Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Chris André Brombach

Time-of-Flight (TOF) depth camera for Navigation and Mapping

Master's thesis in Cybernetic and Robotics
Supervisor: Professor Tor Arne Johansen,
July 2019

NTNU
Norwegian University of
Science and Technology

Forord

Denne oppgaven er en masteroppgave skrevet i forbindelse med et 5 årig studium i kybernetikk og robotikk ved Norges Teknisk-Naturvitenskaplige Universitet (NTNU). Oppgaven er skrevet våren 2019 og danner det avsluttende arbeidet i studiet, og innenfor spesialiseringa autonome systemer.

Oppgaven er skrevet i samarbeid med Scout Drone Inspection, et oppstartsfirma fra NTNU, som spesialiserer seg innen autonome droner for innendørs inspeksjon av industrianlegg. Arbeidet gjort i denne oppgaven er ikke direkte knyttet til det Scout DI jobber med, men de har av interesse å se mulighetene Time of flight kamera kan ha for innendørs navigering og kartlegging. Til tross for dette så har det i stor grad vært selvstendig arbeid uavhengig av Scout DI. Det Scout DI og NTNU bidratt med er de nødvendige sensorene brukt i denne oppgaven og i prosjektoppgaven høsten 2018. Dr. Kristian Klausen (CTO ved Scout DI) har også vært tilgjenglig som rådgiver, og har gitt generelle råd om hvilken retning oppgaven burde ta, og hvordan jeg burde gå frem.

Denne oppgaven har gitt meg mulighet til å ta et enda dypere dykk inn i datasynverdenen, ett av fagområdene innenfor kybernetikken som jeg finner svært spennende. Jeg har også blitt bedre kjent med ROS som verktøy i arbeidet med å utvikle programvare til autonome systemer.

Jeg vil takke hovedveilederen min Tor Arne Johansen, samt koveileder Kristan Klausen for å være tålmodige og støttende når det dukket opp utfordringer underveis. Jeg vil også takke Ascend NTNU og alle de ressursene som jeg hadde tilgjengelig der.

Abstract

This master thesis examines the use of time of flight(ToF) camera as a visual sensor for problems such as localization, mapping, collision avoidance and other relevant computer vision tasks. The thesis starts by introducing the concept of time of flight, and aims to provide a good understanding of the behavior of the camera. Then we shed some light on traditional computer vision and SLAM with the use of ToF. The thesis mainly looks at the use of indirect SLAM, and is tested with ORB SLAM.

A mobile robot is designed for testing a holistic system. The electronic components and sensors are discussed as well as their role for the robot and the system as a whole. Furthermore, a simple simulated model is also made in the gazebo, where both robot and camera are simulated and tested.

The system as a whole is discussed, and we discuss how the various modules in the software work and how they work together. The main focus has been on the use of ORB SLAM with ToF camera, but we also look at collision avoidance and tracking of a predefined pattern in the image. Finally a set of tests is carried out on the various systems implemented, and a discussion of these.

The conclusion is that the time of flight camera is a worthy sensor for the use of navigation and collision avoidance, given some prerequisites. The camera has some limitations in terms of range and performance, and it is influenced by the environment and the features in it. If certain requirements are met it seems that time of flight cameras are a good alternative as an active visual sensor.

Sammendrag

Denne masteroppgaven ser på bruken av time of flight(ToF) kamera som en visuell sensor til problematikk som lokalisering, kartlegging, kollisjonsunngåelse og andre aktulle datasynsoppgaver. Den starter med en introduksjon til konseptet time of flight, og har som mål å gi en god forståelse av virkemåten til kameraet. Videre så settes dette i lys av tradisjonell datasyn og SLAM. Oppgaven ser i hovedsak på bruken av indirekte SLAM, og er testet med ORB SLAM.

En mobil robot blir konstruert for testing av et helverdig system. De elektroniske komponentene og sensorene blir diskutert, og hvilken rolle de har for roboten og systemet som helhet. Videre blir det også laget en enkel simulert model i gazebo, hvor både robot og kamera blir simulert og testet.

Systemet som helhet blir diskutert, og vi ser på hvordan de ulike modulene i programvaren fungerer og hvordan de virker sammen. Hovedfokuset har vært på bruken av ORB SLAM og ToF kamera, men vi ser også på kollisjonsunngåelse og sporing i bildet. Til slutt så ser vi på testene gjennomført på de ulike systemene som er blitt implementert, samt diskuterer disse.

Konklusjonen er at time of flight kamera er en verdig sensor til bruk av navigering og kollisjonsunngåelse, gitt noen forutsetninger. Kameraet har noen begrensinger med tanke på rekkevidde og ytelsen er preget av hvordan miljøet rundt er oppbygget. Dersom visse krav er oppylt så virker det som time of flight er et godt alternativ som en aktiv visuell sensors.

Contents

Chapter 1

Introduction

1.1 Problem formulation

The purpose of this thesis is to examine whether time of flight cameras is an ideal alternative as a visual depth sensor by constructing a mobile robot, use the visual data from the camera as the primary source for localization of the robot, mapping of the environment and obstacle detection. The scope of this text can be summarized in the following keypoints:

- Describe the principles of the depth camera, and the specification of the chosen sensor
- Build a simulated environment for a ground robot and test SLAM in this environment
- Design and construct a moving ground robot for experimental trails
- Demonstrate navigation in an unknown environment with the sensor
- Develop a system for obstacle avoidance and tracking.

1.2 Motivation

Even though time of flight cameras isn't a new invention, smaller more lightweight low power cameras has become available in the recent years, which could be ideal for mobile robots where weight and space is an issue that has to be considered. Given that this kind of sensors are able to decently localize and map the environment it could be a good alternative sensor.

Time of flight cameras are active sensors which make them able to work independently of ambient light, and they should have an advantage over standard cameras in enclosed and poorly lit environments.

This thesis is a continuation of the author's fall project where the goal was to argue whether ToF cameras could be used as a visual sensor for localization and mapping by comparing the performance with the performance of a active IR projecting stereo camera, and is discussed in some more detail in section 2.2.

1.3 Structure of thesis

Chapter 1

Chapter 1 forms the basis for the thesis by introducing the goal and motivation behind the project.

Chapter 2

In this chapter the reader is brought up to speed on the basic theory making the foundation of this thesis. Primarily it about the working principle of time of flight (ToF) cameras, computer vision in general and how it all fit together.

Chapter 3

This chapter contain an overview of the hardware used to construct the robot used in this project, along with some specifications and how the different components are connected and working together.

Chapter 4

A short introduction of the simulation environment gazebo is given, as well as the configuration of gazebo to simulate a robot in a simple environment with a depth camera.

Chapter 5

In chapter 5 the different software implementation on the robot is presented, the working principles and the communication flow between the different nodes in the software. The different parts of the software can be divided into motor and sensor control, motor controller, SLAM, obstacle detection and avoidance and target tracking.

Chapter 6

In this chapter the results of the implementations in simulator and on the robot is presented. This include ORB SLAM, obstacle avoidance and tracking.

Chapter 7

In the final chapter the results of the testing is discussed. Then further work is discussed based upon the results and possible improvements. The chapter ends with a short final conclusion.

Chapter 2

Background theory

In the following sections the background theory making the grounds for this thesis will be presented, with focus on the Time of flight sensor, and how it fits into the overall theory and thesis. In section 2.1 the working principle of time of flight cameras is presented. Section 2.3 presents some basic computer vision principles used in this project, and finally in section 2.4 VSLAM is discussed.

2.1 Time of flight principle

Time of flight (ToF) cameras are active sensors, that is, it uses light in the infrared spectrum to illuminate the environment, making it independent of the amount of ambient light in the environment. The main principle is to project the infrared light onto the scene and calculate the time it takes for the light to return, hence the name "time of flight". Time of flight systems use primarily two kinds of systems, either pulse modulation or continuous wave modulation. In this project continuous wave modulation is used, but a short introduction to pulse modulation will also be included.[1].

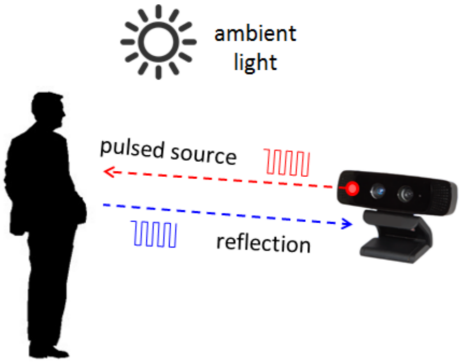


Figure 2.1: Basic principle of ToF cameras, [1]

2.1.1 ToF pulse modulation

Pulse modulation ToF systems solves the problem of finding the time of flight with the most obvious solution, i.e. send out a single pulse of IR-light and measure the time it takes to return. There are in general two methods to achieve this; one could start use a fast counter which is stopped when the first returning light is measured or by integrating photoelectric energy from the reflected light. Since the speed of light is ≈ 0.3 meters per nanosecond, the former case is highly dependent on really fast hardware if a high resolution depth is to be estimated. Thus the arrival time has to be detected very precisely, where just a small offset will have a major impact on the depth measurement. These kind of components are usually very expensive so the cheaper method is usually used, by integrating the photoelectric charge from the reflected light. [1][2].

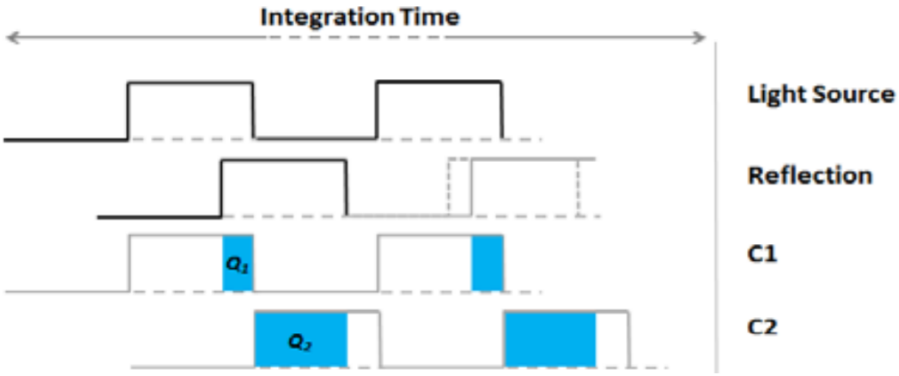


Figure 2.2: Pulse modulation, [1]

The method of integrating the photoelectric charge is as following. A single pulse of IR-light is sent out into the environment with a duration of Δt. The returning light is sampled in every pixel of the camera in parallel, by the use of two out of phase windows C_1 and C_2 as seen in fig. 2.2. Both of the windows has a time duration of Δt. The depth in every pixel is then calculated by eq. 2.1 where Q_1 and Q_2 is the accumulated photoelectric charge in each of the respective windows.

$$d = \frac{1}{2} c \Delta t \frac{Q_2}{Q_1 + Q_2} \tag{2.1}$$

2.1.2 Continuous wave (CW) modulation

As with the pulse modulation in the previous subsection continuous wave modulation make the use of the accumulated photoelectric change in different windows. In continuous wave modulation the method to find the depth in the environment is by measuring the phase shift of a modulated infrared signal. This signal is typically a sinusoid or a square wave. The infrared light is modulated according to eq 2.2, where A_E is the maximal amplitude of the signal and f_{mod} is the

modulation frequency. The signal strength is between $[0, A_E]$[2].

$$S_E(t) = A_E(1 + sin(2\pi f_{mod} t)) \tag{2.2}$$

The reflected signal is given by eq 2.3a where $\Delta\phi$ is the phase shift and the signal B_R is the interference from the ambient light in the infrared spectrum. By setting $B = B_R + A_R$ and $A = A_R$ we get the simplified eq. 2.3b. A is called the amplitude of the signal and B is the offset.

$$S_R(t) = A_R(1 + sin(2\pi f_{mod} t + \Delta\phi)) + B_R \tag{2.3a}$$

$$S_R(t) = Asin(2\pi f_{mod} t + \Delta\phi)) + B \tag{2.3b}$$

The next step is to to estimated the variables A, B and $\Delta\phi$ from equation 2.3b. This is done by sampling the signal of the reflected modulated in four windows. Each of the windows is out of phase by 90 degrees as shown in the table below.

Time	**Samples of S_R**
t = 0	$Q_0 = S_R^0(t)$
$t = \frac{1}{4f_{mod}}$	$Q_1 = S_R^1(t)$
$t = \frac{2}{4f_{mod}}$	$Q_2 = S_R^2(t)$
$t = \frac{3}{4f_{mod}}$	$Q_3 = S_R^3(t)$

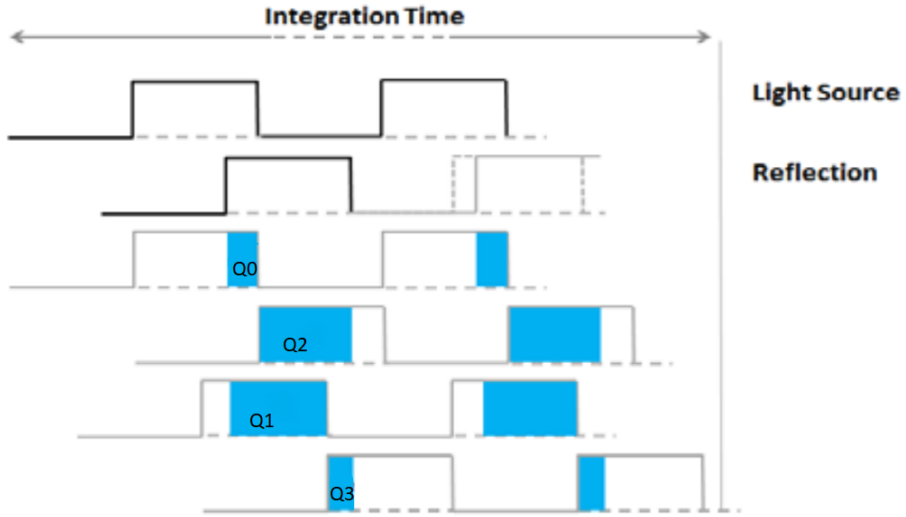


Figure 2.3: Wave modulation measurement principle, describing the windows and the integration time, [1]

The estimated variables $\hat{A}, \hat{B}, \hat{\Delta\phi}$ are then found by minimizing the square error between the measured charge in each window, and the value of $S_R^N, N \in [0, 3]$ for

each of the different windows given by eq. 2.4.

$$(\hat{A}, \hat{B}, \hat{\Delta\phi}) = argmin_{A,B,\Delta\phi} \sum_{n=0}^{3} (Q_n - Asin(\frac{\pi}{2}n + \Delta\phi) + B)^2 \tag{2.4}$$

The solution is then given by the following three equations [2]:

$$\hat{A} = \frac{\sqrt{(Q_0 - Q_2)^2 + (Q_1 - Q_3)^2}}{2} \tag{2.5a}$$

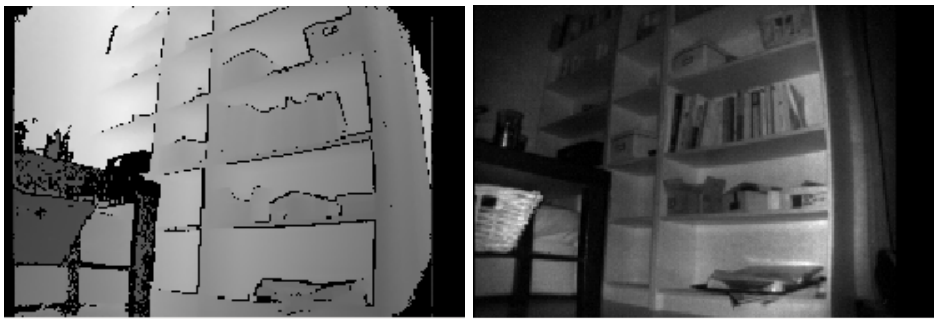
$$\hat{B} = \frac{Q_0 + Q_1 + Q_2 + Q_3}{4} \tag{2.5b}$$

$$\hat{\Delta\phi} = arctan2(Q_0 - Q_2, Q_1 - Q_3) \tag{2.5c}$$

Now that the estimated variables $\hat{A}, \hat{B}, \hat{\Delta\phi}$ is written as a function of the accumulated photoelectric charges Q_0, Q_1, Q_2 and Q_3, the distance can in each pixel can be found by equation 2.6, where c is the speed of light.

$$\hat{d} = \frac{c}{4\pi\ f_{mod}} \hat{\Delta\phi} \tag{2.6}$$

The output of the ToF camera is two or three pictures, where the pixel values is based upon the estimated variables $\hat{A}, \hat{B}, \hat{d}$. In most cases only the intensity image and depth image are used, based upon $\hat{A}$ and $\hat{d}$ respectively.



(a) Depth image (b) IR intensity image

Figure 2.4: Example of the depth and IR image from the Pico Flexx

According to [1] the depth measurement variance can be approximated by the Gaussian 2.7 where c_d is the modulation contrast which describe the sensors ability to collect and separate the photoelectric charges.

$$\sigma = \frac{c}{4\sqrt{2}\pi\ f_{mod}} \frac{\sqrt{A + B}}{c_d A} \tag{2.7}$$

As seen above in this and the previous subsection the continuous wave modulation and pulse modulation work by the same principle by measuring the accumulated

photoelectric charge from the incoming IR-light, then why use the more complex continuous wave modulation? The answer is quite clear when we look at the equations of the two. The pulse modulation will be directly effected by a constant effect from ambient light, while the CW modulation is immune to this effect. This is made clear by equations where a constant q_c photoelectic effect from ambient light is added.

$$d = \frac{1}{2}c\Delta t\frac{Q_2 + q_c}{2q_c + Q_1 + Q_2} \neq \frac{1}{2}c\Delta t\frac{Q_2}{Q_1 + Q_2} \tag{2.8}$$

$$\Delta\phi = arctan2((Q_0 + q_c) - (Q_2 + q_c), (Q_1 + q_c) - (Q_3 + q_c)) \tag{2.9a}$$

$$= arctan2(Q_0 - Q_2, Q_1 - Q_3) \tag{2.9b}$$

As seen in equation 2.8 the distance estimate is affected by the constant q_c while the CW modulation sum out to zero in equation 2.9, where the phase is calculated. Since the distance estimate is the product of constants and the phase, eq. 2.6, ambient light have a reduced effect on the CW modulation depth estimate.

While the ambient light don't have an effect on the phase estimate, it has an effect on the repeatability of the measurement, as seen in equation 2.7, which is a function of A and B. A is unaffected by q_c but B is not, and an increase in the ambient light will increase the variance of the depth measurement.

It is in our interest to minimize the variance of the depth measurement. The variables that has a direct effect on this is A, B, c_d and f_{mod}. The key takeaway is a small value of B, minimize ambient light, large values of A, c_d and f_{mod}. A is saturated by the light source of the ToF camera, a stronger light source could increase A.C_d is limited by the hardware and f_{mod} is limited by aliasing. Since the phase wraps around every 2π, i.e $sin(t) = sin(t + 2\pi)$, the distance will have an aliasing distance, called the ambiguity distance. This ambiguity distance is given by eq. 2.10. This is also the maximum distance the ToF cameras are able to measure. If the maximal measurement is to be increased, the modulation frequency has to be decreased, which will increase the variance of the measurements [1].

$$d_{amb} = \frac{c}{2f_{mod}} \tag{2.10}$$

To improve the depth estimate and reduce noise a common technique is to average the photoelectric charge in each window over several periods, called the integration time, and use these averages to calculate $\hat{A}, \hat{B}, \hat{\Delta\phi}$. While this reduce the amount of noise in the measurements, it has some side effects that must be considered. First of all, since the photoelectric charge is averaged over a given interval, usually between ¡1,100¿ milliseconds, the camera may move during the integration time. This may result in motion blur. Another side effect is the case of saturation, which happens when received quantity of IR-light exceeds the maximum capability of the hardware. The effect of saturation is mostly visible in

the case of high external infrared illumination, like from the sun, or from highly reflective surfaces. [2].

2.2 Earlier work

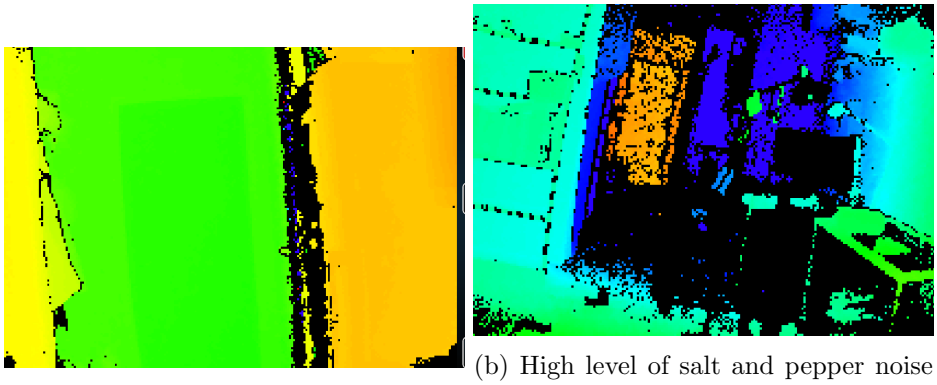
In an earlier project, fall of 2018 [3], a comparison between time of flight camera and active stereo camera was conducted. The purpose of the project was to argue whether time of flight cameras were an ideal sensor to the use of SLAM and obstacle avoidance, the scope if this paper. A series of small test was conducted to test the performance of the Pico Flexx ToF camera against the Intel D435 stereo camera.

The general findings were that the performance was somewhat similar between the two cameras, where the Intel D434 stereo camera had some longer range, while the accuracy was within 1%-2% of the distance to the object for both cameras, i.e. at distance of 2 meters the measurement would be within $\pm$ 2 - 4 centimeters of the actual distance.

Another finding was the range of the time of flight camera was highly dependent on the background and how reflective that and of the objects in the pictures were. As seen in the table below the maximal range is more than halved if the background is dark and little reflective. The different modes will be discussed further in section 3.3.2. However it is not a surprise that the range is reduced when less IR-light return to the sensor, but it is important to keep in mind during the test scenarios.

Mode	**Max range, reflective background**	**Max range, dark background**
1	6.9 meters	2.9 meters
2	5.5 meters	2.2 meters
3	4.5 meters	1.75 meters
4	3.5 meters	1.5 meters
5	2.4 meters	1.2 meters
6	2.3 meters	1.0 meters

As seen in fig 2.13b the amount of salt and pepper noise increase as the distance from the object reaches the maximal distance of the sensor. The reason for this is the amount of the returning light in the pixel is below a threshold to give reliable depth estimates.



(a) Minimal salt and pepper noise

(b) High level of salt and pepper noise in the background.

Figure 2.5: Significant increase in invalid measurements when the camera is close to its maximum range[3]

2.3 Basis of computer vision

The following subsection introduce the various computer vision algorithms and principles used in this project. The primary focus will be on camera model, intrinsic and extrinsic camera parameters, calibration of camera, key point and feature detection and last but not least, VSLAM.

2.3.1 Camera model

A camera model is a description of the mapping between the world (3D) to the image (2D). There are many different camera models and which one to choose depend a lot on the camera in question. For many purposes and for the uses in this thesis the pinhole camera model should be sufficient.

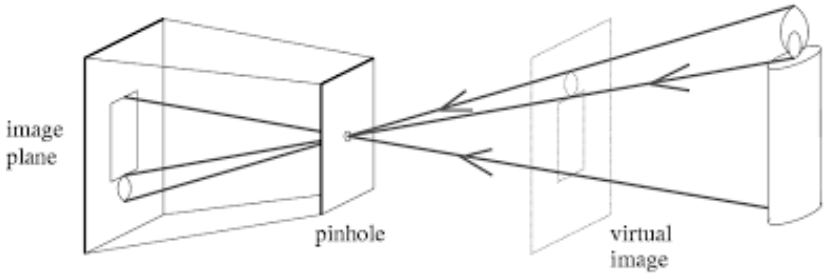


Figure 2.6: Illustration of the pin hole model, [4]

As seen in fig. 2.6 the pinhole model is based on that the light from the motive out in the world only passes through a tiny little hole in the camera. In this case a point in the world will only project to a single point in the image plane, which is a feature we want.

A common method is to set a virtual image in front of the pinhole instead of

using the image in the image plane, which is upside down. Given the use of the virtual image, we can use a simple mathematical mapping between this image an the world described by eq. 2.11, where X_w is the point in the world frame, and X_i is the point in the image.

$$x_i = KT_{cw}X_w \tag{2.11}$$

2.3.2 Intrinsic and extrinsic camera parameters

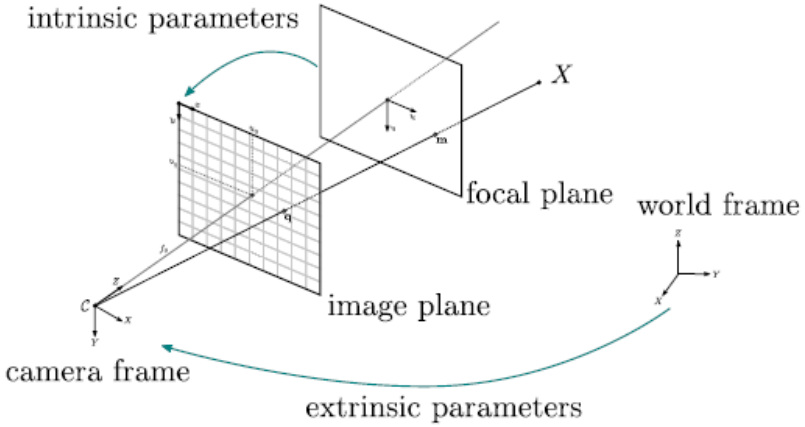


Figure 2.7: The connection between the intrinsic and extrinsic camera parameters and the different frames, [4]

In eq. 2.11, K is called the intrinsic matrix and T_{cw} is the extrinsic matrix. Together they represent the transform from the world frame into the image plane. As seen in the fig. 2.7 above the T_{cw} is the transform from the world frame to the camera frame. K is the transform from the 3D camera frame into the 2D image plane.

In the equation below, eq. 2.12, the equation is written in full form. The vectors x_i and X_W are homogeneous vectors, making the transformation between world and image frame easier with simple matrix multiplications. As seen in the equation, T_{cw} is a simple homogeneous transformation between the world frame and the camera frame, i.e it describes the rotation and translation between the two. In this case $[t_x, t_y, t_z]$ is the vector from the camera frame to the world frame.

The K matrix in the same equation consist of five parameters. f_u and f_v is the focal length given in pixels along the u and v axis. The parameters c_u and c_v is the principal point given in the u-v-plane. The final parameters s is the skew of

the image, and is most cases equal to zero.

$$x_i = \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix} \begin{bmatrix} f_u & s & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_x \\ r_{2,1} & r_{2,2} & r_{2,3} & t_y \\ r_{3,1} & r_{3,2} & r_{3,3} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ W_w \end{bmatrix} \tag{2.12}$$

2.3.3 Camera calibration

Camera calibration is the process of finding the unknown parameters given in the previous section, i.e. f_u, f_v, s, c_u and c_v, as well as some lens distortion parameters. Most cameras have one or more lenses and in one way or another it will effect the picture. However the pinhole camera model do not consider the effect of the lenses since there are none in the model.

Two types of lens distortion that are typical to model are radial distortion and tangential distortion. Radial distortion is due to fact that the light is bend more at the edges of the lens, than at the optical point. The distortion can be modelled as in eq. 2.13, where x_{dis}, y_{dis} is the distorted points, x and y is the undistorted pixel locations, $r = x^2 + y^2$ and k_1, k_2 and k_3 is the radial distortion coefficients of the lens.

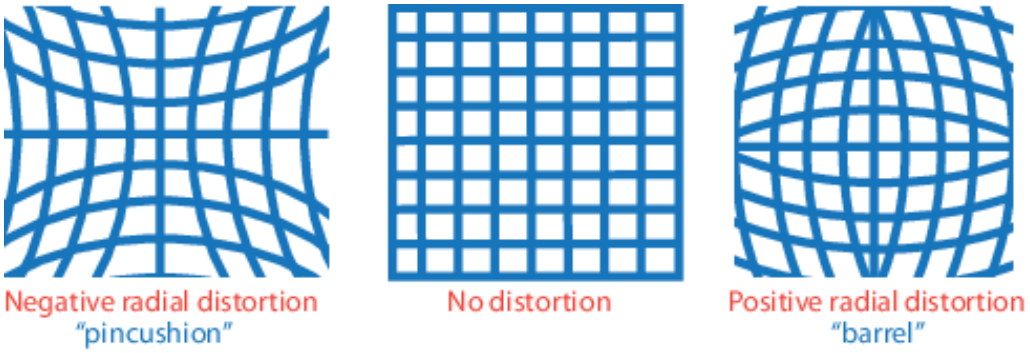


Figure 2.8: The different effects radial distortion has on the picture, [5]

$$x_{dis} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.13a}$$

$$y_{dis} = y(1 + k_1 * r^2 + k_2 r^4 + k_3 r^6) \tag{2.13b}$$

Tangential distortion happens when the lens and image plane is not parallel. To model this effect we use eq. 2.14, where p_1 and p_2 is the tangential distortion coefficients of the lens.

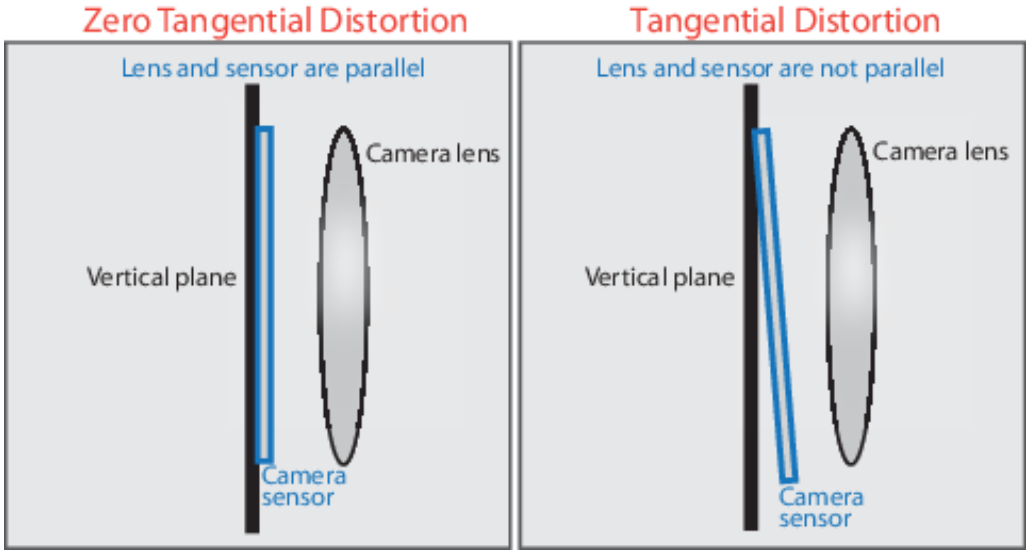


Figure 2.9: Example of tangential distortion,[5]

$$x_{dis} = x + (2p_1xy + p_2(r^2 + 2x^2)) \tag{2.14a}$$

$$y_{dis} = y + (2p_2xy + p_1(r^2 + 2y^2)) \tag{2.14b}$$

Thus the process of calibrating a camera is finding the parameters f_u, f_v, s, c_u, c_v, $k_1, k_2, k_3 p_1$ and p_2. There a many open source camera calibrations programs out there, openCV, ROS and matlab to mention some. The program generally works by taking a series of pictures of a checkerboard pattern from different angles and distances. The calibration parameters of the pico flexx camera was found by the use of the openCV camera calibrator and are presented in the table below.

f_u	f_v	c_u	c_v	k_1	k_2	k_3	p_1	p_2
208.02	208.02	111.29	87.18	0.8685	-7.175	12.12	≈ 0	≈ 0

2.3.4 Segmentation and thresholding

Segmentation is the process of the process of partitioning a digital image into multiple segments, usually based on the intensity of the pixels. If a set of pixels are close to each other and have have similar intensity, it is probably an area of interest. In the case of obstacle detection and avoidance the use of segmentation could be ideal to detect obstacles.

The simplest method of segmentation is by thresholding, however it has some major drawbacks. The process is as following; given the intensity in an image, set a threshold **t** and change the intensity of each pixel x in the image given by eq. 2.15. This will separate the picture into two regions, making them easier to work with. However some of the drawback are; what value to choose for the

threshold, separating into more than two regions require several thresholds, and increase the complexity of the problem.

$$I_{new}(x) = \begin{cases} 1 : I(x) \geq \mathbf{t} \\ 0 : I(x) < \mathbf{t} \end{cases} \tag{2.15}$$

Another method for segmentation is region growing. As seen in fig 2.10 a set of seeds is placed in the image. These seeds spread to nearby pixels with similar properties, usually based on the intensity of the neighboring pixels. While this method efficiency is based on where the seeds are put, it solves many of the problems with thresholding when segmenting into several regions.

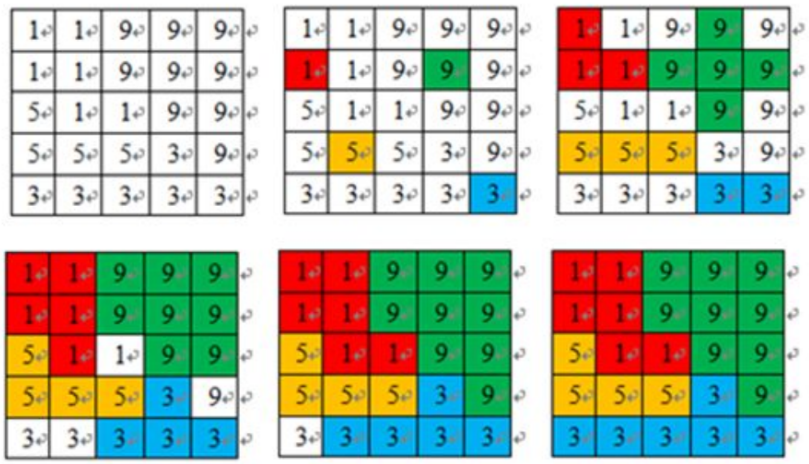


Figure 2.10: Process of segmentation by region growing, [6]

2.3.5 ORB

Oriented FAST and rotated BRIEF (ORB) is feature matching method, which consist of the FAST keypoint detector and BRIEF descriptor, to find corresponding points in different images,

Feature detection is the process of finding interesting points or key points in an image. This is usually done by looking at a pixel and the surrounding pixels to evaluate if this area is distinct enough to be labeled a feature in the picture. In general areas with large change pixel value in either direction is of interest, i.e edges and corners.

Features from accelerated segment test, better known as "FAST", is a high speed low computational corner keypoint detector. The algorithm can be summarized in the following points:

- Select a pixel p in the image. The intensity of that pixel is I_p
- Select a intensity threshold $\mathbf{t}$

- Make a circle of radius 3 pixels and consider the 16 pixels in this circle, as seen in fig 2.11.
- The pixel p is a corner if n contiguous pixels (typically n = 12) are brighter than $I_p + t$ or darker than $I_p - t$.
- Repeat for each pixel

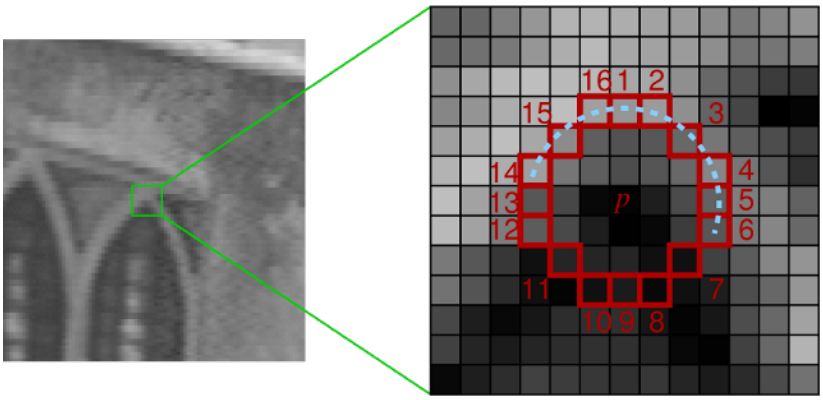


Figure 2.11: Illustration of how FAST corners work, [7]

To improve the speed the of the FAST algorithm one first should compare the intensity of pixel 1, 5, 9, 13 in the circle. If less than 3 of these don't satisfy the threshold criterion the point can be discarded as a corner. Otherwise continue with the rest of the points in the circle.

Once a corner has been found a descriptor is needed to separate one corner from another and such that feature matching is possible. ORB do this by using Binary robust independent elementary feature (BRIEF).

Brief make use of all of the keypoint found by FAST, and describe it and its neighborhood with a binary feature vector. The feature vector, also know as binary feature descriptor, contains only ones and zeroes. The length of the vector varies, but are usually a 128-512 bit string.

To make the binary feature descriptor BRIEF start by smoothing the image with a Gaussian kernal to make the descriptor more robust high frequency noise in the image. Then it select a random pair of pixels in the neighborhood around the keypoint. This neighborhood is called a patch, at a predefined size. The first pixel is drawn from a Gaussian distribution around the feature point. The second pixel is then drawn from a Gaussian distribution around the first pixel. The binary value is found by eq. 2.16 where p(x) and p(y) is the pixel intensities for the two pixels. This test is run 128-512 times filling up the binary feature vector. Feature correspondence is then found by comparing the binary feature vectors using Hamming distance. [8][9]

$$\tau(p, x, y) = \begin{cases} 1 : p(x) < p(y) \\ 0 : p(x) \geq p(y) \end{cases} \tag{2.16}$$

2.4 Visual simultaneous localization and mapping

Visual simultaneous localization and mapping (vSLAM) is a method/set with algorithms that are able to build a map of a unknown environment while performing localization by the use of visual sensor, i.e. mono and stereo cameras. This problem is often the most important problem to be solved when working with autonomous vehicles.

vSLAM algorithms are divided into two main categories, direct and indirect methods. These two can be divided into dense and sparse SLAM algorithms, which is strictly correlated to the amount of pixels used in the mapping (dense use more, sparse use less). The most typical is direct dense/sparse and indirect spare. The major difference between direct and indirect SLAM is whether the picture is preprocessed or not. Direct SLAM use the image directly, hence the name, while the indirect method make use of feature extraction first,like the type discussed in the previous subsection, 2.3.5. Based on the movement of these point in one image to another it is possible to find the transformation between these frames.

Indirect SLAM preprocesses the image by using a detector/descriptor like ORB and extract these features in each image. The problem can be defined as a minimization problem, described in eq. 2.17. Here the geometric error (reprojection error) is given by the term $u_i - \pi(T_{cw}X_i^W)$ and $\pi()$ is projection of a point from camera frame to image frame like in subsection 2.3.2. The variables to be found is the transform between world and camera, T_{cw} and X_i^* which is the set of feature points given in world coordinates. The problem of finding both T_{cw} and X_i^* is called full bundle adjustment. By linearizing the the measurement prediction function with a local Taylor expansion the problem can be formulated as a least square problem which has a known solution.

$$T_{cw}^*, x_i^* = argmin_{T_{cw},x_i^*} \sum_i \left\| u_i - \pi(T_{cw}X_i^W) \right\| \tag{2.17}$$

Direct SLAM algorithms make use of all or many pixels in the image along with their intensity. The transformation T_{cw} is found by minimizing the photometric error, as given in eq. 2.18. The photometric error is given as the difference in the pixel intensities in the two different pictures. This is done by transforming one of the pictures by T_{cw} and minimizing the error between the different pixel intensities.

$$T_{cw}^* = argmin_{T_{cw}} \sum_i \left\| (I_i - I_c(\pi(T_{cw}X_i^W))) \right\|^2 \tag{2.18}$$

2.4.1 ORB-SLAM

ORB-SLAM is a indirect sparse SLAM algorithm. There are a several different indirect and direct SLAM algorithm to choose from, but in this project ORB-SLAM will be the primary focus. There are several reason for this fact. Today

ORB-SLAM is one of the best performing vSLAM algorithms including both direct and indirect algorithms [10]. Another reason is that it uses ORB-features which are fast and less computationally heavy, ideal for embedded systems. Last but not least, ORB-SLAM is open source and is implemented as a package in ROS, which simplify the use and integration into the rest of the system [11].

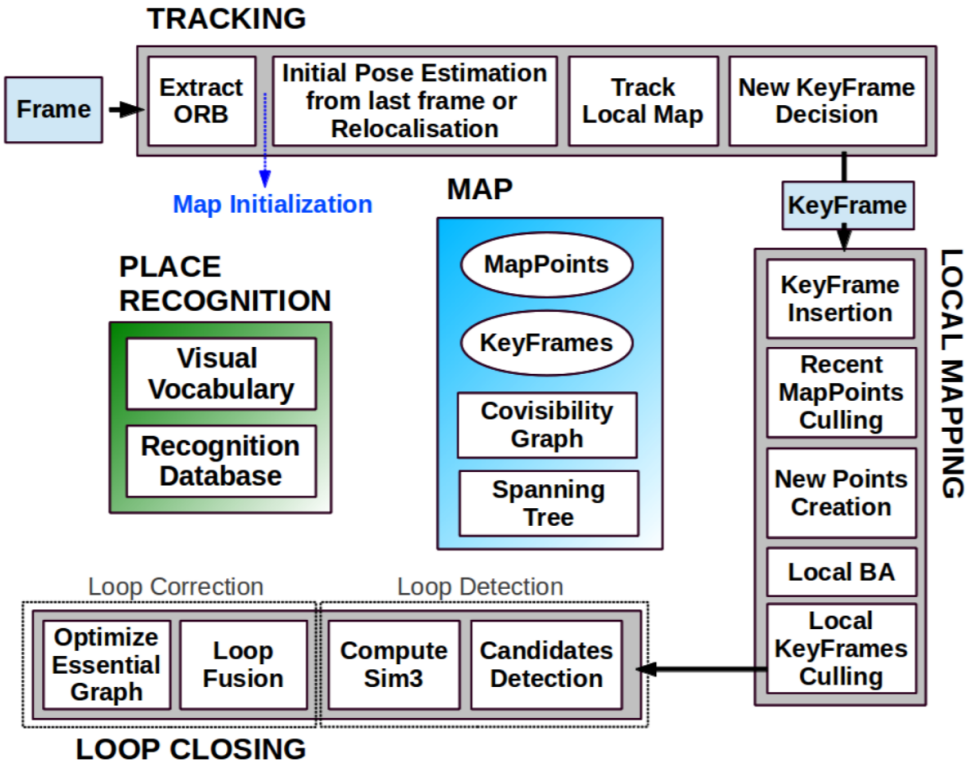
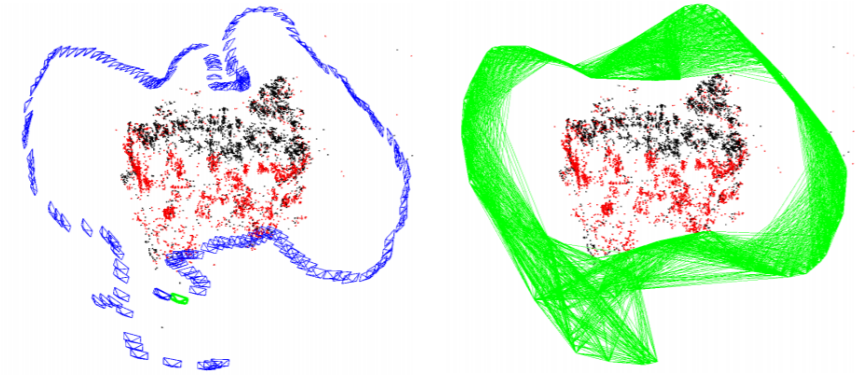


Figure 2.12: Overview of the ORB SLAM system, and the main events in each module, [10]

In fig 2.12 an overview of the ORB-SLAM system is presented. The sub modules tracking, local mapping and loop closing will be discussed under with focus on the important key points in each module.

Initialization

Before the ORB-SLAM algorithm starts it need to initialize the map by computing the relative pose between two frames. This is used to triangulate a set of initial map points. In parallel the system compute two geometrical models, a homography for planer scenes and one for non-planar scenes, a fundamental matrix. Based on a robust heuristic a model is chosen and when model is found a full bundle adjustment is performed.



(a) Keyframes (blue), current frame (green), local map point (red) and map points (red & black)

(b) The covisibility graph (green) which connect all the keyframes with common map points in a graph structure

Figure 2.13: Keyframes, map points and the covisiblity graph in ORB-SLAM,[10]

Tracking

The features are extracted with ORB as described in subsection 2.3.5. In ORB-SLAM the corners are extracted with a scale factor 1.2 at eight different scale levels. Depending on the resolution of the image a number between 500 - 2000 corners are extracted.

If tracking from the previous frame was a success, it uses a constant motion model to predict the next frame, and use a guided search for the map points observed in the previous frame. The pose is then optimized with a motion only bundle adjustment with the found corresponding map points.

Once an estimation of the camera pose is found and a set of feature matches, the points in the map is projected onto the current frame and a search for more map point correspondences is performed. Only the local map is projected because of the complexity in larger maps. The local map consist of two sets with keyframes, K_1 that share map points with the current frame and K_2 that are neighbors to the keyframes K_1 in the covisibility graph. In the set K_1 a K_{ref} keyframe exist, which is the frame that share the most map points with the current frame.

The last step is to decide whether the current frame is going to become a keyframe or not. In the local mapping thread there is a mechanism that culls redundant keyframes, which allows to keyframes to be inserted fast. To become a keyframe a set of conditions must be satisfied:

- More than 20 frames must have passed since the last global relocalization
- The local mapping is idle, or more than 20 frames have passed since the last keyframe insertion global relocalization
- At least 50 points is tracked in the current frame
- The current frame tracks less than 90% points from K_{ref}.

Local Mapping

When a frame is converted to a keyframe the covisibility graph is updated by by adding a new node for the new keyframe K_i, and then updating the edges to the keyframes in the map with shared map points. Also update the spanning tree which link K_i to the keyframe with the most common points. Then a bags of words representation of the keyframe is computed.

As well as keyframes, the map points is also culled. If the map point is to be retained in the map it must pass a test during the first three keyframes after creation, to ensure that they are trackable. A map point must fulfull two conditions:

- The tracking must find the point in more than the 25% of the frames in which it is predicted to be visible.
- If more than one keyframe has passed from map point creation, it must be observed from at least three keyframes.

New map points are created by triangulating ORB-features from the connected keyframes, K_c in the covisibility graph. ORB-feature pairs are triangulated, and are accepted when positive depth, scale consistency, reprojection error and parallax are checked.

Local bundle adjustment optimizes over the current keyframe K_i, all of the keyframes K_c that is connected to it in the covisibilty graph, and all of the common map points. All of the keyframes that see the same map points, but are $\notin K_c$ is included but kept fixed in the optimization.

To keep the map as compact as possible the local mapping tries to detect redundant keyframes and delete them. This effect reduces the load on bundle adjustment since the complexity grow with the number of keyframes. Another advantage is that the number of keyframes won't grow unbounded, unless the move into new visual content.

Loop closing

The last thread is the loop closing thread. It takes the K_i from local mapping, the last keyframe processed by the local mapping, and tries to detect and close the loop.

First the similarity between bag of words vector of K_i and its neighbors in the covisiblity graph is computed, and the lowest score S_{min} is then used as a threshold and discard all keyframes in the database that has a lower score than S_{min}. Additionally all the keyframe directly connected to K_i is discarded. A loop candidate is accepted if at least three loop candidates is connected in the covisibility graph.

To close the loop a similarity transform from the current keyframe K_i to the loop keyframe K_l has to be computed, which informs about the error accumulated in the loop. With 3D to 3D correspondence for each candidate, RANSAC is used to

find a similarity transform. If a similarity transform with enough inliers is found the transform is optimized and a guided search for more correspondences between the frames K_i and K_l are performed. The loop is accepted if the optimization has enough inliers after the guided search.

The first step after a loop detection is to fuse the duplicated map points and new edges are inserted into the covisibility graph. The pose T_{iw} of the current keyframe K_i is corrected by the similarity transform, along with all of the neighbors of K_i. The common map points are projected into K_i and all matches found in a small area around the projection are fused. All the edges of the keyframes involved in the fusion will then be updated in the covisibility graph, and then closing the loop.

Chapter 3

Hardware specification

3.1 Wall-E 2.0

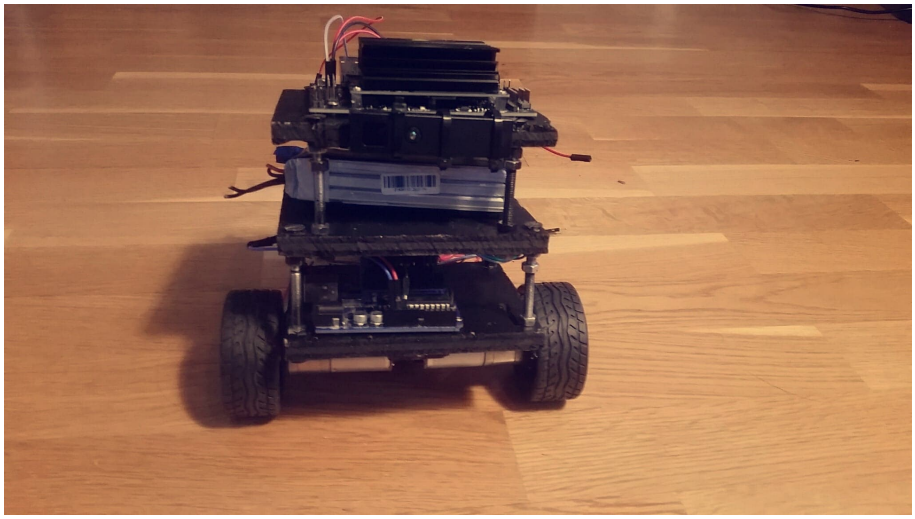


Figure 3.1: A picture of Wall-E 2.0.

Wall-E 2.0 is a three degree of freedom robot, able to move in the xy-plane and around its own axis. It has two wheels and a tail to keep the body stable. The robot consist of three wooden plates on top of each other. At the bottom the hardware responsible for motor control and odometry is found. In the middle the battery is placed along with a battery elimination circuit for voltage regulation down to 5V. At the top the primary computer is placed, the Jetson Nano, along with the Time of flight camera in the front. The following section will present the reader to the hardware used.

3.2 Onboard computers

Wall-E 2.0 has two computers, one meant for real time control of the robot, Arduino Uno, and the Jetson Nano for more computationally heavy operations like image processing and SLAM.

3.2.1 Arduino uno

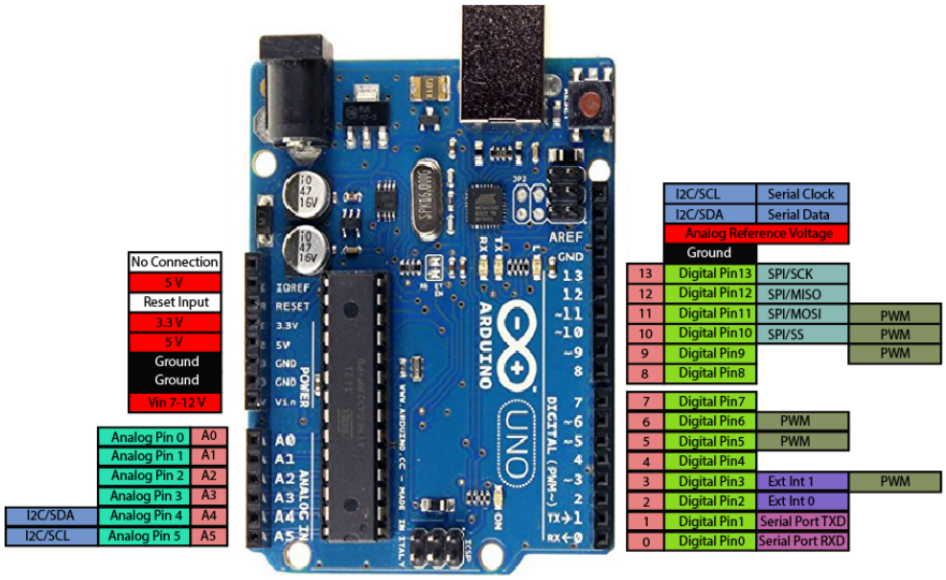


Figure 3.2: Arduino uno, with the different input and outputs , [12]

The Arduino Uno is a small 8-bit, 5V microcontroller board, based upon the ATmega328P, and it is quite suitable for simple control, and sensor readings. The micro controller board has 14 digital I/O pins which 6 of them can provide PWM output. In addition it has 6 analog input pins, and capability to use a set of popular protocols like SPI, I^2C and UART. The CPU runs at 16 MHz which is more than sufficient for simple task like sensor reading and less complex computations.

In this project the computer is used as a slave of the Jetson Nano via the serial port. The Arduino is responsible for communication and with the MPU6050 via the I^2C protocol, PWM control of the motors, and estimate motor speed based upon encoder readings. The implementation of these systems are discussed in section 5.2.

3.2.2 NVIDIA Jetson Nano

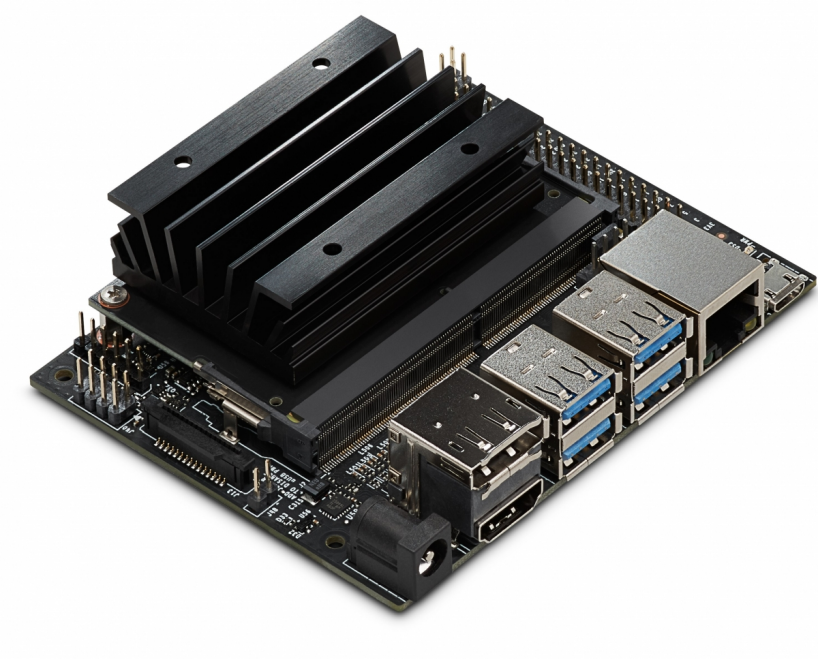


Figure 3.3: Picture of the Jetson Nano Dev board, [13]

The Jetson Nano is the most recent embedded platform from NVIDIA and is the low cost embedded computer version in the NVIDIA Jetson series. The computer comes with the development board and has a series of connections as specified in the table below [14].

Specifications	**Techonlogy**
GPU	128 NVIDIA CUDA cores
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI 2.0 and eDP 1.4
USB	4x USB 3.0, USB 2.0 Micro-B
I/O	GPIO, I^2C, I^2S, SPI, UART
Size	100 mm x 80 mm x 29 mm

The Jetson Nano runs a complete desktop Linux environment based upon the Ubuntu 18.04 LTS. This makes the module easy to work with, and enable the use of frameworks like ROS. In this project the Jetson Nano will be used as the primary module running ROS, SLAM, user interface and high level control of the robot, more on this in Chapter 5. A WiFi dongle is connected to one of the USB ports used for SSH and remote control of the robot.

3.3 Sensors

In the following subsections the sensors used on the robot will be presented.

3.3.1 MPU6050 IMU

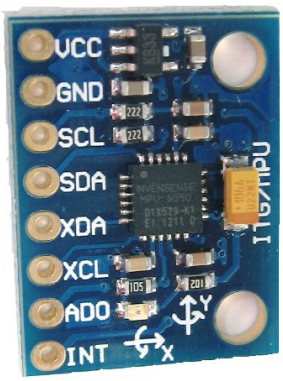


Figure 3.4: The MPU6050 6DOF IMU, [15]

The MPU6050 is a lowcost 16-bit 6 degree of freedom IMU, i.e. accelerometer and gyroscope. The MPU-6050 version uses the I^2C protocol at 400 Khz, where other version like the MPU-6000 also has an SPI interface at 1 Mhz. The interrupt pin can be programmed to The sensor has a series of different modes, and configurations of the sensitivity of the sensors, as given in the table below [16].

Mode	Gyroscope sensitivity	Accelerometer sensitivity
0	$\pm 250\ ^\circ/s$	± 2 g
1	$\pm 500\ ^\circ/s$	± 4 g
2	$\pm 1000\ ^\circ/s$	± 8 g
3	$\pm 2000\ ^\circ/s$	± 16 g

The IMU also has a integrated digital low pass filter for gyroscope and the accelerometer ideal for filtering out high frequency noise in real time. The low pass filter has a series of different modes, described in the table below, where the bandwidth (bw) ranges from 5 Hz to 260 Hz. The drawback is that the measurements will be delayed dependent on the chosen bandwidth of the filter.

Mode	Acc. bw [Hz]	Delay [ms]	Gyro bw [Hz]	Delay [ms]
0	260	0 g	256	0.98 g
1	184	2.0 g	288	1.9 g
2	94	3.0 g	98	2.8 g
3	44	4.9 g	42	4.8 g
4	21	8.5 g	20	8.3 g
5	10	13.8 g	10	13.4 g
6	5	19.0 g	5	18.6 g

3.3.2 Pico flexx

Figure 3.5: Picture of the PMD Pico flexx camera, [17]

Parameter	Camera Properties
Dimensions	68 mm x 17 mm x 7.35 mm
Weight	8g
Camera Resolution	224 x 171
Viewing angle (H x V)	62° x 45°
Measurement range	0.1 - 4 m
Frame rate	5 - 45 fps
Illumination	850 nm IR
Depth resolution	$1\% - 2\%$ of distance
Output	Depth map and grayscale intensity

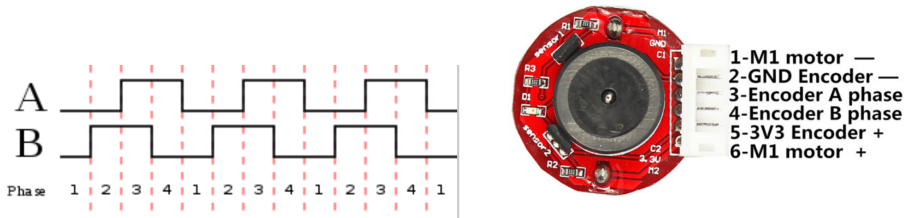
The PMD Pico flexx is a time of flight camera. The working principle is as described in theory, 2.1. The output from the camera is a depth map, point cloud and intensity map, see figures 2.4 and 5.6. The pico flexx has a series of different modes where the frame rate varies from 5 fps to 45 fps, and the corresponding maximum range from 4 meters to 1 meter.

Nr	Use Case	Name	Range [m]	Framerate	max. Exposure Time (us)
1	Indoor room reconstruction	MODE_9_5FPS_2000	1 - 4.0	5 fps	2000
2	Room scanning, indoor navigation	MODE_9_10FPS_1000	1 - 4.0	10 fps	1000
3	3D object reconstruction	MODE_9_15FPS_700	0.5 - 1.5	15 fps	700
4	Medium size object recognition, face reconstruction	MODE_9_25FPS_450	0.3 - 2.0	25 fps	450
5	Remote collaboration, step by step instruction, table-top gaming	MODE_5_35FPS_600	0.3 - 2.0	35 fps	600
6	Small object/product recognition, Hand tracking	MODE_5_45FPS_500	0.1 - 1.0	45 fps	500
7	Mixed Mode	MODE_MIXED_30_5		30/5fps	300/1300
8	Mixed Mode	MODE_MIXED_50_5		50/5fps	250/1000

Figure 3.6: The different modes of the Pico flexx camera

3.3.3 Encoder and motors

The motors are a simple bidirectional 12 V DC motors. The encoder is attached to the end of the motor. The encoder used is a rotational incremental encoder, which output is two phase shifted square waves. The phase is shifted by +/- 90 degrees depending on the direction of the circular motion, as seen in fig 3.8a. The encoder will send a given number of pulses per rotation of the motor, 341.2 pulses per rotation to be exact. The number of pulses can be measured by the Arduino micro controller, and the rate of rotation can be estimated. To get the rate and the direction of the rotation both of the phases, A and B, has to be measured. However if only the rate is of interest, it is sufficient to measure only the pulses from one of the signals A or B.

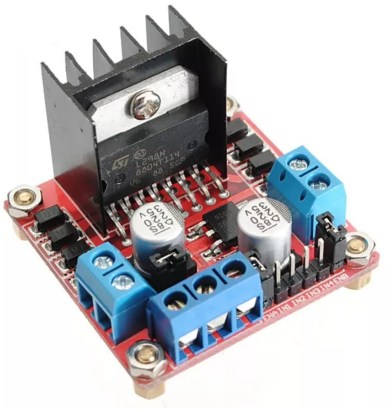


(a) Phases of the encoder, [18] (b) Pinout of the encoder, [19]

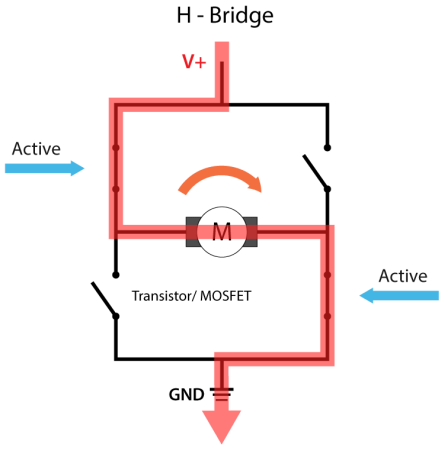
Figure 3.7

3.4 Other components

3.4.1 L298N Dual H Bridge



(a) L298N Dual H Bridge, [20]



(b) Working principle of H-bridge, [21]

The L298N Dual H Bridge is the motor controller of the robot. The controller can drive DC motors from 5 to 35 V. It also has a on board voltage regulator, down to 5 V, which can be used if the input voltage is 12 V or less. This is necessary for the integrated circuit to work properly, and it also have a 5 Volt output which can be used to power other modules, like the Arduino Uno.

The board has a set of input pins, $E_A, IN_1, IN_2, IN_3, IN_4$ and E_B.E_A, IN_1, IN_2 are used to control motor A, while the rest is used to control motor B. The set of combinations of signals are presented below. To enable speed control PWM-signal is sent to EN_A and EN_B, turning the voltage over the motors on and off, lowering the rate of the rotation of the motor.

Description	EN_A/EN_B	IN_1/IN_3	IN_2/IN_4
Motor is off	0	x	x
Break and stop	1	0	0
Forward motion	1	0	1
Backward motion	1	1	0
Break and stop	1	1	1

3.4.2 Universal Battery eliminator circuit (UBEC)

The battery used in this project is a 3 cells LiPo battery with a voltage around 12 V. Both the Arduino Uno and the Jetson Nano have a input voltage of 5 V and the Jetson Nano also have a quite large current draw which makes it necessary to have battery eliminator circuit. While the L298N has a voltage regulator on board it has a max current of 2 ampere, which isn't enough to drive the Arduino, MPU6050, encoders, Jetson Nano and the Pico flexx camera. That is why the

UBEC is set to powering the Jetson Nano, and the connected Pico Flexx, while the L298N powers the Arduino Uno and all of the connected sensors.

3.5 Connection and power

In fig 3.9 a diagram of the connections on the robot is presented. The key idea is to show the reader how the different modules are connected and how they work together.

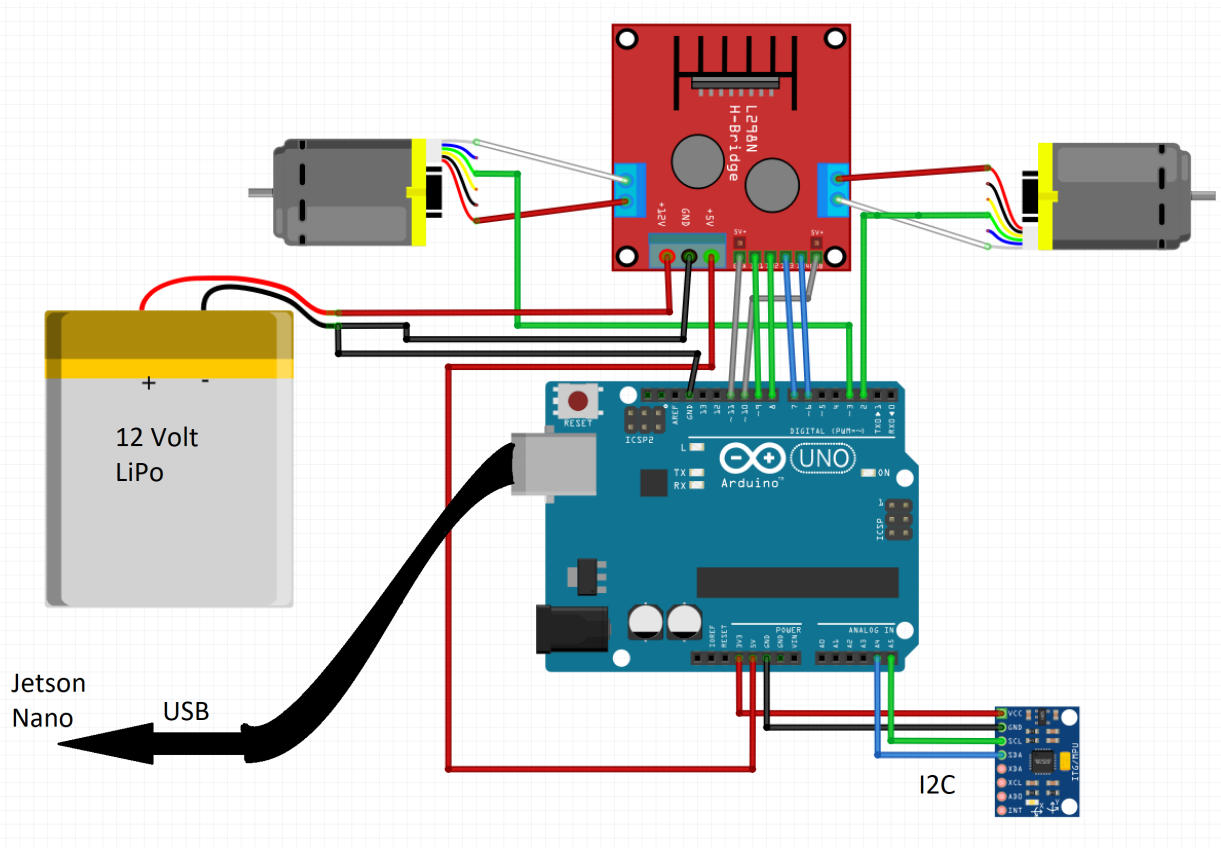


Figure 3.9: Connection schematics

Chapter 4

Simulation

4.1 Robotic operating system (ROS)

ROS is a popular open source framework for robot software development. It is not an operating system, as the name might make you believe, but rather a collection of libraries, tools and drivers making the development of robotic software easier. ROS is compatible with the most recent versions of Ubuntu, and in the latest release *Melodic*, it has also become available for Windows 10.

ROS is the software skeleton in project that binds the different parts of software together primarily via communication and information sharing. Thus it is seen fit to introduce some of the important concepts in ROS and terminology.

ROS Graph

The ROS framework can be viewed as a network of programs sharing data, in a graph like structure. A ROS system is a set of smaller programs that communicate via defined messages. In the ROS Graph these programs become the *nodes*, while the communication between them becomes the edges. The communication is based on publishing and subscribing. One node may publish a message on a *topic* and any number of nodes may subscribe on this topic. A simple real life example would be a radio broadcast, and anyone who would like to could tune in on that channel.

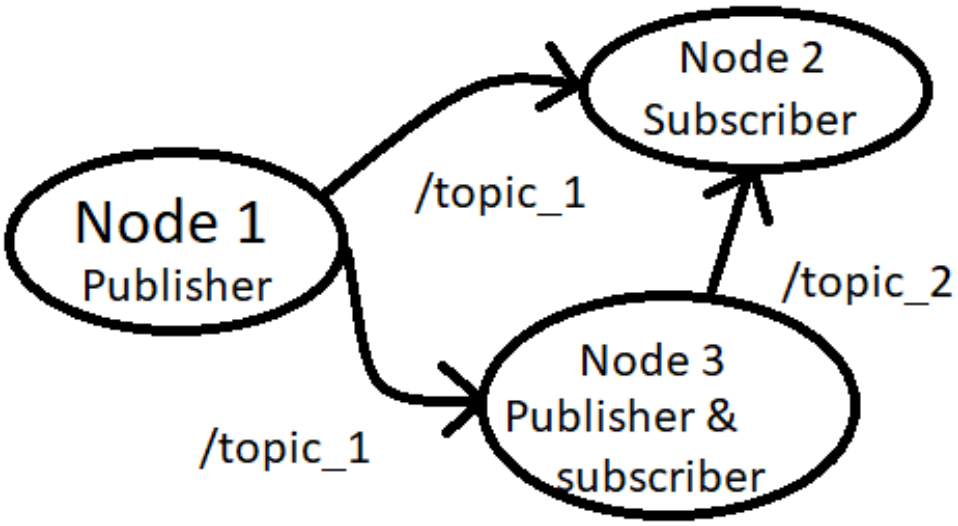


Figure 4.1: A simple example of a ROS graph. Node 2 and 3 subscribe on topic 1 published by node 1 and node 3 also subscribe on topic 2 published by node 2.

ROS master

The ROS master is the core in the ROS graph network. Before any node my run, an instance of *roscore* must be launched first. The ROS master is responsible for setting up the communication between the nodes.

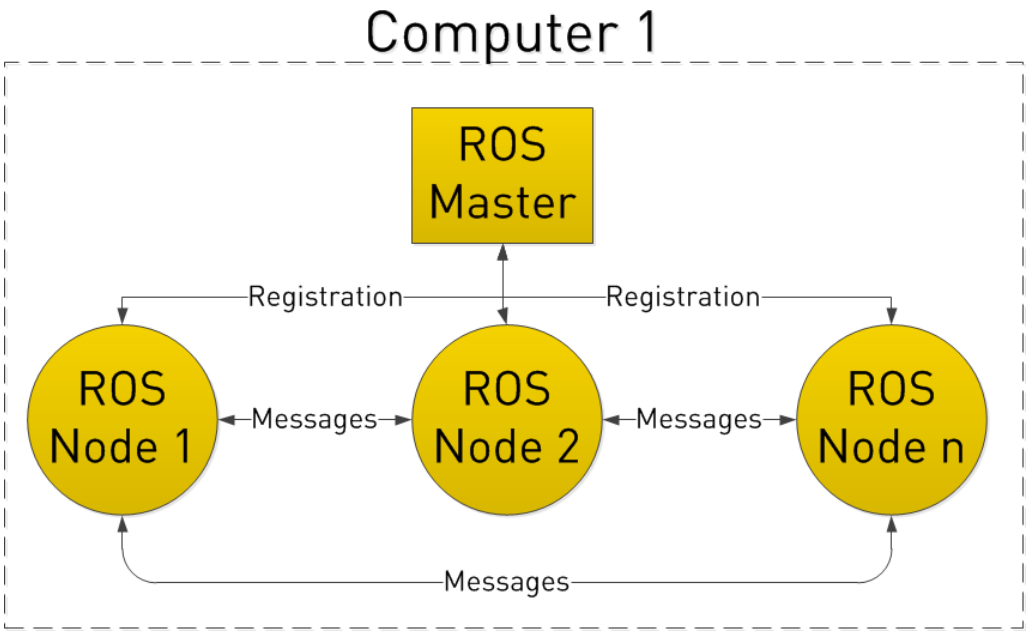


Figure 4.2: Connections of the ROS network, [22]

ROS messages

ROS messages are the data that is sent between nodes in the ROS network. When two nodes communicate they have to agree on the structure of the data. ROS has a many predefined message data structures, like **std_msgs/Bool** or **geometry_msgs/Pose**. ROS has messages for almost every need in robotics, but if needed it is possible to generate custom messages.

Tools

In ROS there are a series of tools making the world of developing robotic software a little easier:

- *RVIZ:* Visualization in 3D, combined robot model, sensor data, obstacles in a combined view.
- *ROS_bag:* Save messages published on one or several topics, able to replay the data
- *rqt_plot:* Plot scalar data published on ROS topics

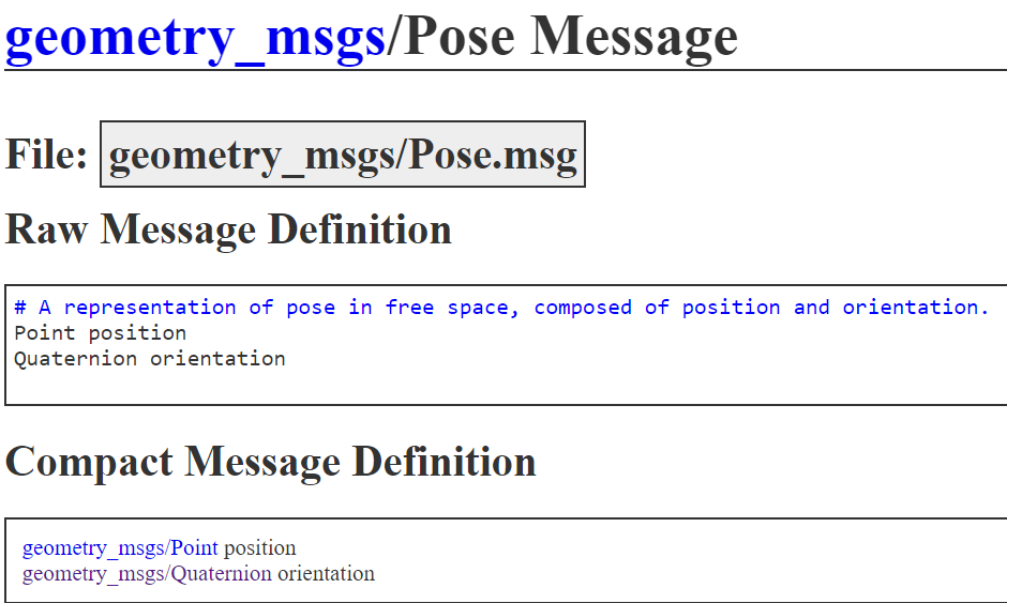
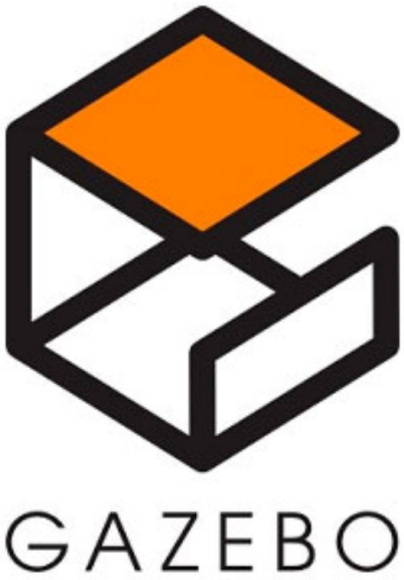


Figure 4.3: Structure of the pose message, [23]

4.2 The simulation environment Gazebo



Gazebo is a real-time simulator with a robust physics engine, quality graphics and interfaces linked to ROS. It is possible to build a robot model, and use a wide variety of sensors like accelerometer, gyroscope, LIDAR, camera and so on.

In gazebo it is possible to construct a world of your choosing. The software has a series of predefined static objects that one may place in the world using the graphical user interface. It is also possible to construct the world and the objects in it by using world-files to define the world and Unified Robot Description Format (URDF), to describe the models in the world. In both files the syntax is XML.

Since the gazebo-simulator can communicate with ROS over the local network, using TCP/IP, it is an ideal simulator to test software developed in ROS, as well to simulate a time of flight camera, and use the simulated data as input into the ORB-SLAM algorithm in ROS. The purpose of following sections is to discuss how the implementation of the robot, world and time of flight camera was done in gazebo, as well as shed some light on the communication and interfaces between gazebo and ROS.

4.3 Simulated robot

4.3.1 Building the robot model

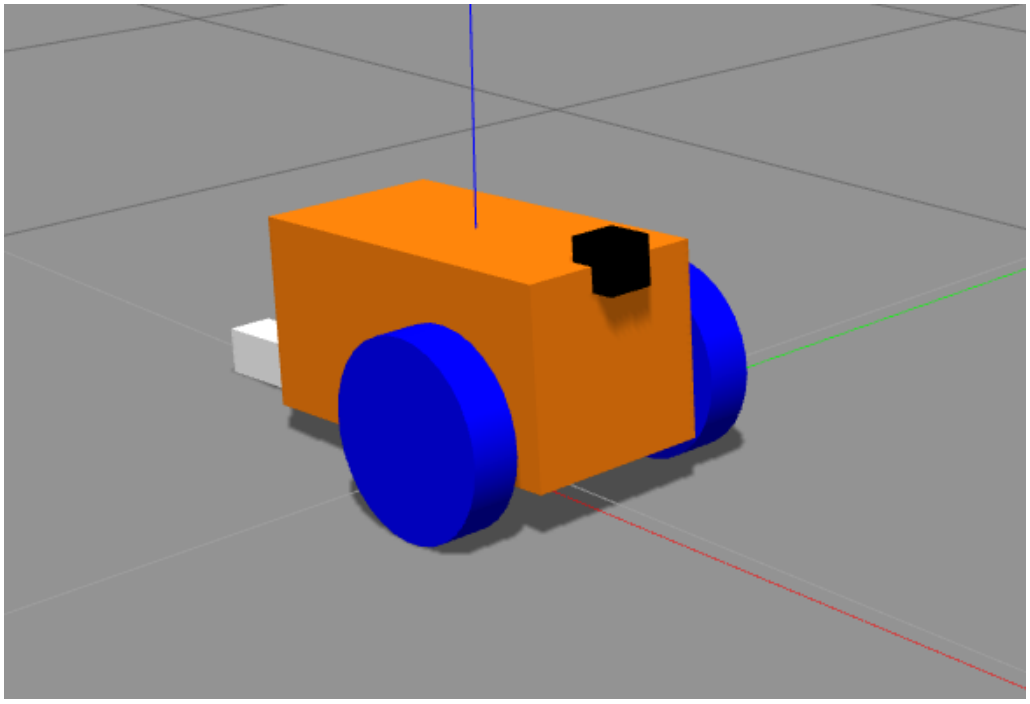


Figure 4.4: The robot model is Gazebo

We use URDF to define and describe the different parts of the robot. The idea is to keep the model simple since the purpose of the simulation is testing ToF-camera and ORB-SLAM, and not simulating an advanced robot. The robot consist of five parts; the chassis, two wheels, a tail and the camera.

The parts are defined as links in the URDF file, and are connected by joints. These joints can continuous which allows the links to move in a defined direction, or it could be fixed, holding two objects together in place. A small example of how to define links using URDF is found in fig 4.5a. As seen in the figure the chassis consist of a mass, and two defined boxes, where the visual box is what we see in the simulation while the collision box is the volume used in the collision engine in Gazebo.

```
<link name='chassis'>
  <inertial>
    <mass value="1.0"/>
    <origin xyz="0.0 0 0.2" rpy=" 0 0 0"/>
  </inertial>
  <collision name='collision'>
    <geometry>
      <box size=".4 .2 .2"/>
    </geometry>
  </collision>
  <visual name='chassis_visual'>
    <origin xyz="0 0 0" rpy=" 0 0 0"/>
    <geometry>
      <box size=".4 .2 .2"/>
    </geometry>
  </visual>
</link>
```

(a) Example of how to define a link in Gazebo. Here it is the chassis used in the simulation.

```
<gazebo reference="tof">
  <sensor type="depth" name="tof_frame_sensor">
    <always_on>true</always_on>
    <update_rate>20.0</update_rate>
    <camera>
      <horizontal_fov>${62.0*3.141/180.0}</horizontal_fov>
      <image>
        <format>R8G8B8</format>
        <width>224</width> <!-- Pixel width -->
        <height>171</height> <!-- Pixel hight -->
      </image>
      <clip>
        <near>0.01</near>
        <far>5</far>
      </clip>
    </camera>
```

(b) XML code to define the ToF sensor in Gazebo.

Figure 4.5

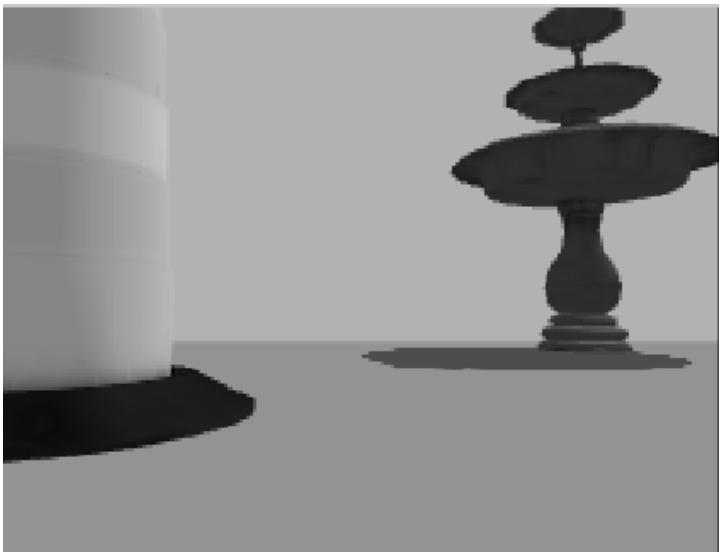
4.3.2 Simulated sensor

To add a sensor to the robot, first a sensor-link has to be created. This is necessary for the simulator to know where the sensor is place, in what direction it is facing and so one. Once the link is created it can be defined as a sensor. This is done by defining the specifications of the sensor, whether it is a IMU, LIDAR or a camera. The sensor must also be linked to a plugin, which is the code that simulate the sensor based on data from the simulation environment.

As seen in fig 4.5b we include the specifications of the camera, like the pixel width and height, update rate, and the field of view. The parameters used is based of the Pico Flexx. It is also possible to include the calibration parameters found for the Pico Flexx and add it to the plugin for a more realistic simulation of the camera.

In the figures below, 4.6a and 4.6b we see the simulated intensity and depth image, which both is grey scale images. In the case of the depth image, the

darker the pixel, the closer the object is to the camera. Note that if no depth information is available the value will be set to zero, as seen in the background of the depth image.



(a) IR image generated in gazebo

(b) Depth Image generated in gazebo

Figure 4.6

4.3.3 Controlling the robot

The robot is now able to view the world around it, but it still need a way to move around. We solve this by using another plugin which we can link directly to the continuous joints between the chassis and the wheels. This plugin is called *differential drive controller* one of the standard plugins in the gazebo library. This plugin allows for the control of the robot via ROS. The robot is controlled by sending a Twist message, a data structure containing two 3D vectors, linear and angular. This message is sent on the topic *cmd_vel*, where linear.x is the

body frame velocity forward in the positive direction, and angular.z is the rate of rotation around the z axis. To make the user interface easier a ROS-node is developed to take input from the keyboard. The arrow keys is used to increase or decrease linear velocity and angular velocity, while the space bar is used to stop the robot.

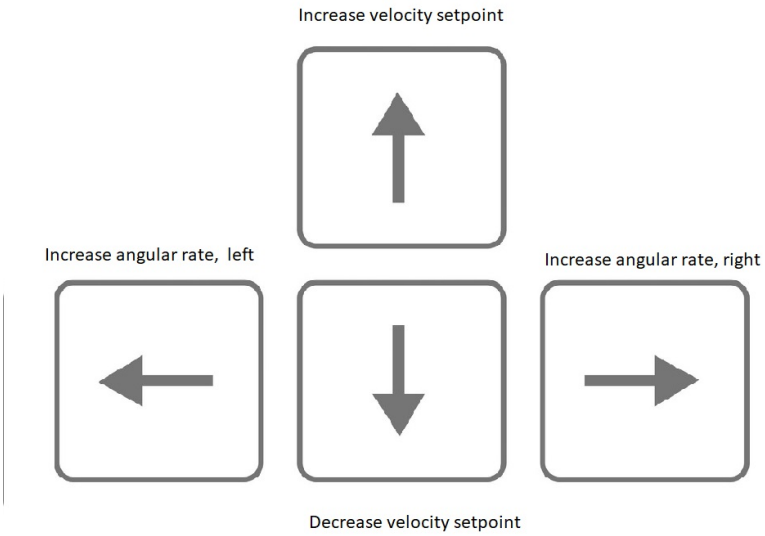


Figure 4.7: Keyboard inputs, [24]

4.3.4 Path following control

Given the implementation in the previous subsection it is now possible to control the robot using input from the keyboard. However it would be practical if the robot could be able to follow and track a set of predefined points.

To solve this we use a vector field that converges to the point of interest, see fig 4.8. The details of this implementation is given in section 5.6 where we combine different kinds of vector fields for obstacle avoidance.

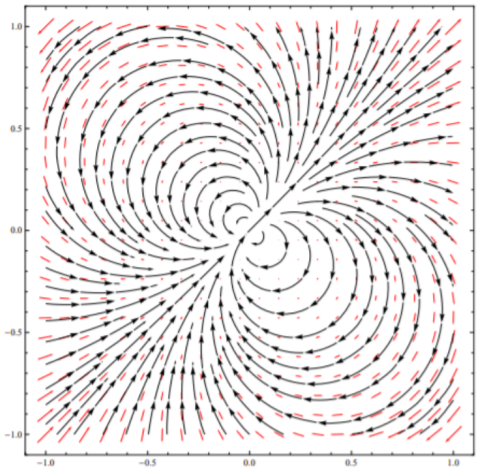


Figure 4.8: Dipol vector field, centered at the target,[25]

4.4 Simulated world

The gazebo world would be empty unless some objects are put into it. To test ORB-SLAM in the simulator, a simple environment is created. It consist of a closed space with some walls forming a hallway in a square shape. The idea is having the robot follow a predefined path through this environment.

Since ORB-SLAM is feature based it is important that the objects in this environment has features. For that reason brick walls are used, which contains a lot of edges and corners that are detectable by the algorithm. A set of objects are also put into the environment, as seen in fig 4.9. The detail of the test itself is specified in section 6.1.

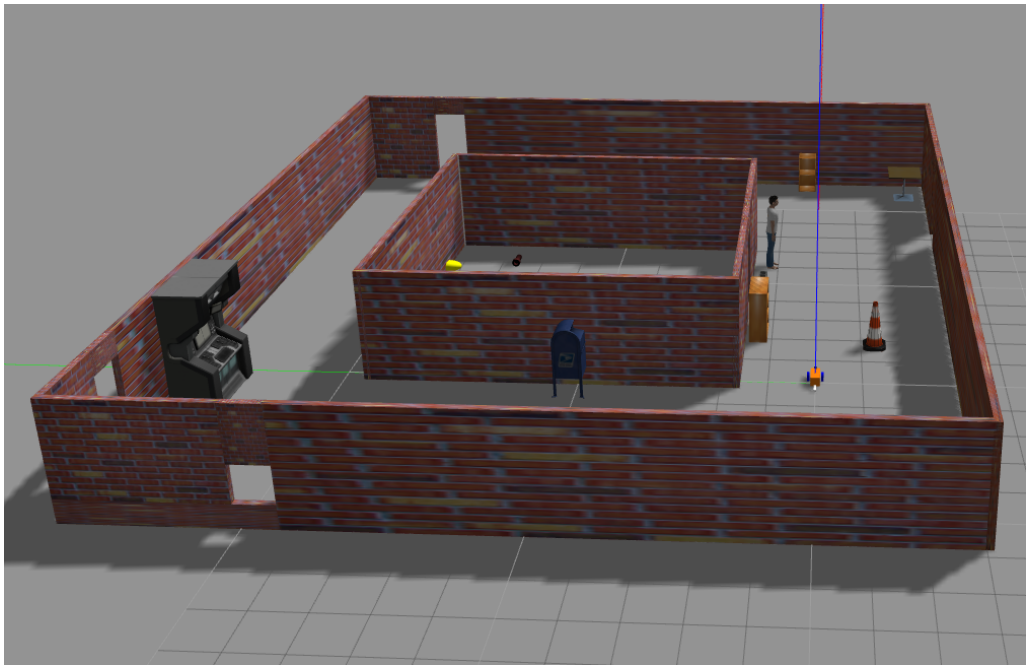


Figure 4.9: Overview of the test world environment

Chapter 5

System implementation

5.1 System overview

The following sections will present the different modules in the system, and will discuss in some detail how they where implemented. A system overview of the ROS network is presented in the figure below. ROS in the core of the implementation and binds the different modules together.

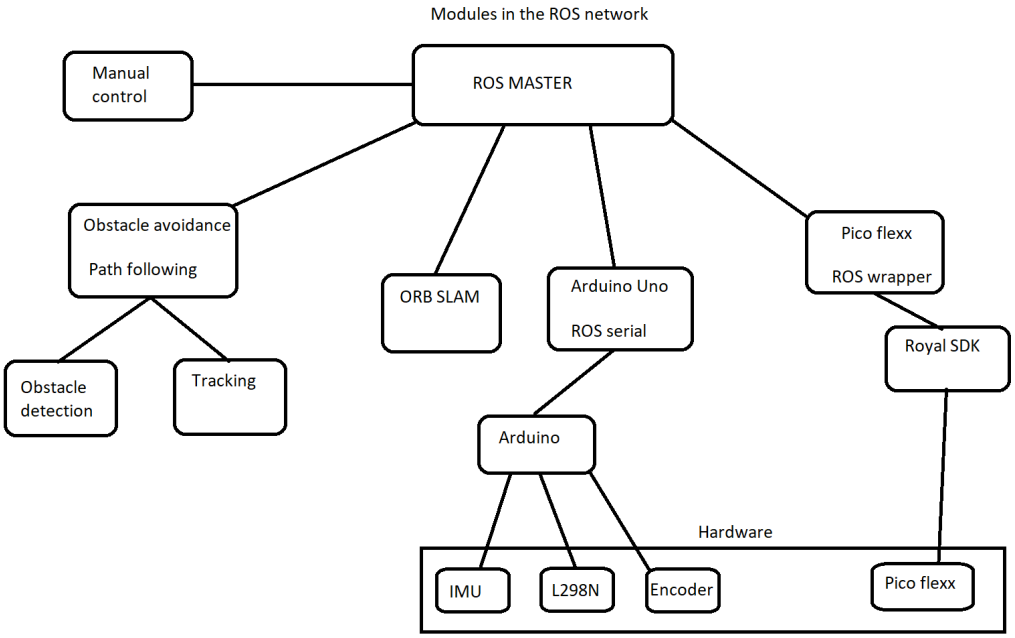


Figure 5.1: Overview over all the software systems and how they are connected

5.2 Drivers

The following three subsections will discuss the implementation of drivers on the arduino Uno, i.e. the software to communicate with the sensors MPU6050 and

the encoder as well generating a pwm signal to the L298N for motor control. By using the Arduino IDE and the associated libraries, we can avoid developing code for the most low level hardware of the micro controller.

5.2.1 MPU6050

The MPU6050 communicate with the Arduino via the I^2C protocol. By using the Arduino I^2C driver communication between the sensor and the micro controller is set up with some simple function calls.

Initialization

The process of the initialization is setting the proper registers on the MPU6050. This involves enabling the the sensor, chose configuration for low pass filtering, and decide scale of the measurements.

```
void setup() {
  Wire.begin();                 // Init I2C
  Wire.beginTransmission(0x68);// Adress = 0x68


  Wire.write(0x6B);         // Status register
  Wire.write(0x00);         // Write 0x00 into the register
                            // to enable the sensor
  Wire.write(0x1A);         // Config register
  Wire.write(0x00);         // Set to standard config
  Wire.write(0x1B);         // Gyro config
  Wire.write(0x00);         // Scale +/- 250 deg
  Wire.write(0x1C);         // Accel config
  Wire.write(0x00);         // Scale +/- 2 g
  Wire.endTransmission(true); // End transmission
```

Figure 5.2: Initialization code for MPU6050

Reading the data

The accelerometer and gyroscope data is read out from 12 eight bits registers, two for each measurement, addresses 0x3B-0x40 and 0x43-0x48 respectively. The code runs in loop, and the data is read whenever the data is available. To convert the data to the right format the bits from the first register must be shifted 8 bits, since they are the 8 most significant bits of the data, and then scaled depending on the scale chosen in the initialization, as seen in fig. 5.3.

```
void loop() {
  float acc_x;

  Wire.beginTransmission(0x68);
  Wire.write(0x3B);
  Wire.endTransmission(false);
  Wire.requestFrom(MPU, 2, true);
  acc_x = (Wire.read() << 8 | Wire.read()) / 16384.0; // X-axis value
```

Figure 5.3: Example of how to read the x-axis accelerometer data. The format is similar for the other axis, and the gyro readings

5.2.2 PWM motor

The control of motor is one of the simplest parts of the code on the robot. As discussed in chapter 3. the motor is controlled by the L298N, which have 6 inputs, which 4 of them are digital. For pins on the Arduino Uno is set as output pins and are connected as described in subsection 3.5. Using the table in subsection 3.4.1 forward motion is enabled by setting $IN_1/IN_3 = 0$ and $IN_2/IN_4 = 1$ and sending a PWM signal to EN_A and EN_B. For backward motion the digital signal are inverted. The PWM signal is generated by the Arduino analogWrite(pin, value) function where the value ranges from 0 to 255 as seen in fig. 5.4. In terms of the motor speed a value of 0 would mean no motion, and a value of 255 would mean full speed.

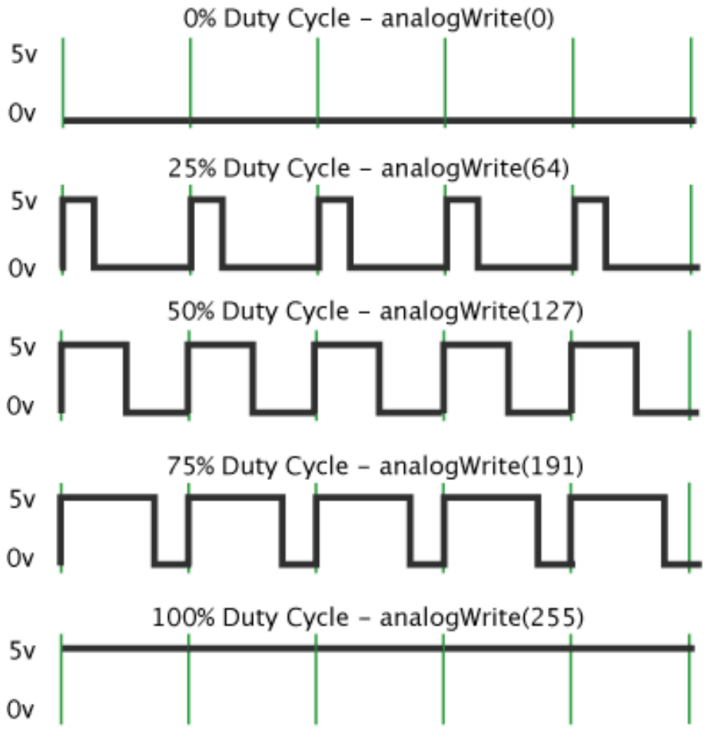


Figure 5.4: Pwm signal for different values, [26]

5.2.3 Encoder

As described in Chapter 3 the encoder send out two pulses out of phase by $\pm 90^o$. The direction of rotation can be determined by the order of the pulses. If the A pulses before B it turns forward, in the other case it turns backwards. While it would be ideal to use both of the phases, the Arduino Uno has only two interrupt pins, and it is necessary to use interrupts to avoid polling and stalling in the software, and pulses might be lost if the input pin isn't check regularly. However with some assumptions the direction of the motor can be determined by the input signal to the L298N. One would expect the motor to in that direction it is told to. It is only in the case where the robot goes in one direction and suddenly is command to go the other direction, where there is an ambiguity. This problem is solved all together by avoiding these cases with sudden changes in the direction of velocity.

Since the Arduino only has two interrupt pins only the pulse A of each motor is measured and counter. It is programmed such that each time the pulse goes from low to high a counter is incremented by the interrupt service routine (ISR). Every 100 ms the counter value is saved and passed on in the program, and the counter is reset to zero. The velocity of the wheel can then be determined by eq.5.1, where 341.2 is the number of pulses per rotation, $\Delta t = 100$ms and D_{wheel} is diameter of the wheel.

$$v_{wheel} = \frac{counter}{341.2\Delta t} D_{wheel}\pi \tag{5.1}$$

5.3 Control system

The control system consist of two parts. A controller to control the speed of each of the motors, and a simple state estimator fusing the measurements.

5.3.1 Model

To make a mathematical model a few parameters and variables needs to be defined. The velocity and rate of rotation is a function of several variables, as given in eq. 5.2. The $\dot{x}_b, \dot{y}_b, \dot{\Theta}_b$ are the state variables, which is a function of; the length from a wheel to the center of the robot l, the diameter of the wheels d, and the rotational rate of the right and left motors $\dot{\phi}_r$ and $\dot{\phi}_l$ [rad/s].

$$\begin{bmatrix} \dot{x}_b \\ \dot{y}_b \\ \dot{\Theta}_b \end{bmatrix} = f(l, d, \dot{\phi}_r, \dot{\phi}_l) \tag{5.2}$$

The states of the robot can be model by the function f(l, d, $\dot{\phi}_r$, $\dot{\phi}_l$) which needs to be found.

The driving force of the robot is the two motors turning the wheels. Given a point P at the center of the robot and assume the local reference frames origin is in this point, as in fig 5.5.

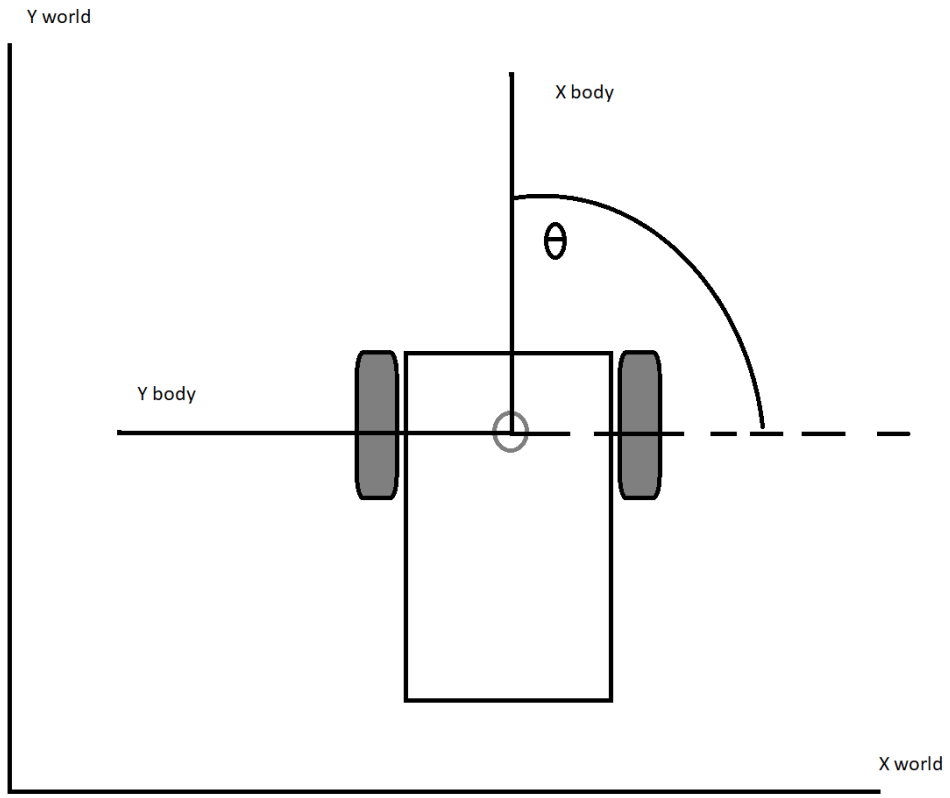


Figure 5.5: Illustration of the coordinate systems

The translations velocity of the point P is given by eq. 5.3. This comes from the effect that the wheel rotation speed also contribute to the rotation of the robot, like in the case where $\dot{\phi_r} = -\dot{\phi_r}$ when the robot spin around it own axis and the translational velocity is zero.

$$\dot{x_b} = \frac{1}{2} d\dot{\phi_r} + \frac{1}{2} d\dot{\phi_l} \tag{5.3}$$

The rate of rotation is given by eq. 5.4.

$$\dot{\Theta_b} = \frac{d\dot{\phi_r}}{2l} - \frac{d\dot{\phi_l}}{2l} \tag{5.4}$$

Since it is impossible for the robot to move in the y_b-direction, the system is

modelled as in eq. 5.5.

$$\begin{bmatrix} \dot{x}_b \\ \dot{y}_b \\ \dot{\Theta}_b \end{bmatrix} = \begin{bmatrix} \frac{1}{2}d\dot{\phi}_r + \frac{1}{2}d\dot{\phi}_l \\ 0 \\ \frac{d\dot{\phi}_r}{2l} - \frac{d\dot{\phi}_l}{2l} \end{bmatrix} = \begin{bmatrix} \frac{d}{2} & \frac{d}{2} \\ 0 & 0 \\ \frac{d}{2l} & -\frac{d}{2l} \end{bmatrix} \begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} \tag{5.5}$$

5.3.2 State estimator

The states to be estimated is x_b and Θ_b, based upon the measurement from the IMU, encoders, and the pose from ORB SLAM. To solve this a weighted sum of the different measurements is used, eq, where $a + b + c = 1$.

$$\hat{\dot{x}}_b = a_x \int \ddot{x}_{acc} dt + b_x(\frac{1}{2}d\dot{\phi}_r + \frac{1}{2}d\dot{\phi}_l) + c_x \dot{x}_{SLAM} \tag{5.6a}$$

$$\hat{\dot{\Theta}}_b = a_\theta \dot{\theta}_{gyro} + b_\theta \frac{d\dot{\phi}_r}{2l} - \frac{d\dot{\phi}_l}{2l} + c_\theta \dot{\theta}_{SLAM} \tag{5.6b}$$

5.3.3 Controller

The setpoints sent to the controller is of the type $\dot{X}_{ref}$ and $\dot{\Theta}_{ref}$, i.e. body frame velocity and rotation rate. Using the eq. 5.5, the state variables are a function of the rotational velocity of the wheels. By inverting the matrix and removing $\dot{y}_b$, the equation can be transformed to eq. 5.7.

$$\begin{bmatrix} \dot{\phi}_r \\ \dot{\phi}_l \end{bmatrix} = \begin{bmatrix} \frac{1}{d} & \frac{l}{d} \\ \frac{1}{d} & -\frac{l}{d} \end{bmatrix} \begin{bmatrix} \dot{x}_b \\ \dot{\Theta}_b \end{bmatrix} \tag{5.7}$$

This equation can then be used to convert $\dot{x}_{ref}$ and $\dot{\Theta}_{ref}$ to rotational velocity setpoints of the motors. What remains is to control the rotation of the of the motors, which is done by a standard PI controller, eq. 5.8.

$$u = K_p(\dot{\phi_{ref}} - \dot{\phi}) + K_i \int (\dot{\phi_{ref}} - \dot{\phi}) dt \tag{5.8}$$

5.4 ORB SLAM

5.4.1 ORB-SLAM-2

The ORB-SLAM software is developed and built upon the work done in [10]. The ROS package used in this project is called orb_slam_2_ros, [11]. The orb_slam_2 software is an expansion of the algorithm discussed in the background theory,

subsection 2.4.1. While the ORB SLAM algorithm is designed to use monocular cameras, the orb_slam_2 software has been expanded to use stereo cameras and other depth cameras like the time of flight camera. The major difference is the position and depth of the map points is estimated based on both the image and the depth image.

The advantage of using this ROS package is that it is fully compatible with ROS and all of the input and output go via ROS topics, which is ideal since the rest of the software is implemented in ROS. The data on the topics are as follows:

Published

- Live image containing the currently found key points and a status message
- A PointCloud2 (ROS datatype) which contain all the key points in the map
- A tf from pointcloud frame to the camera frame, i.e the pose of the camera.

Subscribed

- Topic /camera/rgb/image raw for the intensity image
- Topic /camera/depth_registred/image_raw for the depth image

In addition to feeding the correct data into the program, a configuration file for the camera and an appropriate ROS launch file has to be generated. The configuration file contain some important parameters like the number of features that should be extracted, the scale factor and some threshold values, as well as the calibration parameters of the camera. The launch file is just a file to start up the program, point to the correct configuration file, and set up the correct topics. More info on this in Appendix A

5.4.2 Pico Flexx software

To make use of the Pico Flexx camera, and be able to communicate with it, we make use of "royale SDK" which is the code and software included with the camera. However this software isn't compatible with ROS in any way, so we make use of a ROS wrapper to connect the data stream to the ROS network [27]. The data and topics used in the ROS wrapper are as follows:

- Depth image, topic: /pico_flexx/image_depth, is a 32-bit float image where the distance is along the optical axis
- IR image, topic: /pico_flexx/image_mono16 is a 16 Bit intensity image
- Noise image, topic: pico_flexx/image_noise, is a 32 float image, standard deviation of each of the depth pixels
- Point cloud, topic: /pico_flexx/points, generated by the sensor

Along with the data published on the topics the ROS-wrapper also have some services that can be used during runtime:

- Change use cases, subsection 3.3.2
- Set exposure mode, Automatic or Manual
- Set exposure time in manual mode
- Noise filtering by setting max noise

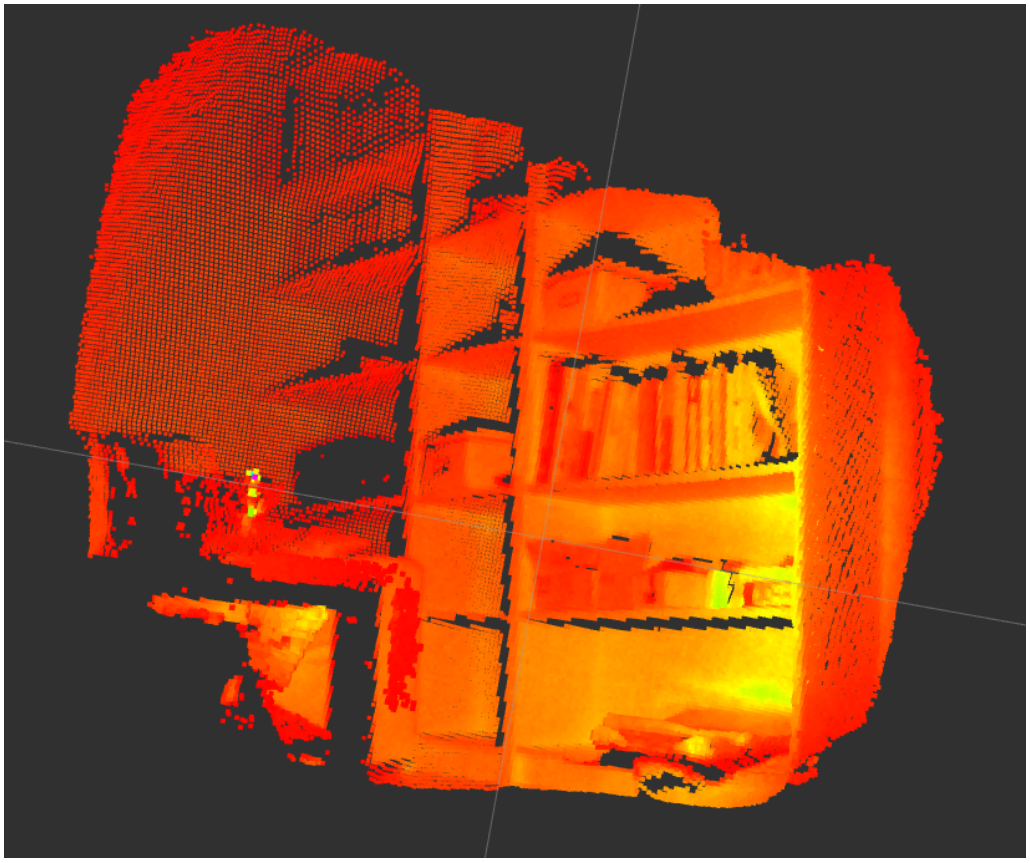


Figure 5.6: Pointcloud in ROS, visualized in RVIZ

5.5 Obstacle detection

There are many was to process the visual data to get information about potential obstacles in the field. One could use the point cloud output directly from the camera, or from the SLAM algorithm or it is possible to process the depth image, the latter is done here.

As seen in fig 4.6b obstacles are relatively easy to spot visually in the depth image, and thus visual processing will be used to find possible obstacles. The

implementation is inspired by segmentation with region growing, discussed back in subsection 2.3.4.

Obstacle detection algorithm

```
Data: Depth image
Result: Segments of obstacles
set segment threshold;
set maximal range;
while there are unprocessed pixels do
    if pixel within max range then
        create new segment;
        while there are pixels in segment with unchecked neighbours do
            for Each unprocessed neighboring pixel do
                if within max range then
                    if pixel value within segment threshold then
                        add pixel to segment;
                    end
                else
                    mark pixel as processed;
                    continue;
                end
            end
            mark neighbors of pixel as checked;
        end
    else
        mark pixel as processed;
        continue;
    end
end
```

Algorithm 1: Finding segments in depth picture

The algorithm works by setting some parameters first. The maximal range is a threshold for which distances it should be created segments. From a obstacle avoidance view the primary focus should be on the objects that are close to the robot, and skip processing pixels which has a distance value above the maximal distance threshold.

The segment threshold is used to decide which pixels that are going to get included in the segment. The threshold is used as an upper and lower limit around the first pixel value added to that segment, i.e. if a pixel value is larger or smaller than this value, it won't be added to the segment.

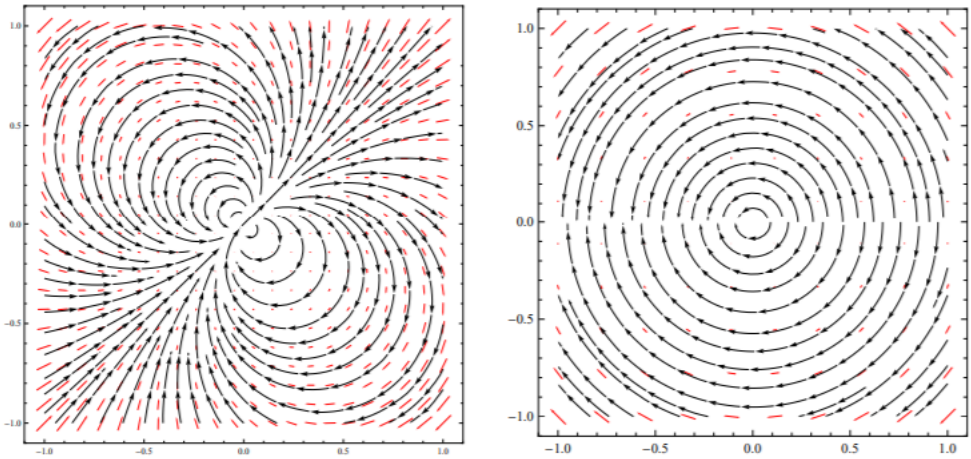
After all the pixels are processed there are found N segments, which represent obstacles in the depth image. The segment object has a set of variables that describe the object it is representing, and these are the position of the leftmost and rightmost pixels, the position of the center in the segment as well as the depth. From a obstacle avoidance perspective the only necessary information needed is the position of the obstacle and the size of the obstacle, which these four variables describes. The position of the obstacle relative to the robot is

found by using the inverse of the intrinsic matrix, described in 2.3.2, where now the known values are the pixel positions u and v, as well as the depth Z_c relative to the camera.

5.6 Obstacle avoidance

The obstacle avoidance system is based upon the work done in [25]. The idea is to use a vector field to guide the robot to the target, as well as around any obstacles.

The vector field is defined in eq. 5.9. Where $\boldsymbol{r}$ is the position of the robot in a coordinate system where the origin is at the target. The vector $\boldsymbol{p}$ is the direction of the field as described in fig. 5.7



(a) $\lambda = 2$, $p_x = 1$, $p_y = 1$. (b) $\lambda = 1$, $p_x = 1$, $p_y = 0$.

Figure 5.7: The field F for different values of λ, [25]

$$F(r) = \lambda(\boldsymbol{p}^T\boldsymbol{r})\boldsymbol{r} - \boldsymbol{p}(\boldsymbol{r}^T\boldsymbol{r}) \tag{5.9}$$

The idea is to combine different vector fields, and in which area they have effect on the robot. There is one *global field* that pull the robot in the direction of the position the robot is wanted to go to, and this field is dominant everywhere. Then there is smaller local fields around the obstacles, similar to that of in fig 5.8. The local field around obstacles is divided into two. The bottom part, where $\Lambda = 1$ and the top part where $\Lambda = 0$. The bottom part is meant to push the robot around the obstacle, while the top part pushes the robot in the direction of position target.

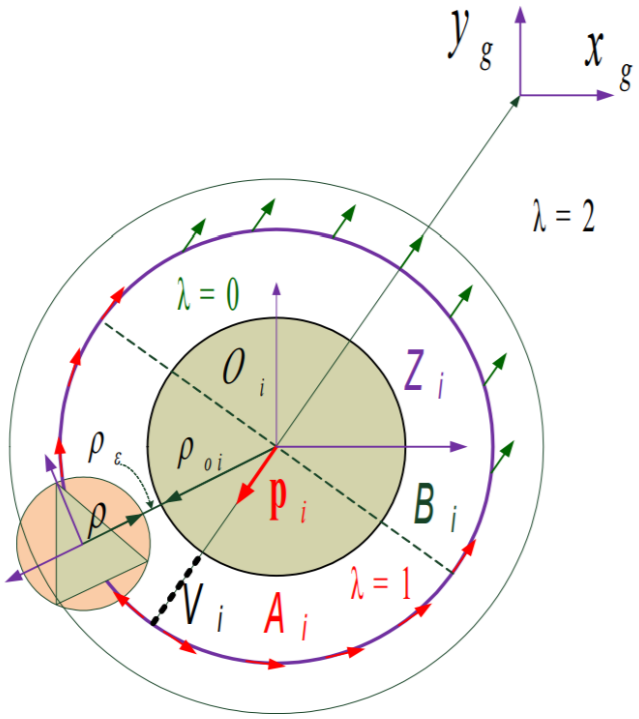


Figure 5.8: Example of the vector field around an object, [25]

All of the different vector fields are summed together and scaled by σ_i which depend on the position of the robot. If the robot is within the field of obstacle i, $\sigma_i = 1$, otherwise zero. The summation is described in eq. 5.10, where N is the number of obstacles, F_g is the global field and F_i is the field around obstacle i.

$$F^* = \prod_{i=1}^{N} \sigma_i F_g + \sum_{i=1}^{N} (1 - \sigma_i) F_i \tag{5.10}$$

The vector F^* is updated whenever there is a change in position of the robot, obstacle or if there is a new destination. This vector is the input to the controller defined in eq. 5.11, where u is the forward velocity input, ω is the rate of rotation, Θ is the heading and $\varphi := arctan(F_y^*, F_x^*)$, the components in the vector field.

$$u = K_u tanh(x^2 + y^2) \tag{5.11a}$$

$$\omega = -K_\omega(\Theta - \varphi) + \dot{\varphi} \tag{5.11b}$$

5.7 Tracking

To help the tracking of a object we make use of a predefined simple pattern, fig 5.9 and search for this pattern in the image. To speed up the process the

segments found in the obstacle detection algorithm is used, which reduces the search space.

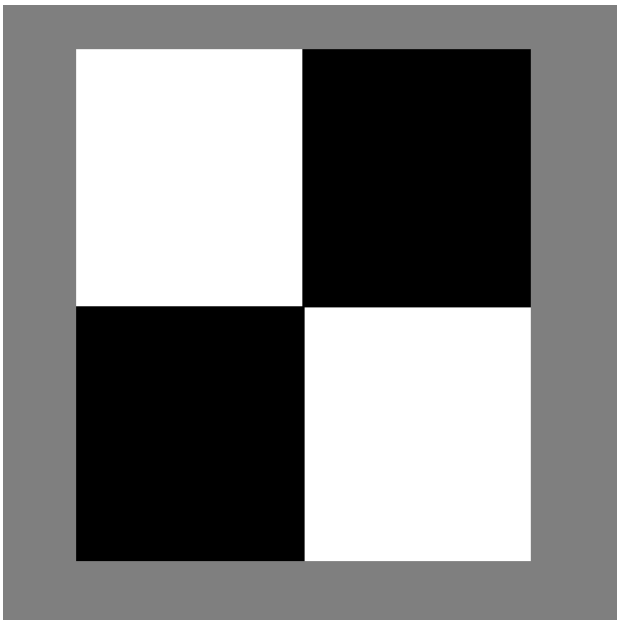


Figure 5.9: Trackable pattern

To find the pattern as sliding window method is used over the search area. The window is on the form given in eq. 5.12. Each of these entries in the matrix are multiplied with the corresponding pixel value and summed up. To reduce computation a only the diagonals in the matrix are used in the actual implementation. If the window is right above the pattern in fig 5.9 the sum will be a large value. A match is found when the sum is above a given threshold. In the case of several matches found, the one with highest value will be chosen.

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{bmatrix} \tag{5.12}$$

The size of the window is dependent on the scale. Since the pattern is a fixed size in the real world, it would be large in the image if it is close, and small if it is far away. To avoid searching all the image with different scales, the depth information is used to set the size of the window.

When the pattern first has been found it is possible to reduce the search area even further. By the assumption that the pattern will not move much in between each picture, means that the search area can be reduced by searching for the pattern around the area where the pattern was found recently.

Chapter 6

System testing

6.1 ORB-SLAM

6.1.1 Simulator

To test ORB SLAM with the simulated camera in gazebo the environment described in subsection 4.4 is used. The robot follows a set of predefined points in the environment, using the vector field discussed in 5.6. Since the ROS plugin is used with the simulator the actual position of the robot is published on the ROS network. The pose estimate from the ORB SLAM package is also published on the network, which are then plotted together in fig. 6.1.

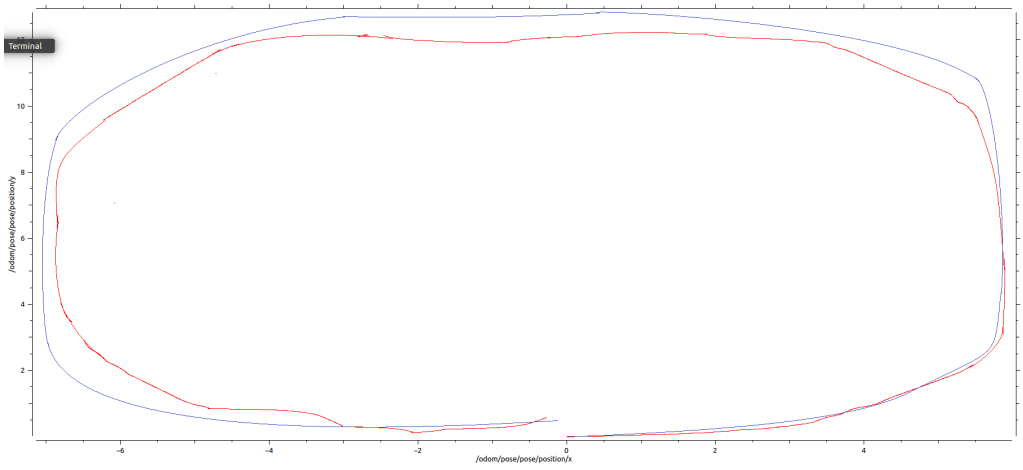


Figure 6.1: The red line is the ORB estimate while the blue line is the ground truth position of the robot

As seen in the figure the estimate and the ground truth is quite close in the beginning, and start to diverge a bit after the first turn, and significantly after the second turn, but stays somewhat constant with a slight decrease in error along the straight paths. At the end a loop closure is detected, and the two converge

back together at the end of the path.

6.1.2 Robot

Like in the case with the simulator we would like to have a ground truth to compare the estimate of the ORB SLAM algorithm. To do this we make use of "OptiTrack" which is a motion capture system [28]. With millimeter precision it serves well a credible source of the actual position an orientation of the robot.

The robot is programmed to follow a simple square path, similar to that used in the simulator. The configurations are the same and the pico flexx camera is put into mode 2, which is intended for indoor navigation, described in 3.3.2.

Before the ORB SLAM start in need to calibrate, however the algorithm had some issues finding enough feature points in the picture. By increasing the threshold value of the ORB extraction it would finally initialize. Another setting that improved the feature detection was increasing the integration time. The intensity pictures are really dark sometimes and increasing this value helps the images become brighter.

The error of the estimates plotted in fig 6.3, where the total position error is defined as $\sqrt{(x_{orb} - x_{true})^2} + \sqrt{(y_{orb} - y_{true})^2}$ and the heading error is defined as $\sqrt{(\Theta_{orb} - \Theta_{true})^2}$.

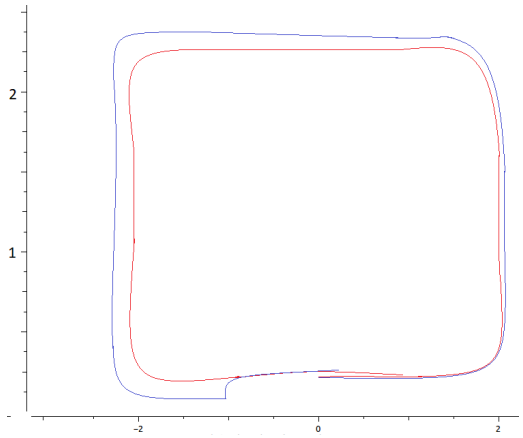
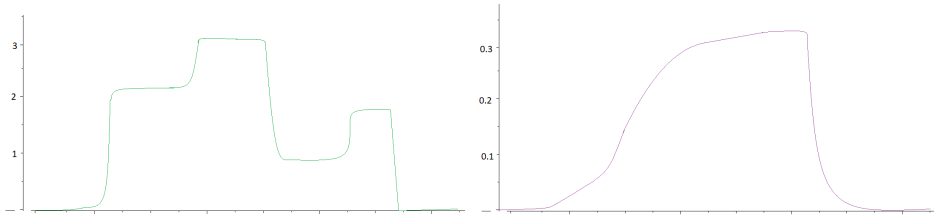


Figure 6.2: ORB SLAM estimate and ground truth



(a) Error of the heading Θ (b) Error of the position

Figure 6.3

The error seem to be increasing linearly with the distance, but as seen in fig. 6.3a the heading estimate is off whenever the robot is making a turn. This results in an increased error of the position since the path is rotated by the error in the heading. At the end of the loop ORB SLAM closes the loop which pulls the errors downward to zero.

6.2 Obstacle avoidance test

To test the obstacle avoidance implementation a set of obstacles are set in a straight hallway. The robot has a predefined destination, marked as x, behind the obstacles, and the only task is to avoid the obstacles and get to the target.

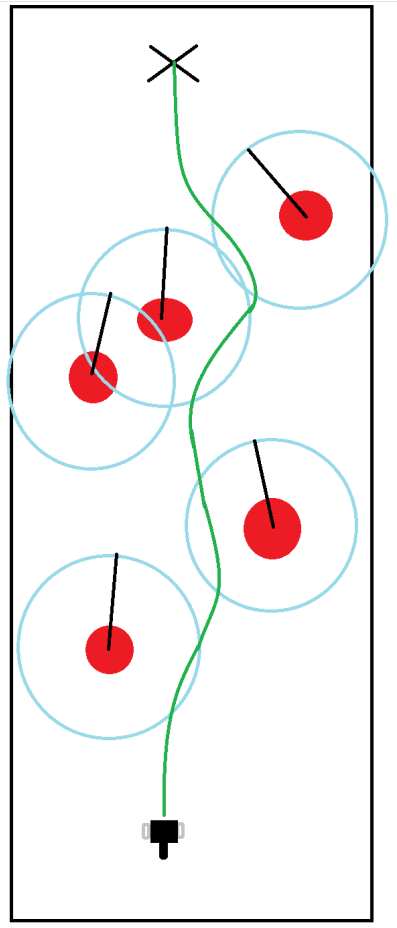


Figure 6.4: Description of the obstacles, the fields around them, and the path taken.

The picture above show a figure of the test environment, and path of the robot through this environment. The blue circles around the obstacles are to represent the area which the local field comes into effect. The black lines describe the direction of the obstacle field, all pointing in the direction of the target position. The robot is able to detect all the obstacles and successfully avoid all of the on

the path to the target.

6.3 Tracking

To test the tracking system, one simple test were conducted. The pattern was printed out on a paper, and the detection of the pattern was tested at different ranges. The robot was able to find and track a person holding the paper, walking in front of the robot. The range was somewhat limited, and at the border of the maximal range of the sensor, 3.3.2, it had a hard time detecting the pattern at all.

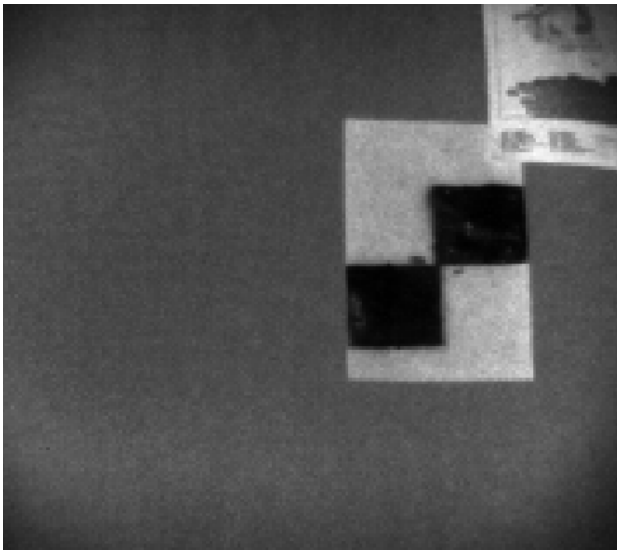


Figure 6.5: The pattern is clearly visible in the picture up to a distance of 3-4 meters

Chapter 7

Discussion

7.1 Discussion on tests

7.1.1 ToF and SLAM

The time of flight camera seems to work quite well under certain conditions. The error of the pose increase with time, but it is the orientation estimate that has the most influence on the error. As the error in heading increases, so will the error of the position of the robot. This problem is seen often in SLAM, especially the problem of orientation as it also will effect the position estimate.

In the simulation the problem with error in the heading occur, but in a much smaller degree than in the real life test. The effect is seen in the upper right corner of fig. 6.1, where there is a error during the turn, but after the turn the error in heading seem to be small since the error in position doesn't increase. There may be several reason for this, SLAM algorithm are a bit sensitive to rotations.

During the test of robot there were some issues getting ORB SLAM initialized, a process that needs feature points to decide which model to use, discussed in 2.4.1. The general experience is that ORB SLAM has a bit more trouble finding features in the IR-image of the Pico flexx camera compared to the image in the simulation. This might effect the performance of the algorithm and could explain the difference in the plots of the simulation and the actual real world.

The Pico flexx works quite well for navigation and as shown in the obstacle avoidance test the robot are able to navigate around obstacles and find its way to the target. However it is dependant on features in the image, and because of the relative short range, these has to be close to the camera. It also require that the environment is reflective so that enough IR light is returned to the camera. If the environment is black (not dark), is would have trouble finding any features at all because of the low amount of returned IR light. This problem could be reduced by using ToF cameras with a stronger light source, but usually these cameras are considerably larger and heavier than the camera used in this project.

7.1.2 Obstacle avoidance and detection

The segmentation algorithm seems to be working quite well. The robot was successful in detecting and avoiding all of the obstacles during the test.

Some of the drawbacks with the implementation is that it is hard to separate an object in the path and for example a wall along the path. Since segmentation is used, where thresholding is the metric to separate the different segments, a wall in the seen in the image will turn out to be a set of several segments placed in a row. This isn't a problem for the obstacle avoidance system implemented, it doesn't care what the the obstacles are, but it do generate a lot of segments that are unnecessary. Because of this, when a wall is observed in the image, a lot of obstacles are generated in the obstacle avoidance system. This increases computation time, and should be avoided one an embedded system by finding another solution to process walls.

7.1.3 Tracking

The tracking seems to work well within the range of the camera, however when the distance reaches the maximal range of the camera, the intensity of the picture reduces as well as salt and pepper noise start to show up. A possible solution to handle the reduced intensity is to lower the threshold of the summation over the window, but that might also increase the amount of false positives in the area close to the camera where the intensity is still strong. The algorithm do however work well within the range of the camera, and because of the use of the segmented areas found in obstacle avoidance processing time is reduced, which is practical on embedded system where computational power isn't an abundant resource.

7.2 Further work

7.2.1 Improve pose estimation

The robot has a series of sensors, but this data is just weighted by constants based upon the credibility of the sensor information. A better approach would be to implement a model based Kalman filter, model the sensors, as well as the noise. A common solution when using SLAM in robotics is to fuse the pose estimate from the SLAM algorithm in a Kalman filter with the other measurements. Another approach would be to fuse the measurements in the SLAM algorithm itself, which overall could improve the performance of the mapping and localization. It exist SLAM algorithms that make use of IMU-data, but a step up would be to include all the data available.

7.2.2 Other SLAM algorithms

The scope of this thesis was limited to only one SLAM algorithm, an indirect one. However it could be of interest to test ToF camera with different kinds of SLAM algorithm, especially testing with a direct SLAM algorithm and compare the performance. Since features are harder to find at some range, it might happen that direct SLAM algorithms perform better.

7.2.3 Adapt SLAM to ToF

In the project the ToF camera has been used as any other depth camera in ORB SLAM, where the IR image replaces the standard camera image. However as discussed in the background theory about ToF, 2.1, the camera has some interesting properties with the IR image. The Gaussian of the measurement could be estimated based upon the intensity of the pixels in the IR image. In other words, a low value on the IR pixel mean a corresponding uncertain depth estimate. This information could be useful in SLAM algorithms to improve the performance and filter out feature points where the depth is uncertain.

7.2.4 Preprocess the ToF IR image

During the testing ORB SLAM had some struggles finding enough feature points in the IR-image to work properly, which was solved by tweaking some of the hyper parameters of the algorithm. A solution could be to preprocess the image to make the edges in image clearer for the ORB extractor, for example increasing the contrast in the image or use a filter to sharpen the the picture.

7.3 Conclusion

The time of flight camera is a capable sensor in the question of navigation and obstacle detection. It does however have some limitations in regards to range, and is quite dependable upon the amount of features and how reflective the environment is to work properly. Within its range and preferred environment the sensor seems to be a good alternative active visual sensor in regards til its size and light weight.

Bibliography

[1] L. Li, "Time-of-flight camera–an introduction," *Technical white paper*, no. SLOA190B, 2014.

[2] C. D. Mutto, P. Zanuttigh, and G. M. Cortelazzo. (2013) Time-of-flight cameras and microsoft kinecttm. [Online]. Available: http://lttm.dei.unipd.it/nuovo/Papers/ToF-Kinect-book.pdf

[3] C. A. Brombach, "Time-of-flight (tof) depth camera for navigationand mapping," 2018.

[4] O. authors. Pinhole camera model. [Online]. Available: https://openmvg.readthedocs.io/en/latest/openMVG/cameras/cameras/#pinhole-camera-model

[5] Mathworks. What is camera calibration. [Online]. Available: https://se.mathworks.com/help/vision/ug/camera-calibration.html

[6] Segmentation. [Online]. Available: https://slideplayer.com/slide/9306887/

[7] E. Rosten. Fast corner detection. [Online]. Available: https://www.edwardrosten.com/work/fast.html

[8] D. Tyagi. Introduction to orb. [Online]. Available: https://medium.com/software-incubator/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

[9] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary robust independent elementary features," in *European conference on computer vision*. Springer, 2010, pp. 778–792.

[10] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[11] L. Haller. Orb slam 2. [Online]. Available: http://wiki.ros.org/orb_slam2_ros

[12] A. Aqeel. Introduction to arduino uno. [Online]. Available: https://www.theengineeringprojects.com/2018/06/introduction-to-arduino-uno.html

[13] NVIDIA. Jetson nano developer kit. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[14] Jetson nano. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/

[15] Mpu-6050 accelerometer + gyro. [Online]. Available: https://playground.arduino.cc/Main/MPU-6050/

[16] Mpu-6000 and mpu-6050 product specification. [Online]. Available: https://store.invensense.com/datasheets/invensense/MPU-6050_DataSheet_V3%204.pdf

[17] pmdtechnologies. Pico flexx. [Online]. Available: https://pmdtec.com/picofamily/flexx/

[18] Rotary encoder. [Online]. Available: https://en.wikipedia.org/wiki/Rotary_encoder

[19] Dc motor + encoder. [Online]. Available: https://www.banggood.com/6V-210RPM-Encoder-Motor-DC-Gear-Motor-with-Mounting-Bracket-and-Wheel-p-1044064.html?utm_design=41&utm_source=emarsys&utm_medium=Shipoutinform171129&utm_campaign=trigger-emarsys&utm_content=Winna&sc_src=email_2671705&sc_eh=c6bb30944d7c76f41&sc_llid=7298077&sc_lid=104858042&sc_uid=sVgRKuosig&cur_warehouse=CN

[20] Dejan. Arduino dc motor control tutorial – l298n — pwm — h-bridge. [Online]. Available: https://howtomechatronics.com/tutorials/arduino/arduino-dc-motor-control-tutorial-l298n-pwm-h-bridge/

[21] L292. [Online]. Available: https://www.banggood.com/no/10-Pcs-L298N-Dual-H-Bridge-Stepper-Motor-Driver-Board-For-Arduino-p-1054211.html?gmcCountry=NO&currency=NOK&createTmp=1&utm_source=googleshopping&utm_medium=cpc_union&utm_content=2zou&utm_campaign=ssc-no-euw-all-july&ad_id=359014888303&gclid=Cj0KCQjwjrvpBRC0ARIsAFrFuV8cjTinAqG3tR3lLZ9xyn5B32ob8PJxeVvLMXseQEgHCRjprxpHwcB&cur_warehouse=UK

[22] Ros 101: Intro to the robot operating system. [Online]. Available: https://robohub.org/ros-101-intro-to-the-robot-operating-system//

[23] geometry_msgs. [Online]. Available: http://wiki.ros.org/geometry_msgs

[24] [Online]. Available: https://www.vectorstock.com/royalty-free-vector/arrows-buttons-keyboard-vector-2729970

[25] D. Panagou, "Motion planning and collision avoidance using navigation vector fields," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 2513–2518.

[26] Pwm. [Online]. Available: https://www.arduino.cc/en/tutorial/PWM

[27] T. Wiedemeyer. Pmd camboard pico flexx driver. [Online]. Available: https://github.com/code-iai/pico_flexx_driver

[28] Optitrack. [Online]. Available: https://optitrack.com/

Appendices

Appendix A

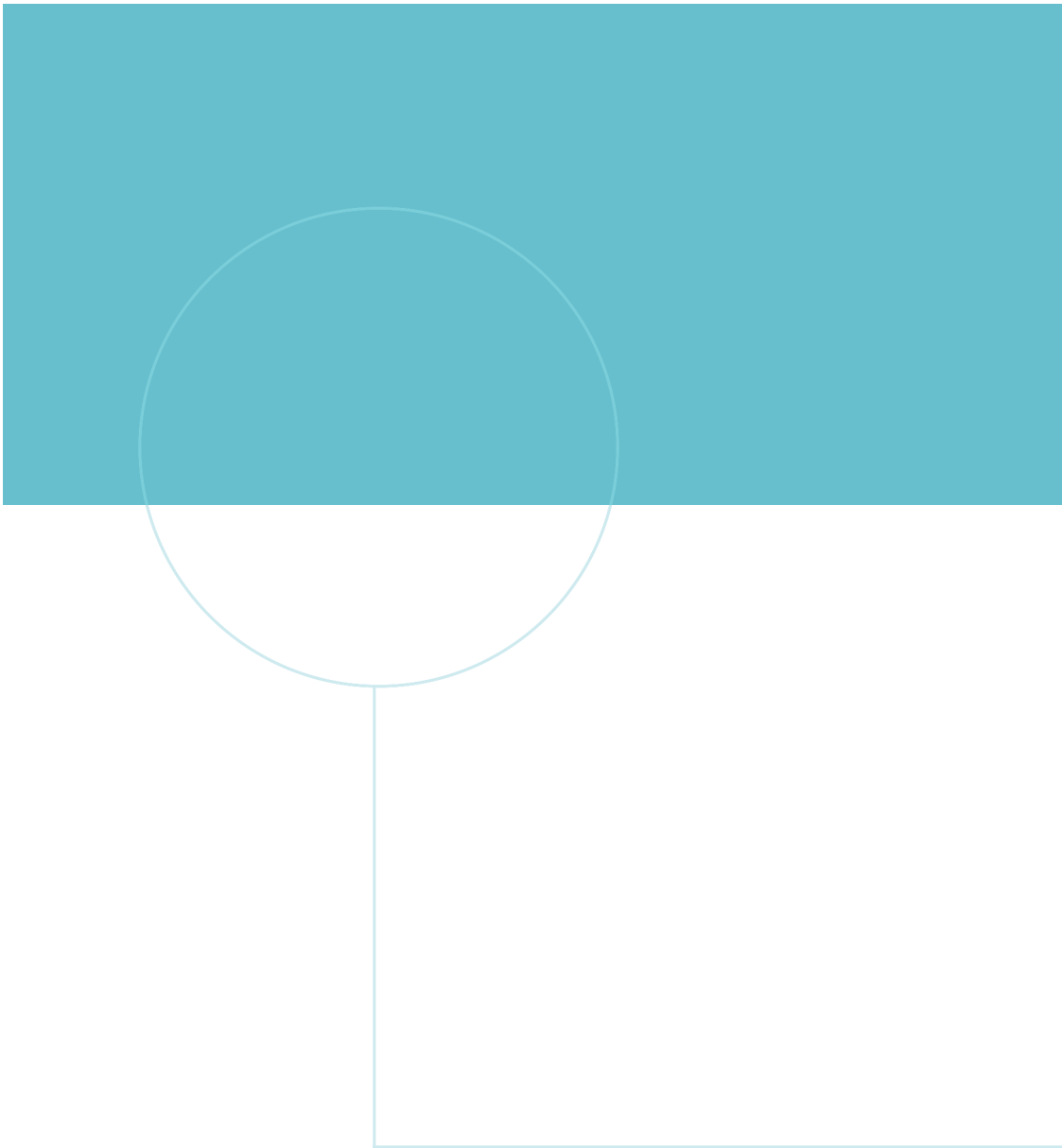
Setting up ORB SLAM

```
<launch>
  <node name="orb_slam2_rgbd" pkg="orb_slam2_ros"
      type="orb_slam2_ros_rgbd" args="
          $(find orb_slam2_ros)/orb_slam2/Vocabulary/ORBvoc.txt
          $(find orb_slam2_ros)/orb_slam2/config/pico_flexx.yaml"
      output="screen">
       <remap from="/camera/rgb/image_raw" to="/tof/tof/ir/image_raw" />
       <remap from="/camera/depth_registered/image_raw" to="/tof/tof/depth/image_raw" />

       <param name="publish_pointcloud" type="bool" value="true" />
       <param name="publish_pose" type="bool" value="true" />
       <param name="localize_only" type="bool" value="false" />
       <param name="reset_map" type="bool" value="false" />
       <param name="pointcloud_frame_id" type="string" value="map" />
       <param name="camera_frame_id" type="string" value="camera_link" />
       <param name="min_num_kf_in_map" type="int" value="5" />
  </node>
</launch>
```

Figure A.1: launch file to start orb slam 2

```
%YAML:1.0

# Camera calibration and distortion parameters (OpenCV)
Camera.fx: 208.02
Camera.fy: 208.02
Camera.cx: 111.29
Camera.cy: 87.18

# Camera distortion parameters (OpenCV)
Camera.k1: 0.8685
Camera.k2: -7.175
Camera.p1: 0.0
Camera.p2: 0.0
Camera.k3: 12.12

Camera.width: 224
Camera.height: 171

# IR projector baseline times fx (aprox.)
Camera.bf: 0.0

# Camera frames per second
Camera.fps: 20.0

# Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale)
Camera.RGB: 1

# Close/Far threshold. Baseline times.
ThDepth: 50.0

# Deptmap values factor (what pixel value in the depth image corresponds to 1m?)
DepthMapFactor: 1.0

#--------------------------------------------------------------------------------------------
# ORB Parameters
#--------------------------------------------------------------------------------------------

# ORB Extractor: Number of features per image
ORBextractor.nFeatures: 500

# ORB Extractor: Scale factor between levels in the scale pyramid
ORBextractor.scaleFactor: 1.2

# ORB Extractor: Number of levels in the scale pyramid
ORBextractor.nLevels: 8

# ORB Extractor: Fast threshold
# Image is divided in a grid. At each cell FAST are extracted imposing a minimum response.
# Firstly we impose iniThFAST. If no corners are detected we impose a lower value minThFAST
# You can lower these values if your images have low contrast
ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7
```

Figure A.2: Pico flexx config file



NTNU
Norwegian University of
Science and Technology