



Norwegian University of
Science and Technology

Mapping a maze with a camera using a Raspberry Pi

Kristian Bjørnsen

Master of Science in Cybernetics and Robotics

Submission date: June 2017

Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

This master thesis features the implementation of a complete image processing, mapping and communication system on a credit card-sized computer, the Raspberry Pi, for the characterization and mapping of a maze from the air using a drone. The system was implemented in Python and has been verified in real-life testing scenarios.

The system captures an image, detects the wall segments of the maze in the image and through the integration of sensors the system calculates the real life lengths and position of the walls. The system is designed to be run remotely, and incorporates a communication platform for the exchange of mapping data.

Image processing and mapping

By translating an existing implementation of image processing and mapping algorithm from MATLAB and further developed in Python, and through the use of a new image processing library, OpenCV, the system is able to detect and extract information about wall position and length.

Sensor implementation

A camera and a range sensor has been integrated in the hardware and software of the system and provides sensor data to the system for the image processing and mapping algorithm. The camera has been determined to give excellent sensor data, while the sensor data from the range sensor can be improved to obtain better accuracy.

Communication

A communication system, together with a defined communication protocol has been implemented using SSH. Libraries that facilitate for a smooth, fast and secure connection between the system and a host computer has been installed and verified.

Sammendrag

Denne masteroppgaven beskriver implementasjonen av et komplett bildebehandlings-, kartlegging- og kommunikasjonssystem på en Raspberry Pi. Raspberry Pi er en data-maskin på størrelse med et kredittkort. Systemet er ment for å kartlegge en labyrint fra luften ved å monteres på en drone. Systemet var utviklet i Python og har blitt verifisert gjennom tester på alle komponentene.

Systemet tar et bilde, detekterer veggsegmenter i labyrinten og gjennom integrasjon av sensorer kan systemet kalkulere de faktiske verdiene i labyrinten i form av lengde og posisjon. Systemet er designet for å bli kjørt eksternt og blir kontrollert gjennom en kommunikasjonskanal der den utveksler kartleggingsdata.

Bildebehandling og kartlegging

En tidligere implementasjon av bildebehandling og kartlegging har blitt oversatt fra MATLAB og utviklet videre i Python. Gjennom bruken av et nytt bildebehandlingsbibliotek, OpenCV, klarer systemet å detektere og ekstrahere informasjon om veggene i labyrinten.

Sensorimplementasjon

Et kamera og en avstandsmåler som innhenter informasjon til databehandlings- og kartleggingsalgoritmen har blitt integrert i Raspberry Pi maskinvaren. Kameraet har blitt testet og gir ut gode sensordata i form av bilder. Avstandsmåleren har også blitt testet, og kan bli utbedret for å øke nøyaktigheten til målingene.

Kommunikasjon

Et kommunikasjonssystem med en definert kommunikasjonsprotokoll har blitt implementert i systemet basert på SSH. Biblioteker som åpner for bruken av en sikker og rask SSH forbindelse har blitt installert og verifisert.

Problem description

We wish to implement the image processing method developed in the project thesis Bjørnsen 2016 [19], which is an algorithm for the detection and mapping of a maze captured by a camera from the air. It should be implemented on a mobile single board computer with the necessary processing power for computer vision applications. A camera system will also need to be integrated with the single board computer. The goal is to develop a computer that can be mounted on an unmanned aerial vehicle such as a quadcopter. There are several pieces of information the image processing method requires from the drone, and needs to be discussed.

Moments in the thesis should include:

- Determine a single board computer suitable for our application
- Determine a camera that is compatible with this computer and the camera characteristics needed for the image processing method
- Implement the image processing method on the single board computer
- Implement communication from the single board computer to a PC
- Determine how information such as attitude, height and position can be extracted from the drone, and what the difficulties are with respect to the accuracy of this information

Acknowledgements

I want to thank my supervisor throughout my project report and master thesis; Tor Onshus. This project started for the first time fall of 2016. Thank you to Mikal Nielsen for helping me find the documentation needed for the image processing part of the project. Thank you to all the other masters students under Tors guidance.



Kristian Bjørnsen

Contents

List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Project Vision	1
1.2 Project Overview	2
1.3 Report Structure	3
2 Theory	4
2.1 Ground Sample Distance - GSD	4
2.2 Image Processing	7
2.2.1 Edge Detection	7
2.2.2 Edge Detection methods based on first derivatives	7
2.2.3 Edge Detection methods based on the second derivative	10
2.2.4 Canny edge detection method	10
2.3 Edge Linking	12
2.3.1 Hough Transform	12
2.4 Mapping	13
2.4.1 Determining the length of the walls	15
2.4.2 Determining the position of the walls	15
2.5 Ultrasonic range sensor	17
2.6 Image Sensor	18
2.7 Technical hardware requirements	21
2.8 Positioning of the system	22
2.8.1 Pure IMU based positioning	22
2.8.2 GPS positioning	23
2.8.3 Camera assisted indoor positioning	23
3 Hardware specification	24
3.1 Raspberry Pi 3 Model B	24

3.2	Raspberry Pi Camera Module v2	25
3.3	HC - SR04 Ultrasonic Ranging Module	26
3.4	Raspberry Pi Universal Power Supply	27
4	Hardware Setup	28
4.1	System Architecture	28
4.2	System Setup	28
4.3	Raspberry Pi	30
4.4	Camera	31
4.5	Range sensor	32
4.6	3D printed case	34
5	Software Setup	35
5.1	Operating System, Raspbian Jessie	35
5.1.1	Installation	36
5.2	Python 2.7	36
5.2.1	Installation	37
5.2.2	Dependencies	37
5.2.3	Virtual Environment	38
5.3	OpenCV 3	39
5.3.1	Installation and Compilation	40
5.4	Remote Desktop and Connection	42
5.4.1	Determining IP address	42
5.4.2	Remote Desktop	43
6	Software Implementation	45
6.1	Camera and Image Capturing	45
6.2	Image Processing	47
6.2.1	Edge detection	48
6.2.2	Hough Transform	48
6.3	Range sensor implementation	50
6.4	Mapping	51

6.5	Main function	53
6.6	Communication	54
6.6.1	Protocol	55
7	Testing	57
7.1	Range sensor	57
7.1.1	Setup	57
7.1.2	Measurements	58
7.2	Edge Detection	60
7.2.1	Setup	60
7.2.2	Results	62
7.3	Mapping	63
7.3.1	Setup	63
7.3.2	Accuracy of mapping with accurate height measurement	65
7.3.3	Accuracy of height measurement over the maze	67
7.4	Communication test	69
7.4.1	Ping Raspberry Pi	69
7.4.2	Connect through SSH (PuTTY)	70
7.4.3	Run the software on the Raspberry Pi	70
7.4.4	Output	71
8	Sources of Error	72
8.1	Inaccurate height measurement	72
8.1.1	Range sensor gives wrong values	72
8.1.2	Difference in height from camera to height sensor	72
8.2	Incomplete mapping of wall segments	73
8.3	Detection of the same wall segment more than once	75
8.4	Communication failure	76
9	Discussion	77
9.1	Findings already established	77
9.2	Hardware Improvements	77

9.3	Software Improvements	78
9.4	Accuracy	79
9.4.1	Length of Walls	79
9.4.2	Height measurement	80
9.4.3	Hough Transform	81
10	Conclusion	82
11	Future Work	83
11.1	Bluetooth Communication	83
11.1.1	Technical Issue	83
11.2	Mobile Power Supply	83
11.3	Develop a Drone	84
11.4	Implementation of Positioning	84
12	References	85
Appendix A		88
A.1	Figures	88
A.2	Test Images	88
A.3	Software	88
A.4	Previous Reports	88
Appendix B		89
B.1	Hardware included	89

List of Figures

1	Project Vision	1
2	Ground Sample Distance. From [2]	5
3	(1) Horizontal intensity profile. (2) First derivative. (3) Second derivative with zero crossing	8
4	Mapping setup	14
5	Ultrasonic range sensor. From [14]	17
6	Top left: Continuous image Top right: A scan from A to B in the continuous image Bottom left: Sampling and quantization Bottom right: Digital scan line [1]	19
7	APS CMOS Image Sensor	20
8	Raspberry Pi 3 Model B	25
9	Raspberry Pi Camera Module v2	26
10	HC - SR04	26
11	Raspberry Pi Universal Power Supply	27
12	System architecture	28
13	Normal hardware setup	29
14	Debug hardware setup	29
15	Raspberry Pi 3 Model B. [20]	30
16	GPIO Pins on the Raspberry Pi 3 [16]	30
17	Raspberry Pi Camera Module V2 with CSI cable	31
18	CSI Cable mount	31
19	Voltage divider	32
20	Range sensor setup	33
21	HC-SR04 pins. Vcc, Trig, Echo, GND	33
22	3D printed case	34
23	Raspberry Pi IP	42
24	Windows Remote Desktop Connection	43
25	Raspberry Pi Login	44
26	PuTTY in Windows	54
27	PuTTY terminal in Windows	55

28	Range Sensor Setup	58
29	Test maze	60
30	Test Image	61
31	MATLAB Image Processing Toolbox Canny Implementation	62
32	Python OpenCV Canny Implementation	62
33	Tripod setup used for testing	64
34	Bulls eye spirit level on the tripod	64
35	Output from algorithm	65
36	Height tests images with the maze at different locations	67
37	Raspberry Pi IP	69
38	PuTTY terminal in Windows	70
39	Output from the mapping algorithm	71
40	Difference in height between range sensor and image sensor	73
41	Hough lines drawn over original image	74
43	Two wall segments detected	75

List of Tables

1	General 3x3 Spatial Filter Mask	10
2	Username and Password	43
3	Update message protocol	55
4	Range sensor measurements	58
5	Username and Password	70
6	Hardware differences between the two implementations	78

Abbreviations

APS CMOS Active pixel sensor Complementary metal–oxide–semiconductor

IMU Inertial measurement unit

GSD Ground sample distance

BLE Bluetooth Low Energy

UAV Unmanned aerial vehicle (Drone, quadcopter)

GPS Global positioning system

RAM Random access memory

I/O Input/Output

OS Operating system

LIDAR Light detection and ranging

CPU Central processing unit

GPU Graphics processing unit

1 Introduction

1.1 Project Vision

Exploration and mapping has been an important part in the field of robotics for many years. In order to create fully autonomous systems, one often needs to be able to identify the surrounding environment for control purposes. In mapping, camera sensors are often used to collect a great amount of environmental variable data, and only the relevant pieces of information should be extracted.

The vision of this project is to contribute towards developing a detection and mapping platform meant to be mounted on an unmanned aerial vehicle (UAV) capable of mapping a maze from the air. Is it possible to develop such a system using only cheap off-the shelf hardware and open source software?



Figure 1: Project Vision

1.2 Project Overview

There has been an ongoing effort since 2004 to create a system of LEGO-robots capable of the exploration and mapping of a maze. Using only cheap off-the shelf hardware and through the design of algorithms for exploration, mapping and communication the LEGO-robots have become proficient in mapping and exploring. However, one biggest issues in the current implementation of the LEGO-robots is the accuracy of the mapping.

In an attempt to improve the overall accuracy of the maze mapping, it has been proposed that the use of a camera sensor capable of detecting and mapping the maze can compliment the existing implementation and improve its accuracy. We envision a different mapping system from the LEGO-robots, where an image sensor together with an image processing capable computer is mounted on an unmanned aerial vehicle (UAV, drone).

Some work has been done in the image processing part of this new mapping system in the project report Bjørnsen fall 2016 [19], where an algorithm for detecting and mapping the maze was developed in MATLAB. To further develop this system, this thesis seeks to implement the mapping algorithm on a hardware solution suitable for my application.

1.3 Report Structure

Here is a brief description of the contents of each chapter:

- **Chapter 1 - Introduction**
- **Chapter 2 - Theory.** Covers the theoretical basis needed in image processing, mapping and the theory behind the sensors used in the system. This seeks to lay the foundation for the application.
- **Chapter 3 - Hardware Specification.** Covers the hardware used in the thesis, and the specification of each piece of hardware.
- **Chapter 4 - Hardware Setup.** Covers the setup of the hardware used in the system.
- **Chapter 5 - Software Setup.** This chapter covers the software setup of the Raspberry Pi and the installation and compilation of OpenCV and other software dependencies for the application.
- **Chapter 6 - Software Implementation.** This chapter covers the implementation of the mapping algorithm in Python. This is the translation of the algorithm developed in the project report [19].
- **Chapter 7 - Testing.** This chapter documents how the system works in varying tests in terms of accuracy and project goal.
- **Chapter 8 - Sources of error.** This chapter explores the potential sources of error in the implementation of the system.
- **Chapter 9 - Discussion.** Discussion of the implementation, its accuracy and how it compares to previous solutions.
- **Chapter 10 - Conclusion.** This section concludes the thesis and attempts to draw a conclusion if any can be made from the data presented.
- **Chapter 11 - Further work.** Explores the potential further work to make the project vision a reality.

2 Theory

In order for my single-board computer to be able to extract useful information from the digital images it receives, it has to use fundamentals from image processing and mapping theory together to create a complete mapping system. There are also some requirements in terms of the processing power in the single-board computer, and to be able to run the mapping algorithm in a timely manner, these needs to be met.

The following section presents the theoretic basis for image processing and mapping, some sensor theory, as well as the requirements on the single-board computer. The theory section will share many similarities with the project report [19] I wrote fall of 2016 for Tor Onshus. The same fundamentals are being used in this thesis, but in terms of implementation; they vary. The theory section will focus on the theory for the techniques actually employed in the implementation, and less on presenting all the available techniques and selecting which ones to implement.

About half of the theory needed in this thesis is excerpt from Bjørnsen 2016 [19].

2.1 Ground Sample Distance - GSD

In order to associate image properties with real life properties of the mapping object, I can use what is known as Ground Sample Distance. When analyzing an image, this is a way to express and measure what one pixel in the image represents in real world units. The result will be a scaling factor given in $[m/pixel]$, where this factor can be used to scale pixel values to meter values.

Ground Sample Distance can be viewed as a measure of resolution limitations of an image sensor due to sampling [2]. With the assumption that the sensor is pointed normal to the ground, it is in effect a measure of the real world distance between two pixels on the ground. The angular distance between sensor samples is given by pixel pitch p divided by focal length of the sensor f . Assuming this angular distance is projected normal to the ground, it defines GSD:

$$GSD = \frac{p}{fW} R \quad [\text{meters/pixel}] \quad (1)$$

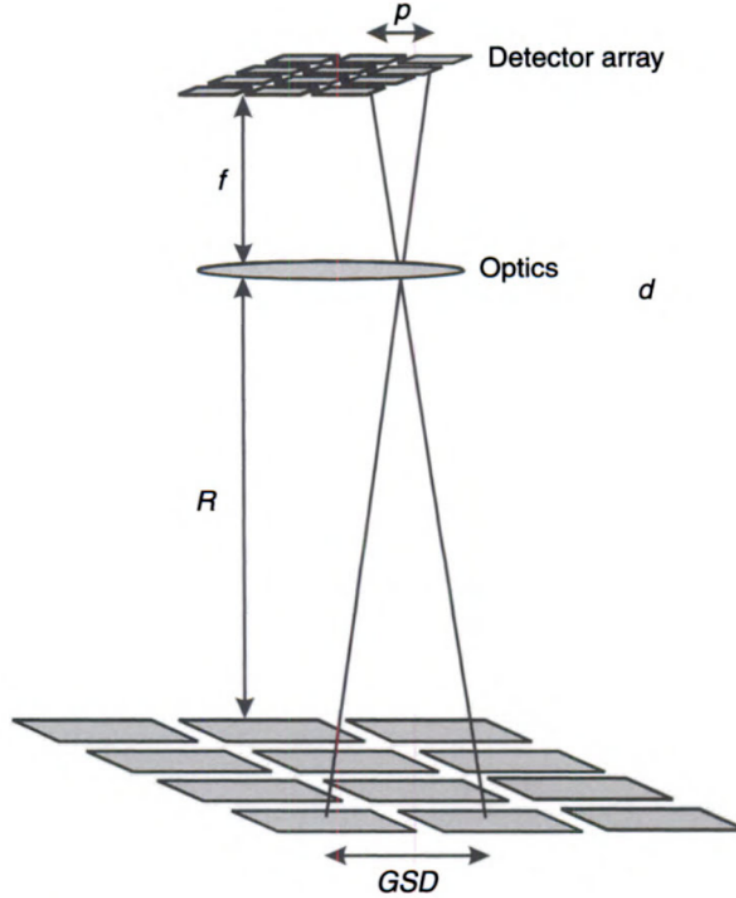


Figure 2: Ground Sample Distance. From [2]

Where R is the distance from the optics to the ground and W is the width of the image in pixels. GSD is illustrated in Figure 2. Equation 1 assumes that the sensor is directed normal to the ground surface. If the sensor is not aligned normal to the ground, the

GSD must be expanded for the look angle between the ground and the sensor θ :

$$GSD = \frac{pR}{fW \cos \theta} \quad [\text{meters/pixel}] \quad (2)$$

Where θ is the angle the sensor is facing with respect to the ground plane. By taking the geometric mean of the horizontal and vertical ground sample distances you get a two-dimensional system GSD [2] factor.

To calculate GSD it is assumed that certain camera properties are known and that we have a way to measure or calculate the distance from the object to which we are calculating the GSD. With these pieces of information, it is possible to convert an image from pixels to real life units.

2.2 Image Processing

The digital camera is a powerful sensor that has many provides many advantages in technical applications and implementations. It is a sensor that can be used for the identification of environmental variables within the line of sight, and can easily be combined with other sensors. Many autonomous applications employ cameras, and together with image processing it provides a powerful tool to automate advanced tasks and real life objectives. Self-driving cars [13] use cameras as well as other sensors to extract useful information from the environment.

As an example, in a self-driving car application, one might be interested in determining the size and location of other cars on the road in order to prevent collisions and provide an autonomous driving experience. Image processing is a topic that can be related to several fields of engineering, from simple automation to complex artificial intelligence.

In the following section I will present the fundamental underlying theory of edge detection and some of the image processing methods I use in my thesis.

2.2.1 Edge Detection

A powerful tool to use when trying to identify the characteristics of a the environment is edge detection. Edge detection is based on detecting sharp, local changes in intensity in an image. At a fundamental level, abrupt, local changes in intensity can be detected by using derivatives, where first- and second order derivatives are the most used [1].

Figure 3 illustrates the differences between the intensity response of the first- and second order derivatives on an edge with a ramp response.

2.2.2 Edge Detection methods based on first derivatives

As mentioned previously, edges are characterized as a change in intensity in an image. Digital images are discrete, therefore we have to use an approximation of the first derivative with the requirements that; (1) it must be zero in areas of constant intensity,

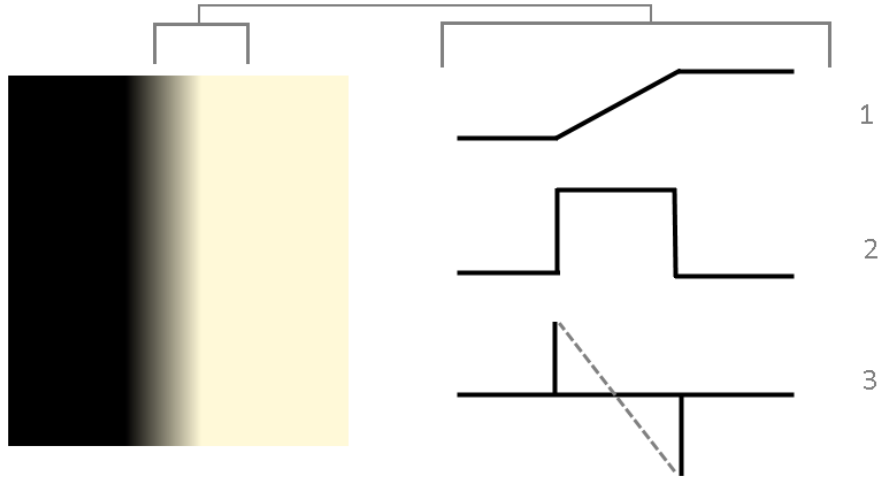


Figure 3: (1) Horizontal intensity profile. (2) First derivative. (3) Second derivative with zero crossing

(2) it must be nonzero at the onset of an intensity step or ramp, and that (3) it must be nonzero at points along an intensity ramp [1]. It is important to note that most edges in an image does not change its value immediately, but tends to change more gradually, in a ramp-function fashion. The requirements covers this, where the derivative must be non-zero along an intensity ramp.

I first consider the one-dimensional function $f(x)$, I approximate by Taylor expansion about x of $f(x + \Delta x)$, where I let $\Delta x = 1$, and keep the linear terms:

$$\frac{\partial f}{\partial x} = f'(x) = f(x + 1) - f(x) \quad (3)$$

I used the partial derivative here because the image is a function of two variables. I approximate the the derivative by Taylor expansion in the y dimension just like I did

above:

$$\frac{\partial f(x, y)}{\partial x} = f(x + 1, y) - f(x, y) = g_x \quad (4)$$

$$\frac{\partial f(x, y)}{\partial y} = f(x, y + 1) - f(x, y) = g_y \quad (5)$$

A powerful tool in edge detection is to define the gradient, ∇f as

$$\nabla f \equiv grad(f) \equiv \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} \quad (6)$$

$$M(x, y) = \sqrt{g_x^2 + g_y^2} \quad (7)$$

Where the ∇f vector gives us information about the edge strength as well as the direction of the greatest rate of change of f at location (x, y) .

The direction of the edge can be expressed as:

$$\alpha(x, y) = \arctan(g_y/g_x) \quad (8)$$

Obtaining the gradient of an image involves calculation the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at every location of the pixels in the image. In order to do this I use spatial filtering in the image (also known as masking). The process of spatial filtering consists of moving a filter mask from point to point in an image, and calculating the filter response of the original image at each point (x, y) in the image. The filter values are pre-defined and characteristics of the filter can be modified in order to achieve different effects in an image such as image enhancement.

I want to look for edges in two dimensions; x and y direction, thus the most used edge detection algorithms use 2D spatial filters to find edges. A general 3x3 spatial filter mask can be represented as a table of intensity values z_i as illustrated in Table 1.

Edge detection methods based on the first derivative are generally more primitive than those based on the second order derivative. They are generally less computationally intensive, but offer limited functionality and robustness. For my application I will

Table 1: General 3x3 Spatial Filter Mask

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

implement a method based on the second derivative.

2.2.3 Edge Detection methods based on the second derivative

The requirements set on the approximation of the second order derivative is similar to that set on the first order derivative. The only difference is that I now only require the second order derivative to be non-zero at the onset and end of a ramp in intensity value. This means that the first order derivative methods will create "thicker" edges since its non-zero along the whole ramp, while the second order derivative methods will lead to "thinner" edges, since the values are non-zero only at the beginning and the end of a ramp.

2.2.4 Canny edge detection method

A complex edge detection algorithm is the Canny edge detector [18]. In general, the Canny method is regarded as superior to most other edge detection algorithms, including other edge detector based on the second derivative, and is based on three objectives:

- Low error rate. The algorithm should find all the edges in an image, regardless of orientation. Should be as close as possible to the actual edges.
- Edge points should be well localized. Detected edges should be close to the actual edges.
- Single edge point response. Should not return multiple edge points where only a single edge point exist.

The Canny method starts with applying a Gaussian filter to convolve with the image in order to smooth out noise. After this the gradient magnitude (7) and the direction

(8) are calculated:

$$M(x, y) = \sqrt{g_x^2 + g_y^2}$$

$$\alpha(x, y) = \arctan(g_y/g_x)$$

After this it applies a non-maximum suppression technique to the gradient magnitude image which thins the edges. This helps suppress all the gradient values to zero except the local maximal, which is the the sharpest change in intensity value, thus fulfilling the objective of giving a single edge point response.

The last part of the algorithm is to use double thresholding to determine strong and weak edges. Edges with intensity values below the lowest threshold are suppressed, which is noise in most cases. This is a tuning parameter and needs to be determined empirically. The last step of the algorithm also involves tracking edges and conducting a connectivity analysis to detect and link edges together, where edges that are weak and not connected to strong edges are suppressed.

The Canny algorithm is the method employed later in the implementation, and is the fundamental edge detection method used in this paper.

2.3 Edge Linking

After the edge detection algorithm has been applied to the image, I have ideally detected all the edge pixels in the image perfectly. Most of the time, this is not the case. The edge detection is therefore followed up by edge linking, where I attempt to assemble edge pixels into meaningful edge segments. Since actual edges will have the property of being fully or partly connected straight lines, noise in the image will naturally be suppressed and not detected.

2.3.1 Hough Transform

The Hough transform is a global processing method of edge linking [5]. Considering a point (x_i, y_i) in the xy-plane, I can describe infinitely many lines running through this point as $y_i = ax_i + b$ for any a and b . I can rewrite this as $b = -ax_i + y_i$, and consider the ab-plane, which is known as the parameter space. This gives an equation for a single line for a single xy-pair (x_i, y_i) . A different xy-pair (x_j, y_j) also has a line expressed in parameter space $b = -ax_j + y_j$, and if the lines are not parallel, the lines will intersect at a point (a', b') . a' is the slope, and b' is the intercept of the line containing both (x_i, y_i) and (x_j, y_j) . The points on this line will have lines in the ab-plane that intersects at (a', b') .

One fundamental problem representing lines on the form $y = ax + b$ is that a approaches infinity as the line gets closer to vertical. It is therefore beneficial to represent the lines as:

$$x \cos \theta + y \sin \theta = \rho$$

This is a transformation to the $\theta\rho$ -plane, which is very similar to that of the ab-plane. (θ', ρ') corresponds to the line that passes through both (x_i, y_i) and (x_j, y_j) . When (θ', ρ') has a high concentration of curves passing through it, it indicates that these curves are a part of a connected line in the xy-plane. The Hough transform algorithm can be expressed as [1]:

1. Use edge detection on an image and obtain a binary image edge image

2. Transform image to the $\theta\rho$ -plane
3. Count and examine lines intersecting in the $\theta\rho$ -plane
4. Determine the lines based on number of intersects in (θ', ρ')

By using Hough transform the length of each wall segment can be extracted in pixel values. This is very important to make us able to map the maze in real-life units.

2.4 Mapping

This section is from Bjørnsen 2016 [19]

Techniques explained previously lays the ground work for how I am able to extract useful information from the image to be used for mapping. In addition to the information extracted by the image, I need some additional information for mapping:

- Height H_{tot} above ground the image was taken from as well as x-y position
- Assume we know the height of the walls H_w
- Information about the image sensor (pixel pitch p and focal length f)

I also put the following assumptions on the maze and image:

- The wall height H_w is constant on the entire maze
- The walls are straight (no curves)
- The walls are normal to the ground (no angle)
- The line of sight of the image covers the entire maze (one image sufficient)

Figure 4 displays the setup and different heights and planes associated with the mapping looking perpendicular to the ground. I can see that the wall plane height $H_p = H_{tot} - H_w$.

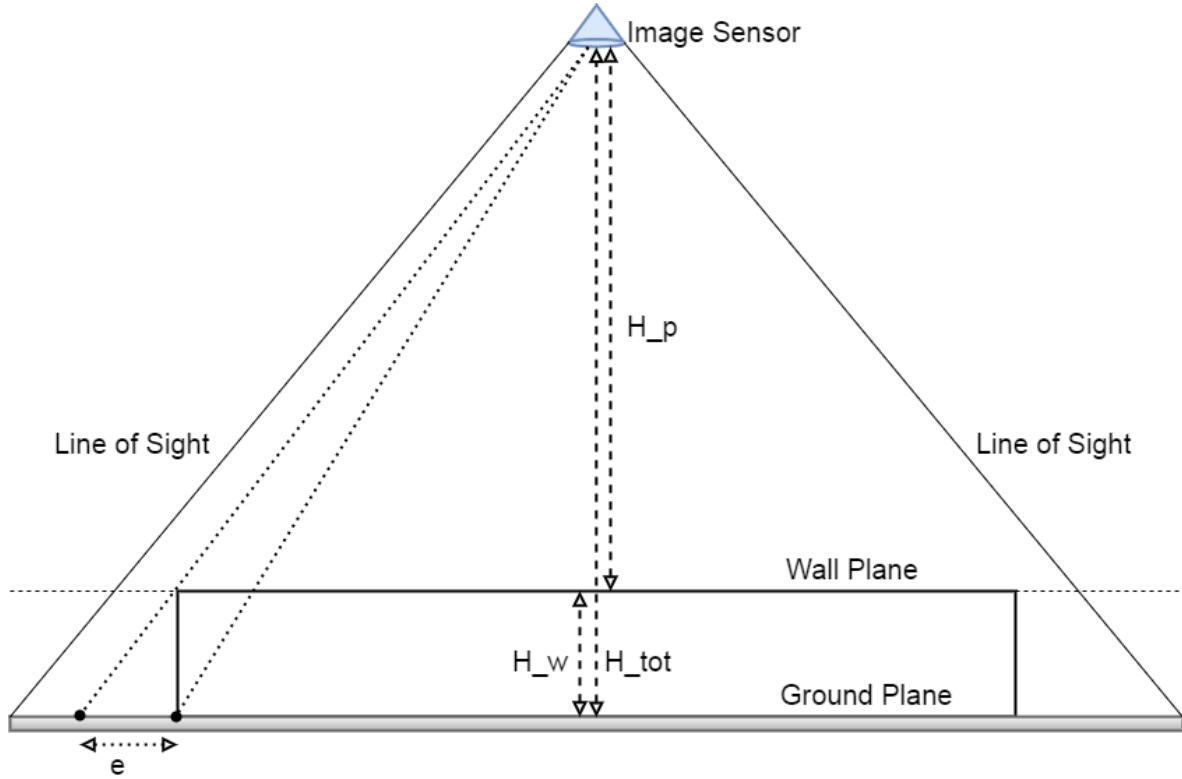


Figure 4: Mapping setup

One key insight in this setup is a result of taking the image normal to the ground and wall plane. Looking at Figure 4, I can see that if we used the total height H_{tot} in our calculation, the top of the wall would be projected an error e away from the actual base of the maze. Since I am taking the image normal to the ground, and I know the height of the wall H_w , I can move to the wall plane and do my calculations here instead of doing it at the ground plane, thus eliminating the error e . I can simply move the projected map a distance $z = -H_w$ after the mapping is done to coincide the top of the wall with the base of the wall.

2.4.1 Determining the length of the walls

As mentioned previously, Hough transform gives us the straight line segments of the maze and their respective lengths in pixels are easy to extract. The Hough transform algorithm outputs a list of x-y coordinates in pixels that define the start- and end-point of each detected line segment. To extract the lengths of each line segment I can use:

$$\text{Length of line} = l_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \quad [pixels] \quad (9)$$

Ground Sample Distance (GSD) can now be used to convert these lengths in pixels to lengths in meters. The length L can be expressed in [m]:

$$L_i = l_i \times GSD \quad [m] \quad (10)$$

$$L_i = l_i \times \frac{p}{fW} H_p \quad [m] \quad (11)$$

Where H_p is the distance from the optics to the top of the wall and W is the width of the image in pixels. p is the pixel pitch and f is the focal length of the image sensor.

2.4.2 Determining the position of the walls

The information used when finding the length of the walls can be also used to transform the position in the image to their respective real-life position. The Hough transform outputs a list of the start and end position of the line segments in pixels. To transform these values to real life values we need information about the location of the image sensor. When taking the image, I assume I have measurements for the x-y position in the real world of the image sensor. Since I am taking the picture normal to the wall- and ground plane, the x-y position of the image sensor in the real world corresponds to the center of the image.

There are several ways to extract and transform the position of the walls from the Hough transform to real-life units. One way is to transform the start- and end point coordinates in pixels of each wall segment to meters with respect to the image center. This can be done by first finding the GSD of the image and after that subtracting the

x- and y component of each start- and end point by the x- and y component of the center and multiplying by the GSD.

If I have the position of the center (x_c, y_c) and the start- and end point of a line segment (x_1, y_1) and (x_2, y_2) in the image respectively. The position of each of these points relative to the center can be expressed as:

$$\text{Position of start point} = GSD \times (x_1 - x_c, y_1 - y_c) \quad [m] \quad (12)$$

$$\text{Position of end point} = GSD \times (x_2 - x_c, y_2 - y_c) \quad [m] \quad (13)$$

Since I know the start and end point coordinates in real life units of all wall segments, I have all the information I need to express the characterization of the maze. I assume I know the orientation of the image sensor when the image is taken, giving us a complete characterization of where each wall segment is in the real world.

It is important to note that I am not calculating or implementing the orientation of the image sensor when the image is taken. It is assumed we have a measurement of the orientation about an inertial axis in the ground- and wall plane, so that the mapping directly relates to this inertial frame. This means that we require the UAV to measure the attitude when mapping the maze.

2.5 Ultrasonic range sensor

In order to calculate a GSD, the system needs a way to measure the distance from the camera to the ground. This information can be obtained by using an ultrasonic range sensor.

The ultrasonic range sensor utilizes high-frequency sound wave pulses to determine the range from the sensor to the object reflecting the pulse back to the sensor. The sensor outputs the time taken from the sound transmission to the detection of the reflected signal, and is then used to calculate the distance from the sensor to the object.

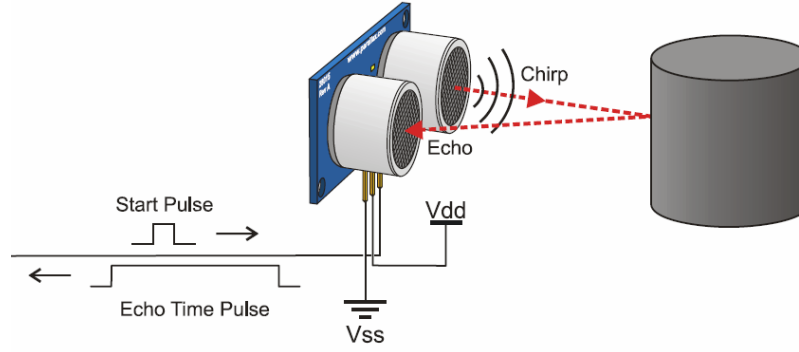


Figure 5: Ultrasonic range sensor. From [14]

The distance is calculated as follows:

$$speed = \frac{distance}{time} \quad (14)$$

In dry air at 20 C, the speed of sound is 343 meters per second [15]. The equation will give the total distance traveled by the sound pulse, so it must be divided by two. Giving:

$$distance = \frac{speed \times time}{2}$$

$$distance = 17150 \times time \quad [cm]$$

2.6 Image Sensor

The image sensor is one of the most important components in the system. There are several different types of image sensors available, but most of the image sensors I would use consist of an array of small sensors capable of detecting incoming illumination energy and transforming it to digital images.

Gonzales explains the process of the digital image sensor as follows: *"The idea is simple: Incoming energy is transformed into a voltage by the combination of input electrical power and sensor material that is responsive to the particular type of energy (wavelength) being detected. The output voltage waveform is the response of the sensors, and a digital quantity is obtained from each sensor by digitizing its response."* [1]

That means the response of each sensor in the sensor array of the image sensor is a continuous voltage waveform, and to create digital images from this data the process of sampling and quantization must be applied. Figure 6 illustrates the the basic idea behind sampling and quantization.

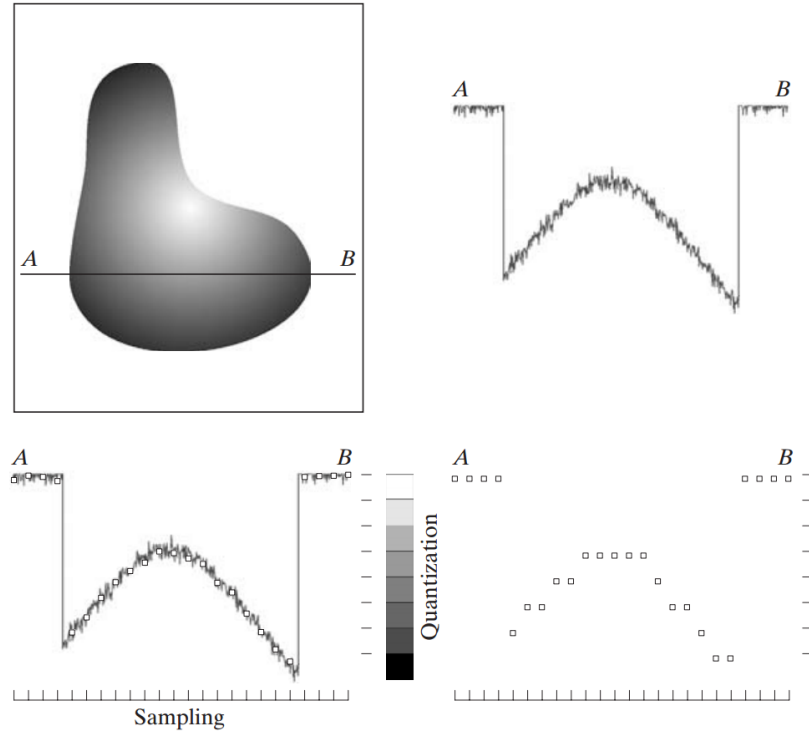


Figure 6: Top left: Continuous image
 Top right: A scan from A to B in the continuous image
 Bottom left: Sampling and quantization
 Bottom right: Digital scan line [1]

Sampling can be described as digitizing the coordinate values of the image, while quantization is the process of digitizing the amplitude values in the image. The amplitude of any given coordinate in the detected image is the intensity value. In terms of this application, it is not vital to understand every detail of how the image sensor works, but a general understanding of the technology is preferred.

The image sensor used in this thesis is the IMX219, which is a active pixel sensor CMOS (APS CMOS) image sensor using CMOS technology. APS means that the integrated circuit in the sensor contains an array of pixel sensors, where each pixel contains a photodetector and amplifier.

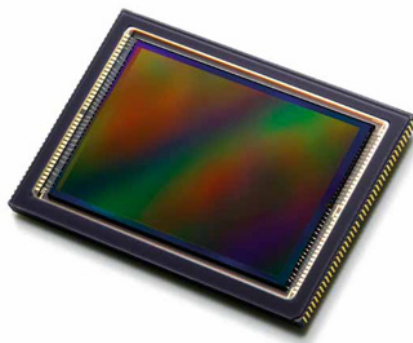


Figure 7: APS CMOS Image Sensor

Figure 7 depicts an APS CMOS image sensor. CMOS is one of the leading technologies for small image sensors often used in mobile phone cameras. CMOS is a technology for designing integrated circuits, and is preferred in image sensors for its small power consumption and small image lag.

2.7 Technical hardware requirements

If a complete image processing algorithm is to be implemented on a mobile computation device, the hardware selected needs to be able to compute and run image processing in a timely manner. Image processing is known to be demanding on hardware, and most advanced implementations such as autonomous cars use high-end dedicated graphic cards [13] to be able to run in real time. The algorithm I am implementing does not map in real time, thus the requirements for my hardware performance is reduced. However, the hardware should feature enough processing power and storage to be able to run the image processing algorithm as well as storing the image in between operations.

These are some of the over-arching hardware requirements:

- Capable of convolving high resolution images ($> 1920 \times 1080$ pixels)
- Enough RAM to store images during operations
- Dedicated graphics chip
- Powerful enough to be suitable for further project developments, potentially into real-time

The size of the image output is maximum 3280×2464 , which makes approximately 8.08M pixels. Since the bit depth of the sensor is 10-bit, I can calculate the file size of the image assuming a 10 bit depth:

$$\begin{aligned} 3280 \times 2464 \quad [pixels] \times 10[bit] &= 80819200 \quad [bit] = 10102400 \quad [bytes] \\ 10102400 \quad [bytes] \div 1048576 &= 9,63 \quad [Megabytes] \end{aligned}$$

I assume the image file size is at a maximum 10 Megabytes for calculation purposes. This means that the hardware needs to provide sufficient RAM to store images of several Megabytes. The GPU should also be able to do operations on pictures of this size, which means a dedicated graphics chip should be present in the system.

2.8 Positioning of the system

In the problem description of this paper, the following is stated:

"Determine how information such as attitude, height and position can be extracted from the drone, and what the difficulties are with respect to the accuracy of this information"

There are several potential solutions available to be able to extract useful positional data from a drone. The determining factor in positioning is its accuracy, and if the positioning needs to be indoors or outdoors.

2.8.1 Pure IMU based positioning

This is the most simple and cheapest potential solution to positioning. One would think using an inertial measurement unit (IMU) some useful information about position could be extracted, but this is rarely the case.

IMU works great when measuring orientation, this is because it uses a gyroscope to measure linear velocity, and integrate it once to get orientation. There is some linear drift in there, but since I can measure the magnetic north and the gravity vector, it can be corrected for.

However, this not true for positional tracking. In order to track the position, the accelerometers needs to be used to measure acceleration, as opposed to linear velocity. The acceleration needs to be integrated twice to get the position. And thus the big issue occurs when integrating twice, since you are not accumulating error in the linear fashion, but in the quadratic fashion. That means the drift will always increase, thus making it not suitable for positioning. [31]

During the initial phases of this paper, a meeting was conducted with a researcher at the Department of Engineering Cybernetics together with my advisor. The researcher confirmed the fact that IMU positioning is not feasible for the paper, and that other avenues should be evaluated.

2.8.2 GPS positioning

If the drone is designed to operate outdoors, GPS positioning is a great way to control the drone. It is the most commercially available technology in autopilots for drones, and is used in many different applications. One major disadvantage in this technology is that it performs poorly indoors, thus making it only accurate for outdoor use.

By the use of triangulation between at least three satellites, the position can be extracted in three dimensions. GPS chips are available in very small sizes and range from very cheap off-the shelf hardware to expensive dedicated GPS with high accuracy.

Many drone autopilots feature a built-in GPS chip that provides position relatively accurate, making the operator able to control the drones position. The system which I am working on is meant to be used indoors, so GPS positioning would not be a good solution to the positioning needs of the system.

2.8.3 Camera assisted indoor positioning

Camera assisted positioning can be many things, but what I mean by this is an array of cameras providing positioning by detecting an element in the images captured by the cameras. A system similar to this is implemented on NTNU in the B333 "Bevegelseslab" room in the Electronics Department.

The system works by having several cameras placed at different locations in a room detect an object from different angles, and then place this object in a 3D reference frame in relation to a coordinate system selected by the user by triangulation. This system provides a very accurate position estimate, as well as a very accurate orientation estimate, but is often very expensive to implement because of the hardware required.

In most cases, such a system is used for testing purposes, where a position- or orientation estimator is compared to the position and orientation provided by the camera system. This is because the camera system will provide a very accurate position in three dimensions.

3 Hardware specification

The platform selected needs to fulfill the requirements set in the theory section, and will be a hub where different sensors such as a camera and distance sensor can be mounted and integrated. The following section describes the hardware specification of the complete system.

3.1 Raspberry Pi 3 Model B

The Raspberry Pi, developed in the United Kingdom by the Raspberry Pi Foundation, is the single-board computer selected for my application. It is a powerful computer and has the I/O peripheral support expected from a regular PC. It features a powerful processing unit suited for image processing, and enough RAM to support such applications. The Raspberry Pi is open source and compatible with many operating systems, opening up for further improvement by other project- and master students.

Specifications [16]:

- **System-on-chip** Broadcom BCM2837
- **CPU** 1.2 GHz 64-bit quad-core ARM Cortex-A53
- **Memory** 1 GB LPDDR2 RAM at 900 MHz
- **Graphics** Broadcom VideoCore IV 400 MHz
- **Power** 10.0 W (2 A)

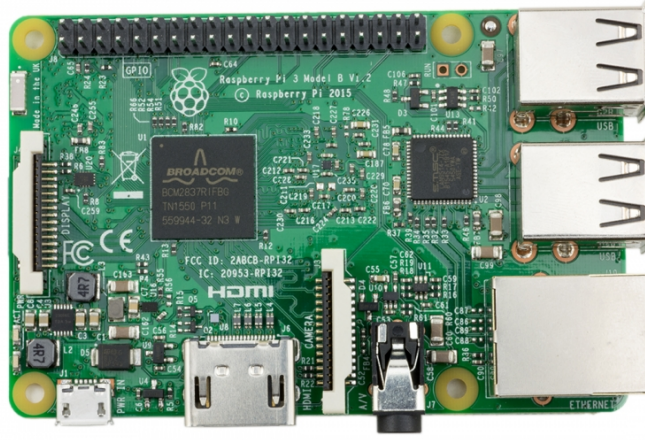


Figure 8: Raspberry Pi 3 Model B

Other technical specifications [16]:

- Bluetooth Low Energy (BLE)
- BCM43438 WiFi
- 40pin extended GPIO
- 4 x USB 2 ports
- 4 pole Stereo output and Composite video port
- Full size HDMI output
- CSI camera port for connecting the Raspberry Pi camera
- DSI display port for connecting the Raspberry Pi touch screen display
- Micro SD port for loading operating system and storing data

3.2 Raspberry Pi Camera Module v2

The Raspberry Pi Camera Module v2 is the second version of the camera module produced by the Raspberry Pi Foundation. It features excellent compatibility with the Raspberry Pi, and is open-source.

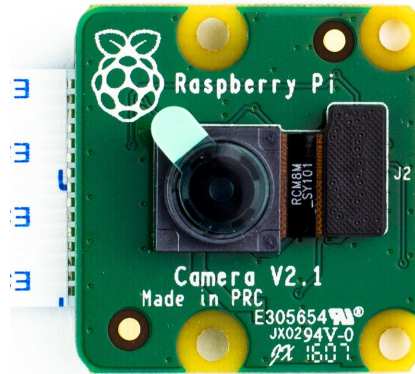


Figure 9: Raspberry Pi Camera Module v2

Specifications [17]:

- **Sensor** Sony IMX219
- **Sensor Resolution** 3280×2464 pixels
- **Sensor Image Area** 3.68×2.76 mm (4.6mm diagonal)
- **Pixel Size** $1.12 \mu\text{m} \times 1.12 \mu\text{m}$
- **Focal length** 3.04 mm

3.3 HC - SR04 Ultrasonic Ranging Module

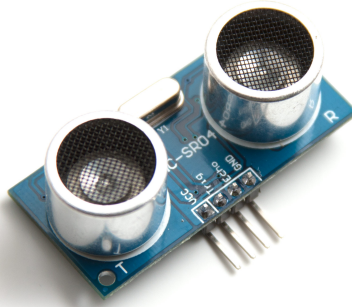


Figure 10: HC - SR04

- **Max Range** 4 m
- **Min Range** 2 cm
- **Resolution** 0.3 cm
- **Working Voltage** DC 5V

The HC - SR04 is an ultrasonic range sensor made by Cytron Technologies. It features a range interval suitable for the application in this project, and has a good resolution for a low price. It is compatible with the Raspberry Pi, and has a track record of usage in Raspberry Pi projects.

3.4 Raspberry Pi Universal Power Supply



Figure 11: Raspberry Pi Universal Power Supply

- **Input:** 100-240V
- **Output:** 5.1V/2.5A

The selected power supply is the official power supply of the Raspberry Pi foundation.

4 Hardware Setup

4.1 System Architecture

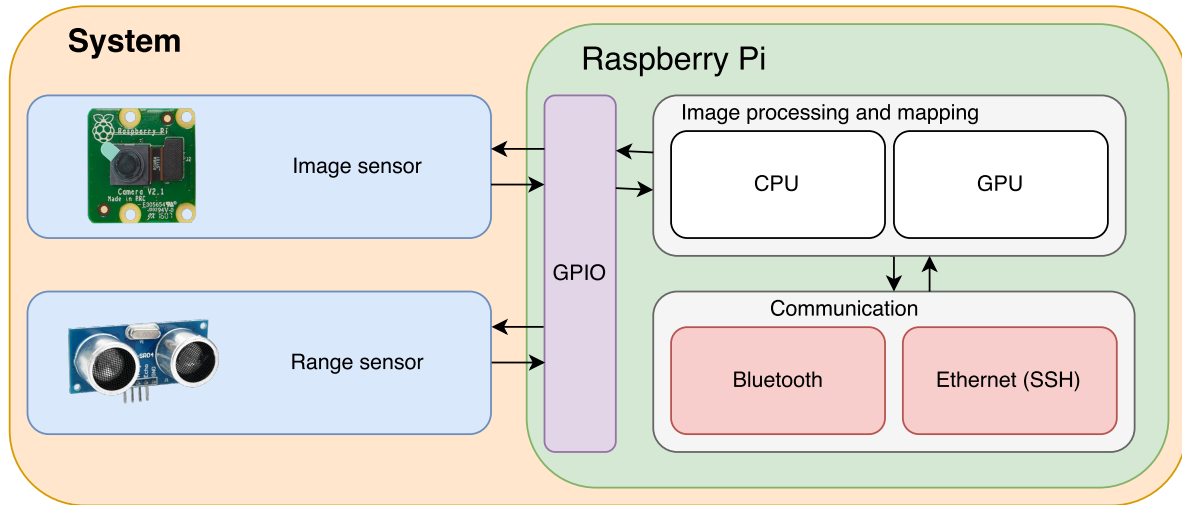


Figure 12: System architecture

The overall system architecture is illustrated in Figure 12. With the exception of a power source and memory, this is a view simplification of the system interconnections. The sensors are connected through the GPIO interface on the Raspberry Pi, and all the image processing is done internally in the CPU/GPU.

4.2 System Setup

Figure 13 and 14 shows the complete system in a picture of how the all the hardware is connected for normal operation and for debugging. Since the Raspberry Pi is running Raspbian OS which features a GUI, debugging can be done on a monitor with USB peripherals connected such as a keyboard and mouse. During normal operation however, the system should not be connected to anything else than power and a communication protocol.

I will describe each component of this system closer in this section.

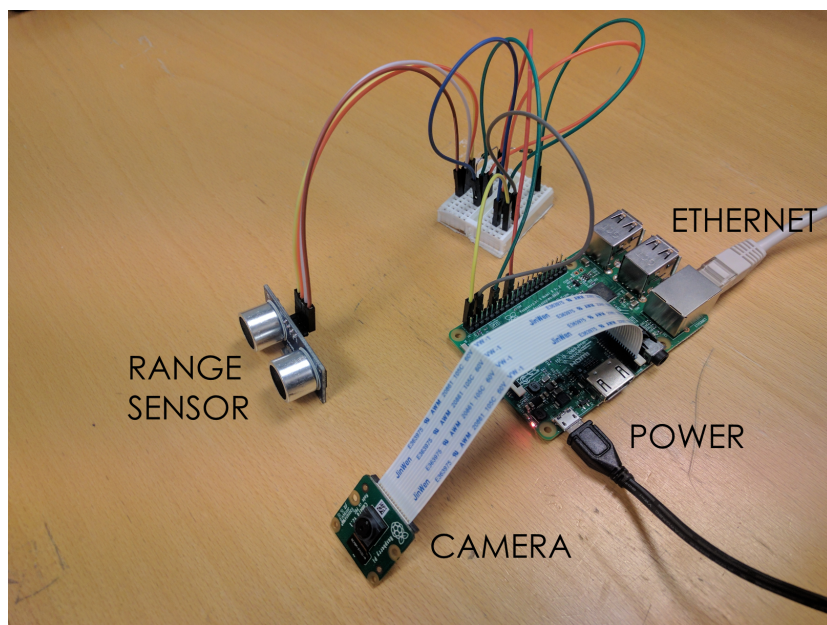


Figure 13: Normal hardware setup

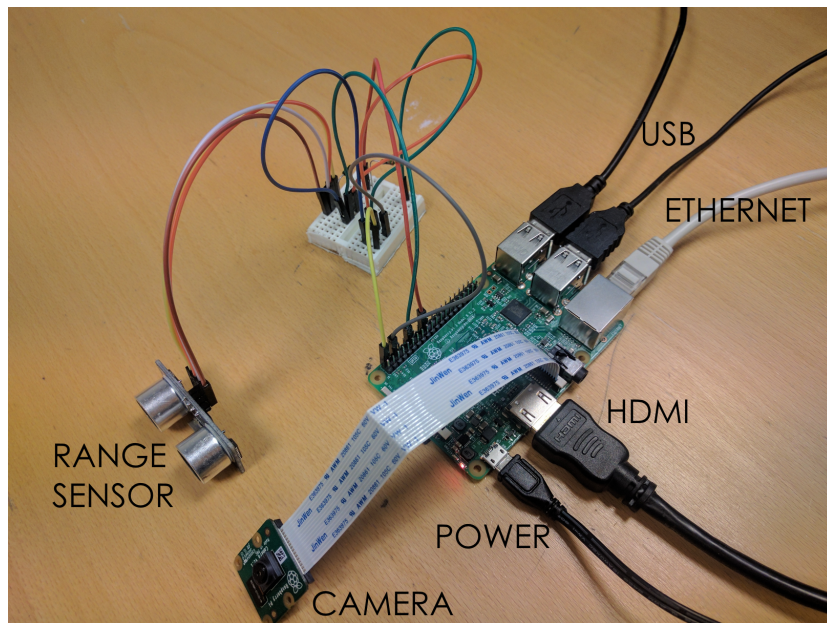


Figure 14: Debug hardware setup

4.3 Raspberry Pi

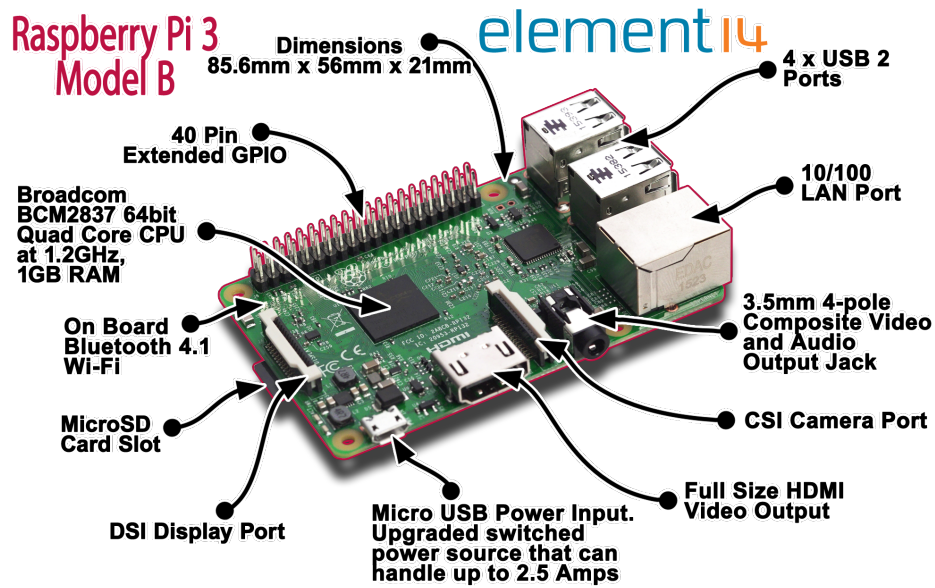


Figure 15: Raspberry Pi 3 Model B. [20]

Figure 15 displays most of the I/O interfaces on the Raspberry Pi 3 Model B.

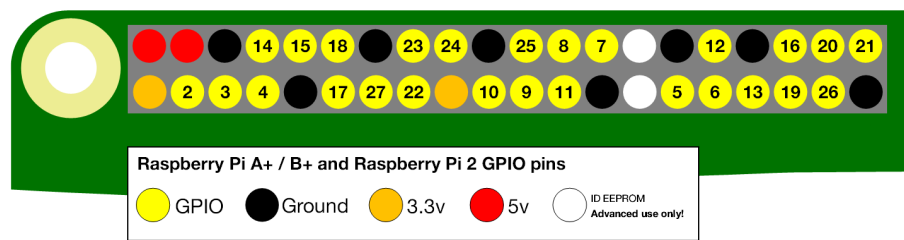


Figure 16: GPIO Pins on the Raspberry Pi 3 [16]

Figure 16 details the GPIO pins on the Raspberry Pi. This map is used to define the correct pins in the range sensor programming.

4.4 Camera

I am using the Raspberry Pi Camera v2 Module optimized for the Raspberry Pi. The camera includes a CSI cable that is connected to the Raspberry Pi through the CSI camera port on the board. The blue markings on the CSI cable should face the LAN port.

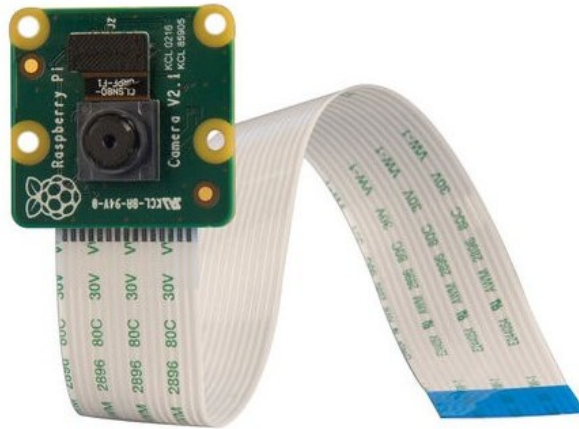


Figure 17: Raspberry Pi Camera Module V2 with CSI cable

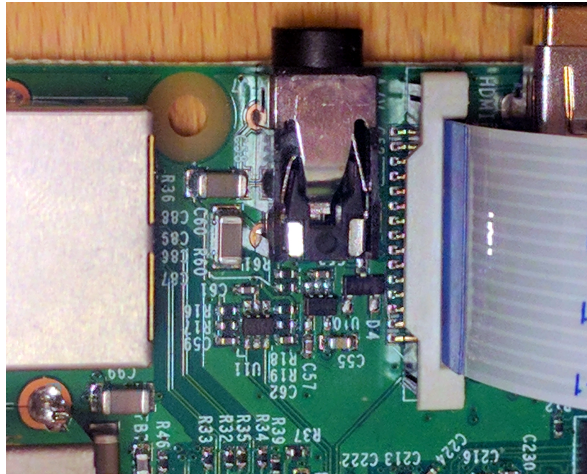


Figure 18: CSI Cable mount

4.5 Range sensor

The HC-SR04 is a very easy sensor to use, but in order to make it compatible with the Raspberry Pi, there are some modifications needed. The GPIO input pins on the Raspberry Pi is made for 3.3V while the ECHO output pin on the HC-SR04 delivers 5V. So in order for the Raspberry Pi to interact with the HC-SR04, I need to reduce the voltage to 3.3V.

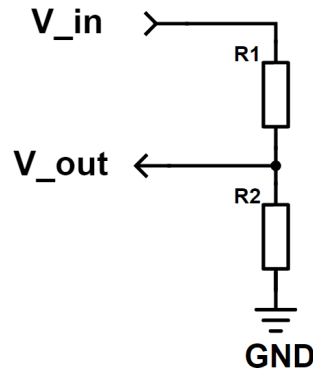


Figure 19: Voltage divider

This can be done with a simple voltage divider. Figure 19 illustrates a simple voltage divider circuit that can be easily created with two resistors since we know the input and output voltages. The formula for a voltage divider is as follows:

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2} \quad (15)$$

I want the V_{out} to be 3.3V, and the V_{in} is 5V. By a combination of resistors R_1 and R_2 the desired output voltage can be obtained. By picking $R_1 = 820\Omega$ and $R_2 = 1500\Omega$ we get:

$$V_{out} = 5 \times \frac{1500}{820 + 1500} = 3.23V$$

This is close enough for our application, and this is verified in the tests later in the thesis. Figure 20 shows how this voltage divider is implemented, and to which GPIO pins on the Raspberry Pi it is connected. Figure 21 shows the four pins on the HC-SR04.

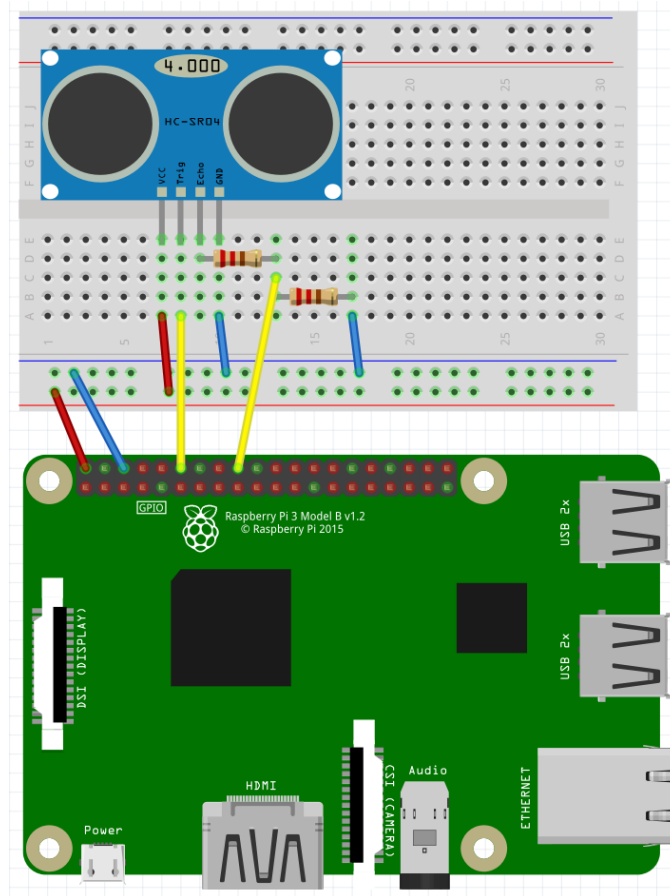


Figure 20: Range sensor setup



Figure 21: HC-SR04 pins. Vcc, Trig, Echo, GND

4.6 3D printed case

To make the system more compact and to facilitate for a better hardware implementation a 3D printed Raspberry Pi case was used. The case was collected from the Thingiverse library [26].

This case is designed so that the CSI cable and the GPIO pins can be accessed easily on the Raspberry Pi. The case was printed on a low-end off-the-shelf 3D printer.

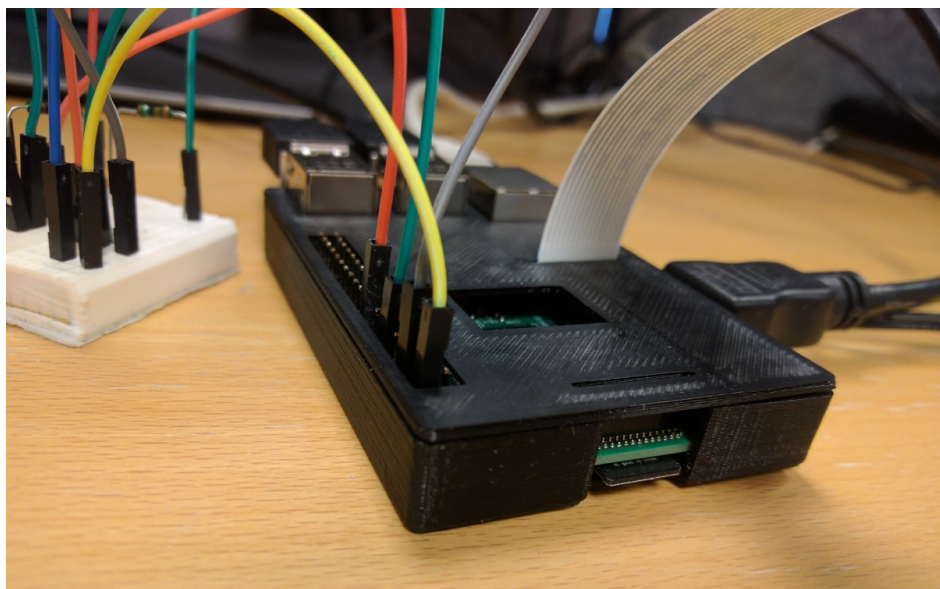


Figure 22: 3D printed case

5 Software Setup

The software setup of the system is presented in the following sections, and will explain all the software setup done on the Raspberry Pi in detail. Most of the setup information; which dependencies, and what packages are needed was obtained from the PyImageSearch website [25].

I will present different code examples in this and the next section, which will consist of Python code and Bash (Unix Shell) code. The bash code will be presented as a runnable script in the Raspberry Pi shell like this:

```
$ sudo apt-get install <SomeSoftware>
```

These commands are mostly used to install the needed framework and dependencies for the application in Python.

The Python code examples will be presented in the following manner:

```
1 #This program prints Hello, world!
2 print('Hello, world!')
```

5.1 Operating System, Raspbian Jessie

The operating system installed on the Raspberry Pi is Raspbian Jessie. This is the most widely used OS for the Raspberry Pi, and provides all I need to make the application work. Keep in mind this OS is chosen as a high-level test platform for my application, to see if it is feasible to implement the algorithm. Raspbian Jessie features a GUI similar to Ubuntu and is easy to use.

The fundamental idea in the ongoing development of this project is to show the feasibility of the implementation on a high-level OS, then gradually optimize the implementation in the future by using a headless OS and a better optimized programming language.

5.1.1 Installation

There are a few steps required to install Raspbian Jessie. Most of the information is found online, and is easily available. There are a few components needed to install the Raspbian Jessie OS.

Requirements:

- SD Card with at least 8GB
- SD Card reader
- Image writing software tool

To install an OS on the Raspberry Pi:

1. Download the Raspbian Jessie image [22]
2. Insert the SD Card into the SD Card reader on a computer
3. Download and install Win32DiskImager [21]
4. Select the OS image and Write to the SD Card

The SD Card now contains the OS needed to run on the Raspberry Pi. The Raspberry Pi boots from the SD Card inserted in the card reader automatically, so simply insert the SD Card in the reader on the Raspberry Pi and plug in the power source.

5.2 Python 2.7

The image processing has been implemented in Python 2.7. This choice was based on the compatibility with OpenCV, which is supported by Python and C++, and the fact that there exists many image processing implementation projects for the Raspberry Pi in Python.

Python is a high-level programming language that is used for general-purpose programming. It is an interpreted language which features a strong focus on code readability

and is an expressive language that often lets programmers express functionality in fewer lines than in for instance C++. [23]

5.2.1 Installation

The development version of the Python header files needs to be installed so I can compile OpenCV with Python bindings later:

```
$ sudo apt-get install python2.7-dev
```

This is so I do not run into errors when running *make* to compile OpenCV later. I also need to install 'pip', a Python package manager:

```
$ wget https://bootstrap.pypa.io/get-pip.py  
$ sudo python get-pip.py
```

5.2.2 Dependencies

Before installing the Virtual Environment in the next section, some dependencies needed for the application needs to be installed. First off update and upgrade the Raspberry Pi to be sure all the packages are up to date:

```
$ sudo apt-get update  
$ sudo apt-get upgrade
```

Cmake needs to be installed so OpenCV can be compiled.

```
$ sudo apt-get install build-essential cmake pkg-config
```

Install image and video I/O packages for Image processing:

```
$ sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev  
↪ libpng12-dev  
$ sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev  
↪ libv4l-dev  
$ sudo apt-get install libxvidcore-dev libx264-dev
```

Install some OpenCV dependencies:

```
$ sudo apt-get install libgtk2.0-dev  
$ sudo apt-get install libatlas-base-dev gfortran
```

5.2.3 Virtual Environment

It is good practice to incorporate and utilize virtual environments to keep the dependencies required by different projects in separate places. This creates isolated, independent Python environments for each of the projects. This will make it easy to keep track of all the dependencies needed in the application. To install:

```
$ sudo pip install virtualenv virtualenvwrapper  
$ sudo rm -rf ~/.cache/pip
```

Update the `~/.profile` file with the following script:

```
$ echo -e "\n# virtualenv and virtualenvwrapper" >> ~/.profile  
$ echo "export WORKON_HOME=$HOME/.virtualenvs" >> ~/.profile  
$ echo "source /usr/local/bin/virtualenvwrapper.sh" >> ~/.profile
```

This will update the `~/.profile` file so that the virtual environment can be accessed. After this I can create the virtual environment, which I will call 'cv', for Python:

```
$ mkvirtualenv cv -p python2
```

I can now install and develop the project entirely independent from any other Python projects on the Raspberry Pi. To work on the virtual environment use the following command:

```
$ source ~/.profile  
$ workon cv
```

When working in the virtual environment, the shell will display '(cv)' on the left side of the command prompt like this:

```
(cv) pi@raspberrypi:~ $ <command>
```

From this point on in the software setup, it is assumed that I will be working in the virtual environment. For numerical processing, I need to install a Python package named Numpy:

```
(cv) $ pip install numpy
```

This installation can take up to 10 minutes to finish.

5.3 OpenCV 3

Open Source Computer Vision (OpenCV) is an open-source image processing toolbox used for computer vision applications, often in real-time [24]. It features a library of image processing functions similar to that of the Image Processing Toolbox™ in MATLAB.

5.3.1 Installation and Compilation

Grab the official OpenCV software with the latest version (April 2017 - 3.2.0) from their Github repository.

```
$ cd ~  
$ wget -O opencv.zip https://github.com/Itseez/opencv/archive/3.2.0.zip  
$ unzip opencv.zip
```

For some extra features I install the full version of OpenCV, by accessing the `opencv_contrib` repository:

```
$ wget -O opencv_contrib.zip  
  ↪ https://github.com/Itseez/opencv_contrib/archive/3.2.0.zip  
$ unzip opencv_contrib.zip
```

After the download has finished, the compilation of OpenCV can be started. First I need to be working in (cv) virtual environment:

```
$ workon cv
```

After this, CMake can be run to setup OpenCV:

```
$ cd ~/opencv-3.2.0/  
$ mkdir build  
$ cd build  
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \  
  -D CMAKE_INSTALL_PREFIX=/usr/local \  
  -D INSTALL_PYTHON_EXAMPLES=ON \  
  -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib-3.2.0/modules \  
  -D BUILD_EXAMPLES=ON ..
```

The output of the CMake should be inspected to make sure the Python 2.7 make location is in the virtual environment (cv). If this is the case, I can make:

```
$ make
```

This will compile OpenCV with one core of the CPU. Some attempts was done with the command 'make -j4' which utilizes all four cores of the CPU. This was not successful, and made the Raspberry Pi crash in all attempts. This operation takes over one hour. After the compilation is finished, the next step is to install:

```
$ sudo make install
$ sudo ldconfig
```

The final step is to link the bindings of OpenCV into the virtual environment:

```
$ cd ~/.virtualenvs/cv/lib/python2.7/site-packages/
$ ln -s /usr/local/lib/python2.7/site-packages/cv2.so cv2.so
```

All the OpenCV functionality is now usable in the virtual environment, and opens up image processing functionality in Python. To use OpenCV in Python projects, include it at the start of the program:

```
1 #OpenCV
2 import cv2
3 #Numpy
4 import numpy
5
6 #Rest of program
```

5.4 Remote Desktop and Connection

I found it beneficial to be able to work remotely on the Raspberry Pi from my workplace in a remote desktop. This means I can run the Raspberry Pi headless when testing the image processing application. The steps involved in connecting to the Raspberry Pi is the following:

1. Determine the Raspberry Pi IP address
2. Connect through the "Remote Desktop Connection" application in Windows

5.4.1 Determining IP address

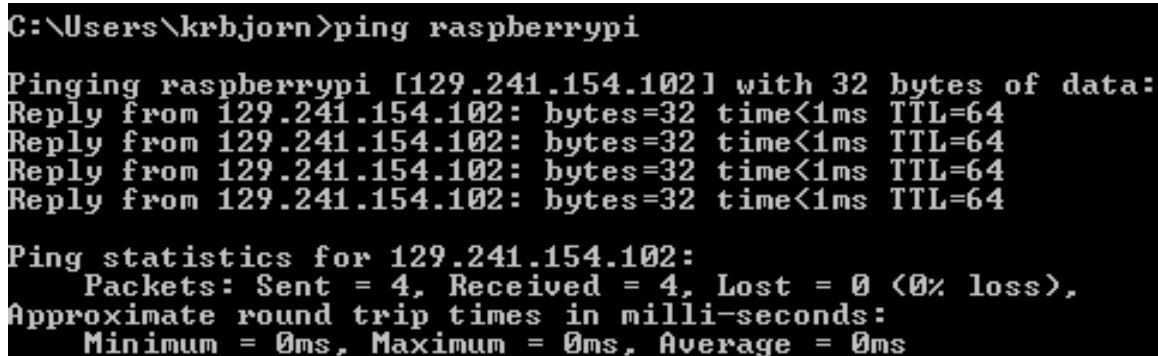
I have installed Samba on the Raspberry Pi, so that the device can be pinged on the local network to obtain the IP address:

```
$ sudo apt-get install samba
```

By typing the following command in the Windows cmd Shell, it returns the IP address of the Raspberry Pi:

```
ping raspberrypi
```

This will output:



```
C:\Users\krbjorn>ping raspberrypi

Pinging raspberrypi [129.241.154.102] with 32 bytes of data:
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64

Ping statistics for 129.241.154.102:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 23: Raspberry Pi IP

5.4.2 Remote Desktop

To connect remotely to the Raspberry Pi, I have installed some dependencies that make this work:

```
$ sudo apt-get install xrdp
$ sudo apt-get install tightvncserver
```

Now that I have the Raspberry Pi IP address, I can connect to it through the Remote Desktop Connection application in Windows, by typing in the Raspberry Pi IP address in the following window:

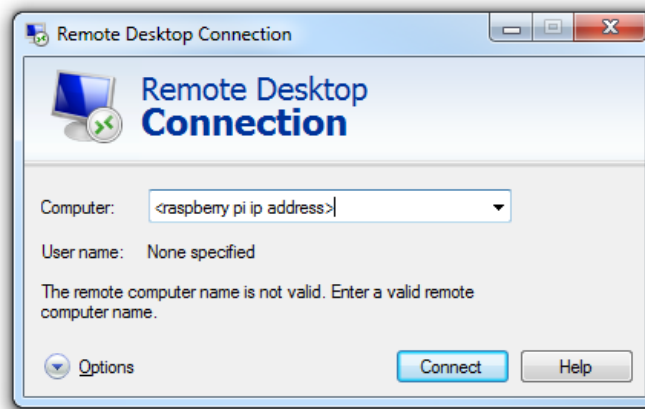


Figure 24: Windows Remote Desktop Connection

This will open a new window that asks for a username and password for the Raspberry Pi. The username and password is:

Username	Password
pi	master2017

Table 2: Username and Password

After entering the username and password in this login prompt, I have complete remote access to the Raspberry Pi.

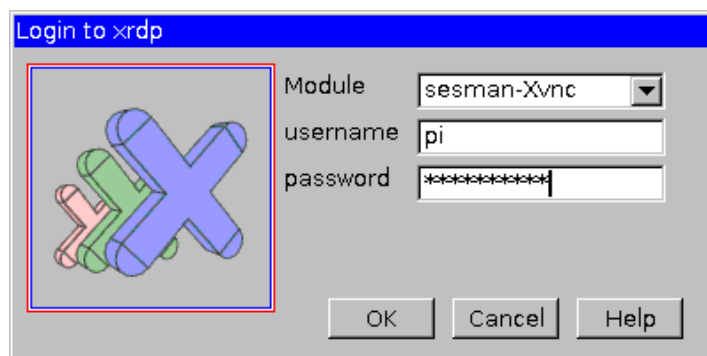


Figure 25: Raspberry Pi Login

6 Software Implementation

In the following section, the actual software implementation of the system will be covered. The image processing algorithm that was developed in Bjørnsen 2016 [19] is further developed and ported from MATLAB to Python. Other than the image processing algorithm, the communication between the Raspberry Pi and a computer, as well as the camera and image capturing is also presented.

The implementation is built up as follows:

1. Image capture by camera with scaling
2. Image processing. Consisting of:
 - (a) Edge detection using the Canny algorithm
 - (b) Hough transform to identify wall segments
3. Mapping. Consisting of:
 - (a) Range sensor implementation
 - (b) Transformation from pixel coordinates to real life coordinates using GSD
4. Communication:
 - (a) SSH
 - (b) Protocol and format

6.1 Camera and Image Capturing

Our system should be able to capture an image with the camera and store it in memory so that the image processing algorithm can be run with it. By using the PiCamera [27] framework, I am able to call functions in Python that will capture and save the image, as well as defining variables and operations relating to the image.

To be able to use the module I must import PiCamera in the beginning of the software:


```
1 from picamera import PiCamera
```

I designed the capture function such that it returns the captured image, and is designed to be run from the main loop function. The resolution of the image can be changed by setting the `camera.resolution`. The maximum resolution for the Raspberry Pi camera is 3280×2464 .

```
1 def capture():
2     #Initialize camera
3     camera = PiCamera()
4     #Set camera resolution
5     camera.resolution = (3280,2464)
6     #Capture image and save
7     camera.capture('Image.png')
8     #Open image in greyscale
9     I = cv2.imread('Image.png',0)
10    #Resize image
11    I = cv2.resize(I, (0,0), fx=0.4, fy=0.4)
12    #Return image I
13    return I
```

This is the most basic implementation of camera capture, and most of the technical aspects such as focus and light correction is done in the `camera.capture` function.

6.2 Image Processing

The image processing part of the software is similar to that found in Bjørnsen 2016 [19], only that it is programmed in Python. Some of the MATLAB functionality found in the Image Processing Toolbox are not the same as that of the OpenCV library, but yields similar results.

The image processing section uses OpenCV and Numpy for the calculations. They must be included in the beginning:

```
1 import numpy as np
2 import cv2
```

The implementation is described below. It includes a part which draws the detected lines on the original picture. This is for testing purposes to see if I manage to detect the whole wall segment.

```
1 def detectededges(img):
2     #Canny edge detection
3     threshold = 400
4     BW2 = cv2.Canny(img,threshold,threshold*0.4)
5
6     #Write the edges detected for diagnostics
7     cv2.imwrite('edge.jpg',BW2)
8
9     #Resize back to original size
10    BW3 = cv2.resize(BW2, (0,0), fx=1/0.4, fy=1/0.4)
11
12    #Hough Transform
13    lines = cv2.HoughLinesP(BW3,5,np.pi/180,500,100,800,200)
14
15    #Draw lines on original picture to illustrate the edges
16    I = cv2.imread('Image.png')
```

```

17     for i in range(len(lines)):
18         for x1,y1,x2,y2 in lines[i]:
19             cv2.line(I,(x1,y1),(x2,y2),(0,255,0),4)
20             cv2.imwrite('houghlines.jpg',I)
21
22     #HoughLinesP saves a 3D array. convert to 2D array
23     lines = lines.squeeze()
24     size = BW3.shape
25
26     #Return lines and size of the image
27     return lines, size

```

6.2.1 Edge detection

The edge detection part of the implementation is done on lines 3 and 4. The threshold value is a empirically tuned variable determined from the image. The maze we are going to map has a dark top side, which enables me to set the threshold value relatively high. Usually it ranges from 300 to 700.

For the actual edge detection I am using the Canny edge algorithm. This is implemented as `cv2.Canny(image,upper_threshold,lower_threshold)` in the OpenCV framework. This returns an array of detected edges, which can then be used in the Hough transform.

```

1     #Canny edge detection
2     threshold = 400
3     BW2 = cv2.Canny(img,threshold,threshold*0.4)

```

6.2.2 Hough Transform

As explained in the theory part of the paper, the Hough transform is used to link the detected edges together into meaningful segments. In this application that means linking in such a way that I can represent the detected edges along a wall as an actual

wall. This is done by setting the minimum line length and maximum line gap that it should fill, and running the `cv2.HoughLinesP(image, rho, theta, threshold[, lines[, minLength[, maxLineGap]])` function. This is an implementation of the probabilistic Hough Transform that finds line segments.

The input variables for this function is tuned for the size of the line segments on the array of detected lines. The other input variables such as `rho` and `theta` can be recognized from the theory part.

The `cv2.HoughLinesP` function returns a 3D array, but what I really want is a 2D array. The reason it does this originates from the official translation of the code from C++ to Python. By running `squeeze()` on the array I delete the empty dimension of the array, and this can be returned.

```
1      #Hough Transform
2      lines = cv2.HoughLinesP(BW3,5,np.pi/180,500,100,800,200)
3
4      #HoughLinesP saves a 3D array. convert to 2D array
5      lines = lines.squeeze()
```

6.3 Range sensor implementation

The range sensor is implemented in a `getdistance()` function which returns the detected range from the HC-SR04. The code was collected from ModMyPi [28] and modified.

The first part of the function defines the GPIO interface, that is, which pins are echo, and which is trig. By detecting the output of these pins, and measuring the time it takes, the distance can be calculated. In order for this function to work the following must be imported at the start of the program:

```
1 import RPi.GPIO as GPIO
2 import time
3 GPIO.setmode(GPIO.BCM)

1 def getdistance():
2     #Define GPIO pins on the Raspberry Pi
3     TRIG = 18
4     ECHO = 24
5
6     #Set which pins is echo and trig
7     GPIO.setup(TRIG,GPIO.OUT)
8     GPIO.setup(ECHO,GPIO.IN)
9     GPIO.output(TRIG, False)
10
11     #Sleep while sensor settles
12     time.sleep(2)
13     GPIO.output(TRIG, True)
14     time.sleep(0.00001)
15     GPIO.output(TRIG, False)
16
17     #Get the time for the calculation
18     while GPIO.input(ECHO)==0:
```

```
19     pulse_start = time.time()
20     while GPIO.input(ECHO)==1:
21         pulse_end = time.time()
22
23         #Calculate the distance
24         pulse_duration = pulse_end - pulse_start
25         distance = pulse_duration * 17150
26         distance = round(distance, 2)
27
28     return distance
```

The theory behind the distance calculation is explained in Section 2.5.

6.4 Mapping

The mapping part of the implementation is implemented in the `mapping(lines, distance, image_size)` function. This function takes in the detected line segments from the Hough Transform, and the detected distance from the range sensor implementation, and the size of the image. From this information it is able to transform the detected lines from pixel values to real life units.

The actual translation is done by multiplying the calculated GSD with each detected line segment, and updating the list.

```
1 def mapping(lines, distance, image_size):
2     ###Calculate GSD
3     #Sensor width
4     Sw = 3.68
5     #Focal length
6     Fr = 3.04
7     #Height of wall
8     wall_h = 30
```

```
9      #Height
10      H = distance-wall_h
11      #Image size
12      Iw = 3280
13      Ih = 2464
14      #GSD in cm/pixel
15      GSD = (Sw*H)/(Fr*Iw)
16
17      #Correct for image center and apply to lines
18      image_x = image_size[1]/2
19      image_y = image_size[0]/2
20
21      length_lines = [0 for i in range(len(lines))]
22
23      for i in range(len(lines)):
24          #Find length of each line
25          length_lines[i] = GSD*math.sqrt((lines[i][2]-lines[i][0])**2
26                                          + (lines[i][3]-lines[i][1])**2)
27
28          lines[i][0] = lines[i][0]-image_x
29          lines[i][1] = lines[i][1]-image_y
30          lines[i][2] = lines[i][2]-image_x
31          lines[i][3] = lines[i][3]-image_y
32
33      #Return scaled lines
34      return GSD*lines
```

6.5 Main function

The main function is based on the implementation presented in the beginning of this section. It does a step-by-step function call procedure that will yield the desired line segments of the walls detected. After this it will print the results to be collected on a SSH remote terminal.

```
1 def main():
2     #Capture and mapping
3     img = capture()
4     lines,size = detectededges(img)
5     distance = getdistance()
6     reallines = mapping(lines,distance,size)
7
8     #Positioning (not implemented; for communication)
9     x = 0
10    y = 0
11    theta = 90
12
13    #Print the lines detected in correct protocol
14    for i in range(len(reallines)):
15        print("{U: %s, %s, %s, %s, %s, %s, %s}" % (x, y, theta,
16            ↪ reallines[i][0], reallines[i][1]
17                ,reallines[i][2],reallines[i][3]))
18
19    #Cleanup
20    GPIO.cleanup()
```


6.6 Communication

The communication with the Raspberry Pi and the software is meant for SSH in this implementation. The communication is achieved by connecting the Raspberry Pi remotely through SSH and running the program while reading the output on the master computer.

The SSH master computer is in this implementation running Windows (similar to the LEGO robot server), and to establish SSH with the Raspberry Pi, a SSH client must be downloaded. The program of choice for this is PuTTY [29]. By typing the local IP of the Raspberry Pi in the "Host Name", one can connect easily.

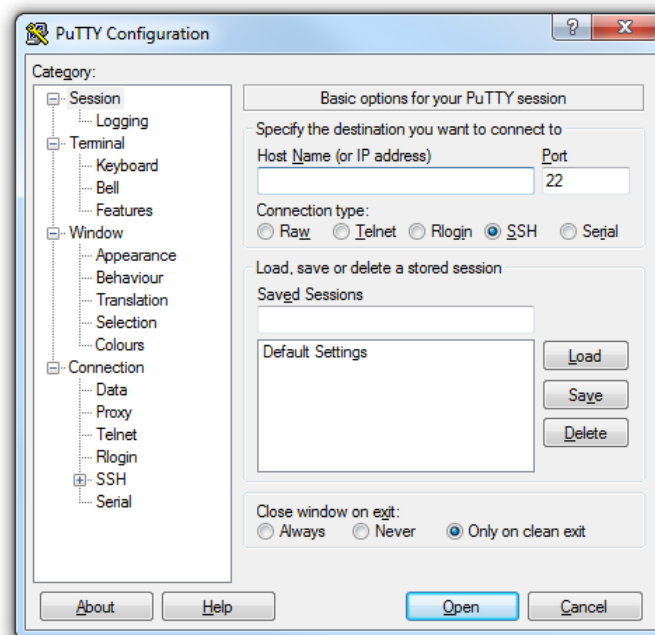


Figure 26: PuTTY in Windows

After typing the Raspberry Pi local IP, the following window will appear:

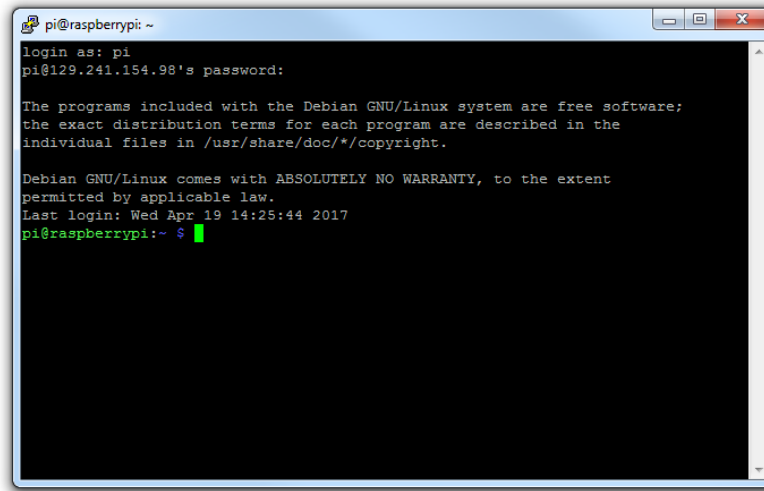


Figure 27: PuTTY terminal in Windows

After logging in, the program can be run remotely from the master computer. By running the program from there the output of the program will also be communicated over SSH in the terminal.

6.6.1 Protocol

The implementation of the LEGO robots features a communication protocol to communicate various pieces of sensor data to the host. Some of the other Master students working on this have provided me with a communication protocol that my implementation can be built around to facilitate for better integration with the existing solution in the future.

The protocol is implemented in the main function mentioned previously, and it is built as follows:

{U:	X	Y	Theta	Line Start X	Line Start Y	Line Stop X	Line Stop Y}
-----	---	---	-------	--------------	--------------	-------------	--------------

Table 3: Update message protocol

The fields should be separated by commas, and initiated with **U:** to identify the mes-

sage as an "Update" message in the LEGO communication protocol. The whole message should be in curly brackets.

The first three fields after the message identifier **U:**, is the X,Y and Theta positioning of the camera of the photo taken. Positioning is not implemented yet in the system, but the communication protocol opens for it in the future. For now, the position variables are set in the main function.

The next four fields are the parameters of the detected lines. All detected lines will have their own message sent, with the four variables needed to position the walls in relation to the camera. Every line is defined by the X and Y position of the start and stop of the line.

7 Testing

In order to see how the system performs in terms of accuracy and feasibility, several tests have been conducted for the various implementations in the system. Since the system is dependent on good tuning and accurate sensor measurements to provide a good mapping of the maze, testing is very important.

I will test the range sensor, the edge detection algorithm and the mapping algorithm. When testing the edge detection and mapping, I will compare it to the work previously done in Bjørnsen 2016 [19].

7.1 Range sensor

The range sensor measures the distance from the measuring device to a reflective object. The data collected from this sensor is used to determine the real life lengths of the walls in the maze.

7.1.1 Setup

The setup of the test is illustrated below. A measuring tape is placed along a surface, and the range sensor is placed at various distances along this measuring tape. At the end of the tape there is a straight wall that will reflect the sound waves.

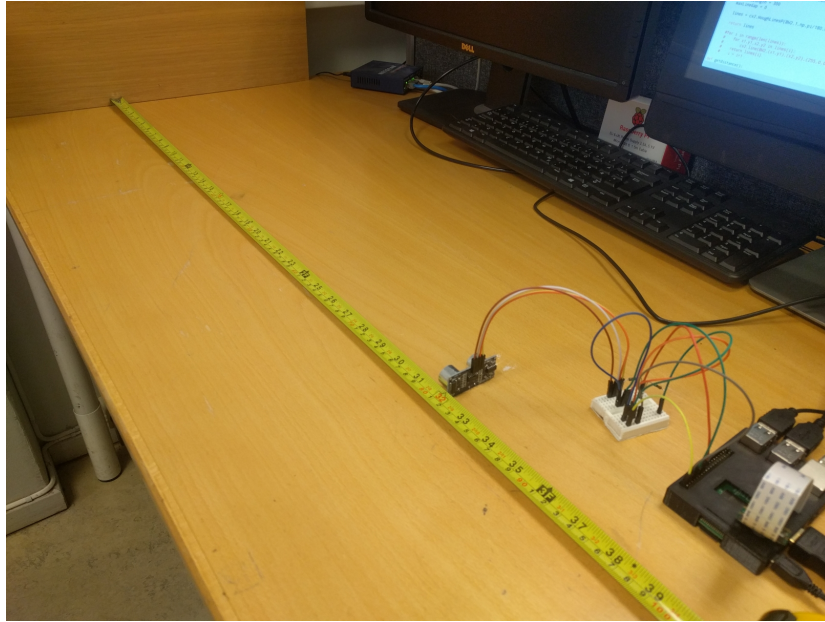


Figure 28: Range Sensor Setup

7.1.2 Measurements

The following measurements were made for varying distances to the wall. The sensor was placed such that the end of the two detectors were at the stated range.

Real Distance	Measured Distance	Diff	% Diff
40 cm	39,12 cm	-0,88 cm	2,2
50 cm	48,96 cm	-1,04 cm	2,08
60 cm	59,12 cm	-0,88 cm	1,46
70 cm	69,03 cm	-0,97 cm	1,38
80 cm	78,8 cm	-1,2 cm	1,5
90 cm	88,46 cm	-1,54 cm	1,7
100 cm	98,53 cm	-1,47 cm	1,47
110 cm	108,9 cm	-1,1 cm	1
120 cm	118,63 cm	-1,37 cm	1,1

Table 4: Range sensor measurements

The measurements range from 40 to 120 cm from the measured object. The difference

in the measurements range from 0,88 cm to 1,54 cm. In terms of percentages in differences, the range was from 2,2% to 1%. What is interesting, is that the actual difference seems to be about the same with varying distances. This means that there might be a "steady state" error which can be corrected for all distances by adding the mean of the differences to the measurement.

$$\text{Average Diff} = \frac{0,88 + 1,04 + 0,88 + 0,97 + 1,2 + 1,54 + 1,47 + 1,1 + 1,37}{9} = 1,1611 \quad [cm]$$

This value can be added to the measurements to improve the average estimation of the distance from the system to the ground. One could argue that since the system is not intended to be used at heights lower than 50 cm, the mean could be calculated from a different range to get a better estimate.

The code for the Range sensor test is included in the Appendix.

7.2 Edge Detection

An implementation of edge detection was developed in Bjørnsen 2016 [19], and to determine the feasibility of the newly implemented edge detection in Python it is beneficial to compare these two implementations with the same input. Since the implementations of the Canny edge algorithm is not entirely similar in OpenCV and in the Image Processing Toolbox in MATLAB, I have attempted to run them with the most similar thresholds and parameters as possible.

7.2.1 Setup

From Bjørnsen 2016 [19], we have the following test maze:

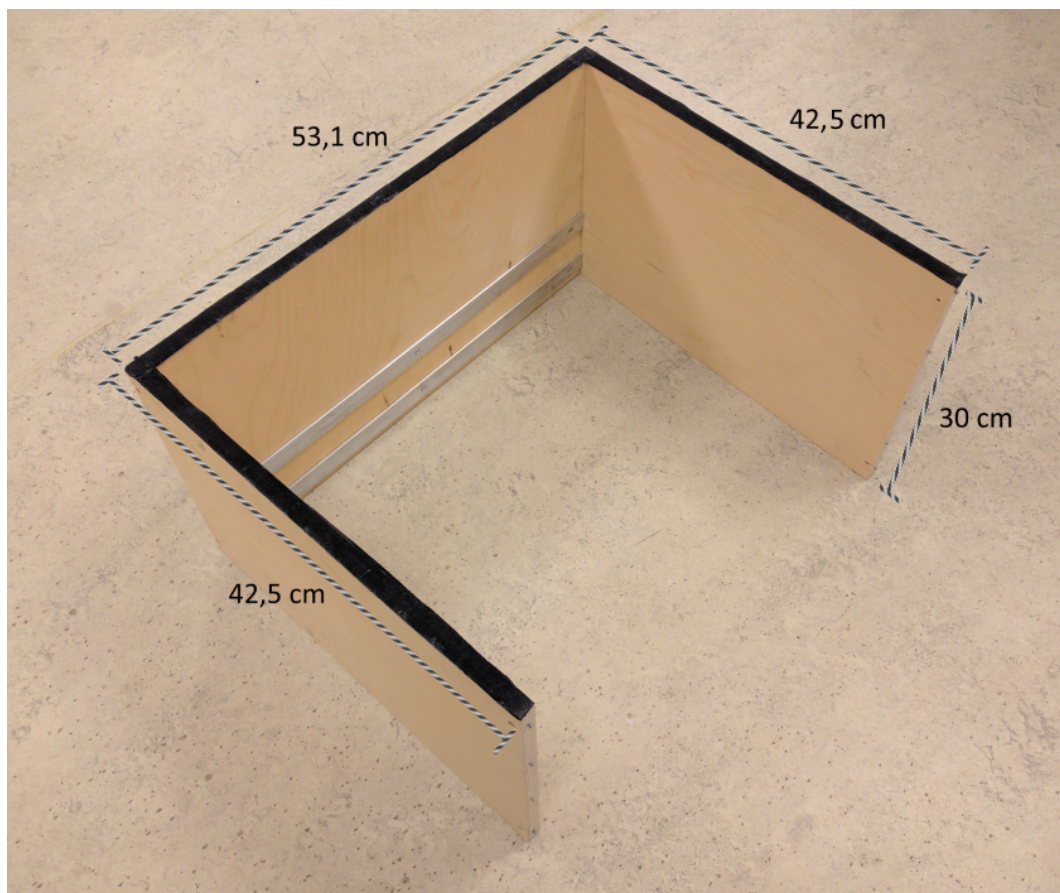


Figure 29: Test maze

Many images were taken of this maze using the Sony Exmor IMX377 in the project thesis, and since I am only testing for the edge detection aspect of the implementation; I can use the same images and compare both implementations. The image I am going to compare the edge detection implementations on is the following:

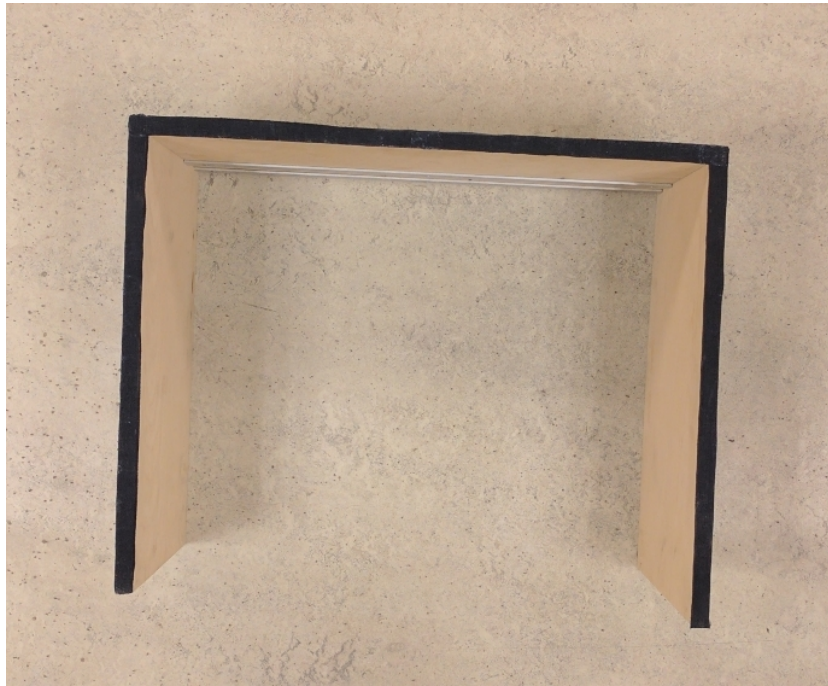


Figure 30: Test Image

The image was taken during the fall in Bjørnsen 2016 [19]. The picture was taken on a tripod facing straight down as described in the project report.

What I am testing for here is a complete edge detection along all the walls of the test maze in the picture. There should be no other edges detected other than the top of the walls and it should be continuous with no holes along the edges.

The software that I used to test the edge detection is included in the appendix.

7.2.2 Results

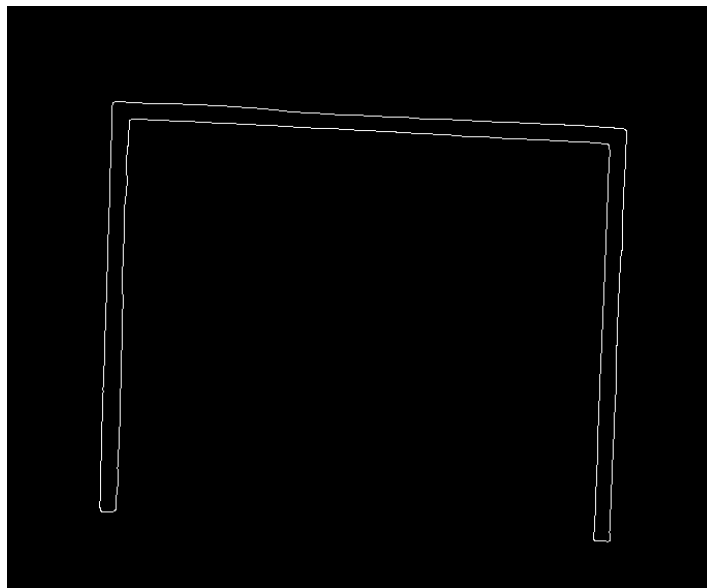


Figure 31: MATLAB Image Processing Toolbox Canny Implementation

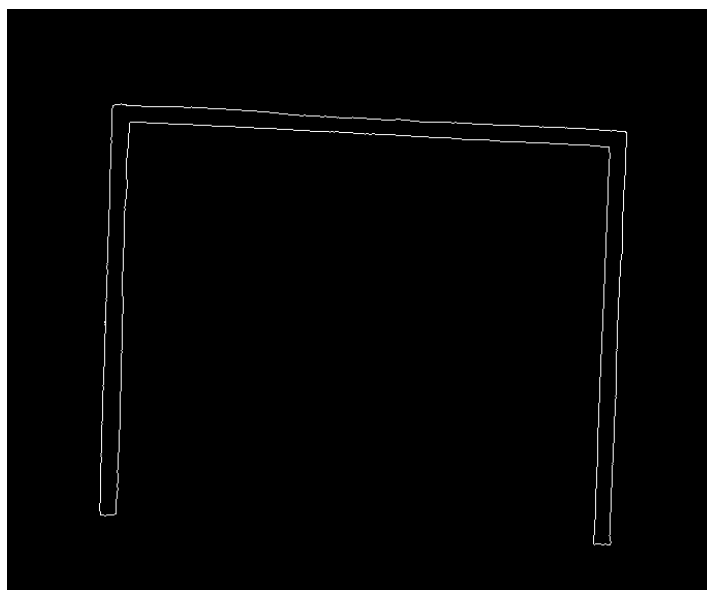


Figure 32: Python OpenCV Canny Implementation

The results show that with similar tuned threshold variables, the Python OpenCV implementation provides a complete edge detection, similar to that of the MATLAB implementation.

7.3 Mapping

What I want my system to provide is the start- and end coordinates (x,y) of each wall segment detected with relation to where the system is positioned. Since I have not implemented any positioning, it is assumed that the system is in coordinates (0,0,90) where 90 is the theta angle, and all the walls are mapped relating to this position. This translates directly to the image center, since the image is captured normal to the ground plane.

There are a couple of factors I want to test with regards to the mapping implementation:

- Accuracy of mapping with accurate height measurement
- Accuracy of height measurement over the maze
- Accuracy of mapping with varying heights

Some of these elements were tested for in Bjørnsen 2016 [19], and I intend to compare how this implementation behaves compared to that done previously.

7.3.1 Setup

The setup in the different mapping test uses the test maze described previously, in addition to other elements. I have rigged the Raspberry Pi on a tripod facing normally to the ground plane. The power and ethernet cable is connected to the Raspberry Pi running up the tripod, and the Raspberry Pi is controlled remotely through the Remote Desktop Connection application in Windows as described earlier.

With this setup, many different tests can be conducted for testing the varying system pieces. The setup is built to replicate the state of which the system might find itself being mounted on a drone flying above a maze.

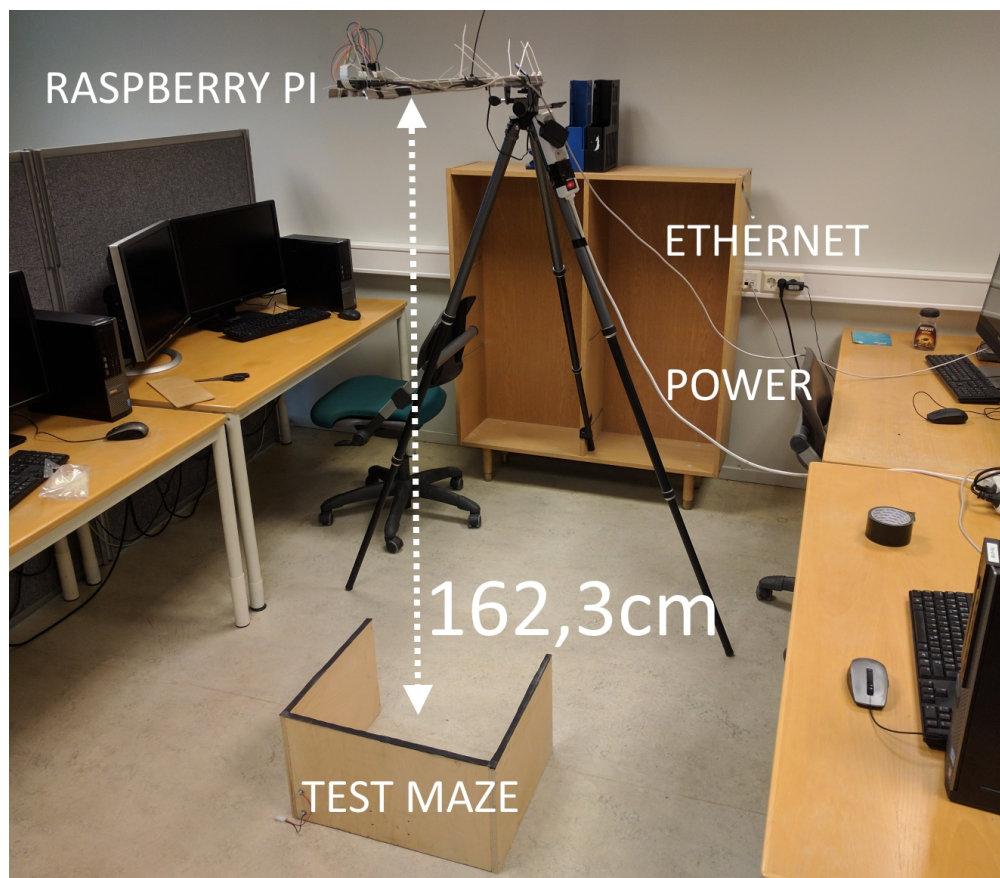


Figure 33: Tripod setup used for testing



Figure 34: Bulls eye spirit level on the tripod

7.3.2 Accuracy of mapping with accurate height measurement

I want to test how accurate the system can map and calculate the wall lengths using a very accurate height measurement. Using the setup described previously, and omitting the measured height from the range sensor, and feeding an accurate height in the program, I can compare the calculated wall lengths with the actual wall lengths.

Since the range sensor has some inaccuracies, I measure the distance from the ground with a measuring tape. The measured height above the ground plane is:

$$Measured_height = 162,3 \text{ [cm]}$$

I have plotted the output of the algorithm when feeding the correct height over the image the algorithm captures:

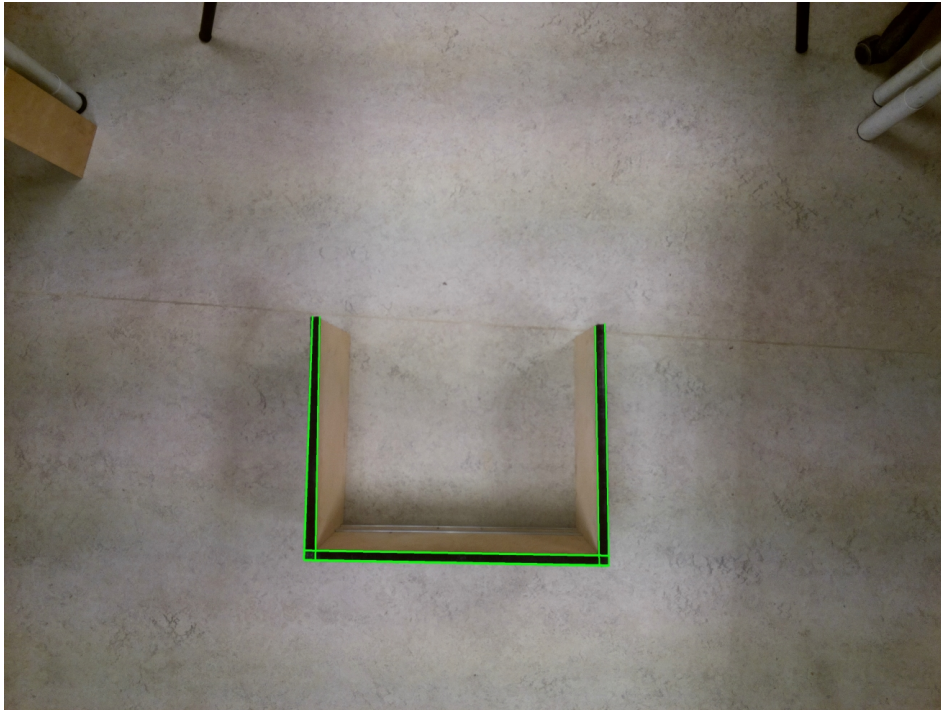


Figure 35: Output from algorithm

The algorithm also calculates the length of each detected wall segment and outputs

them in the terminal, giving the following values in cm:

[52.13614559709601, 51.84253572745204, 41.58182396984803,
41.454211626629274, 40.893884097286644, 52.13614559709601,
41.18835500372284, 51.989768062876635, 52.08792302779498,
41.482826496385684, 41.40344063095725]

A weakness in the algorithm, which I will explain later, is that the algorithm might detect the same wall several times, with slightly different length values. By sorting the values, by back wall and side walls, and taking the mean of each wall segment I get the following:

$$\text{Length Back Wall} = 52.04 \quad [cm]$$

$$\text{Length Side Wall} = 41.33 \quad [cm]$$

I have the actual measurements of the test maze:

$$\text{Actual Length Back Wall} = 53.1 \quad [cm]$$

$$\text{Actual Length Side Wall} = 42.5 \quad [cm]$$

Calculate the difference:

$$\Delta\text{Length Back Wall} = 53.1 - 52.04 = 1.06 \quad [cm]$$

$$\Delta\text{Length Side Wall} = 42.5 - 41.33 = 1.17 \quad [cm]$$

In percent the differences are in terms of the actual value of the wall:

$$\Delta\text{Length Back Wall in \%} = \frac{1.06}{53.1} = 0.0199 = 2\%$$

$$\Delta\text{Length Side Wall in \%} = \frac{1.17}{42.5} = 0.0275 = 2.75\%$$

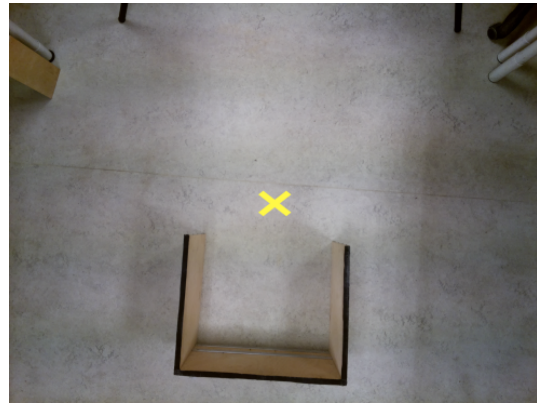
7.3.3 Accuracy of height measurement over the maze

In the last test, I measured the height accurately using a measuring tape. In this test, I want to examine if the range sensor has its performance affected by the maze itself. By comparing the height measurement with and without the maze I can examine if the maze introduces any error.

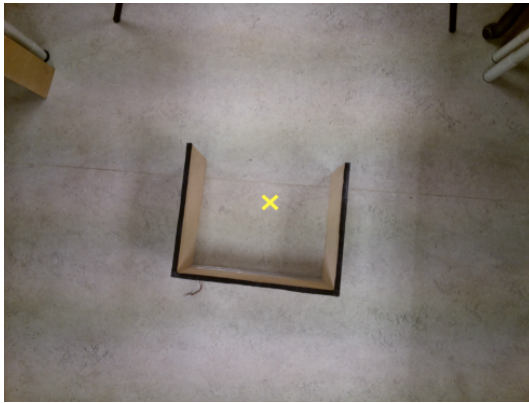
The test will consist of taking different pictures with the maze in different locations in relation to the image center, and comparing the resulting height measurement.



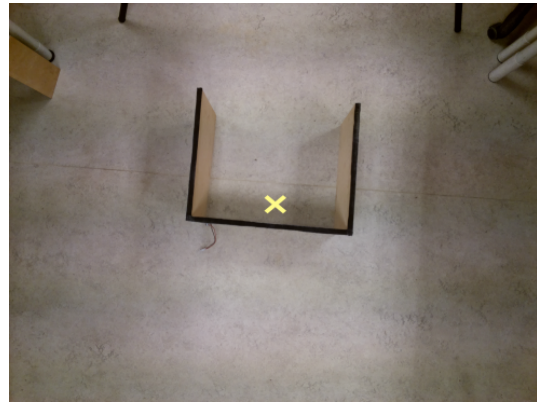
(a) Image 1



(b) Image 2



(c) Image 3



(d) Image 4

Figure 36: Height tests images with the maze at different locations

In figure 36, I have captured four different images with the maze in different locations.

Image (a) is without the maze itself and the others are with the maze. The center of the image is marked with a yellow X to see what is directly below the range sensor.

The results are the following:

$$\text{Measured Height Image 1} = 160,37 \quad [cm]$$

$$\text{Measured Height Image 2} = 160,82 \quad [cm]$$

$$\text{Measured Height Image 3} = 154,90 \quad [cm]$$

$$\text{Measured Height Image 4} = 133,90 \quad [cm]$$

The height of the wall itself is:

$$\text{Height of Wall} = 30 \quad [cm]$$

From the results one can see that the height measurement changes depending on where the maze is located under the range sensor. The trend is shows that the closer the range sensor is to the actual top of the wall, the measured height will decrease. The most interesting measurement is in Image 3, where the sensor is in the middle of the maze, which will be normal in a more complex maze. The measurement shows a decrease in measured height when in the middle of the maze.

In Image 3, the measured height is close to:

$$\text{Actual height - Height of wall} = 162,3 - 30 = 132,3 \quad [cm]$$

$$\text{Measured Height Image 4} = 133,90 \quad [cm]$$

This means the range sensor most likely detects the top of the wall in image 4.

7.4 Communication test

The protocol was explained earlier in Section 6.6.1, and in this section I want to test if the communication system is working; that is, when the system has detected the actual edge segments, the update messages containing the mapping data is received in the other end of the SSH link.

The steps is as follows:

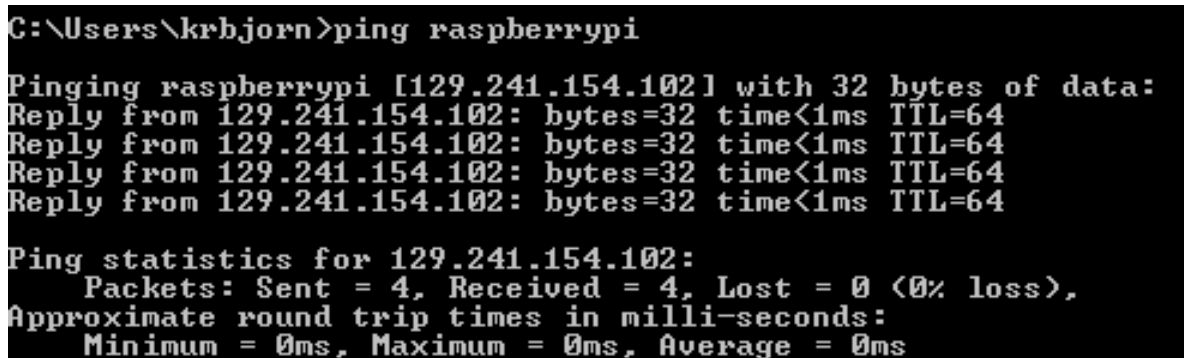
1. Ping `raspberrypi` in Windows shell to get the IP address to the Raspberry Pi
2. Connect to the Raspberry Pi through SSH (PuTTY)
3. Run the software on the Raspberry Pi
4. Examine the output in the SSH client

7.4.1 Ping Rapsberry Pi

Like described in section 5.4.1, a solution to finding the Raspberry Pi IP address in headless mode is by pinging the Raspberry Pi in the Windows shell:

```
ping raspberrypi
```

This outputs the Raspberry Pi IP Address:



```
C:\Users\krbjorn>ping raspberrypi

Pinging raspberrypi [129.241.154.102] with 32 bytes of data:
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64
Reply from 129.241.154.102: bytes=32 time<1ms TTL=64

Ping statistics for 129.241.154.102:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Figure 37: Raspberry Pi IP

Now I have the IP: 129.241.154.102 which I can use in the SSH client.

7.4.2 Connect through SSH (PuTTY)

Connecting to the Raspberry Pi in PuTTY is explained in section 6.6. By entering the correct IP address in the PuTTY client, I get prompted for the username and password, which is the following:

Username	Password
pi	master2017

Table 5: Username and Password

After entering the username and password, the following window appears:

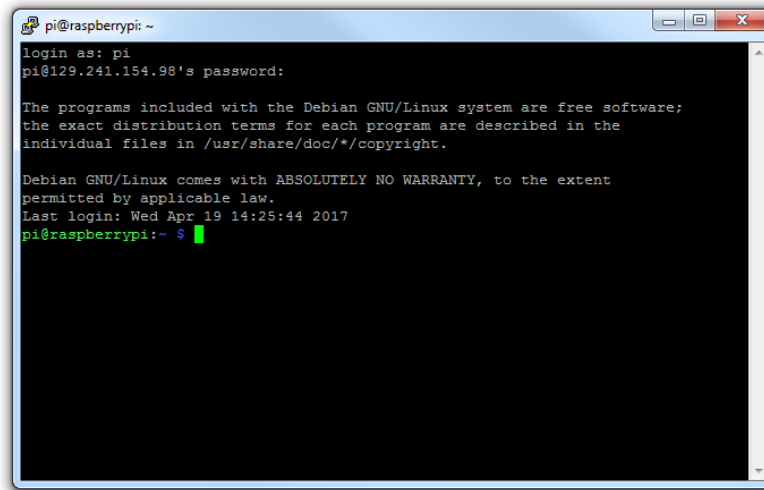


Figure 38: PuTTY terminal in Windows

This indicated I am connected to the Raspberry Pi through SSH.

7.4.3 Run the software on the Raspberry Pi

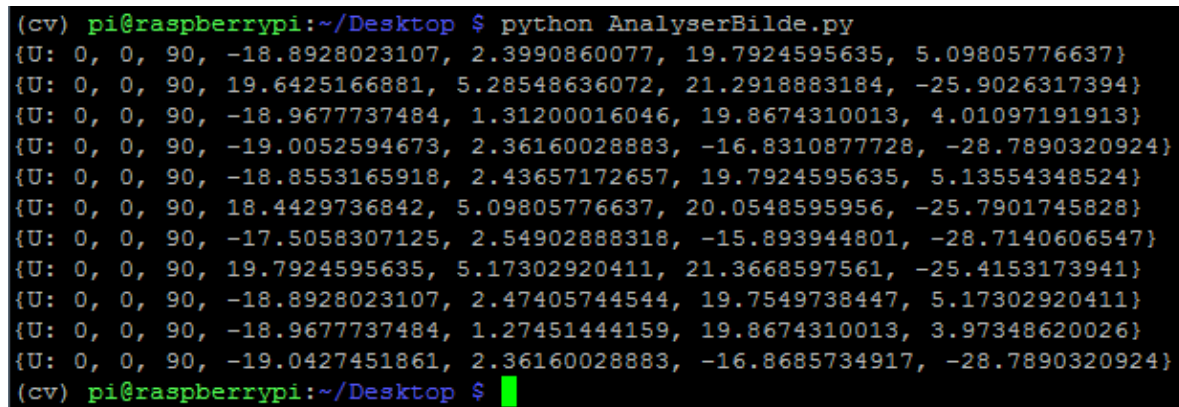
Like described earlier, I have to work in the virtual environment (cv) since this is where the OpenCV library is installed. Then set the directory to the location of the `AnalyserBilde.py` software is located and run it through Python.

```
pi@raspberrypi:~ $ source ~/.profile
pi@raspberrypi:~ $ workon cv
(cv) pi@raspberrypi:~ $ cd Desktop
(cv) pi@raspberrypi:~/Desktop $ python AnalyserBilde.py
```

This will run the mapping algorithm on the Raspberry Pi.

7.4.4 Output

Examining the output in the SSH client after running the mapping algorithm gives the following output:



```
(cv) pi@raspberrypi:~/Desktop $ python AnalyserBilde.py
{U: 0, 0, 90, -18.8928023107, 2.3990860077, 19.7924595635, 5.09805776637}
{U: 0, 0, 90, 19.6425166881, 5.28548636072, 21.2918883184, -25.9026317394}
{U: 0, 0, 90, -18.9677737484, 1.31200016046, 19.8674310013, 4.01097191913}
{U: 0, 0, 90, -19.0052594673, 2.36160028883, -16.8310877728, -28.7890320924}
{U: 0, 0, 90, -18.8553165918, 2.43657172657, 19.7924595635, 5.13554348524}
{U: 0, 0, 90, 18.4429736842, 5.09805776637, 20.0548595956, -25.7901745828}
{U: 0, 0, 90, -17.5058307125, 2.54902888318, -15.893944801, -28.7140606547}
{U: 0, 0, 90, 19.7924595635, 5.17302920411, 21.3668597561, -25.4153173941}
{U: 0, 0, 90, -18.8928023107, 2.47405744544, 19.7549738447, 5.17302920411}
{U: 0, 0, 90, -18.9677737484, 1.27451444159, 19.8674310013, 3.97348620026}
{U: 0, 0, 90, -19.0427451861, 2.36160028883, -16.8685734917, -28.7890320924}
(cv) pi@raspberrypi:~/Desktop $
```

Figure 39: Output from the mapping algorithm

This output displays the detected wall coordinates in the correct predefined communications protocol.

8 Sources of Error

The previous section conducted several experiments to examine the system in terms of accuracy, and to compliment this it is beneficial to look at the potential sources of error in the system. I will focus this section on the topics examined in the previous section, and look at potential sources that might introduce errors in the measurement and mapping.

8.1 Inaccurate height measurement

One of the biggest influences of the system accuracy is the height measurement from the drone to the ground. The actual mapping of the respective walls is done from a calculation derived from the measured height. There are different factors that affect the accuracy of the height measurement.

8.1.1 Range sensor gives wrong values

The range sensor tests showed that the range sensor does not always give accurate range estimates in the application. This can be both hardware and software related, but most likely it is due to hardware. Since I am so dependent on a good measurement of the distance from the drone to the ground, this source of error is very critical.

Potential solutions:

- More expensive sensor
- Implementation of a different range sensor technology like LIDAR
- Better tuning and steady-state offset correction

8.1.2 Difference in height from camera to height sensor

In the hardware system setup I have made, there is a small height difference between the range sensor and the image sensor in the camera is illustrated in Figure 40. This might introduce a small error in the height measurement, but is easy to fix.

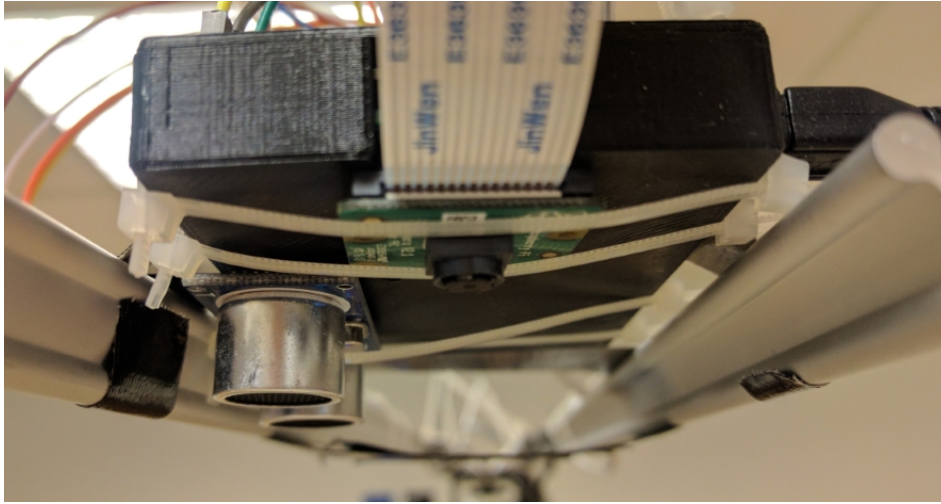


Figure 40: Difference in height between range sensor and image sensor

Potential solutions:

- Mount the range sensor and image sensor in the same plane
- Measure the difference and correct for it in the software

8.2 Incomplete mapping of wall segments

Perhaps the most critical aspects of the algorithm is the ability to detect all wall segments in the image. Bjørnsen 2016 [19] showed that the Hough transform might not always give complete wall segment mapping, meaning the algorithm think that the detected wall is shorter (or longer) than what it really is.

Examining the an image taken by the Raspberry Pi with the detected wall segment lines drawn over the original image:

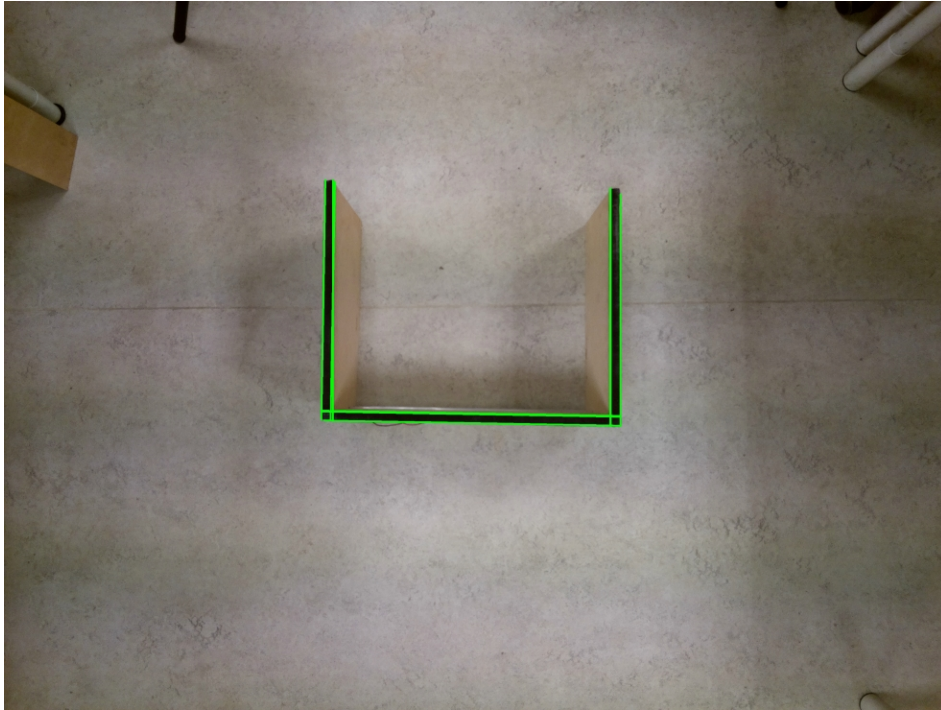
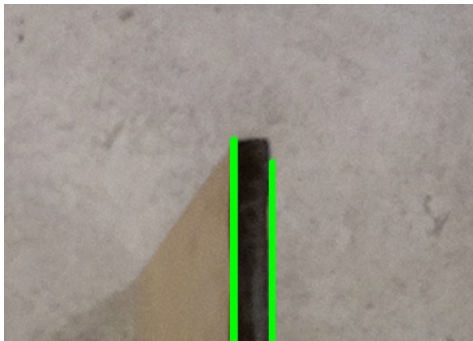
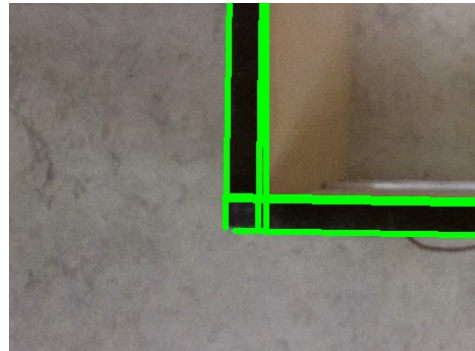


Figure 41: Hough lines drawn over original image

By examining the end points of some of the lines, the same shortening or lengthening of the detected wall segments can be found:



(a) Length of wall error



(b) Length of wall error

Most of these issues stems from the probabilistic Hough transform function run in the implementation of the mapping algorithm.

Potential solution:

- Better tuning of Hough variables
- Tuning to a specific height the images are taken from

8.3 Detection of the same wall segment more than once

An issue that is more prevalent in the Python and OpenCV implementation of the mapping algorithm is the detection of the same wall segment several times. The implementation of the Hough transform in the Image Processing Toolbox in MATLAB has built-in functionality that can eliminate detected edge segments that are close together. This functionality is not the same in OpenCV, and as such, some wall segments will be detected more than once.

The following image illustrates this:

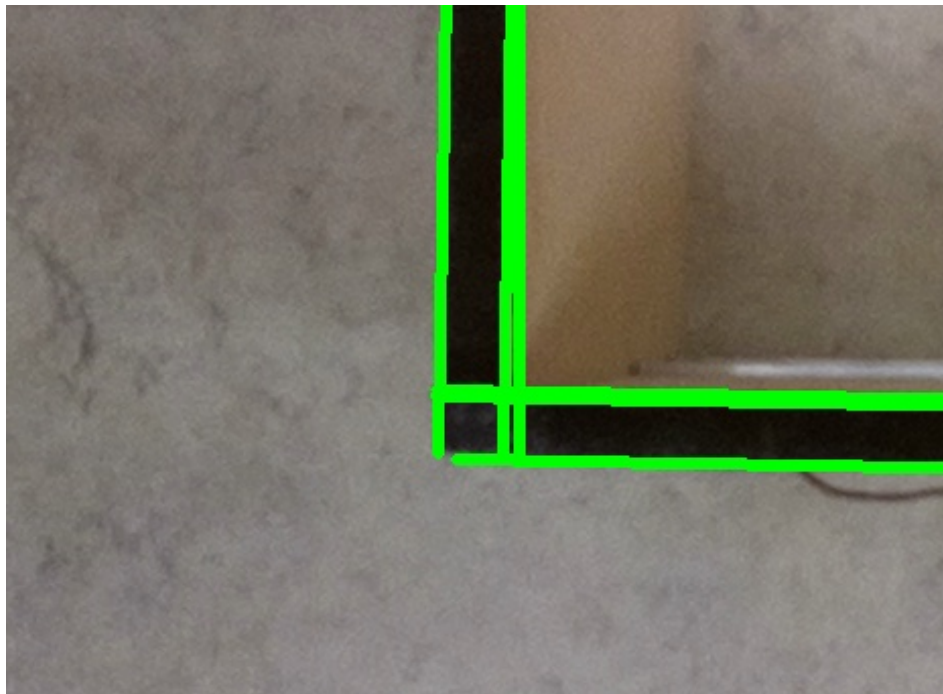


Figure 43: Two wall segments detected

The wall going up in the image has on the right side two detected edge segments on the same wall. This might be an issue since the Raspberry Pi has to use more processing power since it cannot omit lines on top of each other.

Potential solutions:

- Implementation of the elimination of wall segments with very close start and end point coordinates

8.4 Communication failure

Although unlikely, there is a chance that the communication link between the Raspberry Pi and the terminal might fail. This issue is almost non-existent in the SSH implementation of the communication between the Raspberry Pi and the terminal, since it is done as a wired connection. Bluetooth however, might introduce an increased chance of communication failure in the system, and should be accounted for in the implementation.

Potential Solutions:

- Robust implementation of the communication software
- Introduce an optional wireless communication channel such as Wi-Fi
- Implement error handling in the software

9 Discussion

This section will attempt to discuss the findings in the tests and the general experience with developing and implementing the mapping algorithm on a Raspberry Pi with Python and OpenCV. Many of the benefits introduced by using a camera in the identification and characterization of the maze has been thoroughly explained in Bjørnsen 2016 [19]. I will focus the discussion more in terms of a comparison between the implementations done previously in MATLAB and now in Python, rather than restating the findings in Bjørnsen 2016 [19].

9.1 Findings already established

The following has been established in terms of the mapping system [19]:

- The mapping algorithm improves the characterization and mapping of the maze over the traditional LEGO robots
- The use of a camera improves the detection of whole wall segments
- The system is more accurate than the LEGO robots

9.2 Hardware Improvements

In Bjørnsen 2016 [19], the hardware used was a lot different than that of the current implementation. For starters the previous solution did not feature any communication or integration between the camera and the image processing algorithm. The user would have to capture a picture with a mobile camera, measure the distance to the ground manually, upload it to MATLAB, and then run the algorithm.

The current solution features everything the system needs to be run remotely and the internal communication is automated. That means, that the system is able to measure the variables it needs for mapping on the platform itself (excluding positioning).

The biggest difference in terms of hardware is that the current solution offers a complete system, while the previous implementation required a lot of human interaction for it to run the mapping algorithm.

Bjørnsen 2017	Bjørnsen 2016
Range sensor implemented	Measure height by hand
Camera mounted on hardware	Capture with mobile camera
Ethernet and Bluetooth LE enabled	No communication implemented
Dedicated hardware for image processing	Run on multi-purpose PC
Mobile	Not mobile

Table 6: Hardware differences between the two implementations

So in terms of attempting to reach the project vision of a mobile image processing platform being mounted on a drone, the current implementation is a step in the right direction.

9.3 Software Improvements

The main difference between the implementation in this thesis and Bjørnsen 2016 [19] is that the software is developed in Python by using OpenCV and other libraries used for camera access and I/O access, as opposed to using MATLAB and its built in library, the Image Processing Toolbox.

In developing the MATLAB implementation, I focused on the feasibility of the algorithm itself; to see if it was possible to obtain an accurate mapping of a maze using well-researched image processing tools. With this new Python implementation, the focus has shifted more to the integration of the pieces needed to create a complete mapping system. This means that the Python implementation contains more functionality and is generally a more complex and integrated software.

9.4 Accuracy

Accuracy is an important aspect of the implementation, and has been a focus throughout the whole implementation. It is interesting to compare the accuracy of the new and old implementation, to see if the introduction of more sensors and a more mobile and complete platform affects accuracy.

9.4.1 Length of Walls

The same test as in Section 7.3.2 was done in Bjørnsen 2016 [19]. In that implementation, the results was:

$$\Delta\text{Length Left Wall Bjørnsen 2016} = 42,2123 - 42,5 = -0,2887 \quad [cm]$$

$$\Delta\text{Length Back Wall Bjørnsen 2016} = 53,5241 - 53,1 = 0,4241 \quad [cm]$$

$$\Delta\text{Length Right Wall Bjørnsen 2016} = 43,0162 - 42,9 = 0,1162 \quad [cm]$$

In the test run in Section 7.3.2 I had the following results:

$$\Delta\text{Length Back Wall} = 53.1 - 52.04 = 1.06 \quad [cm]$$

$$\Delta\text{Length Side Wall} = 42.5 - 41.33 = 1.17 \quad [cm]$$

Bear in mind that the side walls in the Bjørnsen 2016 [19] was of different lengths, while the side walls in the newest test was equal. This test was done with an accurate height measurement where the height was measured by a measuring tape and fed into the GSD calculation manually. The results show that the accuracy of the detected length of the wall segments has decreased. In terms of percentages the accuracy of the Bjørnsen 2016 [19] is:

$$\% \text{ Accuracy Length Left Wall Bjørnsen 2016} = \frac{0.2887}{42.5} = 0.6\%$$

$$\% \text{ Accuracy Length Back Wall Bjørnsen 2016} = \frac{0.4241}{53.1} = 0.8\%$$

$$\% \text{ Accuracy Length Right Wall Bjørnsen 2016} = \frac{0.1162}{42.9} = 0.3\%$$

While in the Section 7.3.2 test, the accuracy was found to be:

$$\begin{aligned}\% \text{ Accuracy Length Back Wall} &= \frac{1.06}{53.1} = 0.0199 = 2\% \\ \% \text{ Accuracy Length Side Wall} &= \frac{1.17}{42.5} = 0.0275 = 2.75\%\end{aligned}$$

From the results it can be observed that the newest implementation introduces a larger percentwise difference in the measured length of walls compared to the actual length. Possible reasons for this is explored in Section 8, but more specifically, I believe the reason for this decreased accuracy can be attributed to:

- Introduction of a new camera, less expensive than the IMX377
- Potential wrong tuning parameters in the GSD calculation
- Introduction of more sensors will always decrease accuracy compared to setting the variables constant

There needs to be more work done on the tuning and implementation of the GSD and height calculation in the future to make this measurement more accurate.

9.4.2 Height measurement

The test in section 7.3.3 tested if the maze itself affected the measured height from ground to the system, and it did indeed have a big effect on the measured height. Since the system is so dependent on a good measurement of the distance between the system and the ground, this decreased accuracy suggest that the use of an ultrasonic range sensor might not be the best option for this application.

Considerations should be made in determining if a better sensor solution is possible to find a better estimate for the height. Sensor such as LIDAR work with light instead of sound, and might improve the accuracy.

To summarize, the height measurement error introduced in this implementation might introduce unwanted margins of error in the calculations that will affect the mapping.

9.4.3 Hough Transform

A different factor that can affect the accuracy of the mapping is the moments showed in section 8.2 and 8.3, where I had some errors in terms of the complete mapping of the maze. The reason for most of these errors can be attributed to the Hough transformation. By having badly tuned variables in the probabilistic Hough transformation, errors such as the walls not being completely detected can occur, and this was also showed in Bjørnsen 2016 [19].

The only added inaccuracy in this thesis is the fact that the walls may be detected more than once. As explained earlier, this is an error inherent in the OpenCV implementation of the Hough transform, where it is less complex than the equivalent in MATLAB. To solve this, some functionality that removes walls that fall on top of each other should be implemented.

10 Conclusion

A complete mapping system capable of receiving commands, capturing images, running the mapping algorithm internally, and communicating the results back has been implemented in this paper.

Many steps towards the project vision of a mobile mapping platform has been made, but there are still remaining factors that need to be solved before the system can be truly mobile, and capable of being mounted on a drone.

The current implementation of all the needed sensors for an accurate mapping introduces a slight decrease in accuracy in the mapping, but this issue is inherent in the use of cheap sensors such as the distance sensor. The accuracy can be improved either through more accurate sensors or better tuning of the mapping parameters.

The translation of the image processing and mapping algorithm from MATLAB to Python has added a small increase in error, but it is probable that this can be solved through implementation. The usage of Python has unlocked more potential in the software, where sensor integration, computation and communication can be handled in one single program written in Python.

The use of an image sensor in a camera still has a built in benefit of detecting straight line segments, which is the biggest improvement factor over the LEGO robots, which can register straight lines as curved.

More work needs to be done in terms of mobility and hardware integration to reach the project vision of a truly mobile mapping platform able to be mounted on UAV's.

11 Future Work

There are many aspects of the final project vision that has not been implemented in this paper, and which opens up possibilities for many future student papers. Below are some of the suggested improvements and expansions to the project.

11.1 Bluetooth Communication

The ideal communication platform for the application is Bluetooth Low Energy (BLE), and is currently used by the existing LEGO robot implementation. The existing LEGO solution uses an Nordic Semiconductor nRF51 Dongle [30] on each mapping robot and on the server itself. This has facilitated for an easy implementation of the respective robots, since the nRF51 is communicating with the same hardware. The Raspberry Pi has a different BLE chip, and thus the implementation of the communication was more difficult to make work.

11.1.1 Technical Issue

Since the software written on the communication in the existing LEGO robots uses a Nordic Semiconductor BLE chip, a special Nordic Semiconductor UUID is hard-coded into the server software. The server application only detects the special UUID, which is different from the Raspberry Pi UUID. This means to solve this issue, a student must invest time in both the existing server implementation and the Raspberry Pi implementation.

11.2 Mobile Power Supply

In order to make the system completely mobile, then a power supply locally on a drone should be implemented. By using a drone battery to power the Raspberry Pi the system will be able to be operated mounted on a drone.

- Implement a battery powered Raspberry Pi
- Make the system completely mobile

- Can the image processing still be done locally on the Pi?

11.3 Develop a Drone

Perhaps the most exciting part of the system will be to develop a drone suitable for the system. Things to keep in mind:

- Size
- Indoor use
- Positioning of the drone (very important and difficult)
- Autopilot
- Autonomous?

I would recommend the purchase of an off-the-shelf drone, as well as an off-the-shelf autopilot with an open source-code.

11.4 Implementation of Positioning

This is closely tied with the development of the drone, but this will require the student to implement positioning on both the drone hardware and in the Raspberry Pi software code.

- Hardest problem: Indoor positioning
- Command structure (protocol)
- Merge with the LEGO robot server

The topic of positioning is an exciting one, and needs to be solved before the system can be operational.

12 References

- [1] Gonzalez, R. C., & Woods, R. E. (2010). *Digital Image Processing* 3rd Edition. Pearson Education, Inc.
- [2] Leachtenauer, Jon C. & Driggers, Ronald. (2000). *Surveillance and Reconnaissance Imaging Systems: Modeling and Performance Prediction*
- [3] Marr, D., & Hildreth, E. (1980). *Theory of Edge Detection. Proceedings of the Royal Society of London. Series B, Biological Sciences*, 207(1167), 187-217. Retrieved from <http://www.jstor.org/stable/35407>
- [4] Image Sensor Business Division, DSBG. *Sony IMX377CQT Product Summary, Ver. 1.5* Retrieved from http://www.sony-semicon.co.jp/products_en/IS/sensor2/products/imx377.html
- [5] Hough, P. V. C. (1962). *Methods and Means for Recognizing Complex Patterns* U.S. Patent 3,069,654.
- [6] J. Heikkila & O. Silven, *A four-step camera calibration procedure with implicit image correction*. Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Juan, 1997, pp. 1106-1112. doi: 10.1109/CVPR.1997.609468
- [7] Ese, E., *Sanntidsprogrammering på samarbeidande mobil-robotar*. Master 2016
- [8] Ese, E., *Fjernstyring av legorobot*. Prosjekt 2015
- [9] MathWorks. Library of built in functions in Matlab, and their description. <https://se.mathworks.com>
- [10] Tøraasen, J. K *Kartlegging ved hjelp av robot og kamerasensor*. Prosjekt 2009
- [11] Fisher, R. *Image Processing Learning Resources - Laplacian of Gaussian*. Retrieved from <http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm>

- [12] Pix4D. *GSD Calculator*. Retrieved from <https://support.pix4d.com/hc/en-us/articles/202560249-TOOLS-GSD-Calculator>
- [13] NVIDIA. *Tesla Motors' Self-Driving Car "Supercomputer" Powered by NVIDIA DRIVE PX 2 Technology* <https://blogs.nvidia.com/blog/2016/10/20/tesla-motors-self-driving/>
- [14] The Makers Workbench. <http://themakersworkbench.com/>
- [15] HyperPhysics Concepts - Georgia State University. *Speed of Sound* <http://hyperphysics.phy-astr.gsu.edu/hbase/Sound/souspe.html>
- [16] Raspberry Pi Foundation. *Raspberry Pi 3 Model B* <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>
- [17] Raspberry Pi Foundation. *Raspberry Pi Camera Module* <https://www.raspberrypi.org/documentation/hardware/camera/>
- [18] Canny, J. "A Computational Approach to Edge Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 6, pp. 679-698, Nov. 1986. doi: 10.1109/TPAMI.1986.4767851
- [19] Bjørnsen, K. *Mapping a maze with an image sensor*. Prosjekt 2016
- [20] Element 14 Community *Raspberry Pi 3 Model B Technical Specifications* <https://www.element14.com/community/docs/DOC-80899/1/raspberry-pi-3-model-b-technical-specifications>
- [21] Win32DiskImager <https://launchpad.net/win32-image-writer>
- [22] Raspberry Pi Foundation. *Operating Systems Downloads* <https://www.raspberrypi.org/downloads/>
- [23] Summerfield, Mark. (2007) *Rapid Gui Programming with Python and Qt: The Definitive Guide to Pyqt Programming (First ed.)*. Prentice Hall Press, Upper Saddle River, NJ, USA.

- [24] OpenCV team. *OpenCV* <http://opencv.org/>
- [25] PyImageSearch *Install guide: Raspberry Pi 3 + Raspbian Jessie + OpenCV 3.* <http://www.pyimagesearch.com/2016/04/18/install-guide-raspberry-pi-3-raspbian-jessie-opencv-3/>
- [26] Thingiverse *Raspberry Pi 3 Slim Case with Heatsink Cutout.* <http://www.thingiverse.com/thing:1415895>
- [27] PiCamera API. <https://picamera.readthedocs.io/en/release-1.13/>
- [28] ModMyPi. *HC-SR04 Ultrasonic Range Sensor on the Raspberry Pi* <https://www.modmypi.com/blog/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi>
- [29] PuTTY: a free SSH and Telnet client. <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>
- [30] Nordic Semiconductor *Bluetooth Low Energy nRF51 Dongle* <https://www.nordicsemi.com/eng/Products/nRF51-Dongle>
- [31] Siciliano, Bruno, Khatib, Oussama. (2008). *Springer Handbook of Robotics*

A

A.1 Figures

This folder contains all the figures used in the report.

A.2 Test Images

This folder contains some of the test images used in this report when testing the range sensor and edge detection.

A.3 Software

This folder includes the following software:

- `AnalyserBilde.py` This is the complete implementation of the image processing, mapping and communication system presented in this report.
- `AnalyserBilde.m` This is the Bjørnsen 2016 [19] implementation of the image processing and mapping algorithm.
- `cameratest.py` This is the software run to test the camera module on the Raspberry Pi
- `edgetest.py` This is the software run to test the edge detection method implemented in the complete system.
- `rangetest.py` This is the software run to test the range sensor.

A.4 Previous Reports

This section includes the previous reports used in this thesis, which at the moment is only Bjørnsen 2016 [19].

B

B.1 Hardware included

The following hardware is included for the use in future projects:

- Raspberry Pi 3 Model B
- Raspberry Pi Camera Module v2
- Raspberry Pi Power Supply
- 16gb Memory Card
- HC-SR04 Range Sensor with Voltage divider
- Ethernet cable
- 3D printed case