servosetpitch)
#Sets the pitch position for the servo motors
pwm.setPWM(servo_channel_roll, 0, Servosetroll)
#Sets the roll position for the servo motors

cv2.imshow('image', image)
#Shows the image
k = cv2.waitKey(1) & 0xFF
#If "ctrl C" is pushed down the program is ended
if k == 27:
    break
cv2.destroyAllWindows()
#Shuts down all windows
vs.stop()
#Stops video stream
o_s.stop()
#Stops orient stream
```


B

Appendix 2

A flowchart that explains the logic behind the code in appendix A.



C

Appendix 3

The function in the code that finds the balls position and maps it out on the picture.

```
def findball( mask, image):#Finds the biggest contour
                                #in the image and marks it
    global ballposition
    cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)[-2]
    center = ballposition

    # only proceed if at least one contour was found
    if len(cnts) > 0:
        # find the largest contour in the mask, then use
        # it to compute the minimum enclosing circle and
        # centroid
        c = max(cnts, key=cv2.contourArea)
        ((x, y), radius) = cv2.minEnclosingCircle(c)
        M = cv2.moments(c)
        center = (int(M["m10"] / M["m00"]),
            int(M["m01"] / M["m00"]))
        cx = int(M["m10"] / M["m00"])
        cy = int(M["m01"] / M["m00"])

        # only proceed if the radius meets a minimum size
        if radius > 10:
            # draw the circle and centroid on the frame,
            # then update the list of tracked points
            cv2.circle(image, (int(x), int(y)), int(radius),
                (0, 255, 255), 2)
            cv2.circle(image, center, 5, (0, 0, 255), -1)
    return center
```


D

Appendix 4

The class that was created for running the gyroscope separately in the background of the main code with the help of threading.

```
from sense_hat import SenseHat
import time
import os
import cv2
from threading import Thread
import numpy as np

class orientationstream:
    #creates a class called orientationstream
    #initiziates the variables available when addressing the class
    def __init__(self):
        self.sense = SenseHat()
        self.sense.clear()
        self.stopped = False
        self.pitch = 0
        self.roll = 0
        self.o = None

    def start(self):
        #Starts a thread that jumps to the update-function

        Thread(target=self.update, args=()).start()
        return self

    def update(self):
        #Updates the gyroscope-values
        while(self.stopped == False):
            self.o= self.sense.get_orientation()
            self.pitch = round(self.o["pitch"],1)
            self.roll = round(self.o["roll"],1)

    def read_pitch(self):
        #returns the pitch-value when called
        return self.pitch
```

```
def read_roll(self):  
    #returns the roll-value when called  
    return self.roll  
  
def stop(self):  
    #stops the gyroscope-update process when called  
    self.stopped = True
```