UiA Faculty of Engineering and Science

Collaborative SLAM using a swarm intelligence-inspired exploration method

Øystein Eiane
Jakob Einarssønn Lunde
Tor André Andersen Paulsen

Supervisor
Morten Rudolfsen, UIA

University of Agder, 8th June 2020
Faculty of Engineering and Science
Department of Engineering Science

Abstract

Efficient exploration in multi-robot SLAM is a challenging task. This thesis describes the design of algorithms that would enable Loomo robots to collaboratively explore an unknown environment. A pose graph-based SLAM algorithm using the on-board sensors of the Loomo was developed from scratch. A YOLOv3-tiny neural network has been trained to recognize other Loomos, and an exploration simulation has been developed to test exploration methods. The bots in the simulation are controlled using swarm intelligence inspired rules. The system is not finished, and further work is needed to combine the work done in the thesis into a collaborative SLAM system that runs on the Loomo robots.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface. 57

BRIEF Binary Robust Independent Elementary Features. 32, 33

CNN Convolutional Neural Network. 1, 2, 25, 49–52, 54, 74, 114
CSV Comma-Separated Values. 57, 58

DOF Degrees of Freedom. 14, 15
DoG Difference of Gaussians. 30

EKF Extended Kalman Filter. 15

FAST Features from accelerated segment test. 29, 30, 32
FoV Field of View. 63, 74, 111, 114
FPS Frames Per Second. 57, 58, 114

GPU Graphics Processing Unit. 112
GUI Graphical User Interface. 63

ICP Iterative Closest Point. 111
IMU Inertial Measurement Unit. 59
IoU Intersection of Union. 54

mAP Mean Average Precision. 53, 55, 78, 80

ORB Oriented FAST and rotated BRIEF. 24, 28, 30, 32–34, 64, 65, 106, 114

RANSAC Random Sample Consensus. 34, 66
ROI Region Of Interest. 86, 87
ROS Robot Operating System. 2, 3

SDK Software Development Kit. 57, 59, 61, 111
SIFT Scale-Invariant Feature Transform. 24, 30–33
SLAM Simultaneous Location And Mapping. 1–3, 14–17, 19, 20, 24, 34, 37, 57, 58, 64, 111, 114
STL Standard Template Library. 57
SURF Speeded Up Robust Features. 24

VO Visual Odometry. 67–70

YOLO You Only Look Once. 51–53, 78, 114

Chapter 1

Introduction

1.1 Motivation

Simultaneous Location And Mapping (SLAM) is a growing and relevant field of research. Increasingly, people are looking into collaborative SLAM, i.e. combining data from multiple agents to perform SLAM. Many of the existing approaches mostly focuses on how to combine data from multiple agents [1, 2, 3, 4, 5]. Less focus has been given to how the agents should collaborate. In the context of robotics and autonomous vehicles, it is also useful for multiple agents to be able to efficiently explore an unknown environment. Efficient collaborative exploration is a challenging problem. Some research proposes methods for coordinating the behaviour of the involved agents [6, 7].

The *boids* algorithm [8] showed that it was possible to mimic complex swarm behaviour with a very simple rule-set. Instead of coordinating the behavior of the *boids*, i.e. the agents, the boids are given a rule-set for how to act based on what they are able to observe. From this rule-set emerges a collective behavior that resembles that of flocking birds. The elegant simplicity of the boids algorithm became part of the motivation for this thesis.

The University of Agder recently acquired 12 Loomo robots by Segway Robotics. These robots have a range of sensors including multiple cameras and a depth camera that makes them suitable for multi-robot SLAM.

When multiple robots work together, it is useful for them to be able to *see* each other. This can be achieved through machine learning. A Convolutional Neural Network (CNN) is suited for detecting objects in images and can enable Loomo robots to *see* each other.

1.2 Thesis Statement

The original goal for this thesis was that several Loomo robots should perform SLAM together, and through swarm intelligence be able to actively explore an unknown environment. In order to achieve this, three main focus areas were identified.

- Building a SLAM algorithm based on the Loomo's on-board sensors
- Training a Convolutional Neural Network (CNN) so the Loomos are able to identify each other
- Developing swarm intelligence, i.e. a simple rule-set that enables collaborative exploration

The original goal went through a significant revision, but the three focus areas remained. The original goal was oriented towards implementing these three focus areas on the Loomo robots. Software development for the Loomo turned out to be a major challenge. Section 1.3 will describe this challenge in more detail. The revised goals shifted the focus away from implementing software on the Loomo and reduced the scope. Though the three focus areas remained, they became more decoupled from each other.

1.2.1 Original Thesis Goals

- Enable communication between Loomos and a host computer using the ROS framework [9]
- Develop a graph-based SLAM algorithm. The SLAM front-end, which processes the sensor data and does intermediate state estimation, and constructs the graph, is to run on the Loomos. The SLAM back-end, which performs graph optimization, is to run on the host computer, and it is also responsible for combining graphs from multiple Loomos.
- Develop a CNN that runs on the Loomos. Loomo detection should be incorporated with both the SLAM algorithm and the swarm behaviour.
- Develop a rule-set for how the Loomos should act. The rule-set can utilize knowledge about the map state provided by SLAM, and observations of other Loomos provided by the CNN. The resulting behaviour should be so that an unknown environment is explored.

1.2.2 Revised Thesis Goals

- The SLAM algorithm should be able to perform SLAM on recorded Loomo data
- The CNN should still be able to run on a Loomo
- The collaborative exploration should be developed through modelling and simulation

1.3 Project Management

Figure 1.1 shows the Gantt diagram for the original thesis goals. The ”Startup“ task was about getting familiarized with the Loomo’s potential. The thesis as it was proposed by the university was open-ended, so the first two weeks were used by the group to settle for what they wanted to achieve.

All the blue bars in the Gantt diagram were considered part of a learning phase. Because of that, the specific tasks were not as strictly defined. Even though the main implementation phase (red bars) did not start before late February, several functionalities were implemented during the learning phase. For instance, the Loomo was made ROS compatible and was able to publish sensor data and video-streams at a low frame rate to a ROS master. Also a simple utility for taking pictures with the on-board RealSense camera was made, which was used to gather training data for Loomo detection, and checkerboard images for camera calibration.

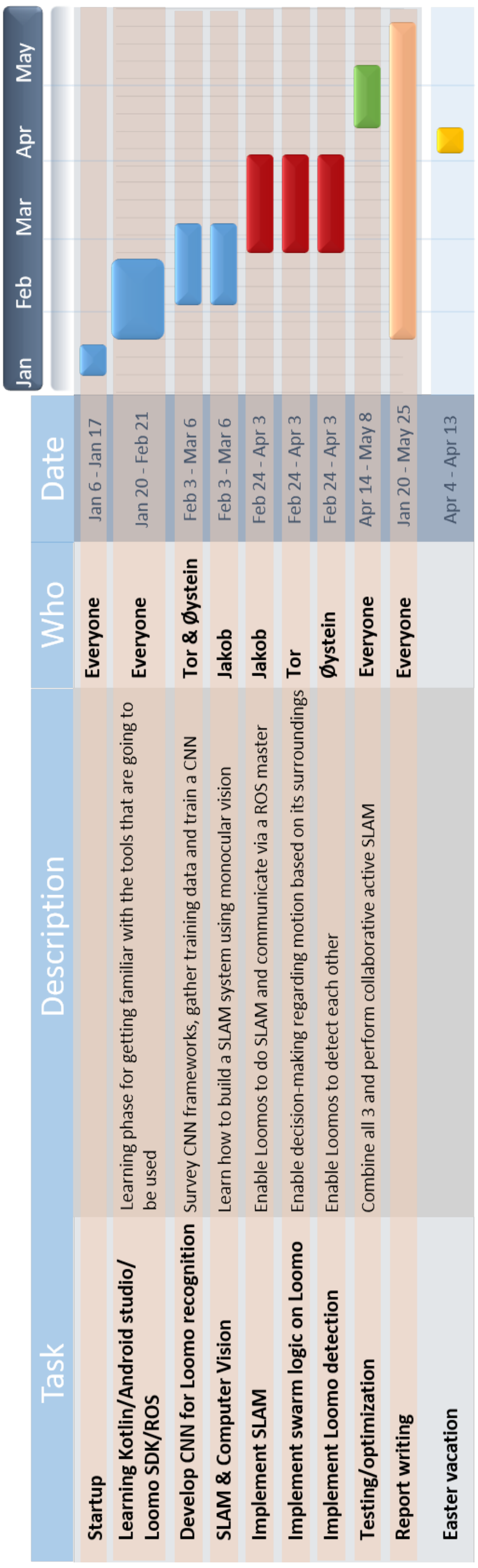
The tasks of the implementation phase reflects the three main focus areas described in the thesis statement. This allowed the group members to work in parallel.

The testing phase (green bar in Gantt chart) is where the three focus areas were to be properly combined.

Early in the implementation phase it became apparent that developing software for the Loomo was becoming a problem. The implementation phase was supposed to be focused on the challenges of SLAM, computer vision, machine learning and swarm behaviour. These were topics that were considered interesting and relevant for the thesis. However, much of our time and focus was dedicated to challenges regarding software. The group had underestimated the difficulty of programming in an Android environment. Thus the group decided to shift their focus away from the Loomo to instead focus on the topics that were considered interesting for the thesis.

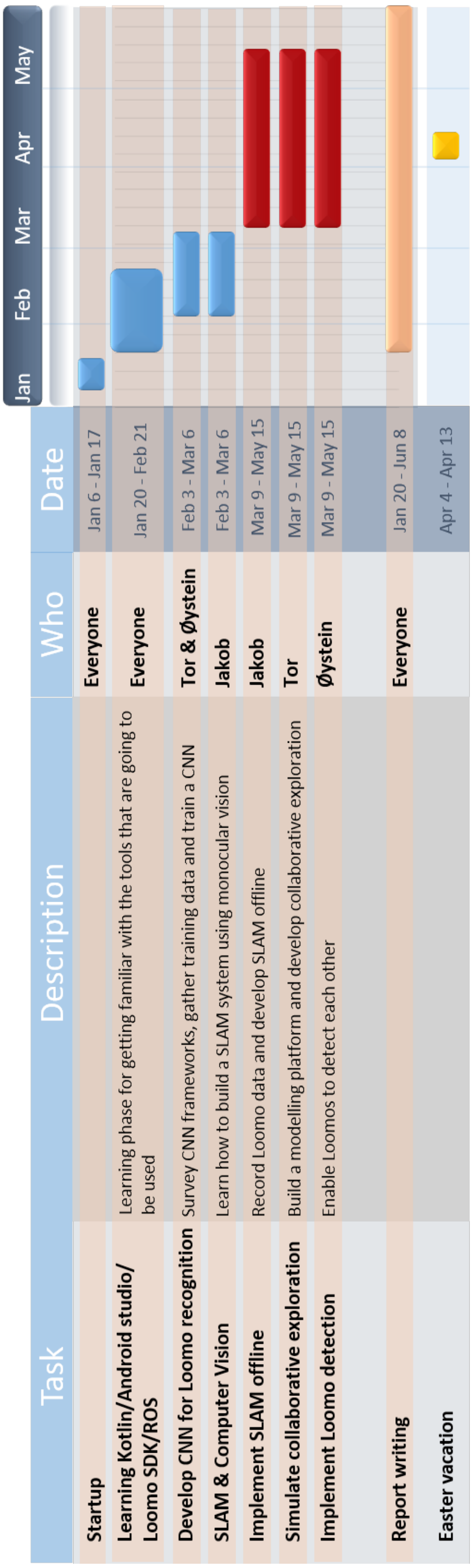
The revision of the thesis goals had most impact on the development of SLAM and collaborative exploration. Developing methods for collaborative exploration is something that is well suited for being solved through modelling and simulation. In hindsight, this is how the problem should have been addressed in the original goals. It was somewhat ambitious to try to develop collaborative exploration directly on a physical system.

The revised Gantt chart is shown in figure 1.2



Task	Description	Who	Date
Startup		Everyone	Jan 6 - Jan 17
Learning Kotlin/Android studio/ Loomo SDK/ROS	Learning phase for getting familiar with the tools that are going to be used	Everyone	Jan 20 - Feb 21
Develop CNN for Loomo recognition	Survey CNN frameworks, gather training data and train a CNN	Tor & Øystein	Feb 3 - Mar 6
SLAM & Computer Vision	Learn how to build a SLAM system using monocular vision	Jakob	Feb 3 - Mar 6
Implement SLAM	Enable Loomos to do SLAM and communicate via a ROS master	Jakob	Feb 24 - Apr 3
Implement swarm logic on Loomo	Enable decision-making regarding motion based on its surroundings	Tor	Feb 24 - Apr 3
Implement Loomo detection	Enable Loomos to detect each other	Øystein	Feb 24 - Apr 3
Testing/optimization	Combine all 3 and perform collaborative active SLAM	Everyone	Apr 14 - May 8
Report writing		Everyone	Jan 20 - May 25
Easter vacation			Apr 4 - Apr 13

Figure 1.1: Gantt diagram for the original thesis goals



Task	Description	Who	Date
Startup		Everyone	Jan 6 - Jan 17
Learning Kotlin/Android studio/ Loomo SDK/ROS	Learning phase for getting familiar with the tools that are going to be used	Everyone	Jan 20 - Feb 21
Develop CNN for Loomo recognition	Survey CNN frameworks, gather training data and train a CNN	Tor & Øystein	Feb 3 - Mar 6
SLAM & Computer Vision	Learn how to build a SLAM system using monocular vision	Jakob	Feb 3 - Mar 6
Implement SLAM offline	Record Loomo data and develop SLAM offline	Jakob	Mar 9 - May 15
Simulate collaborative exploration	Build a modelling platform and develop collaborative exploration	Tor	Mar 9 - May 15
Implement Loomo detection	Enable Loomos to detect each other	Øystein	Mar 9 - May 15
Report writing		Everyone	Jan 20 - Jun 8
Easter vacation			Apr 4 - Apr 13

Figure 1.2: Gantt diagram after the thesis goals were revised

Chapter 2

System

This chapter describe information about the Loomo robot, the sensors used and some useful measurements needed later in the thesis.

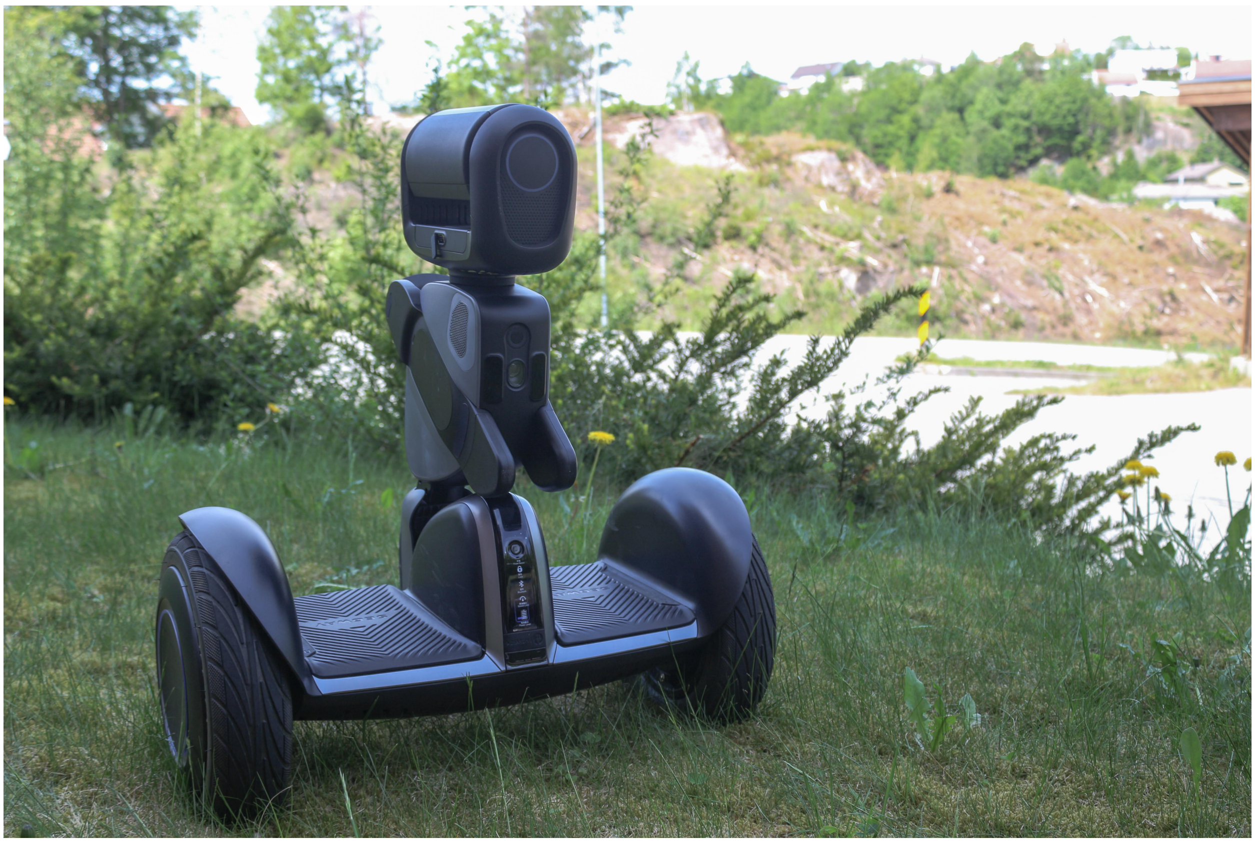


Figure 2.1: Loomo

2.1 Loomo Technical Specification

Parameter	**Description**
Size	650mm height, 310mm length, 570mm width
Weight	∼19kg (∼42 lbs.)
Max Payload	220 lbs
LCD Screen	4.3 inch, 480 * 800 pixels
HD Camera	1080p 30Hz streaming with 104 degrees FOV
3D Camera	Intel RealSense ZR300 Camera for depth-sensing & motion tracking
Sensors Available	Ultrasonic sensors, infrared distance sensors, touch sensors, encoders, IMUs
Processor	Intel Atom Z8750, 4 cores 2.56GHz, x86-64
Operating System	Customized system based on Android 5.1
Memory	4GB
Storage	64GB
Speed Limit	8km/h (4.3mph) in robot mode, 18km/h (11mph) in self-balancing vehicle mode
Typical Range	∼35km (22 miles) per charge
Traversable Terrain	Paved road and sidewalks, packed dirt, slopes $< 15°$, obstacles < 0.4 inch (1cm), gaps < 1.2 inch (3cm)
Mic Array	5 microphones enabling beamforming, voice localization and voice command recognition
Battery	Capacity 329Wh
Waterproof	IPX4

Table 2.1: Loomo technical specification, information taken from: [10]

2.2 Measurements

In a differential drive system, it is important to have an accurate measurement of the base width and the wheel diameter. To make the measurements as accurate as possible, multiple measurements are performed and weighted based on the confidence in the measurements.

The sample standard deviation equation is used to calculate the weight factors

$$\sigma = \sqrt{\frac{1}{N-1}\sum_i (W_i - \hat{W})^2} \tag{2.1}$$

Where

Symbol	Description	Unit
σ	Standard deviation of measurements	m
N	Number of samples	-
W_i	Specific sample	m
$\hat{W}$	Average of samples	m

2.2.1 Base Width

Base width is measured by 3 separate measurements. A laser is used to measure the distance between the inside of each wheel, and a digital caliper is used to measure the width of each wheel. This method is more robust than using a folding rule to measure the distance between the approximate center of each wheel directly. Measurements are performed multiple times to reduce error.

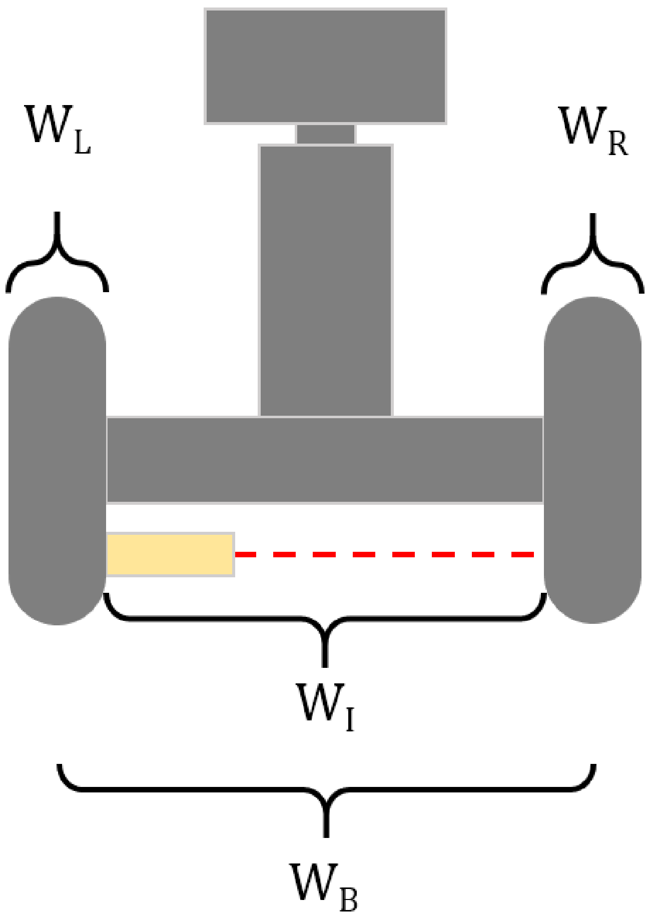


Figure 2.2: Base width measurement

$$C_L = 1 - \frac{\sigma_L}{\sigma_L + \sigma_R} \tag{2.2}$$

$$C_R = 1 - \frac{\sigma_R}{\sigma_L + \sigma_R} \tag{2.3}$$

$$W_B = W_I + 2 \cdot \left(\frac{W_L \cdot C_L}{2} + \frac{W_R \cdot C_R}{2} \right) \tag{2.4}$$

The measurements and calculations are shown in appendix B. The base width is measured to be 488,87mm.

Where

Symbol	Description	Unit
σ_{L}	St.dev left wheel width measurements	m
σ_{R}	St.dev right wheel width measurements	m
C_{L}	Confidence left wheel width measurements	-
C_{R}	Confidence right wheel width measurements	-
W_B	Width of Loomo between center of wheels	m
W_I	Length between the inside of the wheels	m
W_L	Width of left wheel	m
W_R	Width of left wheel	m

2.2.2 Wheel Diameter

Wheel diameter changes with the tire pressure and load on the wheels. This test was performed with the wheels compressed by the weight of the Loomo. Tire pressure was not verified before the test. Loomo is moved the distance it takes for the wheels to complete 2 full rotations. There is a marker fixed to the wheel to indicate the measurement position. The distance traveled is measured by a distance laser mounted on a tripod, and a folding rule laying on the floor. See figure 2.3 and 2.4 for an illustration of the method. It is important that the laser beam is parallel with the floor, this was calibrated prior to the measurements. The laser distance measurement will also be affected by the pitch angle of the Loomo, but since it is in balance mode, the impact is not significant and the measurements will be fairly accurate. Measurements are performed multiple times for each wheel to reduce error.

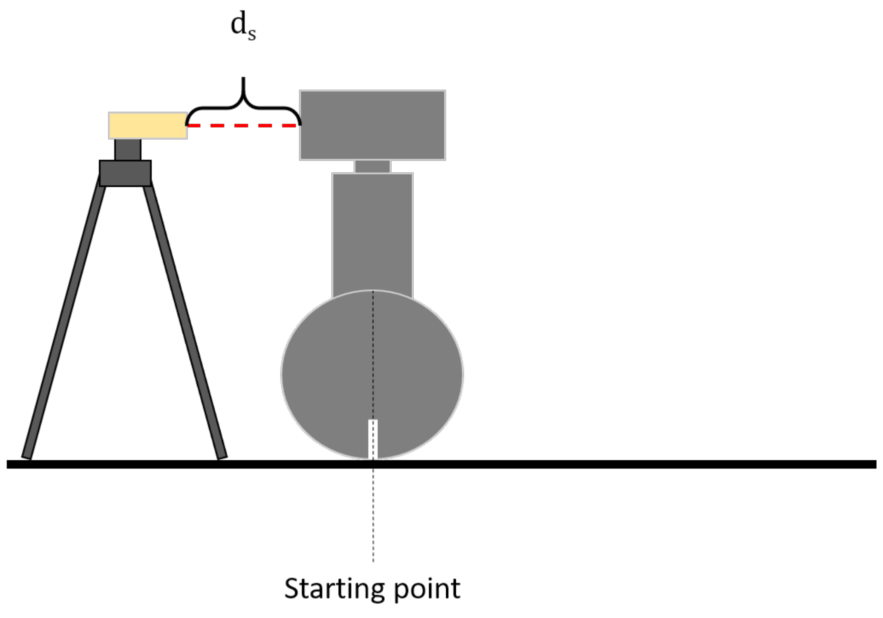


Figure 2.3: Wheel diameter measurement starting point

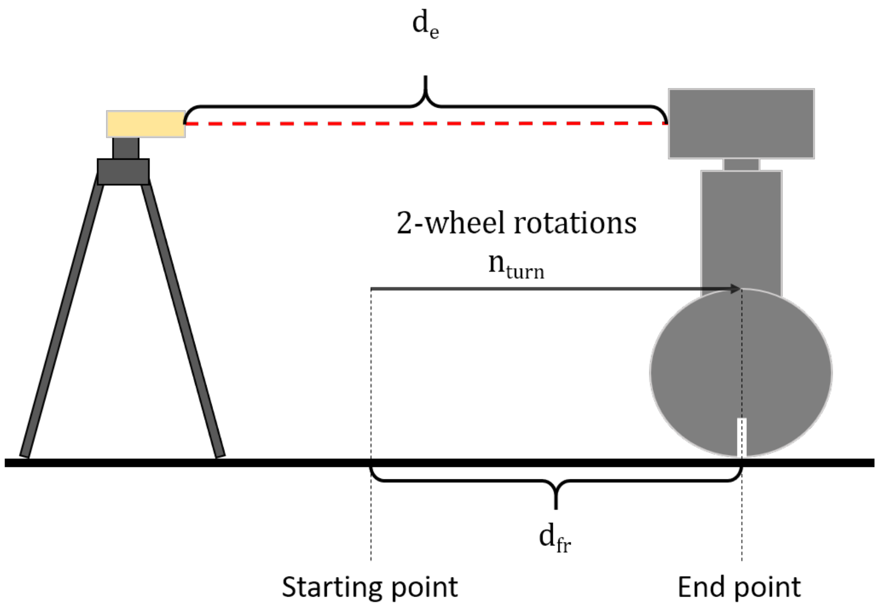


Figure 2.4: Wheel diameter measurement end point

$$d_{\mathrm{l}} = d_e - d_s \tag{2.5}$$

$$C_{\mathrm{l}} = 1 - \frac{\sigma_{\mathrm{l}}}{\sigma_{\mathrm{l}} + \sigma_{\mathrm{fr}}} \tag{2.6}$$

$$C_{\mathrm{fr}} = 1 - \frac{\sigma_{\mathrm{fr}}}{\sigma_{\mathrm{l}} + \sigma_{\mathrm{fr}}} \tag{2.7}$$

$$D_{wheel} = \frac{d_{\mathrm{l}} \cdot C_l + d_{\mathrm{fr}} \cdot C_{\mathrm{fr}}}{n_{\mathrm{turn}} \cdot \pi} \tag{2.8}$$

Where

Symbol	Description	Unit
d_{l}	Distance measured by laser	m
d_e	Distance from laser to Loomo at end point	m
d_s	Distance from laser to Loomo at start point	m
d_{fr}	Distance measured by folding rule	m
σ_{l}	St.dev laser measurements	m
σ_{fr}	St.dev folding rule measurements	m
C_{l}	Confidence laser measurements	-
C_{fr}	Confidence folding rule measurements	-
D_{wheel}	Diameter of wheel	m
n_{turn}	Number of wheel turns	-

The measurements and calculations are shown in appendix B. The wheel diameter was measured to be $270.35mm$

2.3 Sensors

2.3.1 Encoder

The wheels on the Loomo uses hub motors with integrated hall sensor based encoders. The encoder pulse count has been verified by manually turning the wheels 10 turns each direction. The results are 90 pulses per rotation, which is in accordance with the value given in the Loomo API. The measurements and calculations are shown in appendix B

2.3.2 Distance Sensors

There are 3 forward facing distance sensors (not counting the depth camera). One Ultrasonic sensor, and two Infrared sensors. Figure 2.5 illustrate an approximation of the detector field of the sensors. The center cone being the ultrasonic sensor, and the two cones on the left and the right side is the Infrared sensors. The infrared sensors are also pointing downward to enable detection of for example stairwells. The output from the Ultrasonic and infrared sensors are already converted to mm in the Loomo API. These values have been verified with a laser.

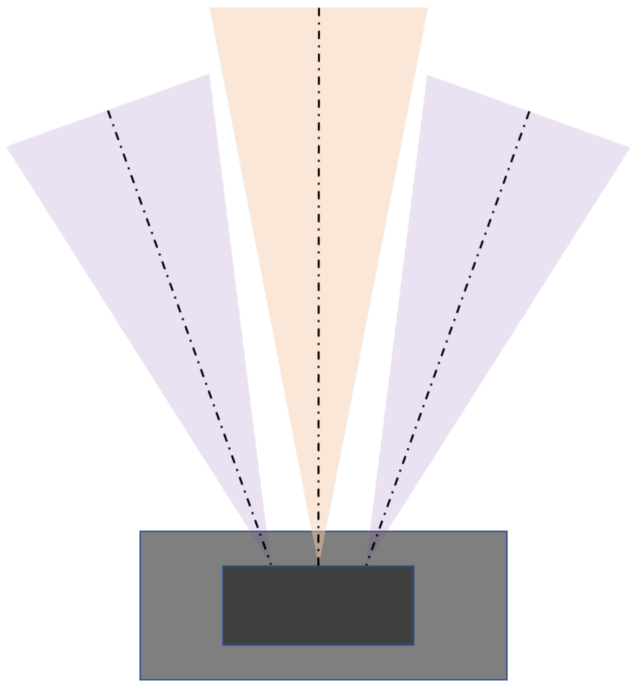


Figure 2.5: IR and Ultrasonic FOV

2.3.3 Intel RealSense Camera

The Intel RealSense ZR300 camera implements a stereo vision depth imaging, color camera, and a fisheye camera into a single module. [11, p. 8]

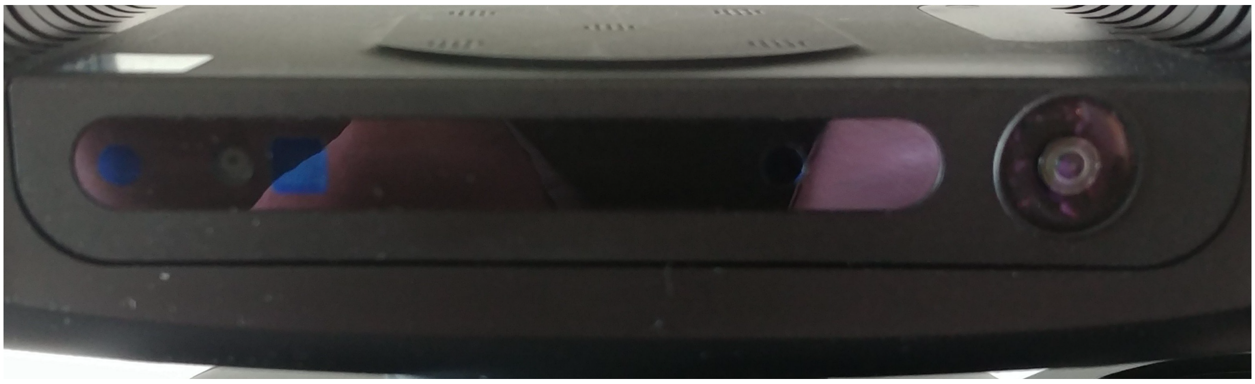


Figure 2.6: Intel RealSense

Component	**Diagonal**	**Vertical**	**Horizontal**
IR Laser Projector FOP	$80° \pm 5\%$	$60° \pm 5\%$	$60° \pm 5\%$
Infrared Camera FOV	$70° \pm 5\%$	$46° \pm 5\%$	$59° \pm 5\%$
Color Camera FOV	$75° \pm 4\%$	$41.5° \pm 2\%$	$68° \pm 2\%$
Fisheye Camera FOV	$166.5° \pm 4\%$	$100° \pm 3\%$	$133° \pm 3\%$

Table 2.2: Realsense FOV: [11, p. 14]

2.3.4 Depth Camera

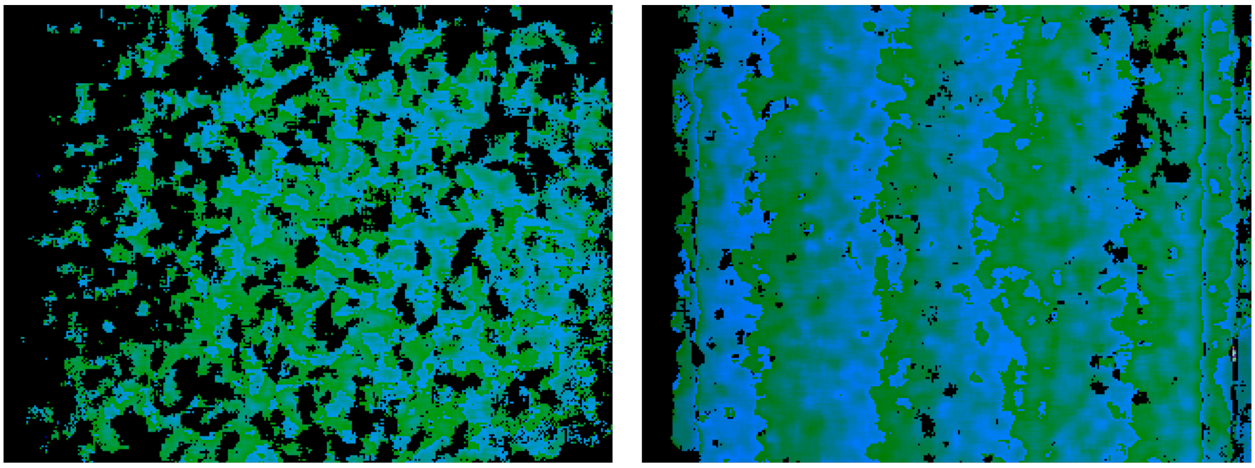
The depth camera is part of the Intel RealSense sensor unit. The IR projector illuminates the environment in front of the Loomo, and the stereo IR cameras observe these projections to extract depth information from the environment. The pixels in the resulting image holds information about the distance to that position in the image from the plane parallel to the camera cover. The depth data has been verified by reading the depth value of specific points in a depth image, and then measuring the distance from the camera to that point in the environment with a laser. The measurements is documented in appendix B.

Interference

This section show some examples of interference that has been observed in the depth camera images during testing.

When there is low variation in the depth, for example when the camera is directly facing a flat wall, there seems to be increased interference in the depth images. Figure 2.7 show an example of this effect.

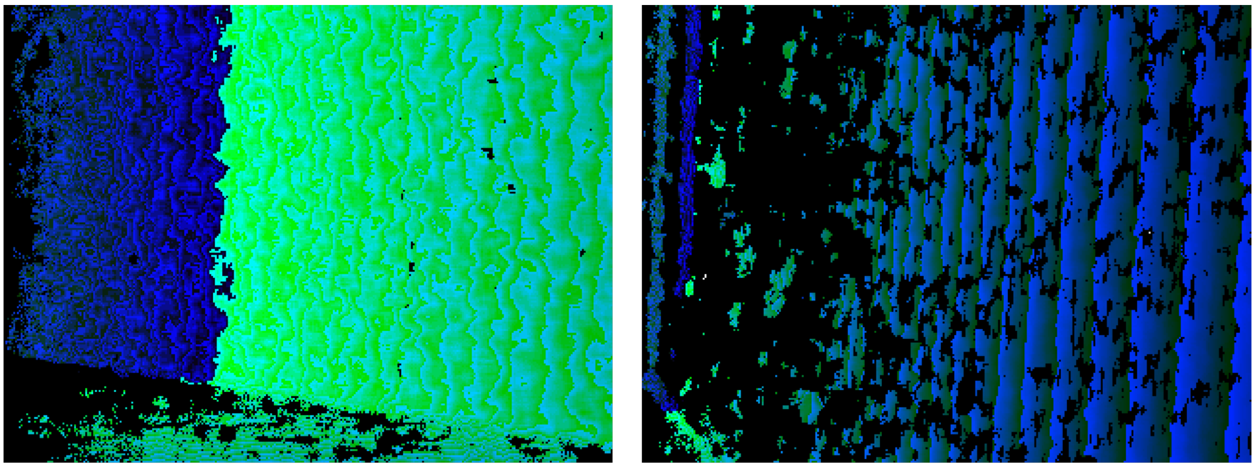
Sunlight will interfere with the function of the depth camera. See figure 2.8 for an indoor vs outdoor comparison.



Depth camera facing a flat surface

Depth camera slightly angled towards flat surface

Figure 2.7: Depth camera interference example: low depth variation



Indoor: Artificial Light

Outdoor: Sunlight

Figure 2.8: Comparing interference indoor vs outdoor

Depth vs range data

The depth camera calculates the depth distance to objects, not range. See figure 2.9 for an illustration. Depth means the distance from an object to the parallel plane in front of the stereo cameras. [11, p. 16]

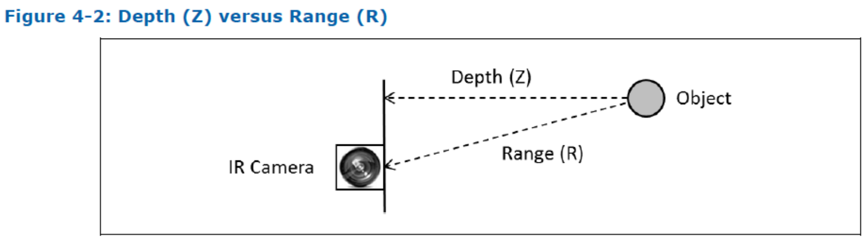


Figure 2.9: Depth vs range [11, p. 16]

However, by inspecting figure 2.10, which is a depth camera image of stairwell, there is clearly some error in depth data. Observe in the right image that the wall on the right side has a curvature, which is not the case in the real stairwell. A hypothesis is that there is some numerical error in the conversion from range to depth in the Intel RealSense ASIC, and the curvature which is observed is an effect from that numeric error.

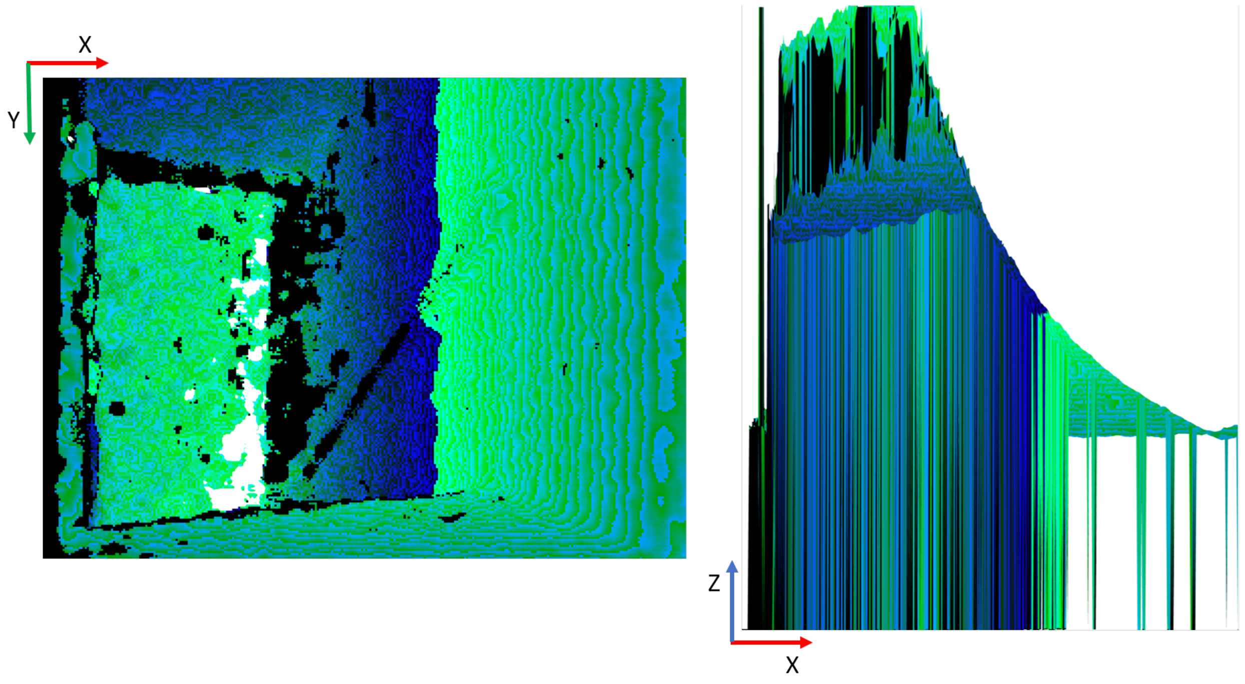


Figure 2.10: Depth image example

Chapter 3

Theory

3.1 Simultaneous Localization And Mapping (SLAM)

Simultaneous Location And Mapping (SLAM) is the problem of constructing a map of an unknown environment while simultaneously locating the agent in that map. Probabilistic formulations of the SLAM problem comes in two main forms: the full SLAM problem, and, the online SLAM problem.

3.1.1 Overview of SLAM

The full SLAM problem is concerned with estimating the probability distribution of the robot's trajectory and the map given a set of observations and inputs. The full SLAM problem is expressed in equation 3.1.

$$p\left(\mathbf{x}_{0:T},\ m \mid z_{1:T},\ u_{1:T}\right) \tag{3.1}$$

The trajectory, $\mathbf{x}_{0:T}$, is the concatenation of the robot's states at time 0 through time T (see equation 3.2). The robot's state is typically represented with 3DOF or 6DOF depending on whether the robot is moving in 2D or 3D space. The inputs, $u_{1:T}$, are often referred to as commands or controls. In practice it is common to use odometry as input.

$$\mathbf{x}_{0:T}^{\top} = \left[\mathbf{x}_0^{\top},\ \mathbf{x}_1^{\top},\ \cdots,\ \mathbf{x}_{T-1}^{\top},\ \mathbf{x}_T^{\top}\right] \tag{3.2}$$

$$\mathbf{x}_n^{\top} = [x_n, y_n, \theta_n] \quad or \quad \mathbf{x}_n^{\top} = [x_n, y_n, z_n, \rho_n, \phi_n, \theta_n] \tag{3.3}$$

Based on its formulation, the full SLAM problem is not suited for online applications because each new input or observation requires a recalculation of the entire trajectory and map. The online SLAM problem seeks to recover only the most recent pose, and is expressed by equation 3.4. Algorithms that solve the online SLAM problem are often called filters [12]. Online SLAM solves the full SLAM problem incrementally. Full SLAM calculates a joint probability distribution of all poses and the map. The probability distributions of individual variables can be determined from the joint probability distribution. This is done by integrating over all the variables that are to be marginalized out. Thus, the online SLAM problem is expressed with respect to the full SLAM problem by integrating out all previous poses as shown in equation 3.5.

$$p\left(\mathbf{x}_t,\ m \mid z_{1:t},\ u_{1:t}\right) \tag{3.4}$$

$$p\left(\mathbf{x}_t,\ m \mid z_{1:t},\ u_{1:t}\right) = \int_{\mathbf{x}_0}\int_{\mathbf{x}_1}\cdots\int_{\mathbf{x}_{t-1}} p\left(\mathbf{x}_{0:t},\ m \mid z_{1:t},\ u_{1:t}\right)\ d\mathbf{x}_{t-1}\cdots d\mathbf{x}_1 d\mathbf{x}_0 \tag{3.5}$$

It is useful to distinguish between the SLAM front-end and the SLAM back-end. The front-end processes sensor data and gives an intermediate representation of the state. It is thus very dependent on the available sensors and the type of locomotion (a 3DOF differential drive robot like the Loomo behaves differently from a 6DOF holonomic vehicle like a quad-copter). The front-end is also responsible for detecting loop closures. Loop closures occur when the robot revisits a location. The back-end performs the state estimation. It is the method used to solve the SLAM problem. The back-end can thus be more general than the front-end.

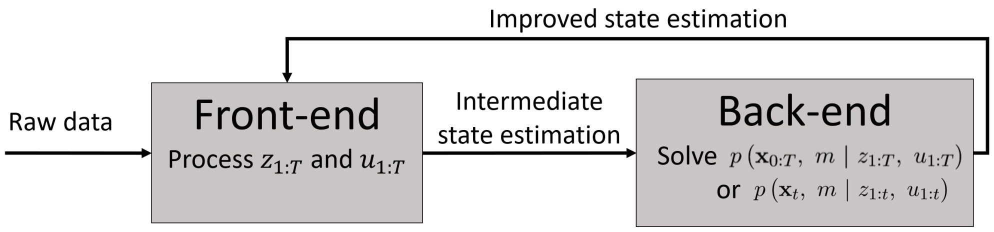

Improved state estimation
Raw data
Front-end
Process $z_{1:T}$ and $u_{1:T}$
Intermediate state estimation
Back-end
Solve $p\left(\mathbf{x}_{0:T},\ m \mid z_{1:T},\ u_{1:T}\right)$
or $p\left(\mathbf{x}_t,\ m \mid z_{1:t},\ u_{1:t}\right)$

Figure 3.1: Interplay of SLAM front-end and back-end

There are three main paradigms for solving the SLAM problem [12]. They are

- Extended Kalman Filter-based approaches
- Particle filter-based approaches
- Graph-based approaches

The EKF, and particle filter-based approaches, as their names suggests, solve the online SLAM problem. Graph-based approaches fundamentally solves the full SLAM problem. However, this approach is highly flexible and there are methods for efficiently re-using previously computed solutions [13, 14, 15].

An underlying assumption in most SLAM methods is the static world assumption. This means that the environment is assumed to be the same each time it is observed. In the real world this is very rarely the case. People and objects can move around. Outdoor environments are highly dynamic because of weather, seasons, and time of day. The sensors that are used play a role. For instance, LIDARs are less affected by changing lighting conditions than cameras. Dynamic effects are sometimes treated as measurement outliers. There are SLAM and localization approaches that specifically addresses dynamic environments [16, 17, 18, 19, 20, 21]. This thesis assumes a static indoor environment where the only dynamic effect comes from the presence of other Loomo robots. Given that the a Loomo can be detected, an observation can be filtered so the Loomo is not interpreted as part of the environment.

3.1.2 Graphs in Graph-Based SLAM

Many problems can be represented by a graph. In a graph, the nodes represent the variables and the edges represent the relationship between variables. For instance, the A* path planning algorithm solves a graph problem where each node is a location and each edge is the cost (often the distance) for moving between nodes.

Figure 3.2 shows the full SLAM problem (equation 3.1) represented by a graph. Graph-based SLAM rarely uses this representation though. The unknowns are the robot poses $\mathbf{x}_{0:T}$ and the landmarks m, and they are usually represented with nodes. A pose-graph approach to SLAM (which is used in this report) even marginalizes out the landmarks. The inputs and observations ($u_{1:T}$ and $z_{1:T}$) are used to find $\mathbf{x}_{0:T}$ and m, so they are usually represented with edges.

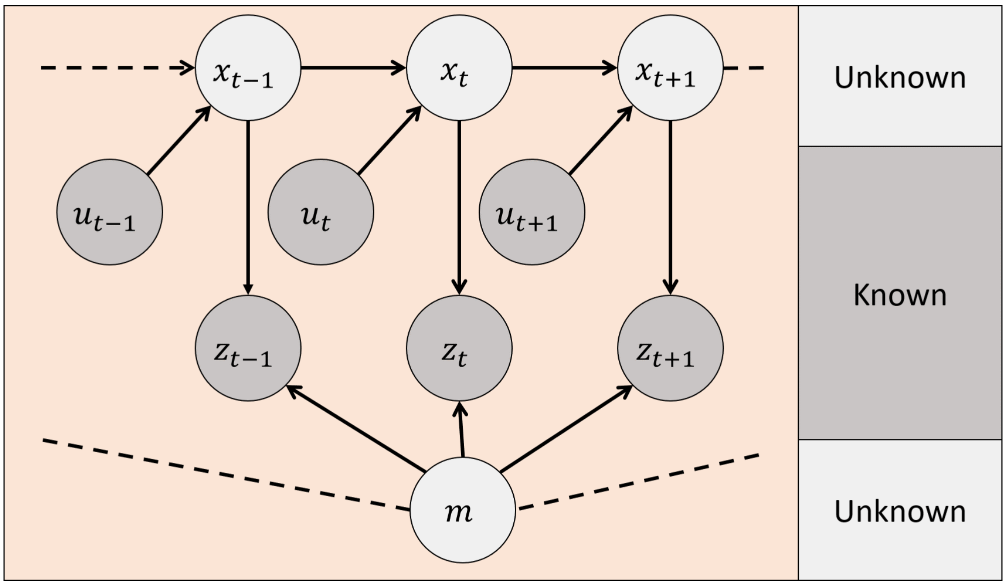


Figure 3.2: Graph-representation of the full SLAM problem

Edges tell the *relationship* between nodes. This relationship is referred to as a constraint. The edges in graph-based SLAM are soft constraints, meaning they are not required to be satisfied, but they have some associated information about to what extent they should be satisfied (i.e. a weighing factor). Graph optimization is the process of adjusting the nodes in a way that best satisfies the edge constraints. Least squares is the most common method for solving the graph optimization in graph-based SLAM. In the context of SLAM in this report, graph optimization and least squares is used interchangeably.

Figure 3.3 shows the interplay of the front-end and back-end in the context of graph-based SLAM. The front-end is still responsible for processing the raw data, but also adds nodes and edges to the graph. The back-end is responsible for solving the SLAM problem, i.e. performing the least squares optimization.

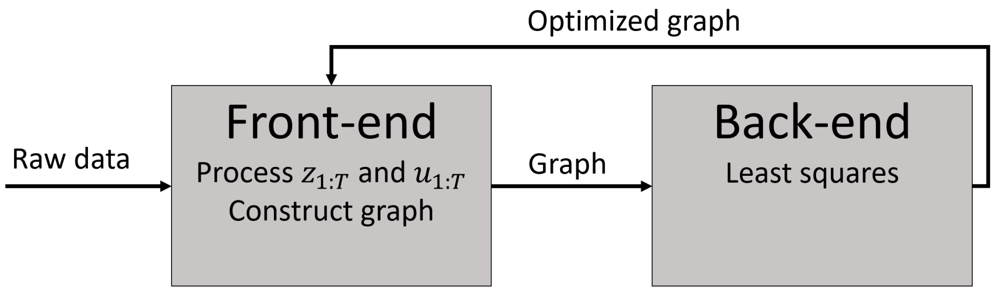


Figure 3.3: Interplay between the front-end and back-end in graph-based SLAM

3.1.3 Least Squares

Graph-based SLAM is often referred to as a *least squares* approach to SLAM. Least squares is an error minimization method. It is an approach for computing a solution for over-determined systems. In a graph representation, a system is over-determined if there is more than one edge connected to a node. All measurements are subject to some uncertainty, so the constraints from different edges tend to *disagree* about what the state of the node is. In other words, trying to solve an over-determined system will lead to errors in some, or all, equations. Least squares minimizes the sum of the squared errors.

The least squares problem can be formulated as follows: Given n noisy measurements $z_{1:n}$ about the state $\mathbf{x}$, estimate the state $\mathbf{x}$ which bests explains the measurements $z_{1:n}$. For each measurement z_k there is a predicted measurement $\hat{z}_k$. The difference between these two will result in an error:

$$\mathbf{e}_k\left(\mathbf{x}\right) = z_k - \hat{z}_k \tag{3.6}$$

Least squares assumes that the error has zero mean and is normally distributed. I.e. Gaussian error with information matrix Ω_k. The squared error of a measurement is a scalar.

$$e_k(\mathbf{x}) = \mathbf{e}_k(\mathbf{x})^\top \Omega_k \mathbf{e}_k(\mathbf{x}) \tag{3.7}$$

The information matrix Ω is a weighing factor in the least squares optimization. It is the inverse of the covariance matrix of the measurements. The lower the values in a covariance matrix is, the more reliable that data is. Ω will in turn increase with lower covariance values. So higher values of Ω means that that particular measurement *matters* more in the optimization.

The global error (equation 3.8) is the sum of all the errors.

$$F(\mathbf{x}) = \sum_k e_k(\mathbf{x}) = \sum_k \mathbf{e}_k(\mathbf{x})^\top \Omega_k \mathbf{e}_k(\mathbf{x}) \tag{3.8}$$

The goal of least squares is to find the state $\mathbf{x}^*$ which minimizes the global error function.

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}}\, F(\mathbf{x}) \tag{3.9}$$

The minimization in equation 3.9 is a complex problem and usually requires numerical solvers. With numerical solvers, finding the global minimum is not guaranteed, so a good initial guess is required. The method that is used in this thesis is the Gauss-Newton solution. It solves the problem with iterative local linearizations. The steps to this method are as follows:

- Linearize the error function around the current state (or initial guess if it is the first iteration): $\mathbf{e}(\mathbf{x} + \Delta\mathbf{x})$
- Use the linearization to form a new expression for the global error in the neighborhood of $\mathbf{x}$: $F(\mathbf{x} + \Delta\mathbf{x})$
- Compute the first derivative of $F(\mathbf{x} + \Delta\mathbf{x})$
- Set it to zero and solve the linear system for $\Delta\mathbf{x}$
- Use this solution to increment the state
- Iterate these steps until the system converges (i.e. the increments approaches 0)

Linearizing the error function around $\mathbf{x}$ is done by approximating it via Taylor expansion.

$$\mathbf{e}_k(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{e}_k(\mathbf{x}) + \mathbf{J}_k(\mathbf{x})\Delta\mathbf{x} \tag{3.10}$$

$\mathbf{J}_k(\mathbf{x})$ is the Jacobian of $\mathbf{e}_k$ with respect to the state.

$$\mathbf{x} = \mathbf{x}_{1:n} \qquad\qquad \mathbf{J}_k(\mathbf{x}) = \left[\frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}_1}, \ \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}_2}, \ \cdots, \ \frac{\partial \mathbf{e}_k(\mathbf{x})}{\partial \mathbf{x}_n}\right] \tag{3.11}$$

Expressing the squared error using the linear approximation in equation 3.10

$$e_k(\mathbf{x} + \Delta\mathbf{x}) \approx (\mathbf{e}_k(\mathbf{x}) + \mathbf{J}_k(\mathbf{x})\Delta\mathbf{x})^\top \Omega_k \left(\mathbf{e}_k(\mathbf{x}) + \mathbf{J}_k(\mathbf{x})\Delta\mathbf{x}\right) \tag{3.12}$$

The notation is simplified from here on as follows

$$e_k \approx (\mathbf{e}_k + \mathbf{J}_k\Delta\mathbf{x})^\top \Omega_k \left(\mathbf{e}_k + \mathbf{J}_k\Delta\mathbf{x}\right) \tag{3.13}$$

Resolving the parentheses in 3.13 and simplifying it yields equation 3.16. $(\mathbf{J}_k\Delta\mathbf{x})^\top$ resolves to $\Delta\mathbf{x}^\top\mathbf{J}_k^\top$. Also, since each term is a scalar, the second and third term on the right hand side in equation 3.14 are equal (if $A \cdot B$ is scalar, then $(A \cdot B)^\top = A \cdot B = B^\top \cdot A^\top$). Equation 3.16 has quadratic form.

$$e_k = \mathbf{e}_k^\top \Omega_k \mathbf{e}_k + \mathbf{e}_k^\top \Omega_k \mathbf{J}_k \Delta\mathbf{x} + \Delta\mathbf{x}^\top \mathbf{J}_k^\top \Omega_k \mathbf{e}_k + \Delta\mathbf{x}^\top \mathbf{J}_k^\top \Omega_k \mathbf{J}_k \Delta\mathbf{x} \tag{3.14}$$

$$e_k = \underbrace{\mathbf{e}_k^\top \Omega_k \mathbf{e}_k}_{e_k} + 2\underbrace{\mathbf{e}_k^\top \Omega \mathbf{J}_k}_{b_k^\top} \Delta\mathbf{x} + \Delta\mathbf{x}^\top \underbrace{\mathbf{J}_k^\top \Omega_k \mathbf{J}_k}_{H_k} \Delta\mathbf{x} \tag{3.15}$$

$$e_k = e_k + 2b_k^\top \Delta\mathbf{x} + \Delta\mathbf{x}^\top H_k \Delta\mathbf{x} \tag{3.16}$$

The new expression for the global error in the neighborhood of $\mathbf{x}$ is as follows:

$$F(\mathbf{x}+\Delta\mathbf{x}) = \sum_k e_k(\mathbf{x}+\Delta\mathbf{x}) \tag{3.17}$$

$$F(\mathbf{x}+\Delta\mathbf{x}) = \sum_k \left(e_k + 2b_k^\top \Delta\mathbf{x} + \Delta\mathbf{x}^\top H_k \Delta\mathbf{x}\right) \tag{3.18}$$

$$F(\mathbf{x}+\Delta\mathbf{x}) = \underbrace{\left(\sum_k e_k\right)}_{e} + 2\underbrace{\left(\sum_k b_k^\top\right)}_{b^\top} \Delta\mathbf{x} + \Delta\mathbf{x}^\top \underbrace{\left(\sum_k H_k\right)}_{H} \Delta\mathbf{x} \tag{3.19}$$

$$F(\mathbf{x}+\Delta\mathbf{x}) = e + 2b^\top \Delta\mathbf{x} + \Delta\mathbf{x}^\top H \Delta\mathbf{x} \tag{3.20}$$

$$b^\top = \sum_k \mathbf{e}_k^\top \Omega_k \mathbf{J}_k \tag{3.21}$$

$$H = \sum_k \mathbf{J}_k^\top \Omega_k \mathbf{J}_k \tag{3.22}$$

Computing the first derivative of equation 3.20. H is symmetrical, so $H = H^\top$.

$$\frac{\partial F(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = 2b + \left(H + H^\top\right) \Delta\mathbf{x} \tag{3.23}$$

$$\frac{\partial F(\mathbf{x}+\Delta\mathbf{x})}{\partial \Delta\mathbf{x}} = 2b + 2H\Delta\mathbf{x} \tag{3.24}$$

$$\tag{3.25}$$

Setting the derivative to 0 leads to the linear system in 3.27.

$$0 = \cancel{2}b + \cancel{2}H\Delta\mathbf{x} \tag{3.26}$$

$$H\Delta\mathbf{x} = -b \tag{3.27}$$

The solution for the increment $\Delta\mathbf{x}$ is

$$\Delta\mathbf{x} = -H^{-1}b \tag{3.28}$$

The steps to the Gauss-Newton solution can now be simplified:

- Linearize the error function around $\mathbf{x}$ and compute for each measurement: $\mathbf{e}_k(\mathbf{x}+\Delta\mathbf{x}) \approx \mathbf{e}_k(\mathbf{x}) + \mathbf{J}_k\Delta\mathbf{x}$
- Compute the terms $b^\top$ and H (equations 3.21 and 3.22) for the linear system
- Solve the linear system $\Delta\mathbf{x} = -H^{-1}b$
- Update the state $\mathbf{x} \mathrel{+}= \Delta\mathbf{x}$
- Iterate until convergence

This is a general explanation of least squares using a Gauss-Newton solver. Section 3.1.4 will go more in detail about what the error function looks like in the context of solving the SLAM problem and how to build and solve the linear system $H\Delta\mathbf{x} = -b$.

3.1.4 A Pose Graph Solution to the SLAM Problem

A pose graph is an approach to graph-based SLAM that does not use landmarks. The graph is reduced to contain only the robot poses, and the landmarks are marginalized out. The graph structure is thus easy to maintain.

A pose graph by itself only solves half the SLAM problem, namely estimating the robot trajectory. However, pose graphs are commonly used as the underlying process in map construction. If the poses are known, it is easier to interpret the observations of the environment.

Each node in a pose graph represents the robot's pose in that instance. It consists of n nodes $\mathbf{x} = \mathbf{x}_{1:n}$, and node number i represents the state vector $\mathbf{x}_i$. The edge constraints are relative transformation between nodes. An edge is constructed between two nodes if:

- a new node is constructed, i.e. the robot moves from node $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$. The edge is odometry-based.
- the robot observes the same part of the environment from node $\mathbf{x}_i$ and from $\mathbf{x}_j$. The edge is referred to as a virtual edge.

All consecutive nodes have odometry-based edges connecting them. Virtual edges are how the pose graph marginalizes out the landmarks. Instead of using observations to perform state estimations of landmarks, the observations from two nodes are compared and their relative transformations are computed directly (given that they are observing the same thing). The edges are called virtual because node i is not directly observing node j.

Figure 3.4 illustrates the structure of a pose graph. When new nodes are added, they are given an initial pose based on the dead reckoning since last node. Thus, if no graph optimization is done, the graph trajectory will be the same as the odometry trajectory. Odometry provides an initial guess for the graph optimization. Virtual connect to the newest of the two nodes that they are connecting. I.e. *to* $\mathbf{x}_j$ *from* $\mathbf{x}_i$ as shown in figure 3.4.

Figure 3.5 illustrates how the error function in a pose graph is obtained. Let the edge from $\mathbf{x}_i$ to $\mathbf{x}_j$ be described by the constraint z_{ij} and information matrix Ω_{ij}. The constraint z_{ij} is the pose of $\mathbf{x}_j$ relative to $\mathbf{x}_i$. The information matrix Ω_{ij} is the inverse of the covariance matrix of the constraint. In the context of least squares, the constraint z_{ij} corresponds to a single measurement. The error function for an edge in a pose graph is thus expressed in equation 3.29.

$$\mathbf{e}_{ij}(\mathbf{x}) = z_{ij} - \hat{z}_{ij} \tag{3.29}$$

$\hat{z}_{ij}$ is the pose of $\mathbf{x}_j$ relative to the pose of $\mathbf{x}_i$ according to the current configuration of the graph. It is convenient to express z_{ij} and $\hat{z}_{ij}$ as homogeneous transformations. Here, capital letters are used for homogeneous coordinates and lowercase is used for vector coordinates. The constraint z_{ij}

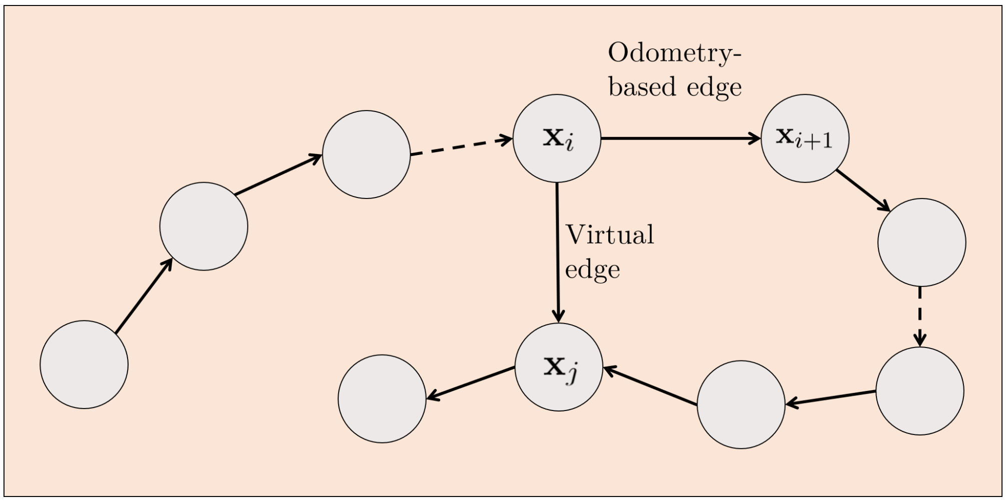


Figure 3.4: The structure of a pose graph

expressed as a homogeneous transformation is thus Z_{ij}. The homogeneous transformation from $\mathbf{x}_i$ to $\mathbf{x}_j$ is $\mathbf{X}_i^{-1}\mathbf{X}_j$. Equation 3.30 is the error function expressed with homogeneous transformations. The least squares optimization still uses the minimal representation of the state, which is a vector. Hence the homogeneous transformation is converted back to vector coordinates (t2v is a function such that $\mathbf{x}_i = \text{t2v}(\mathbf{X}_i)$). The error is zero if $Z_{ij} = \mathbf{X}_i^{-1}\mathbf{X}_j$.

$$\mathbf{e}_{ij}(\mathbf{x}) = \text{t2v}\left(Z_{ij}^{-1}\left(\mathbf{X}_i^{-1}\mathbf{X}_j\right)\right) \tag{3.30}$$

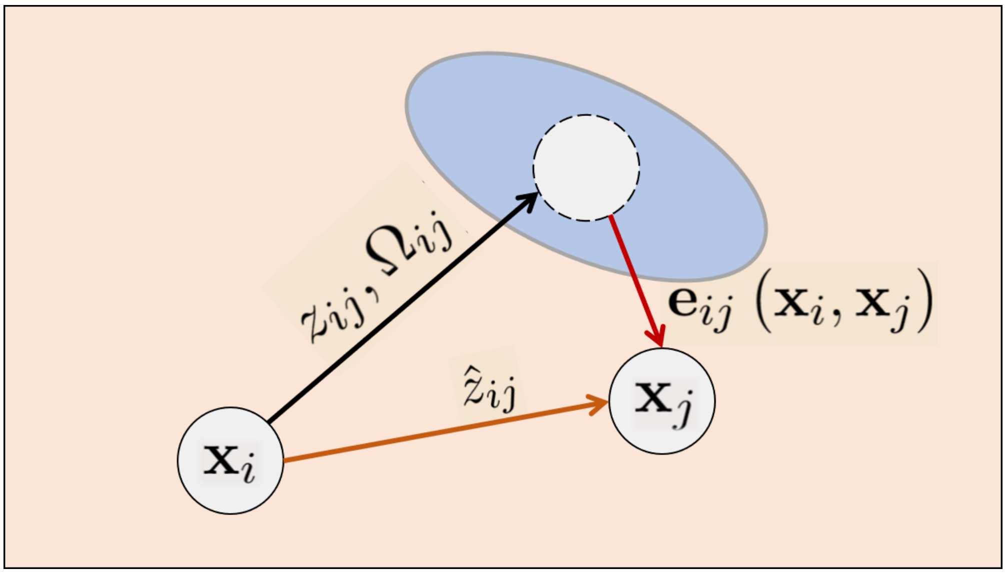


Figure 3.5: Error function for a single edge in a pose graph. The dotted circle is the pose of node j according to the edge constraint, while the other circles are the poses according to the current configuration of the graph. The information matrix Ω_{ij} encodes the certainty of the constraint, and is illustrated by the blue ellipse. Higher values in Ω_{ij} corresponds to a smaller ellipse.

An important property of pose graphs is that any single error term $\mathbf{e}_{ij}$ only depends on $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$\mathbf{e}_{ij}(\mathbf{x}) = \mathbf{e}_{ij}(\mathbf{x}_i, \mathbf{x}_j) \tag{3.31}$$

Linearizing the error function around $\mathbf{x}$:

$$\mathbf{e}_{ij}\left(\mathbf{x}+\Delta\mathbf{x}\right)=\mathbf{e}_{ij}(\mathbf{x})+\mathbf{J}_{ij}\Delta\mathbf{x} \tag{3.32}$$

Because the error only depends on $\mathbf{x}_i$ and $\mathbf{x}_j$, the Jacobian will be non-zero only in the rows corresponding to $\mathbf{x}_i$ and $\mathbf{x}_j$.

$$\mathbf{J}_{ij}=\frac{\partial\mathbf{e}_{ij}(\mathbf{x})}{\partial\mathbf{x}} \tag{3.33}$$

$$\mathbf{J}_{ij}=\left[\mathbf{0}\cdots\frac{\partial\mathbf{e}_{ij}(\mathbf{x})}{\partial\mathbf{x}_i}\cdots\frac{\partial\mathbf{e}_{ij}(\mathbf{x})}{\partial\mathbf{x}_j}\cdots\mathbf{0}\right] \tag{3.34}$$

$$\mathbf{J}_{ij}=\left[\mathbf{0}\cdots\mathbf{A}_{ij}\cdots\mathbf{B}_{ij}\cdots\mathbf{0}\right] \tag{3.35}$$

$$\text{with}\quad \mathbf{A}_{ij}=\frac{\partial\mathbf{e}_{ij}(\mathbf{x})}{\partial\mathbf{x}_i}\quad\text{and}\quad\mathbf{B}_{ij}=\frac{\partial\mathbf{e}_{ij}(\mathbf{x})}{\partial\mathbf{x}_j} \tag{3.36}$$

The next step is to compute the terms $b^\top$ and H for the linear system:

$$b^\top=\sum_{ij}b_{ij}^\top=\sum_{ij}\mathbf{e}_{ij}^\top\Omega_{ij}\mathbf{J}_{ij} \tag{3.37}$$

$$H=\sum_{ij}H_{ij}=\sum_{ij}\mathbf{J}_{ij}^\top\Omega_{ij}\mathbf{J}_{ij} \tag{3.38}$$

Since every node has at least one edge connecting it to another node, the vector $b^\top$ will be fully populated. However, the sparse structure of $\mathbf{J}_{ij}$ will result in a sparse structure of H. H will be non-zero along its diagonal. Otherwise, only the blocks corresponding to node pairs are populated. The structure of H reflects the adjacency matrix of the graph. The elements of an adjacency matrix indicate whether a pair of nodes are connected by an edge. The sparse property of the Jacobian is shown for a single node pair in equations 3.41 and 3.44. The sparse structure of the resulting H matrix is illustrated in figure 3.6.

H will be dense if the sensors can observe most of the environment from all the nodes. This can possibly happen in a small open area.

$$b_{ij}^\top=\mathbf{e}_{ij}^\top\Omega_{ij}\mathbf{J}_{ij} \tag{3.39}$$

$$=\mathbf{e}_{ij}^\top\Omega_{ij}\left[\mathbf{0}\cdots\mathbf{A}_{ij}\cdots\mathbf{B}_{ij}\cdots\mathbf{0}\right] \tag{3.40}$$

$$=\left[\mathbf{0}\cdots\mathbf{e}_{ij}^\top\Omega_{ij}\mathbf{A}_{ij}\cdots\mathbf{e}_{ij}^\top\Omega_{ij}\mathbf{B}_{ij}\cdots\mathbf{0}\right] \tag{3.41}$$

$$H_{ij} = \mathbf{J}_{ij}^{\top}\Omega_{ij}\mathbf{J}_{ij} \tag{3.42}$$

$$= \begin{bmatrix} \vdots \\ \mathbf{A}_{ij}^{\top} \\ \vdots \\ \mathbf{B}_{ij}^{\top} \\ \vdots \end{bmatrix} \Omega_{ij} \begin{bmatrix} \cdots & \mathbf{A}_{ij} & \cdots & \mathbf{B}_{ij} & \cdots \end{bmatrix} \tag{3.43}$$

$$= \begin{bmatrix} \ddots & & & & \\ & \mathbf{A}_{ij}^{\top}\Omega_{ij}\mathbf{A}_{ij} & \cdots & \mathbf{A}_{ij}^{\top}\Omega_{ij}\mathbf{B}_{ij} & \\ & \vdots & \ddots & \vdots & \\ & \mathbf{B}_{ij}^{\top}\Omega_{ij}\mathbf{A}_{ij} & \cdots & \mathbf{B}_{ij}^{\top}\Omega_{ij}\mathbf{B}_{ij} & \\ & & & & \ddots \end{bmatrix} \tag{3.44}$$

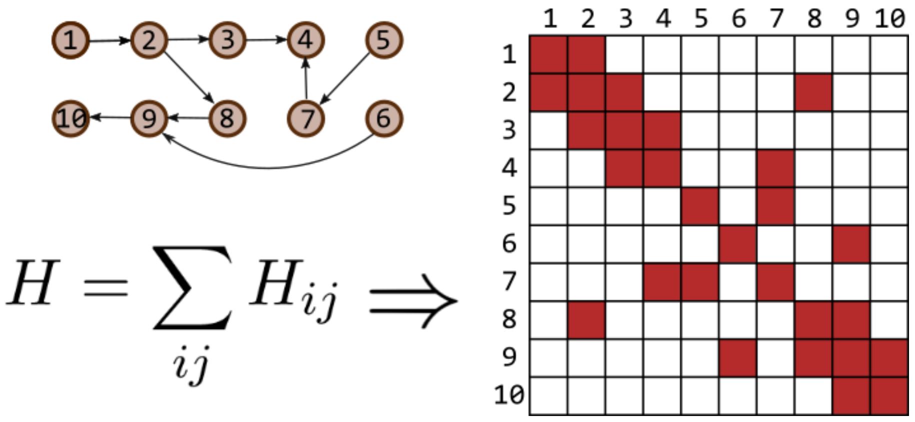


Figure 3.6: Structure of the H matrix for a simple 10-node graph. All white blocks are zero

The linear system can now be built by summing up the contribution of each edge. Initialize b and H with all zero elements.

$$b^{\top} = \begin{bmatrix} \bar{b}_1^{\top} & \bar{b}_2^{\top} & \cdots & \bar{b}_n^{\top} \end{bmatrix} \tag{3.45}$$

$$H = \begin{bmatrix} \bar{H}_{11} & \bar{H}_{12} & \cdots & \bar{H}_{1n} \\ \bar{H}_{21} & \bar{H}_{22} & \cdots & \bar{H}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{H}_{n1} & \bar{H}_{n2} & \cdots & \bar{H}_{nn} \end{bmatrix} \tag{3.46}$$

The *bar*-notation is used here because each element can potentially be the sum of several constraint. This is to distinguish them from element ij. Many of the elements in H are still going to be 0.

For each constraint in the graph:

- Compute the error $\mathbf{e}_{ij}$ (equation 3.30)
- Compute the blocks of the Jacobian, $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ (equation 3.36)
- Update the appropriate blocks in b and H:

$$\bar{b}_i^\top \mathrel{+}= \mathbf{e}_{ij}^\top \Omega_{ij} \mathbf{A}_{ij} \qquad \bar{b}_j^\top \mathrel{+}= \mathbf{e}_{ij}^\top \Omega_{ij} \mathbf{B}_{ij} \tag{3.47}$$

$$\bar{H}_{ii} \mathrel{+}= \mathbf{A}_{ij}^\top \Omega_{ij} \mathbf{A}_{ij} \qquad \bar{H}_{ij} \mathrel{+}= \mathbf{A}_{ij}^\top \Omega_{ij} \mathbf{B}_{ij} \tag{3.48}$$

$$\bar{H}_{ji} \mathrel{+}= \mathbf{B}_{ij}^\top \Omega_{ij} \mathbf{A}_{ij} \qquad \bar{H}_{jj} \mathrel{+}= \mathbf{B}_{ij}^\top \Omega_{ij} \mathbf{B}_{ij} \tag{3.49}$$

This will populate b and H, but the linear system is currently under determined. The graph trajectory must be locked to a world coordinate frame. This can be done by constraining one of the increments $\Delta\mathbf{x}_k$ to zero, which is done by adding the identity matrix I to the kth diagonal block of H. It is typically the first node that defines the coordinate system. By doing $\bar{H}_{11} \mathrel{+}= I$ the linear system can now be solved.

H is potentially a very big matrix. Matrix inversion is a computationally heavy operation and rarely done in practice. More efficient numerical solutions to matrix inversion exist. H is a square, symmetric and positive-definite matrix, so it is suitable to use Cholesky decomposition to solve $\Delta\mathbf{x} = -H^{-1}b$.

3.2 Computer Vision

Computer vision is the discipline of enabling a computer to interpret digital images. There is some overlap between computer vision and machine learning. The use of machine learning for detecting Loomos in images is described in section 3.5. This section describes how computer vision is used in the context of Simultaneous Location And Mapping (SLAM).

A camera is an exteroceptive sensor, meaning it can provide a robot with information about its environment. For SLAM, there are two questions that this information should provide answers to:

- ”What is in my environment?“
- ”Where is it?“

A camera provides a series of images. The problem that computer vision is supposed to solve can roughly be divided into two:

- Finding correspondences between images
- Finding the geometrical relationship between correspondences in images

There is also a real-time requirement. This can be satisfied by abstracting an image into a set of image features. This report used the ORB feature descriptor [22] because of its proved effectiveness in recent SLAM systems [23, 24, 25]. The SIFT feature descriptor was the industry standard for feature detection and description. This section will draw some comparrisons to SIFT where they are relevant.

Feature descriptors like ORB, SIFT [26], and SURF [27] does two things: detects keypoints and computes their descriptors. Detecting keypoints means finding the location of image features that are of interest. Computing descriptors means describing the features so that they can be identified repeatedly.

3.2.1 Key Concepts

A digital image can be seen as a matrix containing light intensity values. Each matrix element corresponds to a pixel. A color image has intensity values for each of the color channels red, green, and blue. Thus, a pixel consists of three values. If color information is not needed, it is common to convert the image to grey-scale.

A camera can be seen as a heading sensor. Light is reflected off an object into the camera sensor. The 3D world is projected onto a 2D image, and each pixel tells the heading of the object that was projected onto it.

An important measure in computer vision is the gradient in an image. The gradient is the directional rate of change of pixel intensities. If g is an intensity value, the gradient is given by equation 3.50. Object edges will usually have an abrupt transition in brightness, an thus a high gradient. This makes edges an easy image feature to detect.

$$\nabla g = \left[\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}\right] \tag{3.50}$$

3.2.2 Convolution

Convolution is a key concept in computer vision and Convolutional Neural Networks. Convolution simplifies a filtering process by doing a repeated operation with a mask known as a filter or kernel. The filter is a matrix which is usually $n \times n$ where n is odd so that the filter has a center pixel. The filter is slid over each pixel in the input image and for each pixel calculates a value for the corresponding output pixel. The output value is the sum of the products of corresponding pixels in the filter and the image. Thus, each output pixel is the result of the neighborhood of the the input pixel. This is illustrated in figure 3.7.

A very simple convolutional filter is one where all the values are equal and the sum of all the values is one (for instance a 3×3 filter where all the values are $1/9$). Applying this filter will cause each output pixel value to be the average of the neighborhood of the input pixel. This will effectively blur the images and is used to reduce noise. It is however more common to use a Gaussian filter for noise reduction. A Gaussian filter is made by discretizing a Gaussian distribution so it fits the filter. The output is then a weighted average, where pixels closer to the input pixel has bigger influence.

Another important type of convolutional filter is the gradient filter. Two common types are the Sobel, and Scharr operators. They use two filters. One that calculates the gradient in the x-direction, and one for the y-direction. If the values on the left and right side of the input pixel are similar, the output value G_x when using the D_x filter will be close to 0. If the values on the left

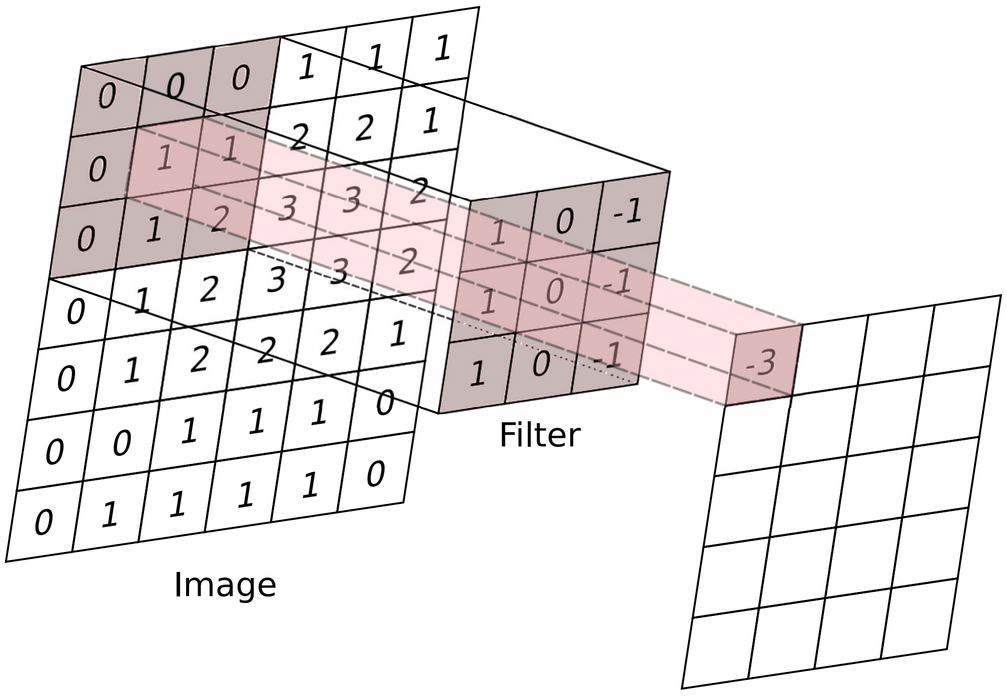


Figure 3.7: Convolutional operation

side are different to the right side, the absolute value of G_x will be high. Let G_x and G_y be the output when convolving with D_x and D_y respectively. The magnitude and direction of the gradient in a pixel is thus found by the equations in 3.53.

The Scharr operator differs from the Sobel operator in that it has some higher accuracy when calculating the gradient direction. These operations are, after all, approximations for calculating the derivatives in an image.

$$\textit{Sobel}: \quad D_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad D_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.51}$$

$$\textit{Scharr}: \quad D_x = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \quad D_y = \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \tag{3.52}$$

$$\textit{Gradient magnitude} = \sqrt{G_x^2 + G_y^2} \qquad \textit{Gradient direction} = \operatorname{atan2}\left(\frac{G_y}{G_x}\right) \tag{3.53}$$

3.2.3 Keypoint Detection

A keypoint is a locally distinct location in an image. One way to evaluate whether a point is locally distinct is to see how the neighborhood of that point changes when the point is shifted. If the gradients in that area are small, the neighborhood of the point will barely change if the point is shifted. In figure 3.8, point p_1 can be shifted in any direction, and those locations will be indistinguishable from each other. Edges on the other hand have steep gradients and are easy to locate. The problem is that only points along the gradient direction are distinct from each other, so point p_2 in figure 3.8 will still be indistinguishable from points along the edge direction. Corners are often highly distinct points because they have gradients in two different directions. Point p_3 is thus the best candidate for being a keypoint.

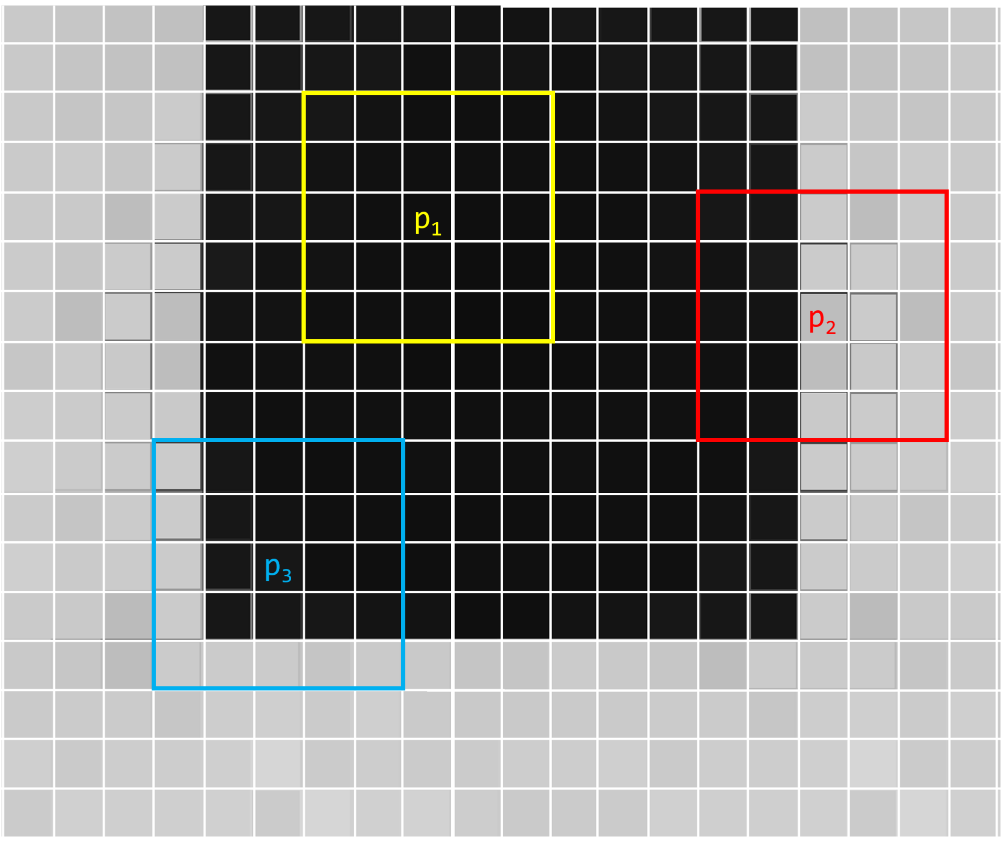


Figure 3.8: Three different keypoint candidates. The neighborhood of p_1 does not change if it is moved. The neighborhood of p_2 only changes if it moves in the x-direction. The neighborhood of p_3 changes regardless of which direction it is moved.

Corners are used in several keypoint detectors [28, 29, 30, 31, 32]. Corners are invariant to translation, rotation and illumination. This means that the keypoint is likely to be found in two images of the same scene, even though the conditions have changed between the images.

A corner can be found by searching for intensity changes in the x and y directions. Let $f(x, y)$ be a function that computes the sum of squared differences of neighboring pixels around pixel (x, y).

$$f(x,y) = \sum_{(u,v)\in W_{xy}} \left(I(u,v) - I(u+\delta u, v+\delta v)\right)^2 \tag{3.54}$$

W_{xy} is a local patch around (x, y). $I(u, v)$ is the intensity value of pixel (u, v) in the local patch W_{xy}. $I(u + \delta u, v + \delta v)$ can be approximated through linearization using Taylor expansion:

$$I(u+\delta u, v+\delta v) \approx I(u,v) + \begin{bmatrix} J_x & J_y \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{3.55}$$

Substituting the linearization into equation 3.54:

$$f(x,y) \approx \sum_{(u,v)\in W_{xy}} \left(\begin{bmatrix} J_x & J_y \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \right)^2 \tag{3.56}$$

$$f(x,y) \approx \sum_{(u,v)\in W_{xy}} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix}^\top \begin{bmatrix} J_x^2 & J_xJ_y \\ J_xJ_y & J_y^2 \end{bmatrix} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{3.57}$$

The summation can be moved inside the matrix. The sums are rewritten to Σ_W for brevity.

$$f(x,y) \approx \begin{bmatrix} \delta u \\ \delta v \end{bmatrix}^\top \underbrace{\begin{bmatrix} \Sigma_W J_x^2 & \Sigma_W J_xJ_y \\ \Sigma_W J_xJ_y & \Sigma_W J_y^2 \end{bmatrix}}_{M} \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \tag{3.58}$$

M is called the structure matrix. It tells the magnitude and directions of the gradients in a local region.

$$M = \begin{bmatrix} \Sigma_W J_x^2 & \Sigma_W J_xJ_y \\ \Sigma_W J_xJ_y & \Sigma_W J_y^2 \end{bmatrix} \tag{3.59}$$

The structure matrix M can tell if a point is a locally distinct point or not (i.e. if there is a corner). If there is one large and one small eigenvalue, you have an edge-like structure (gradients are pointing in the same direction). If you have two large (and of similar size) eigenvalues there is probably a corner.

Jacobians are computed via convolution with a gradient filter. The filter can for instance be the Sobel kernel.

$$J_x^2 = (D_x * I)^2 \tag{3.60}$$

$$J_xJ_y = (D_x * I)(D_y * I) \tag{3.61}$$

$$J_y^2 = (D_y * I)^2 \tag{3.62}$$

The structure matrix summarizes the dominant directions of the gradient around a point. A point can be considered a corner if its structure matrix has two large eigenvalues.

The Förstner [28], Harris [30], and Shi-Tomasi [31] corner detection methods all use the structure matrix. They have different criteria for deciding if a point is a corner or not.

Harris corner criterion (proposed in 1988) was initially the most popular. It was improved by Shi-Tomasi in 1994, which has become the standard keypoint detector in OpenCV. However, Harris corner criterion (also referred to as Harris corner measure, Harris corner score, corner response etc.) is used in the ORB descriptor and will be explained here.

The Harris corner criterion is formulated as follows [30]:

$$R = \det(M) - k\Big(\operatorname{tr}(M)\Big)^2 \tag{3.63}$$

k is a weighting factor. Typically $k \in [0.04, 0.06]$

The determinant of a matrix is the product of its eigenvalues. The trace of a matrix is the sum of its eigenvalues. M has eigenvalues λ_1, λ_2. Thus, equation 3.63 can be rewritten as follows:

$$R = \lambda_1 \cdot \lambda_2 - k\Big(\lambda_1 + \lambda_2\Big)^2 \tag{3.64}$$

Using the above observations about the eigenvalues λ_1 and λ_2 of the structure matrix, the following deduction about the Harris criterion can be made (this is also illustrated in figure 3.9).

$$\begin{array}{cccl} |R| \approx 0 & \Rightarrow & \lambda_1 \approx \lambda_2 \approx 0 & : \text{flat region} \\ R < 0 & \Rightarrow & \lambda_1 \gg \lambda_2 \ or \ \lambda_2 \gg \lambda_1 & : \text{edge} \\ R \gg 0 & \Rightarrow & \lambda_1 \approx \lambda_2 \gg 0 & : \text{corner} \end{array}$$

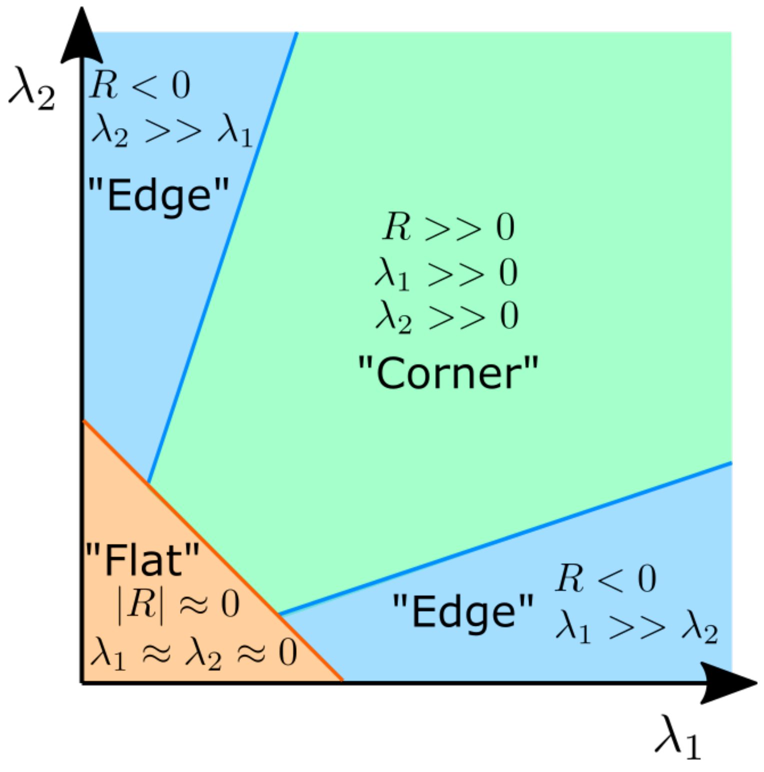


Figure 3.9: Illustration of Harris criterion

The drawback to the Harris corner detection method is that the corner response (i.e. the Harris criterion) must be computed for each pixel. FAST [32] is a corner detector which is highly efficient . FAST is short for Features from accelerated segment test (FAST). The technique is fairly simple. For an image point p, measure the intensity values along the periphery of a circle around p. If n contiguous pixels are all either brighter than p by a certain threshold, or darker than p by a certain threshold, p is considered a corner.

The circle is a Bresenham circle, meaning it has been discretized to pixel-space [33]. The circle has a circumference of 16 pixels, which are numbered in the clockwise direction (see figure 3.10). Rosten and Drummond [32] found that $n = 9$ provides optimal performance. In figure 3.10, 9 contiguous pixels from pixel number 9 through 1 (dotted line) are brighter than p by a threshold, and the point is considered a corner.

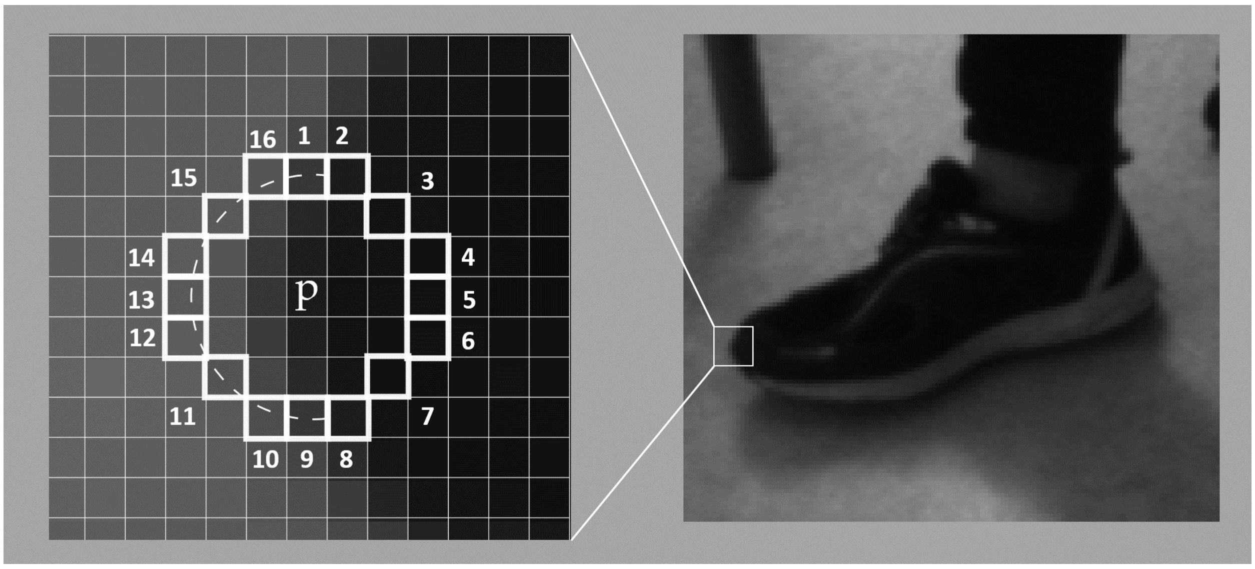


Figure 3.10: FAST corner detector. If n contiguous pixels are either darker or brighter than p by a threshold, p is considered a corner.

FAST does not compute a corner response function. The paper suggests a method for computing the score of the detected points [32], but it is not relevant here as the ORB detector uses the Harris corner criterion (equation 3.64) to score the detected corners.

There are of course many other methods for detecting keypoints in an image. For instance Difference of Gaussians (DoG), which is used by SIFT [26]. DoG will also detect blobs in addition to lines and edges. The methods presented here are those that are relevant for the computer vision task of the Loomo robots.

3.2.4 Feature Description

Given that image features can be reliably detected, it is also necessary to describe them so that they can be compared to keypoints from other images. A feature descriptor is a summary of the local structure around a keypoint.

One common way to summarize the neighborhood of a point is to calculate a histogram of the gradients in that region. This is what the SIFT descriptor does. It calculates the gradients in a 16×16 region around the the keypoint. Note that the 16×16 region does not necessarily correspond to 16×16 pixels in the original image. Scale-Invariant Feature Transform (SIFT) also detects scale and rotation of its keypoints, so this region is relative to that keypoint-information [26]. The 16×16 region is divided into 16 sub-regions, and the histogram is calculated for each of them. The gradient directions are discretized into 8 segments. I.e, angles of 22.5° to 67.6° are evaluated as

45°, angles of 67.5° to 112.5° as 90°, etc. Each histogram then contains 8 floating point numbers where each number represents the sum of the gradient magnitude for that angle. Each histogram is then concatenated into a single vector with a total of $8 \cdot 16 = 128$ floating point numbers. [26]

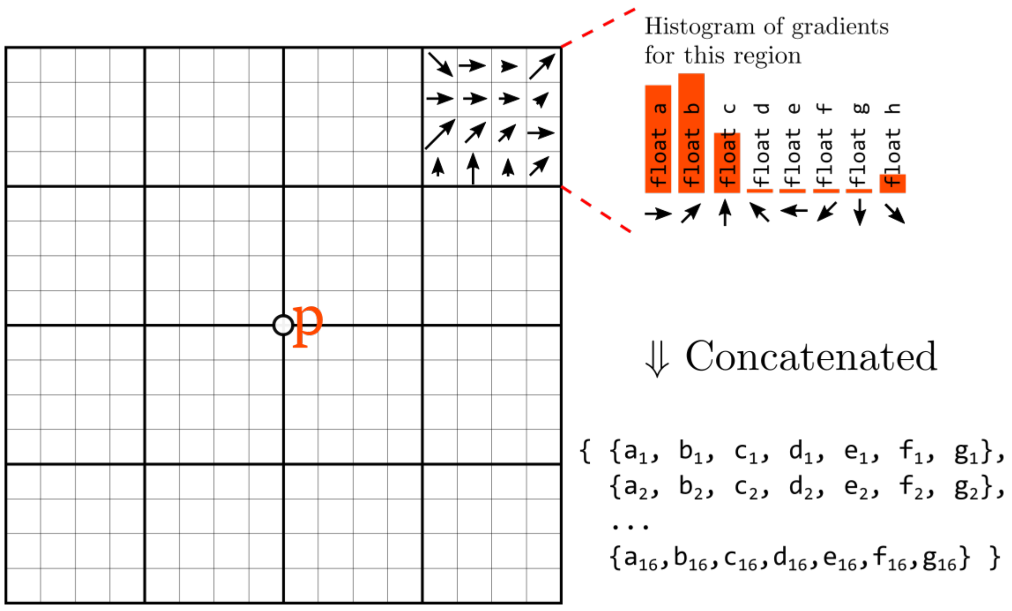


Figure 3.11: SIFT feature descriptor. A histogram of gradients is computed for 16 4×4 sub-regions around a point p. Each histogram is represented by 8 floating point numbers, which are all concatenated into a single 128 element vector

Figure 3.11 illustrates how the SIFT descriptor works. A drawback to this method is its computational complexity. In addition to that, each keypoint descriptor is 512 *bytes* (eq. 3.65), which further complicates the feature matching calculations. Detecting many keypoints can also cause memory problems as more than 600 features will consume more memory than a 640×480 pixel grey-scale image.

$$\begin{gathered} float = 32\ bits \qquad 128\ floats = 4096\ bits = 512\ Bytes \\ 640 \times 480 \text{ grey-scale image} = 307200\ Bytes \\ 600 \text{ SIFT feature descriptors} = 307200\ Bytes \end{gathered} \tag{3.65}$$

A compact and fast alternative to the SIFT method is a so-called binary descriptor. A binary descriptor summarizes the neighborhood of a point with a single bit-string instead of a vector of floating points. The working principal of binary descriptors are as follows:

- Select a neighborhood around a keypoint
- Select a set of pixel pairs in that neighborhood
- For each pair of pixels s_1 and s_2, compare the intensities
- Set the bit b high or low depending on the comparison:

$$b = \begin{cases} 1 & \text{if } I(s_1) < I(s_2) \\ 0 & \text{otherwise} \end{cases} \tag{3.66}$$

- Then concatenate all bits into a bit string

The choice of pairs, and the order in which they are compared must be consistent. Binary descriptors are very compact. They are also fast to compute since it only compares intensity values instead of calculating the gradient for each pixel. Since they are binary strings, it is also trivial and fast to compare features. The similarity of two binary strings B_1 and B_2 is expressed by their Hamming distance. For binary strings, the Hamming distance is the minimum number of bits that has to be flipped for B_1 and B_2 to be equal. In other words, it is the sum of unique bits in the strings and can be computed using *exclusive or*:

$$d_{Hamming}(B_1, B_2) = \sum_{b_1, b_2 \in B_1, B_2} (b_1 \operatorname{xor} b_2) \tag{3.67}$$

The first binary feature descriptor was proposed in 2010 and is called BRIEF[34]. BRIEF stands for Binary Robust Independent Elementary Features. It is commonly a 256 bit descriptor, meaning it samples a set of 256 pixel pairs to construct the bit-string. There are also 127, and 512 bit variants, but the authors found that 256 samples performed near to optimal [34].

The authors propose five different approaches for choosing the sample pairs. These are illustrated in figure 3.12. The four first ones apply random sampling in different ways, and they all achieve higher recognition rates than the fifth which uses an ordered approach [34]. ORB uses BRIEF descriptors where the point pairs are selected with the second method.

A downside to the BRIEF descriptor is that it does not have the same degree of rotation invariance as SIFT does.

3.2.5 ORB Feature Detector

Oriented FAST and rotated BRIEF (ORB), as the name suggests, combines FAST and BRIEF.

ORB uses FAST together with an image pyramid for keypoint detection. An image pyramid is made by reducing the size of the image by a constant factor several times. This gives the detected features some degree of scale invariance (SIFT uses the same method to achieve scale invariance). Harris corner criterion is then applied to the detected points in order to filter out bad ones.

ORB uses a centroid technique to compensate for rotation. The center of mass and orientation of an image patch is calculated by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{3.68}$$

$$\theta = \operatorname{atan2}(m_{01}, m_{10}) \tag{3.69}$$

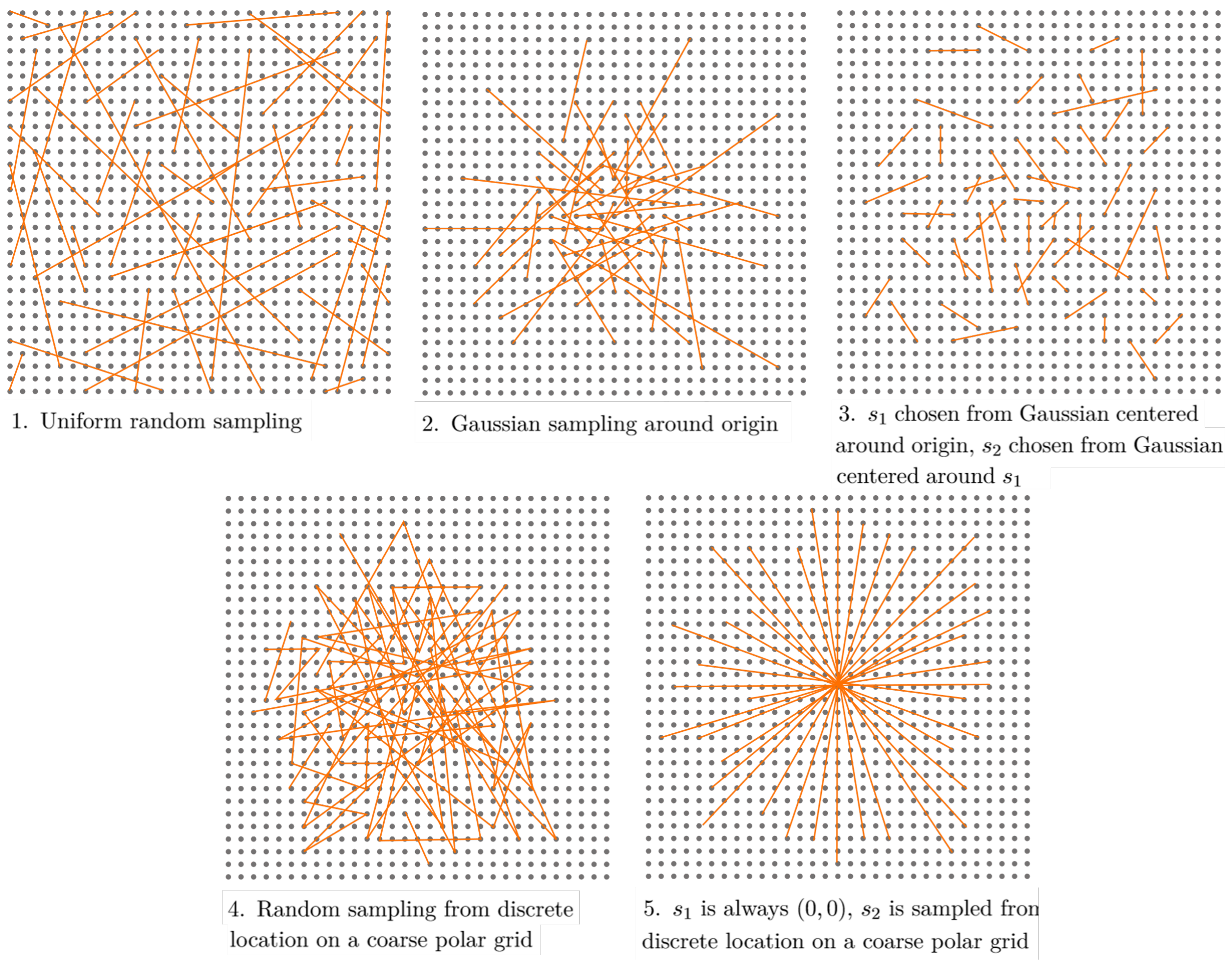

1. Uniform random sampling
2. Gaussian sampling around origin
3. s_1 chosen from Gaussian centered around origin, s_2 chosen from Gaussian centered around s_1
4. Random sampling from discrete location on a coarse polar grid
5. s_1 is always $(0, 0)$, s_2 is sampled fron discrete location on a coarse polar grid

Figure 3.12: Illustration of different approaches for choosing pairs of pixels for comparison in the BRIEF descriptor. Orange lines are drawn between point pairs.

Where m_{pq} it the image moment about a given axis:

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \tag{3.70}$$

This is done for a 31×31 patch around each detected keypoint. Given the center of mass, C, and the orientation, θ, the coordinates of all point pairs used in the BRIEF descriptor can be rotated around C by the amount θ.

3.2.6 Feature Matching

The brute force way of matching features is to compare each feature descriptor in the query image to each descriptor in the second image. For an ORB descriptor, the similarity between two descriptors is their hamming distance. A lower hamming distance means a closer match. Some query descriptors might match with several descriptors in the second image. For instance, repetitive structures will have similar descriptors and thus cause matching ambiguities.

In the SIFT paper [26], Lowe proposed a method for eliminating ambiguous matches referred to as Lowe's ratio test. For a given query descriptor, find the two closest matches in the second image.

The closest match is accepted if it is substantially better than the second best. Given the best distance d_1, and the second best d_2, Lowe suggest that their ratio should be less than 0.7:

$$\frac{d_1}{d_2} < 0.7 \tag{3.71}$$

If this test is not passed, the query descriptor does not have any matches in the second image. Since this algorithm operates with distances, it also works with ORB features.

There will probably still remain some wrong data associations. When the matched features are used to camera's transformation between images, the wrong data associations will not conform to the transformation of the correct data associations. Thus they can be treated as outliers and removed with RANSAC. Random Sample Consensus (RANSAC) is an algorithm for outlier rejection. Given a set of observed data, select a random subset and compute a fitted model based on the subset. Apply this model to the rest of the observed data. The observations that deviate from the fitted model by more than a selected threshold are rejected as outliers. This process is repeated for new random subsets, and the fitted model with the fewest outliers is the one that best describes the dataset.

3.2.7 Camera Model

In order to use the camera for SLAM, its extrinsics and intrinsics must be known. Camera extrinsics are parameters for where the camera is in the world. Camera intrinsics are the parameters that dictate how the world is projected into the camera image.

The goal is to find how a point in the world P_w maps to a point p in pixel coordinates. Assuming ideal projection, this transformation is expressed by equation 3.72.

$$s \cdot p = KHP_w \tag{3.72}$$

s is an arbitrary scaling factor which is not part of the model. H transforms the world point to the camera frame P_c:

$$P_c = HP_w \tag{3.73}$$

K transforms points in the camera frame to image coordinates. K is also known as the camera matrix.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.74}$$

Projection from the camera frame to 2D pixel coordinates is thus

$$s \cdot p = KP_c \tag{3.75}$$

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{3.76}$$

The camera matrix is composed by the focal lengths f_x and f_y, and the principal point (c_x, c_y). The focal lengths are equal if the pixels are perfectly square. The principal point is usually close to the image center. Its offset is determined by a offset between the physical sensor and the camera lens. In equation 3.78, the 3-by-4 projective transformation matrix maps 3D points into 2D pixel coordinates representing normalized camera coordinates:

$$x_d = \frac{X_c}{Z_c} \qquad y_d = \frac{Y_c}{Z_c} \tag{3.77}$$

$$Z_c = \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{3.78}$$

H in equation 3.72 is the extrinsic matrix and is a homogeneous transformation that transforms the world point to the camera coordinate system. R is the extrinsic parameters, while the P_c represents the points in the camera coordinate system:

$$H = \begin{bmatrix} r_1 & r_{12} & r_{13} & t_x \\ r_2 & r_{22} & r_{23} & t_y \\ r_3 & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{3.79}$$

$$P_c = HP_w \tag{3.80}$$

hence:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_1 & r_{12} & r_{13} & t_x \\ r_2 & r_{22} & r_{23} & t_y \\ r_3 & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.81}$$

Combining projection and homogeneous transformation, the projective transformation is obtained to map 3D points in world coordinates into 2D points in pixel coordinates:

$$Z_c \begin{bmatrix} x_d \\ y_d \\ 1 \end{bmatrix} = H \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} r_1 & r_{12} & r_{13} & t_x \\ r_2 & r_{22} & r_{23} & t_y \\ r_3 & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.82}$$

By interconnecting the equations for intrinsic and extrinsic parameters together the output of $sp = AHP_w$ can be written as:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_1 & r_{12} & r_{13} & t_x \\ r_2 & r_{22} & r_{23} & t_y \\ r_3 & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.83}$$

If $Z_c \neq 0$ and using equation 3.81, one can convert equation 3.83 equivalent to:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x \frac{X_c}{Z_c} + c_x \\ f_y \frac{Y_c}{Z_c} + c_y \end{bmatrix} \tag{3.84}$$

All lenses are subject to some distortion. Fisheye lenses are especially affected by this. For ultra wide-angle lenses a different camera model should be used. The standard camera model works for the fisheye camera on the Loomo. Lens distortion is a non-linear problem, but the standard model compensates by doing a linear approximation. Thus the compensation for the fisheye camera will be less accurate than for a regular camera.

The coefficients k and s are the radial and tangential distortion respectively. The parameters x_d and y_d are distorted points, while x_{dd} and y_{dd} are tangential distorted points. Equation 3.85 manage distortion of this pinhole camera model.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x_{dd} + c_x \\ f_y x_{dd} + c_y \end{bmatrix} \tag{3.85}$$

where:

$$\begin{bmatrix} x_d \\ y_d \end{bmatrix} = \begin{bmatrix} \frac{X_c}{Z_c} \\ \frac{Y_c}{Z_c} \end{bmatrix} \tag{3.86}$$

$$\begin{bmatrix} x_{dd} \\ y_{dd} \end{bmatrix} = \begin{bmatrix} x_d(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x_d y_d + p_2(r^2 + 2x_d^2) + s_1 r^2 + s_2^r 4 \\ y_d(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y_d^2) + 2p_2 x_d y_d + s_3 r^2 + s_4 r^4 \end{bmatrix} \tag{3.87}$$

$$r^2 = \left[x_d^2 + y_d^2\right] \tag{3.88}$$

OpenCV and Matlab have tools for finding the camera intrinsics, including the radial and tangential lens distortions. By taking pictures of a regular pattern, for instance a checker board where the size and number of squares is known, these tools can perform a camera calibration. By detecting keypoints in the images fitting them to the camera model, the camera intrinsics can be found.

3.3 Odometry and Motion Model of the Loomo

Odometry is the process of using sensor data to estimate movement over time. The odometry for the Loomo also relies on differential drive kinematics.

All measurements are uncertain. A motion model that also encodes this uncertainty is needed when solving the SLAM problem.

Odometry is closely related to, and sometimes used interchangeably with dead reckoning. Dead reckoning is estimating motion relative to a previously known pose. Odometry does not explicitly estimate motion relative to a previously known pose, though it is usually implied.

3.3.1 Odometry of a Differential Drive Robot

The Loomo is a differential drive robot. The odometry uses the left and right wheel encoders together with the wheel geometry (diameter and base width) to estimate the Loomo's movement. The movement is estimated in 2D space, so the Loomo's pose is described by a position x, y and heading θ.

The odometry is solved incrementally by updating the pose from the previous sample based on an input u_k.

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, u_k\right) \tag{3.89}$$

The input u_k is the linear displacement of the left and right wheel Δs_L and Δs_R. The linear displacement of a wheel is calculated by counting the ticks, n_{tick}, from its rotary encoder. The number of ticks per revolution, n_t is known, and the wheel diameter, D, is known.

$$\Delta s_L = \frac{n_{tick,L}}{n_t} \cdot \pi \cdot D \qquad \Delta s_R = \frac{n_{tick,L}}{n_t} \cdot \pi \cdot D \tag{3.90}$$

The function for incrementing the state, $f(\mathbf{x}_{k-1}, u_k)$, can be written on the following form:

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \Delta\mathbf{x} \tag{3.91}$$

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta\theta \end{bmatrix} \tag{3.92}$$

The calculations for $\Delta\mathbf{x}$ assume no slip between the wheels and ground, and that the increments are small. Small increments allow for the following approximations:

- The robot moves along a circular path between samples, where a straight path is equivalent to a circular path with radius approaching ∞ (illustrated in figure 3.13)
- The distance between two points on an arc is equal to the arc length between those points (illustrated in figure 3.13)
- The sine of an angle is equal to the angle, $\sin(\Delta\theta) = \Delta\theta$

Using these approximations, the displacement of the robot between samples equals the average linear displacement of the wheels (equation 3.93). By using the trigonometric relations illustrated in figure 3.14, the change in angle between samples is expressed by equation 3.94.

$$\Delta s = \frac{\Delta s_L + \Delta s_R}{2} \tag{3.93}$$

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{b} \tag{3.94}$$

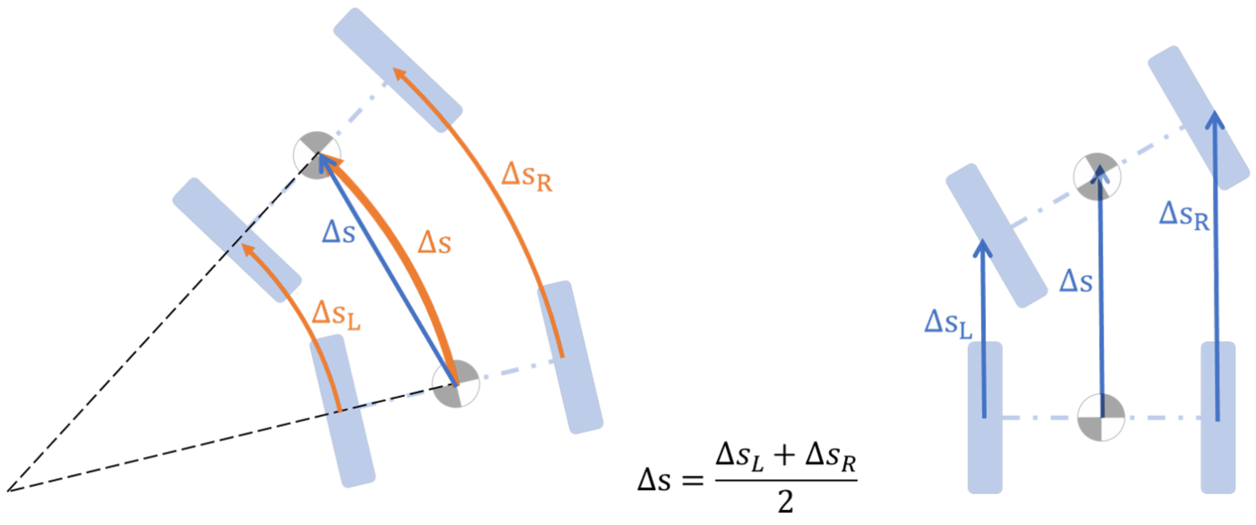
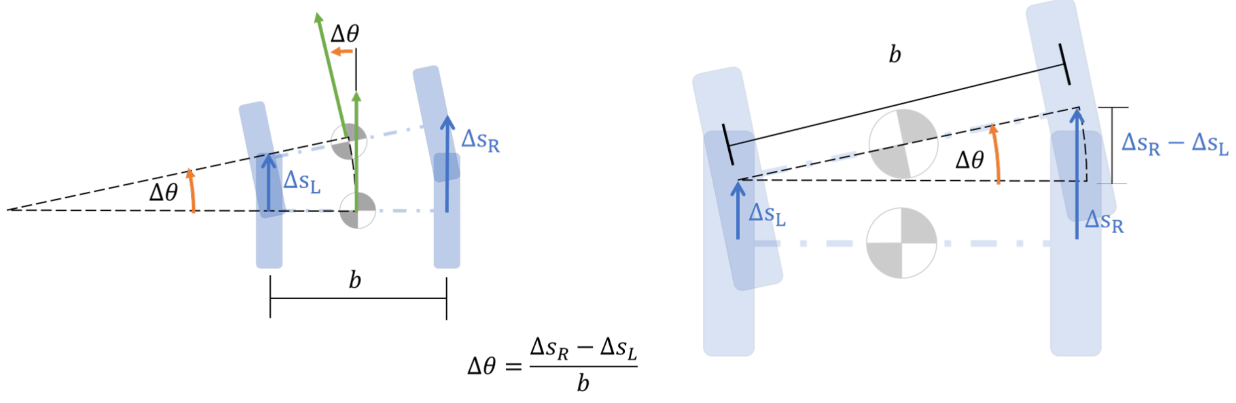


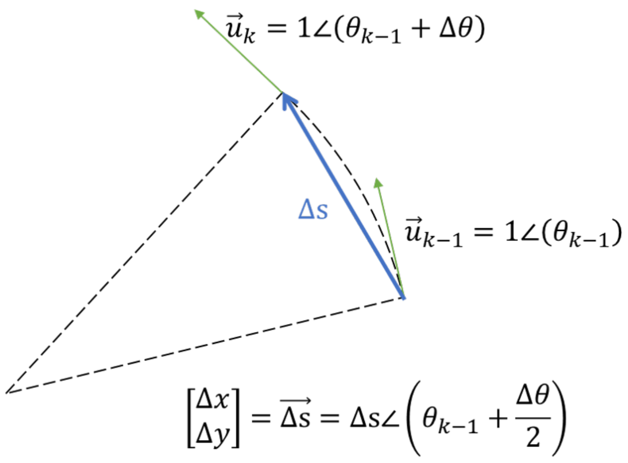
Figure 3.13: Left: Length of an arc between two points approximated to the distance between the points. Right: Robot displacement expressed as the average of the displacements of the wheels

The change in position Δx and Δy is dependent on the heading of the robot. The displacement Δs can be expressed in vector form as illustrated in figure 3.15. Δx and Δy are the components of this vector. The function for updating the robot pose is thus given by equation 3.95.



$\Delta\theta$
Δs_R
Δs_L
b
$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{b}$
$\Delta s_R - \Delta s_L$

Figure 3.14: Change in robot angle derived from trigonometric relations



$\vec{u}_k = 1\angle(\theta_{k-1} + \Delta\theta)$
Δs
$\vec{u}_{k-1} = 1\angle(\theta_{k-1})$
$\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix} = \overrightarrow{\Delta s} = \Delta s\angle\left(\theta_{k-1} + \frac{\Delta\theta}{2}\right)$

Figure 3.15: Expressing Δs as a vector to find the change in position

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, u_k\right)$$
$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \Delta s \cdot \cos\left(\theta_{k-1} + \frac{\Delta\theta}{2}\right) \\ \Delta s \cdot \sin\left(\theta_{k-1} + \frac{\Delta\theta}{2}\right) \\ \Delta\theta \end{bmatrix} \tag{3.95}$$

3.3.2 State Estimation Based on Wheel Encoder Odometry

A state estimation is expressed by the predicted state in equation 3.96, and the uncertainty of that prediction in equation 3.97. The state estimation is a non-linear function, so it can not be applied directly to the covariance Σ_k. The uncertainty is updated by computing the Jacobian of $f\left(\mathbf{x}_{k-1}, u_k\right)$ with respect to the state and with respect to the input (equation 3.98).

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, u_k\right) \tag{3.96}$$

$$\Sigma_k = F_x \cdot \Sigma_{k-1} \cdot F_x^T + F_u \cdot Q_k \cdot F_u^T \tag{3.97}$$

$$F_x = \frac{\partial f(\mathbf{x}_{k-1}, u_k)}{\partial \mathbf{x}_{k-1}}, \qquad F_u = \frac{\partial f(\mathbf{x}_{k-1}, u_k)}{\partial u_k} \tag{3.98}$$

The Jacobians are as follows:

$$F_x = \begin{bmatrix} 1 & 0 & -\sin\left(\theta_{k-1} - \frac{\Delta s_L - \Delta s_R}{2\cdot b}\right) \cdot \frac{\Delta s_L + \Delta s_R}{2} \\ 0 & 1 & \cos\left(\theta_{k-1} - \frac{\Delta s_L - \Delta s_R}{2\cdot b}\right) \cdot \frac{\Delta s_L + \Delta s_R}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{3.99}$$

The following substitution is done in F_u

$$\Theta = \theta_{k-1} - \frac{\Delta s_L - \Delta s_R}{2 \cdot b} \tag{3.100}$$

$$F_u = \begin{bmatrix} \frac{1}{2}\cos\left(\Theta\right) + \frac{\Delta s_L + \Delta s_R}{4\cdot b} \cdot \sin\left(\Theta\right) & \frac{1}{2}\cos\left(\Theta\right) - \frac{\Delta s_L + \Delta s_R}{4\cdot b} \cdot \sin\left(\Theta\right) \\ \frac{1}{2}\sin\left(\Theta\right) - \frac{\Delta s_L + \Delta s_R}{4\cdot b} \cdot \cos\left(\Theta\right) & \frac{1}{2}\sin\left(\Theta\right) + \frac{\Delta s_L + \Delta s_R}{4\cdot b} \cdot \cos\left(\Theta\right) \\ -\frac{1}{b} & \frac{1}{b} \end{bmatrix} \tag{3.101}$$

Q_k is the covariance of the process noise. It encodes the uncertainty of the input.

$$Q_k = \begin{bmatrix} k_R \cdot |\Delta s_R| & 0 \\ 0 & k_L \cdot |\Delta s_L| \end{bmatrix} \tag{3.102}$$

The rotary encoder ticks are counted fairly reliably. The uncertainty comes mostly from inaccuracies when measuring the wheel diameter and base width. For instance, tire inflation is a source of error. The tire deformation varies with load, for instance whether a person is riding the Loomo or not.

Figure 3.16 demonstrates both the accumulated error in odometry, and the estimated uncertainty of the position. The figure is drawn by simulating a differential drive robot driving in a straight line, but for each run it is given a randomized deviation to the wheel geometry. The trajectories are thus the ground truth of the motion, while the predicted state moves along the x-axis.

The uncertainty of the predicted state is drawn with ellipses (they only show the uncertainty of the x and y position). The size is based on three standard deviations (3σ) from the predicted state. In a Gaussian distribution, 99.7% of the values lie within 3σ from the mean. This is clearly not the case for the trajectories (they are grouped in more of a banana-shape than within the ellipses). This is because of the approximation error in the linearization used to calculate Σ_k. It can be seen that the inaccuracy in the linearization also accumulates.

Robot heading, θ, is the most significant of the state parameters x, y and θ in terms of its influence on accumulated dead-reckoning errors. The estimated travelled distance is more accurate than the overall state estimation. The length of each trajectory has much lower variance than the states of each trajectory.

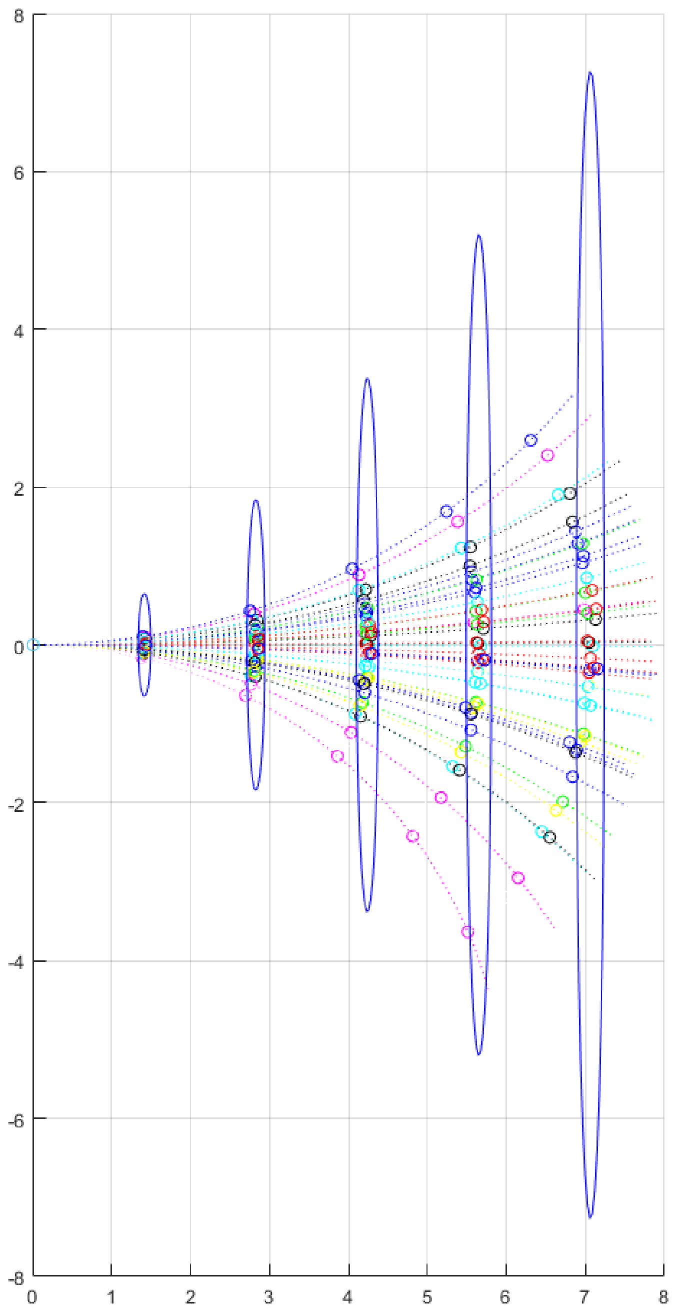


Figure 3.16: Simulated trajectories of several differential drive robots where each robot is given a randomized deviation in wheel geometry. The ellipses are the calculated uncertainties of the position using three standard deviations (3σ)

3.4 Simulating Swarm Intelligence for Collaborative Exploration

This section will go into detail on the theory and ideas that is used to design the simulation of collaborative exploration. The simulation will consist of a randomly generated unknown building, and a variable amount of bots exploring that building. The goal of the simulation is to determine the effectiveness of different exploration methods, and to compare the efficiency of exploring the building as the number of bots increase.

3.4.1 Emergent Behaviour

With the use of multiple robots, research within swarm robotics and flocking behavior is highly relevant for this thesis. The paper 'Flocks, Herds and Schools: A Distributed Behavioral Model'. [8] describes an algorithmic approach to simulate the behaviour of flocking birds, herds of animals or schools of fish, As an alternative to scripting the path of each object individually. In this algorithm each BOID; Bird-oid, behave as independent actors and make decisions based on their perception of the local environment. By following these rules, the whole flock seems to act as a single coherent unit, and we observe what is called emergent behaviour.

The following rules are used in the paper to create flocking behaviour:

Separation: Avoid collision with nearby flock mates.

Cohesion: Stay close to nearby flock mates.

Alignment: Stay in formation with nearby flock mates.

This rule structure has inspired the strategies which is used to control the bots in the simulation, and they will be explained in the implementation chapter.

3.4.2 Occupancy Grid Map

This is a way to represent the environment of a mobile robot. Data from distance sensors and the pose of the robot is used to construct the map as the robot discovers its environment. The map is usually discretized into cells and the state of the cell tells if the cell is free space or an obstacle.

3.4.3 Configuration Space

Configuration space, or C-space represents the set of all transformations that can be applied to a robot given its kinematics. A robot is mapped to a single point, while the obstacle regions are expanded by sliding the robot shape around the obstacles. C-space is useful for motion planning in static environments. However, the environment is not static, both because it is unknown and because there are other Loomo robots present. By using their distance sensors and the swarm rules, the Loomo robots will actively avoid collision with obstacles and other robots as the environment is explored, and thus functions as an approximation to a C-space.

3.4.4 Potential Field

Potential fields are a different method of motion planning. Figure 3.17 show an example of a potential field. The goal position has an attractive component and obstacles have a repulsive component. The steering vector of a robot navigating the work space can be set by By calculating the gradient of the field.

$$\nabla U = \left[\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}\right] \tag{3.103}$$

This is a basic overview of how the potential field can be used. However, a simplified, discretized variant of the potential field method is used as an additional cost in the A* algorithm. This is explained in the implementation chapter in section 4.3.6.

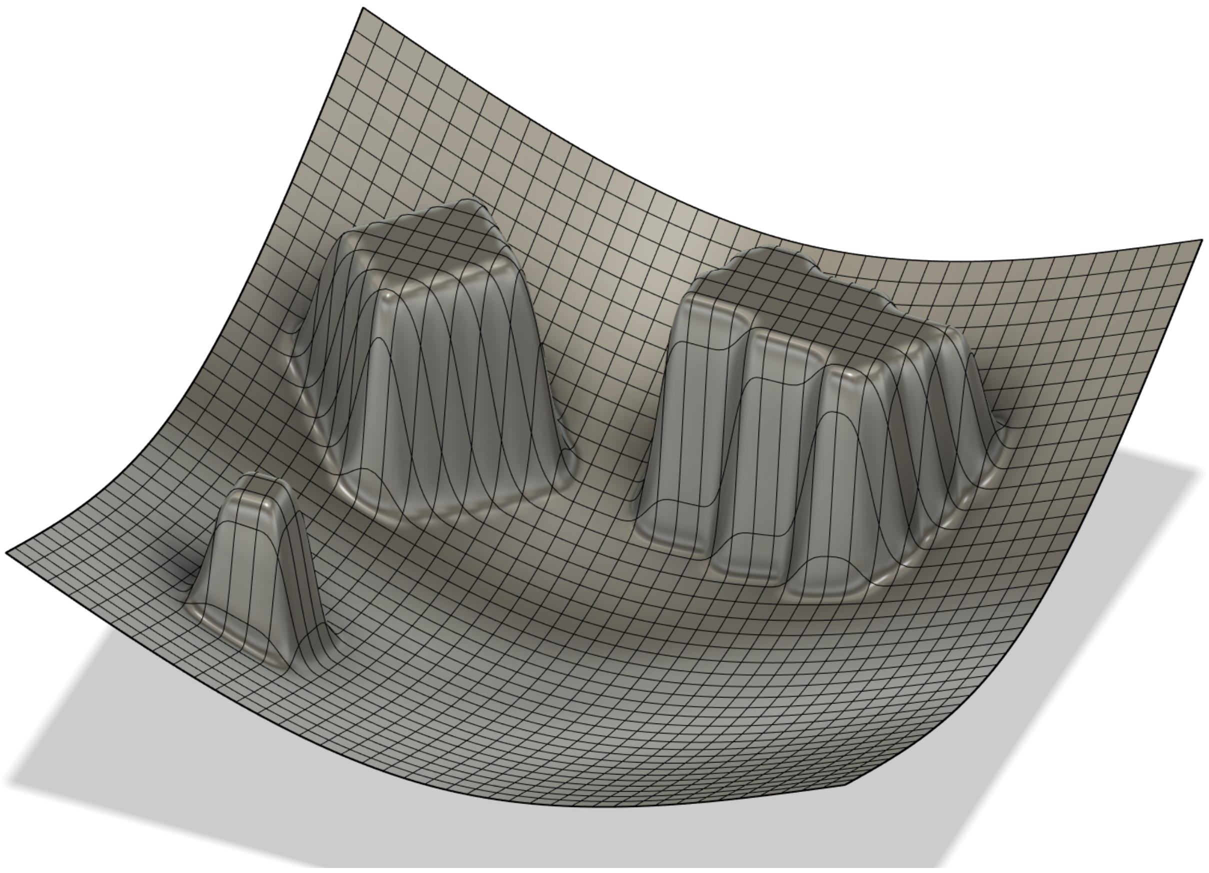


Figure 3.17: Potential field example

3.4.5 Pathfinding

Dijkstra's algorithm is a pathfinding algorithm that will find the shortest path between two nodes on a graph. A disadvantage with this algorithm is that it has to explore all of the nodes on the graph to make sure the shortest path is found. This increases computational load. The A* algorithm is an extension of Dijkstra, the addition being a heuristic, $h(n)$. It represents the distance to the goal node from the evaluated node. The heuristic makes it possible to prioritize which nodes to explore first. By prioritizing, the amount of nodes that is explored before the goal is reached is reduced significantly compared to Dijkstra.

Equation 3.104 show the cost function used in the the A* algorithm. Figure 3.18 illustrates how the cost function is used. The bot is scanning its eastern neighbor, determining the cost of traveling to that node, $g(n)$, and calculating the cost of distance from the neighbor node to the goal node $h(n)$. The total cost $f(n)$ is used to compare this node to other nodes, and prioritize if this node is worth exploring.

$$f(n) = g(n) + h(n) \tag{3.104}$$

Where

Symbol	Description	Unit
n	Node	-
$f(n)$	Total cost of the node	-
$g(n)$	Cost of traveling from current note to neighbor node	-
$h(n)$	Heuristic: Cost of neighbor node to goal node	-

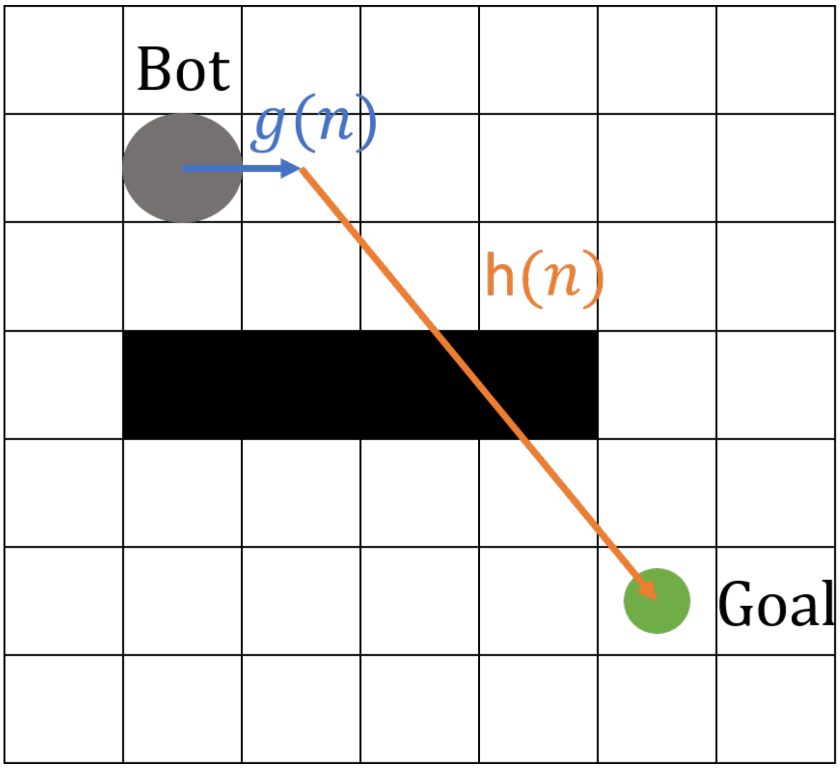


Figure 3.18: A* cost factors

3.4.6 Velocity Control

Every rule controlling the Loomo will provide a resultant vector based on their different criteria, all the rules are multiplied with their weight and the sum of all the resultant vectors are divided by the total number of resultant vectors.

$$\vec{V_c} = \frac{1}{n}\sum_i (w_i \cdot \vec{V_{r_i}}) \tag{3.105}$$

$$v = \left|\vec{V_c}\right| \cdot \cos(\theta_r - \theta_L) \tag{3.106}$$

$$\omega = \left|\vec{V_c}\right| \cdot \sin(\theta_r - \theta_L) \tag{3.107}$$

Where

Symbol	Description	Unit
$\vec{V_c}$	Resultant vector of combined rules	m/s
v	Linear velocity setpoint	m/s
ω	Angular velocity setpoint	m/s
n	Total number of resultant vectors	-
$\vec{V_{r_i}}$	Resultant vector of individual rule	m/s
w_i	Weight factor of individual rule	-
θ_r	Angle of resultant vector in global space	rad
θ_L	Angle of Loomo in global space	rad

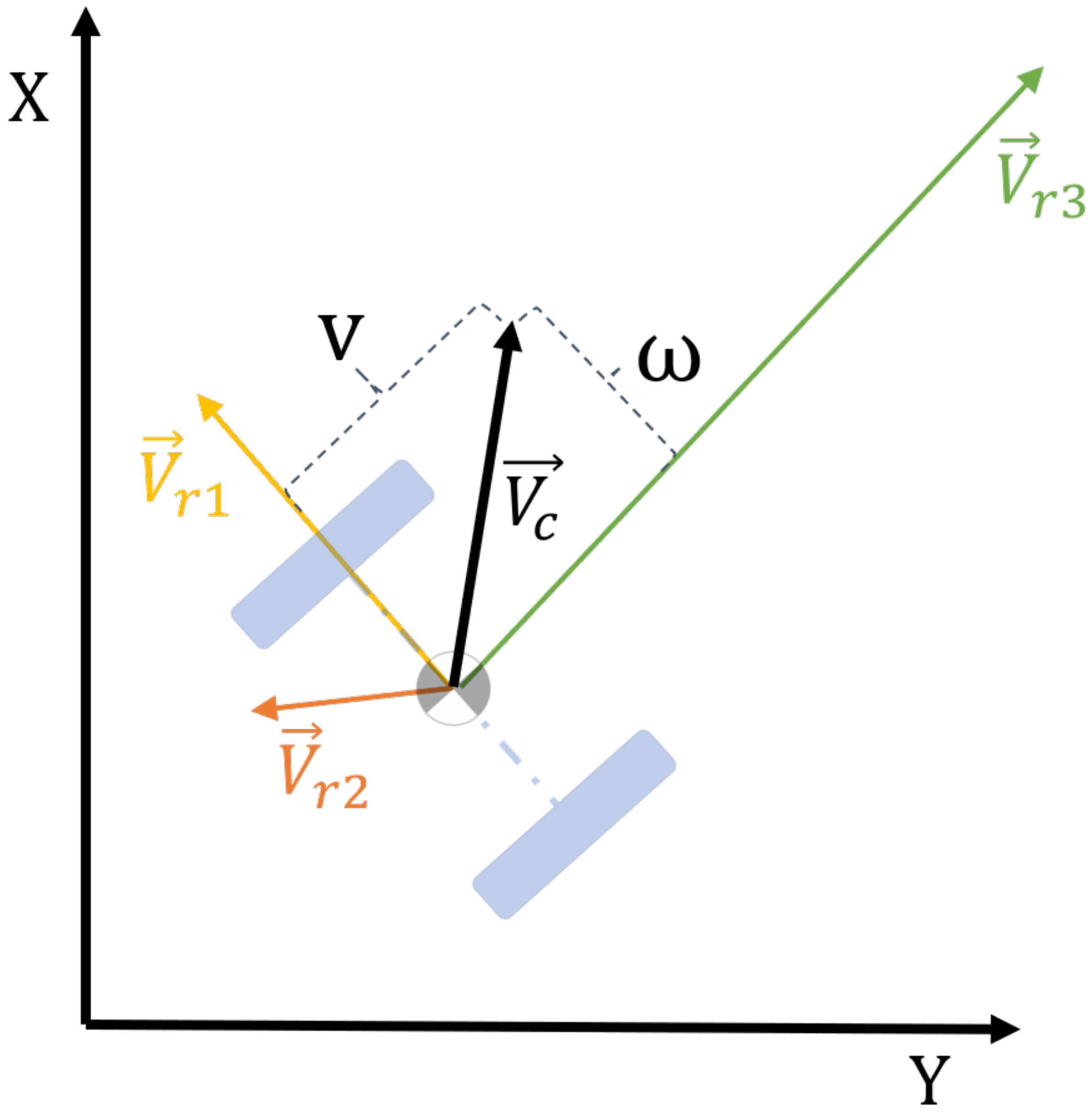


Figure 3.19: Velocity vector

3.5 Machine Learning

Machine learning is the concept of using algorithms that automatically improves through experience. ”Experience“ in the context of machine learning is usually referred to as sample data or training data. The purpose of a machine learning algorithm is to build a model that can make outcome predictions or decisions without explicitly being programmed to do so. Machine learning is thus suited for applications where it is difficult for humans to build these models. For instance, in computer vision it is difficult to ”tell“ a computer exactly what properties a set of pixels needs in order to tell whether there is a person in the image or not.

3.5.1 Supervised Learning

A supervised machine learning model has knowledge of the input and output data, and uses that data to estimate future outputs.

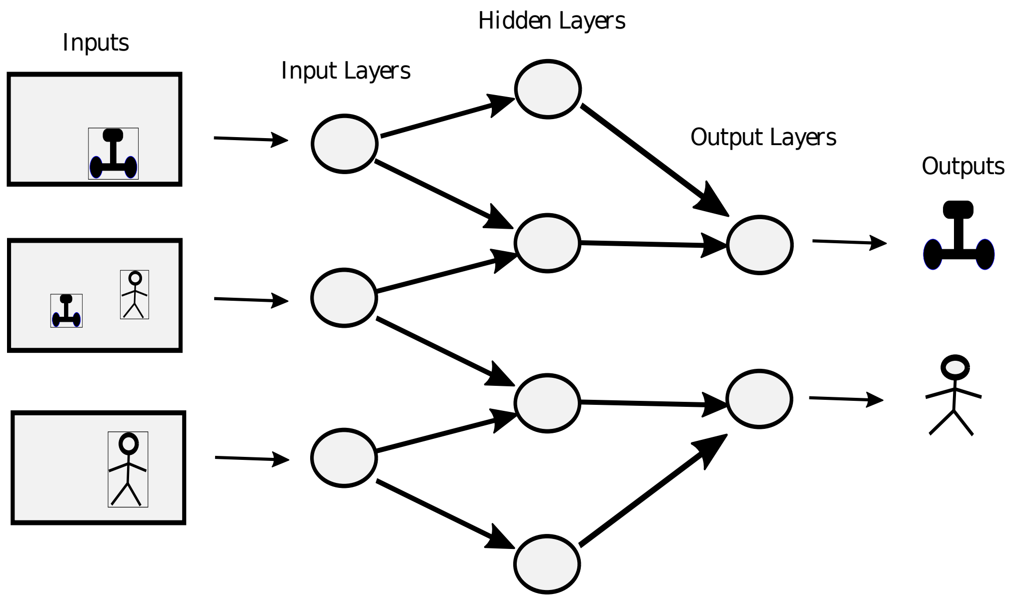


Figure 3.20: Supervised learning (the correct output is known)

All supervised learning algorithms can be categorized as a classification or regression problem. A regression algorithm tries to predict continuous data or responses such as power consumption or stock prices, while classification examples estimates direct response like speech recognition or detecting objects in images.[35]

Savitha [36, p. 8] compares supervised learning with an environment controlled by a ”teacher“. The teacher has knowledge of the environment and tries to adjust how the data is treated in order to minimize output errors. Unsupervised learning on the other hand is when the output is unknown, and the machine learning algorithm tries to find hidden patterns in the input data.[35]

3.5.2 Deep Learning

Deep learning is a branch of machine learning where the purpose is to enable computers to perform tasks that come natural for humans. This can for instance be to understand natural language (through text or audio), or to recognize objects in images.[37]

3.5.3 Neural Networks

A neural network in machine learning takes inspiration from how the brain works. A node, i.e. a neuron, in the network simply receives inputs, and provides outputs. Neural networks in machine learning usually have a more ordered structure than the neurons in a brain. There is an input layer, an output layer, and some amount of hidden layers in between. The most common structure is that the output of each node connects to the inputs of all the nodes in the next layer.

A neuron has a weighing factor, w, for each of its inputs, x. The weighted inputs are summed. A very simple type of neuron, called a perceptron [38], outputs 1 if the sum is above a threshold, and 0 if the sum is below. This threshold is called a bias, b, and is also a parameter of a neuron. The perceptron is a type of neuron where the activation function is a comparison with boolean output. The activation function is often so that the output will be within a lower and upper boundary. For instance, a Sigmoid function will bound all real values so they lie between 0 and 1. The output of a single neuron is expressed in equation 3.108. The weights and biases of a neural network are the parameters that are adjusted through the training process.

$$output = f\left(\sum_{i=1}^{n}(w_i x_i) + b\right) \tag{3.108}$$

Figure 3.21 illustrates an example of a neural network. If the input is a 416×416 pixel grey-scale image, the input layer will have one node for each pixel. If it is a classification algorithm, the outputs can for instance be the probability of there being a cat or a dog in the image.

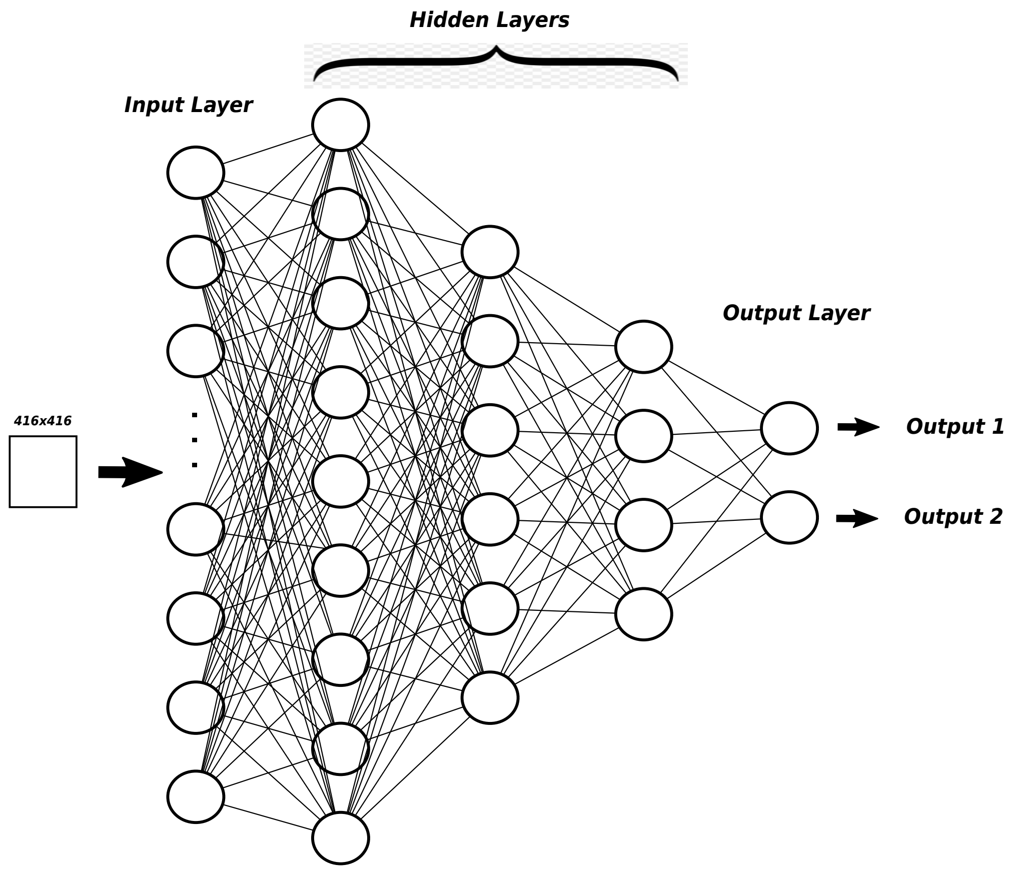


Figure 3.21: A neural network

3.5.4 Convolutional Neural Networks (CNN)

As illustrated in figure 3.22, a CNN has additional convolution layers that perform mathematical operations as the data flows through.[39, p. 101]

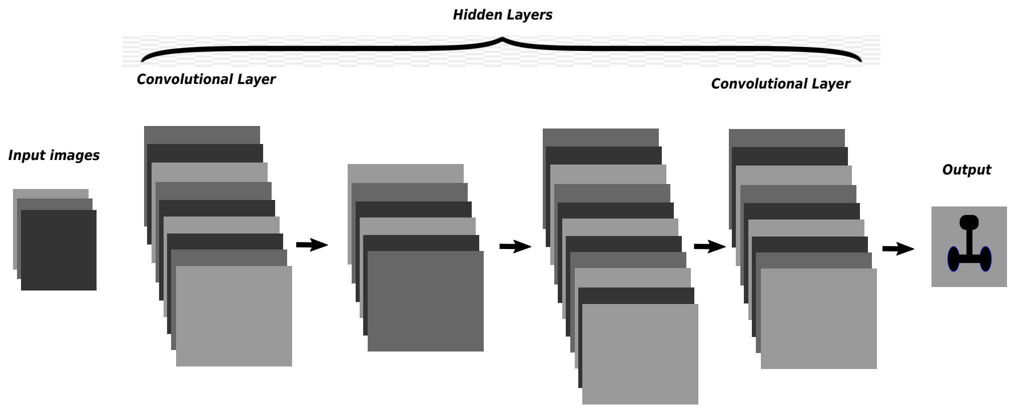


Figure 3.22: CNN workflow

CNNs use 2D convolutional layers, making this approach ideal for computing 2D data such as images. The network breaks the images down into features using several sets of layers. As the name suggests, Deep Neural Networks can be very deep. ResNet for instance, a gigantic classification network can contain 152 layers [40]. The neural network made in this thesis contains 15 layers. The middle layers are often referred as hidden layers since the inputs and outputs of those are not visualized [37]. However, to provide good output data, these hidden layers needs smart mathematical operations inside. Examples of such mathematical operations are convolutional layers, maxpooling and YOLO-output layers [41].

Convolutional operations are repeatedly used in CNNs thus making such operations important when designing neural networks. How convolutional operations work are described in 3.2.2.

3.5.5 Maxpool

Maxpooling can be applied to the output of a convolutional layer to reduce the resolution, and thus the computational load, for following layers. It can also help to avoid overfitting as described in section 3.5.9. An image is segmented into *pools*, and only the maximum value in each pool is carried over to the output of the maxpool-operation. Figure 3.23 has 12 2×2 pools.

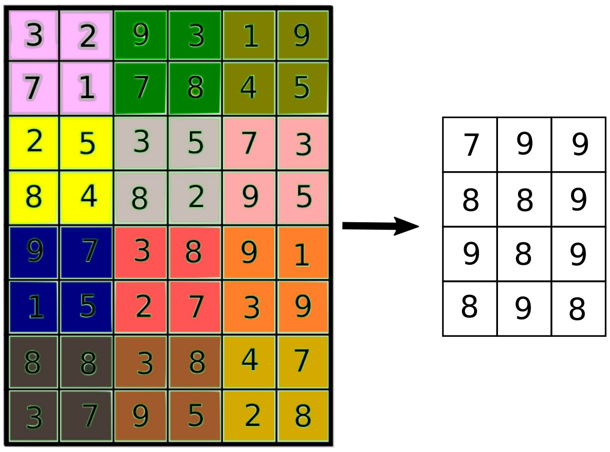


Figure 3.23: Maxpooling

3.5.6 Activations

Each layer inside the CNN is provided with information from the previous layer. A convolutional operation inside the current layer manipulates its parameters and provides some new information to the next layer. This new output information from the layer is called an *Activation*. The Activations are determined by an Activation Function that holds the output number between some lower and upper limit, for instance a Sigmoid function. The closer the upper limit for the output, the stronger activation. Figure 3.24 shows the activation of a convolution operation from a specific filter that highlights gradients in the image. In figure 3.24, the activation from a convolutional operation is visualized. Figure 3.25 shows the activation when maxpooling is applied in addition to the convolution.

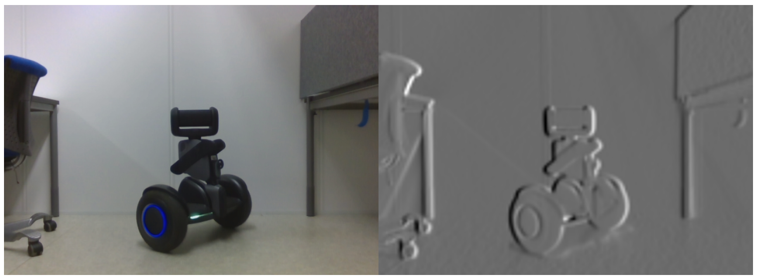


Figure 3.24: Convolutional activation

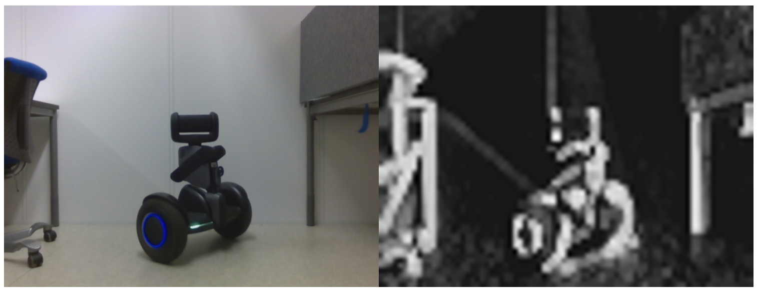


Figure 3.25: Maxpool activation

3.5.7 Object Detection

As mentioned in 3.5.2, Deep Learning is a machine learning strategy to find and classify objects in images. The main difference between classification and object detection algorithms is that bounding boxes are drawn around the region of interest in detection algorithms. A bounding box is represented by five parameters: x, y, w, h and the associated class [42, p. 2] (for instance ”Loomo“ or ”Person“). x and y are coordinates representing the origin point. The width w and height h determines the size. Bounding boxes need to be labeled manually to provide training data, and the answer key is called the ground truth. A sufficiently large set of labeled data is required to train a CNN. During the training process, the output bounding box of the CNN can be compared with the ground truth to evaluate its performance.

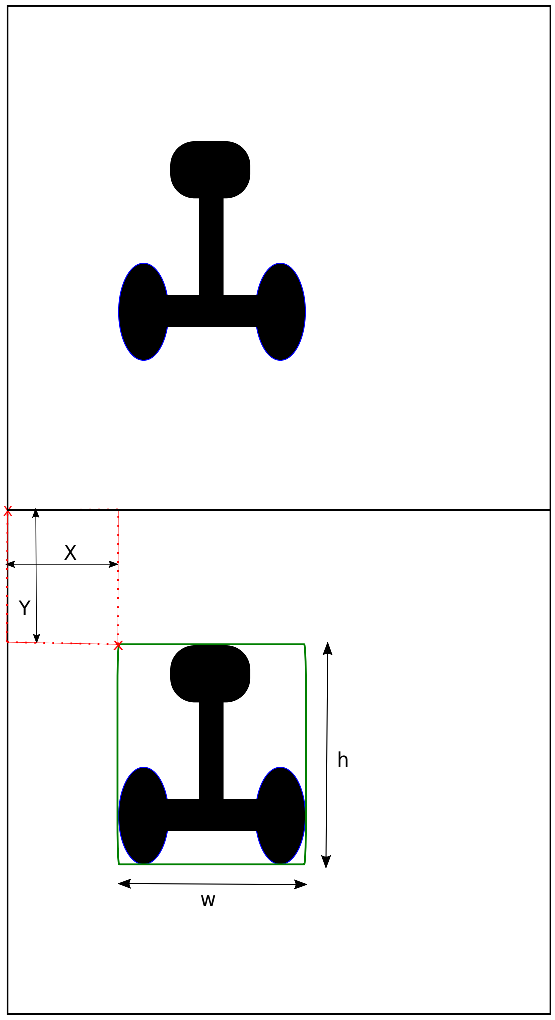


Figure 3.26: Bounding box

There are a plethora of object detection algorithms available. For instance, Regions with CNN features (R-CNN) [43], Fast-RCNN [44], Faster-RCNN [45] and Single Shot MultiBox detector (SSD) [46]. This report uses the You Only Look Once (YOLO) [42] algorithm.

3.5.8 You Only Look Once (YOLO)

You Only Look Once (YOLO) is an object detection algorithm that uses a CNN structure. It is designed for real-time object detection and outperformes R-CNN and its successors. The system contains a single convolutional network where the input image is being resized into a desired form(448x448 in YOLO first version). The output returns bounding boxes like shown in Figure 3.27, if the confidence score is above a given threshold.

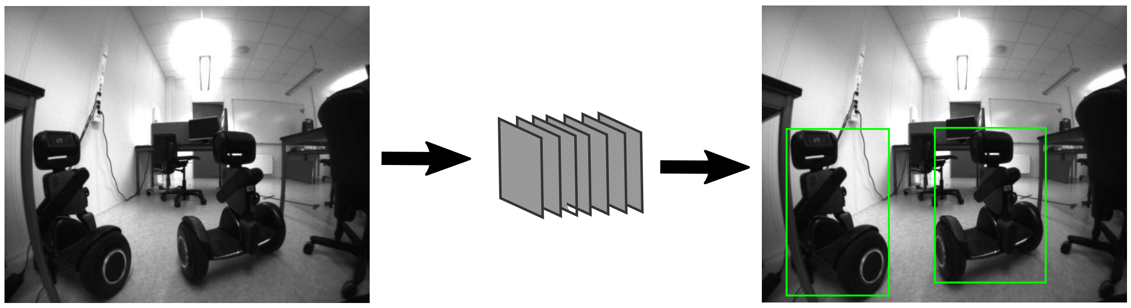


Figure 3.27: Resize image(1), Run CNN(2), Predict Bounding Boxes(3)

The YOLO algorithm is fairly new. The first version was published in 2016 [42], and since then, there have been published several improved versions.

YOLOv2 is second generation YOLO, using same workflow and principles. This version introduces a new strategy for calculating bounding boxes by using Anchor boxes [47]. Anchor boxes are predefined bounding boxes that all have similar width and height as the bounding boxes of the objects you want to detect. The neural network uses the anchor boxes as mutable templates to help it estimate bounding boxes. Including anchor boxes to the neural network improves the ability to detect multiple objects, overlapping objects, and objects of different sizes. The top image in figure 3.28 is covered with tiled anchor boxes. Anchors with low confidence are neglected, while the ones with high confidence are selected using Non Maximum Suppression. Non Maximum suppression is a method for selecting the best candidate out of a group of adjacent candidates. Adjacent candidates in this case is overlapping bounding boxes. Finally the objects are detected as shown in the bottom image in figure 3.28. This example uses only one anchor, while the final model in this thesis uses six. Originally, YOLO predicted 98 bounding boxes per image, but when using anchor boxes, more than 1000 boxes are predicted. This led to growth in Mean Average Precision (mAP) (explained in section 3.5.9) and allows more opportunities when resizing training images. This is a significant improvement compared to the first YOLO version.

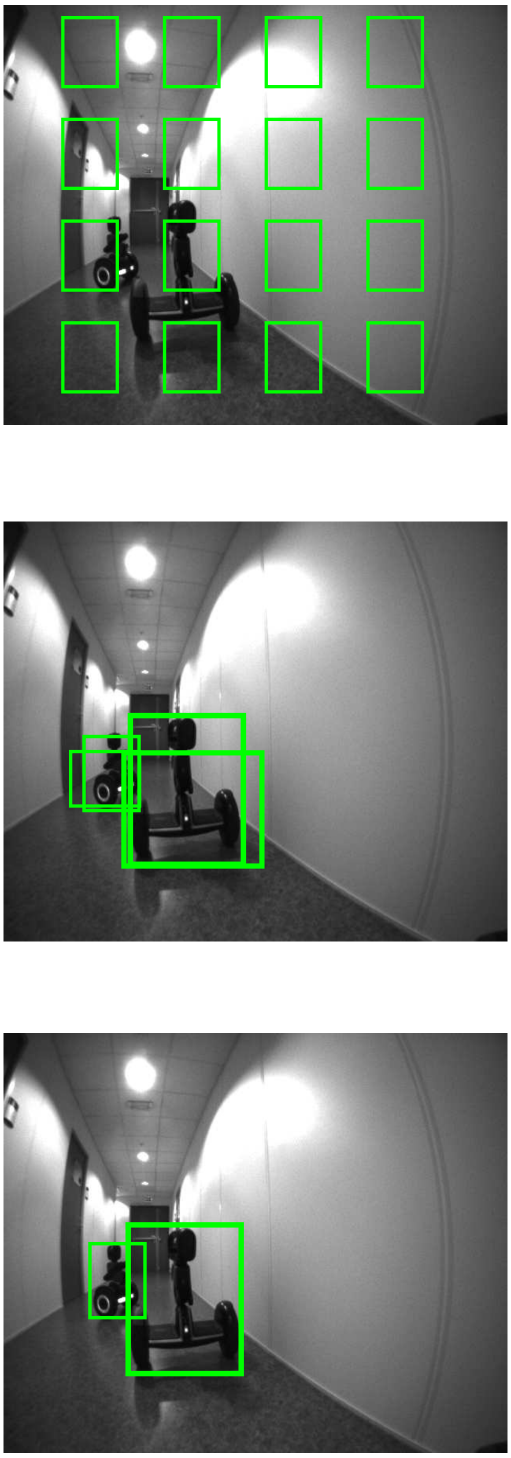


Figure 3.28: Anchor boxes

A third YOLO generation was published in 2018 [41] where small adjustments were made to improve performance. YOLOv3 gained some size in network structure, thereby performing at a significantly higher accuracy, but runs slightly slower. A forth generation (YOLOv4) was recently published [48], but it does not include a *tiny* version.

YOLO-tiny is a light-weight version of the YOLO framework. Even though all YOLO neural nets are targeting real time processing, they do demand strong computational power. The best inference performances are achieved by devices with an integrated GPU. A GPU significantly accelerates the processing time, but YOLO-tiny neural nets are specialized for CPU processing. It does this by minimizing the number of layers and decreasing filter size, and still provides decent precision. YOLO has developed a tiny-neural net for every version except the fourth, which is probably still under development.

3.5.9 Training of Convolutional Neural Networks

All CNNs needs training data. The data provides examples of inputs and their corresponding outputs. This gives the fundamental base of what the neural network are able to learn. The data is inserted into a specific network architecture with tunable parameters or weights and biases that determines the output. The network calculates the loss from the correct output and tries to minimize it by changing the weights during training. This is known as back-propagation. Neural networks with the exact same architecture can be trained to map a set of inputs to any kind of outputs. A fully trained neural net has specific weights and biases, that can classify or determine confident predictions of new inputs.

When your model is fully trained, but you want to include more training data in your neural network, transfer learning is an opportunity. Instead of starting an entire new training session, you use the architecture and calculated weights from the pre-trained model as a fundamental before continuing the training session by including more data. This process is called transfer learning.

Intersection of Union (IoU) is a measure of how accurate the predicted bounding box is overlapping the ground truth bounding box. The term is explained and visualized in figure 3.29.

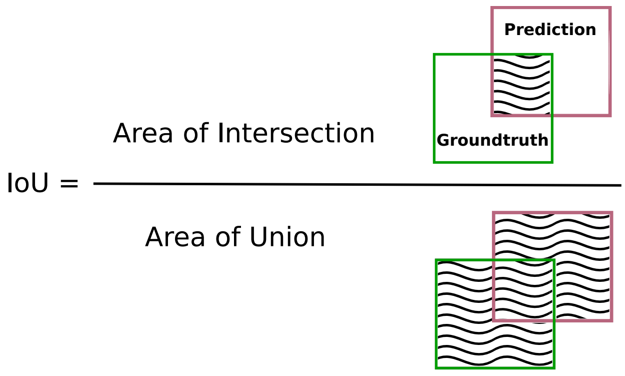


Figure 3.29: IoU

A threshold value for IoU determines whether the prediction is a true positive or negative. A common threshold is 0.5, which provides the following assumptions:

if IoU >0.5	true positive (TP)
if IoU <0.5	false positive (FP)
if IoU >0.5, but predicts wrong class	true negative (TN)

Precision is a measure of the amount of positive predictions that are relevant.

$$Precision = \frac{TP}{TP + FP} \tag{3.109}$$

Recall is a measure of how many relevant objects that are positive.

$$Recall = \frac{TP}{TP + FN} \tag{3.110}$$

Recall and precision are always between zero and one. Figure 3.30 shows an example plot of precision against recall. The plot gives an overview of the overall performance of the entire training set. $p(r)$.

Average Precision, (AP), is per definition the area beneath the precision-recall chart:

$$AP = \int p(r) \tag{3.111}$$

Mean Average Precision (mAP) is the most reliable expression when validating neural networks. It is the mean value of every average precision, and gives an indication of both accuracy and precision.

A *Loss Function* is the error between the network prediction and the known output. The calculated loss is used to determine how to adjust weights and biases.

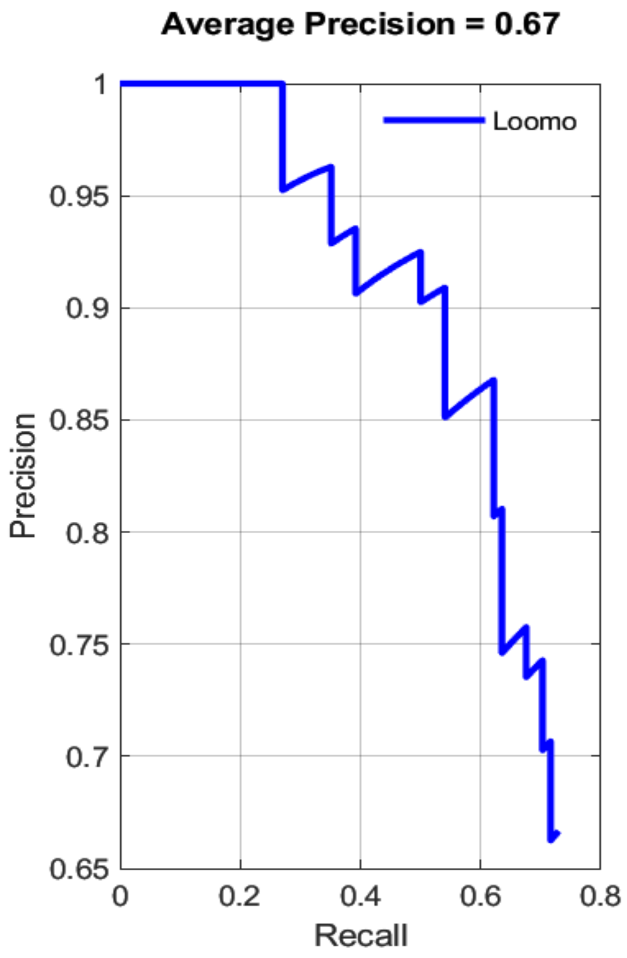


Figure 3.30: Precision against recall example plot

Overfitting can occur when the training process has been too long. The neural network learns details and noise inside the training data, and finds it hard to detect new input-images outside the training set. Figure 3.31 illustrates the effect of overfitting for an arbitrary data set, by making rough changes to fit every node. Information from loss function and mAP can spot indications of overfitting.

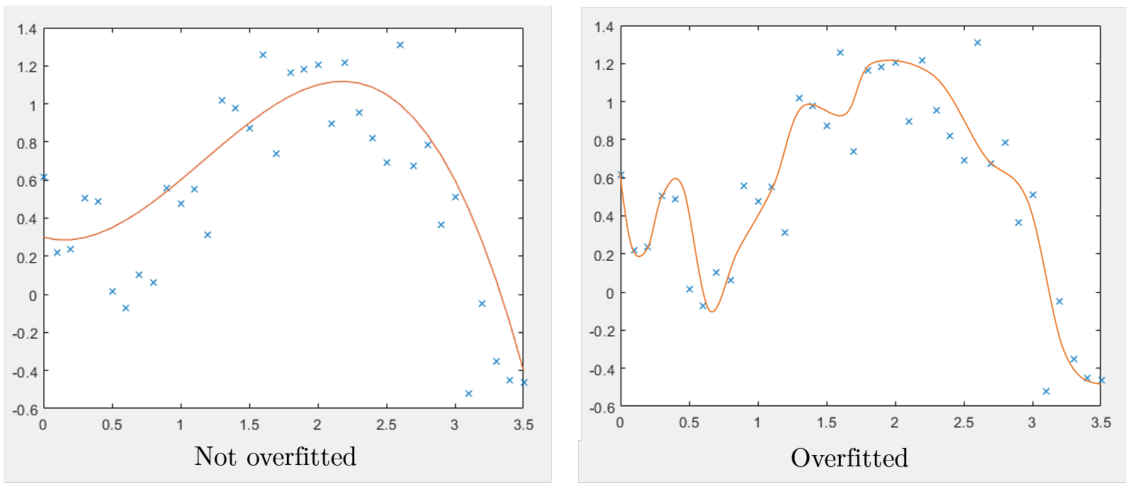


Figure 3.31: Illustration of the effect of overfitting

Chapter 4

Implementation

4.1 SLAM Implementation

The SLAM implementation was done in C++. Source code is available in appendix A. The OpenCV library [49] is used for the computer vision tasks, and Eigen [50] is used for the linear algebra.

The C++ Standard Template Library (STL) has a variety of container templates. If the properties of a container is not particularly relevant, it will simply be referred to as a *list*. Most of the times the container type is a vector (`std::vector<T>`), but this can easily be confused with vectors in math.

4.1.1 Data Recording

The SLAM implementation is not developed on the Loomo itself. Therefor, it is necessary to do recordings of relevant data for use in development. An app was made to record the Loomo's sensors and cameras. The source code for the app is available in appendix A.

Recording the sensors is fairly trivial. The sensors are accessed through the Loomo SDK. This data can be written *as-is* to a CSV file. An external library [51] is used to parse and save the data in CSV format. Each row contains an entry of all the sensors. The first column holds the timestamp for each entry.

Recording the RealSense's camera streams is more difficult. The Android API has utilities for recording video, but the RealSense camera can not be accessed through the Android API (only through the Loomo SDK).

The image frames from the RealSense are received in a byte buffer. A simple way to record video is thus to write the sequence of bytes for each frame to a binary file. Without the use of a video codec, this binary file will be uncompressed. Reading an writing from storage is a slow process, so this method is not fast enough to write the frames at the camera's frame rate of 30 FPS. However, the SLAM implementation in this thesis does not do image processing on each camera frame.

The rate at which the Loomo is able to store uncompressed image frames is dependent on the size of the image. Table 4.1 shows the achieved frame rate when recording video with this method. All 3 cameras where recorded simultaneously. It is the fisheye camera which is of interest in the SLAM implementation. 15 to 16 Frames Per Second is more than good enough for our purpose.

Camera type	Image size in bytes	Achieved frame rate
Fisheye	$640 \times 480 \times 1 = 307,200$ bytes	15-16 FPS
Color	$640 \times 480 \times 4 = 1,228,800$ bytes	6-7 FPS
Depth	$320 \times 240 \times 2 = 153,600$ bytes	26-27 FPS

Table 4.1: Achievable frame rate when storing RealSense camera streams as raw bytes

The recorder app is made to store some extra information in the video files. A video file starts with 3 unsigned integers which tell the width and height of the video, and also the number of bytes per pixel. Before each frame is written to the file, the timestamp for that frame is written. The same timestamp is also included in the CSV file. It is from a different clock than the timestamp for the sensor data, so it is more useful to think of it as a frame index. It is included in the CSV file to enable synchronization of sensor data and video stream.

The sequence of bytes which constitutes an image frame are stored in column-major order. The fisheye camera is grey-scale, so it only stores one byte per pixel. The color camera has four channels, and the bytes are ordered red, green, blue and alpha. The two bytes of each depth camera pixel have little endian order, meaning that the least significant byte is stored first. Figure 4.1 illustrates the structure of the video files.

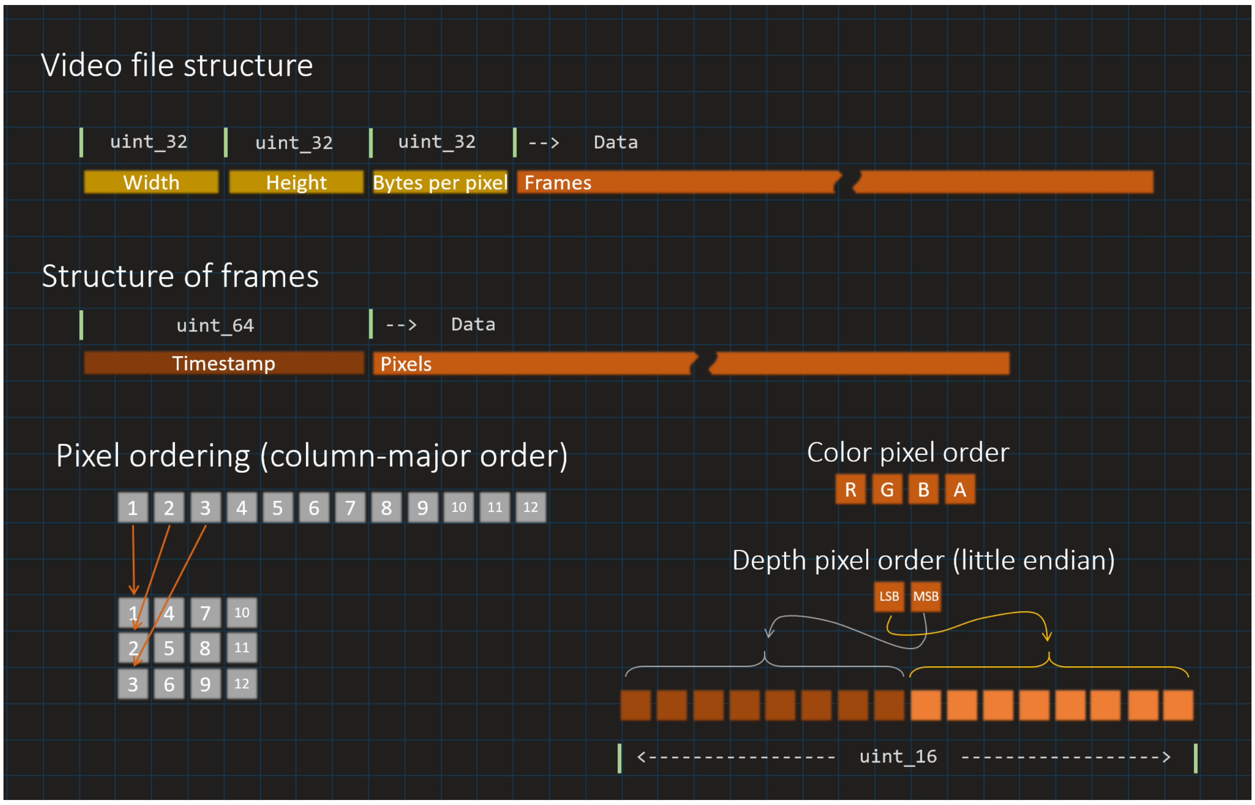


Figure 4.1: Structure of binary files of video recordings

There was a flaw in the data recording. When the Loomo is driven manually, it is steered by tilting the upper body as shown in figure 4.2. None of the recorded sensors registers this motion. The IMU is located in the base. The utility in the Loomo SDK that calculates the transformation between the Loomo's base and the cameras apparently assumes that the upper body is always upright. Considering that the cameras are not supposed to be active when the Loomo is driven manually, this is not an unreasonable assumption. The consequence of the flaw is that the feature matching can not reliably be used to calculate poses between nodes because the camera transformation can not be compensated for.

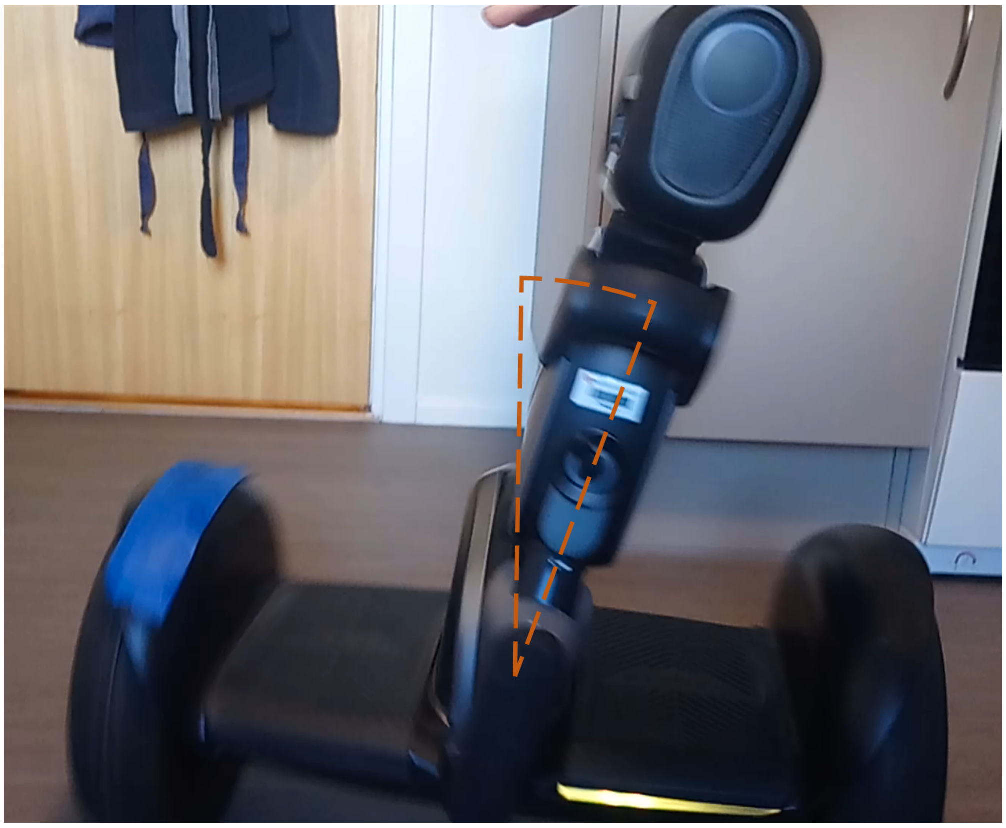


Figure 4.2: When the Loomo is driven manually, turning is done by tilting the upper body of the Loomo

4.1.2 Pose Graph Implementation

The pose graph implementation has three main components

- a data structure representing an edge
- a data structure representing a node
- a class that maintains the graph and performs the graph optimization

An edge contains three things

- a constraint $[x, y, \theta]^\top$
- the covariance matrix for that constraint
- the indices of the two nodes it is connecting

A node contains

- its index
- its current pose $[x, y, \theta]^\top$
- a list of edges that connect to it
- the observation that was done of the environment when the node was created

Notice that all the edges are stored in nodes. A node only contains edges that connect **to** it, not edges that go from it to another node. This is convenient when building the linear system because it is easy to iterate through all the edges of all the nodes.

The observation that is stored in a node consists of the keypoints and descriptors that were calculated from the fisheye camera at that instance.

"PoseGraph“ is the name of the class that maintains and optimizes the graph. It has methods for adding nodes and edges, and for performing the optimization.

The class also has a container with all the nodes in the graph. The container that is used is a map (`std::map<Key, T>`). A map is a container of key-value pairs. The value is the node, and the key is an unique identifier for an element of a map. When a new node is added to the container, the node index is used as the key. Search, insertion and removal operations have logarithmic complexity [52]. Being able to search for nodes by their index is an advantage. It is possible to do this with other container types as well, but not without implementing a search algorithm. Listing 4.1 shows a code snippet for how to iterate through the graph and find all the pairs of nodes that are connected by an edge. The code that is being exemplified is the method that is used to build the linear system when performing graph optimization.

```
for (auto it = nodeMap_.begin(); it != nodeMap_.end(); it++) {
    // 'it' is an iterator. Iterators point to an element in a container
    Node node_j = it->second;
    for (Edge edge : node_j.edges) {
        Node node_i = nodeMap_.find(edge.from)->second; //finding the connected
            node
        // Compute the error and the blocks of the Jacobian
        // Populate b and H
    }
}
```

Listing 4.1: Method for iterating through the graph and finding all the connected node pairs

The node index is incremented for each new node that is added to the graph. When combining multiple graphs, it is necessary that the nodes remain uniquely identifiable. The solution in this implementation is to give the nodes a 64 bit index. Each PoseGraph object is given an identifier which is stored in the top 32 bits of the node index (see figure 4.3). The bottom 32 bits are initialized to zero, so the node index can be incremented approximately $4.29 \cdot 10^9$ times before overflowing into the top 32 bits.

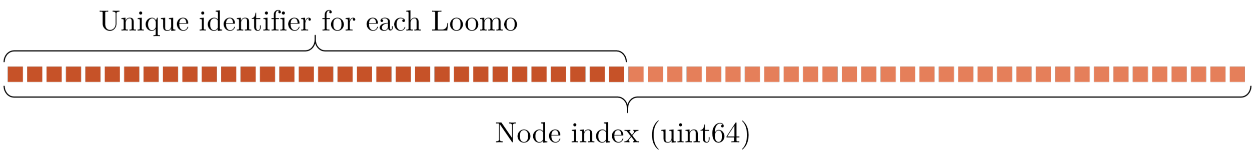


Figure 4.3: An identifier is masked in the top 32 bits of the node index so that graphs can be combined while the nodes remain unique

4.1.3 Processing Encoder Input

The odometry is implemented in the class ”LoomoOdo“. The class processes the encoder inputs to give an estimate of the current pose and its uncertainty based on the equations derived in section 3.3.

The rotary encoders of the left and right wheel are accessed through the Loomo SDK. The SDK returns the net number of encoder ticks, so it is necessary to track the changes in these values.

The encoder ticks are passed to the ”LoomoOdo“-class, which then increments the pose estimate and the covariance estimate. The class has getter-methods that returns these estimates. It also has a getter that returns only the most recent pose increment. This is useful for logging the odometric trajectory of the Loomo, and for measuring the travelled distance of the Loomo.

Figure 4.4 shows an estimated trajectory of a Loomo based only on the wheel encoders. The Loomo was driven a route of approximately 42 meters, and started and stopped in the same spot. A rough sketch of the ground truth overlaid on a map of the building is shown to the left. The bulk of the error in the trajectory comes from turning. This is because the person driving the Loomo has to shift his weight in order to turn. This causes the tires to deform differently during turns. Otherwise, the odometry is subject to little drift.

The odometry starts in pose $\mathbf{x} = [0, 0, 0]^\top$. The state and covariance estimate can be reset back to 0. This is necessary for the graph because odometry based edges hold the relative pose and uncertainty between two nodes.

4.1.4 Adding Nodes to the Graph

Nodes are added to the graph as the Loomo moves around. Each consecutive node is connected by an odometry based edge. Creating a new node always goes in tandem with creating a new edge.

The odometry is reset after new nodes are added to the graph. This means that the current odometry pose estimate is the constraint of the new edge, and the pose's covariance is the new edge's covariance. Let $\hat{z}$ be the odometry pose estimate, and $\hat{\Sigma}$ the covariance. The steps for adding a node to the graph are summarized in algorithm 1.

A node also stores an observation. Doing observations is explained in section 4.1.6.

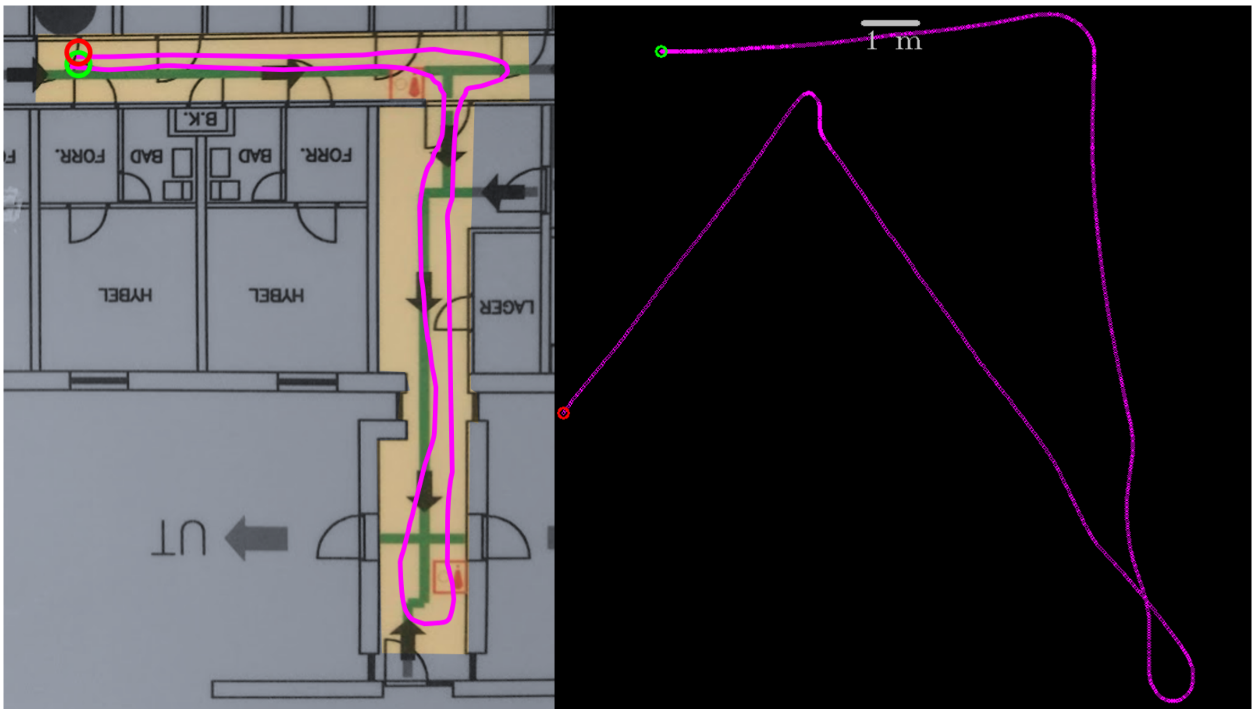


Figure 4.4: Left: a rough sketch of the ground truth of the Loomo's trajectory. Right: the estimated trajectory based on odometry alone

Algorithm 1 Adding node to pose graph

```
newEdge ← $\hat{z}$, $\hat{\Sigma}$
newNode ← newEdge
newNode ← initialPose
newNode ← observation
graph ← newNode
Reset $\hat{z}$, $\hat{\Sigma}$
```

The new node is given an initial pose based on the dead reckoning since last node. The pose $\mathbf{x}_{i+1}$ is found by appending the constraint to the previous pose as shown in equation 4.2. (Note that $\mathbf{R}_i$ is not a 2D homogeneous transformation or a 3D rotation matrix. It is a 2D rotation matrix applied to $[\hat{x}, \hat{y}]^\top$ that conveniently resolves to that form.)

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{R}_i \hat{z} \tag{4.1}$$

$$\mathbf{x}_{i+1} = \begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix} + \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{\theta} \end{bmatrix} \tag{4.2}$$

If no graph optimization is done, the pose graph will resemble the odometric trajectory. Figure 4.6 shows the graph representation of the same dataset as in figure 4.4. The nodes are drawn as triangles so that all three states can be visualized. A new node is added to the graph either every 1 meter travelled, or if the Loomo rotates by more than 45°.

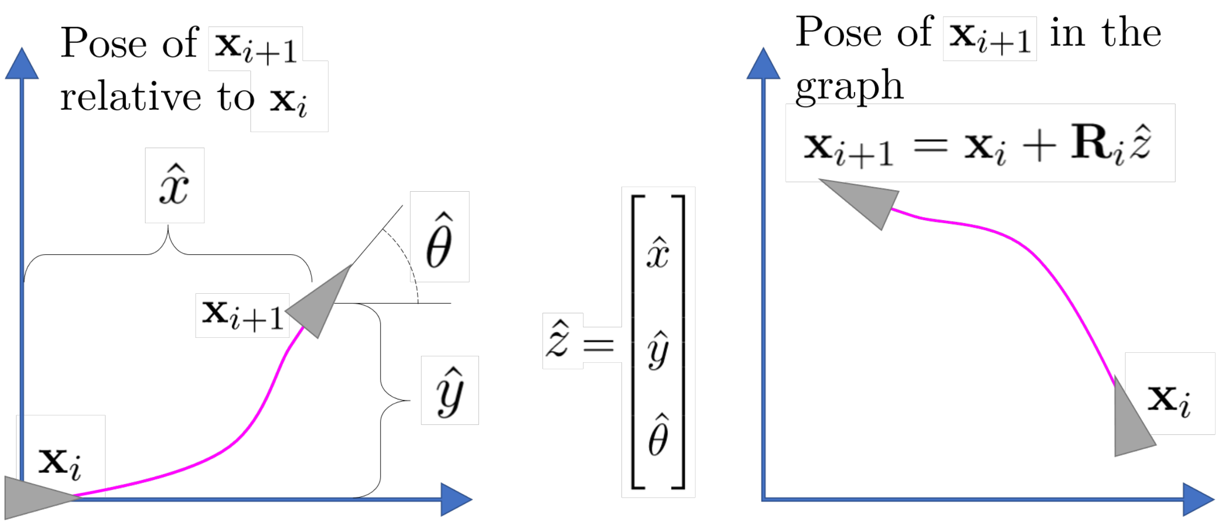

Pose of $\mathbf{x}_{i+1}$ relative to $\mathbf{x}_i$
$\hat{x}$
$\hat{\theta}$
$\mathbf{x}_{i+1}$
$\hat{y}$
$\mathbf{x}_i$
$\hat{z} = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{\theta} \end{bmatrix}$
Pose of $\mathbf{x}_{i+1}$ in the graph
$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{R}_i \hat{z}$
$\mathbf{x}_i$

Figure 4.5: Initializing the pose of new nodes

4.1.5 Camera Calibration

The camera calibration toolbox in Matlab was used to calibrate the RealSense fisheye camera. It has a Graphical User Interface (GUI) that makes it easy to assess the quality of the images that are used for calibration.

The fisheye camera is calibrated by first taking a series of images of a checker-board. The images should be so that the checker-board has positions in most of the FoV of the camera, and has varying angles. Figure 4.7 shows some examples. Many more pictures than were needed were taken. 32 of these were chosen for calibration. Figure 4.8 shows the distribution of the checker-board in those images relative to the camera. Since it is a wide Field of View camera, 3 radial distortion coefficients, k_1, k_2, k_3, are computed. Equation 4.3 shows the calculated camera matrix and equation 4.4 shows the calculated radial and tangential distortion coefficients. The parameters are presented in the form that OpenCV uses.

$$\begin{aligned} K &= \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 294.7754 & 0 & 329.4582 \\ 0 & 294.6943 & 249.6908 \\ 0 & 0 & 1 \end{bmatrix} \qquad (4.3) \\ D &= [k_1, k_2, d_1, d_2, k_3]^\top \\ &= [-0.2641,\ 0.0731,\ 0.0010,\ -0.0003,\ -0.0094]^\top \qquad (4.4) \end{aligned}$$

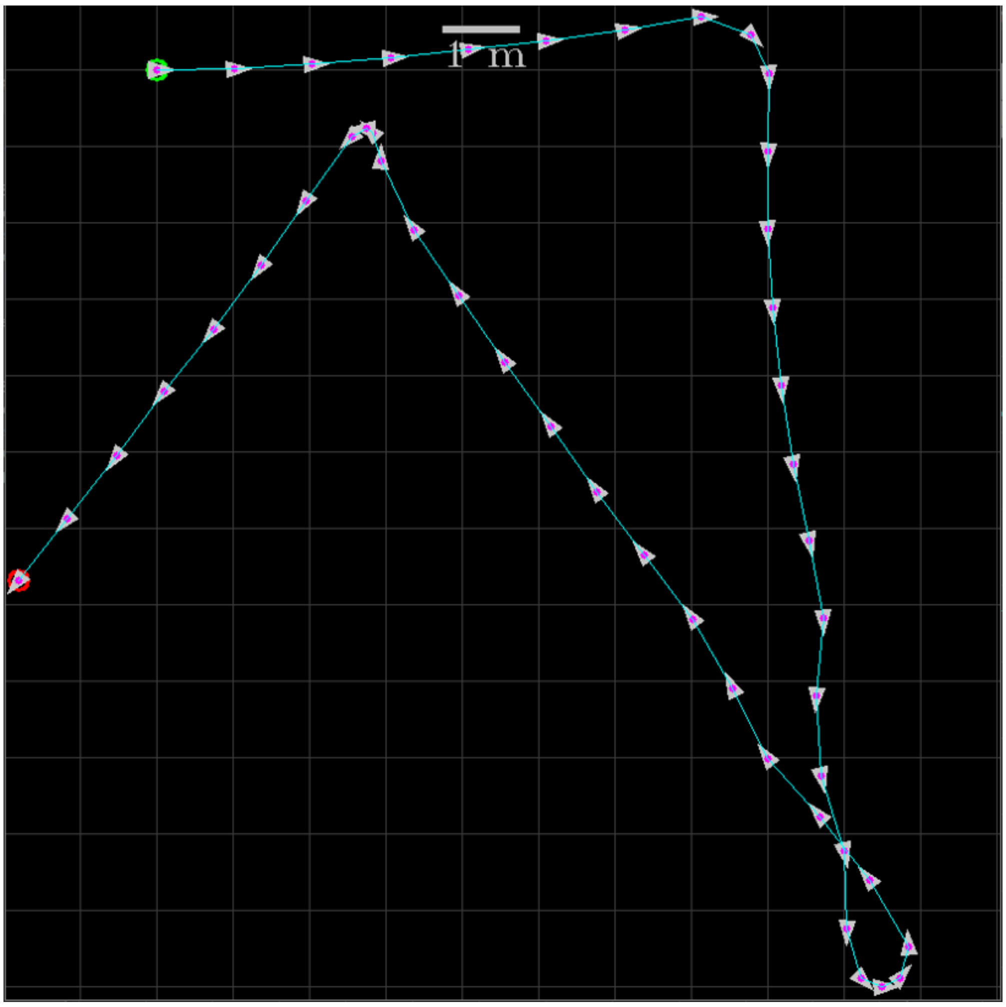


Figure 4.6: Pose graph using only odometry

4.1.6 Computer Vision

[53]

The RealSense's fisheye camera is the exteroceptive sensor used in this SLAM implementation. The computer vision task is twofold:

- Abstract raw images into a compact form, namely identifying keypoints and their descriptors
- Interpreting these observations in a way that is useful for SLAM

The OpenCV library [49] has implementations for most of the required utilities. Here is a brief overview of what this implementation used OpenCV for:

- ORB feature detector and descriptor
- Brute force feature matching
- Undistorting images based on camera parameters
- Estimating essential matrix
- Recover relative camera pose based on essential matrix and point correspondences

The images do not get undistorted. Instead, the keypoints are detected in the raw images, and the

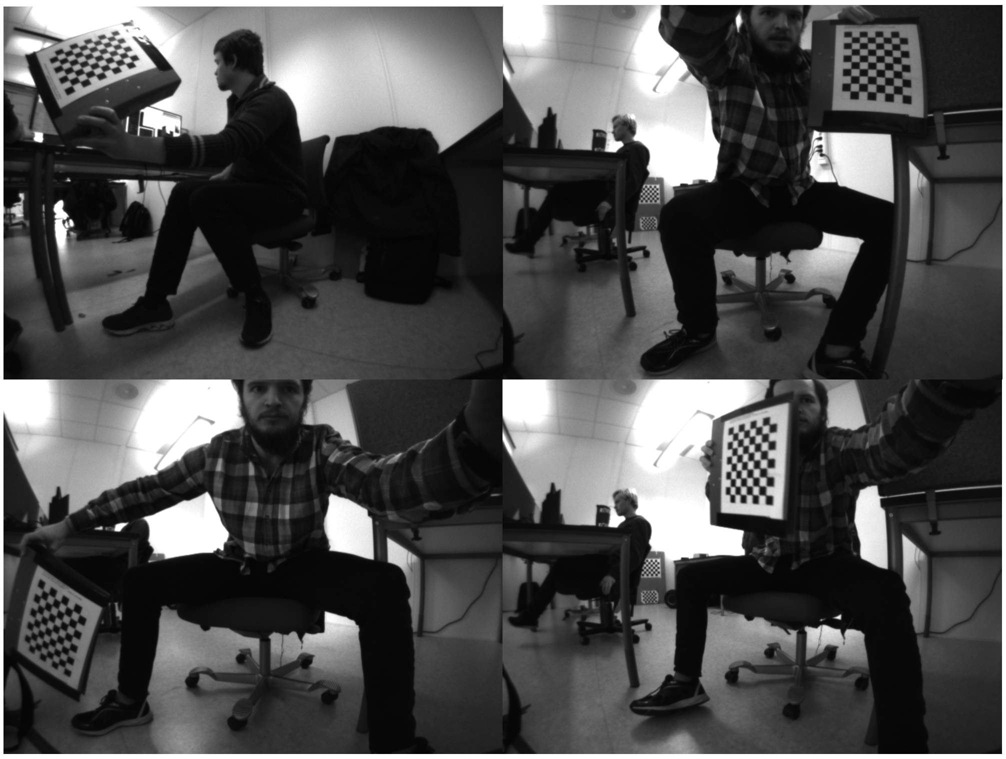


Figure 4.7: Examples of images used in the camera calibration

undistortion is applied to the keypoints only. This reduces computational cost.

ORB is used for keypoint detection and feature description. The default parameters are used with OpenCV's ORB detector except for the maximum number of features to retain. The features are ranked by their Harris score and the poorer ones discarded if above this number. It is set to 3000 instead of the default value of 500. Section 5.4 found that the computation time with respect to the number of features has linear complexity. It was also found that performing the detection has an additional constant time penalty that outweighs the benefit of retaining few features.

The number of features that are detected depends on the environment. It also varies within the same environment as shown in figure 4.9. The detected keypoints and descriptors are stored in separate lists where corresponding keypoint-descriptor pairs share the same index.

When a new node is added to the graph, the current camera frame is processed and the keypoints and descriptors are passed to the node. In order to add virtual edges to the graph, the keypoints and descriptors from different nodes must be matched. The process for selecting candidate nodes for matching is described in section 4.1.7. For now, assume a candidate exist. The data from the current node is referred to as the query descriptors, and those from a previous observation, i.e. the candidate, as the train descriptors.

ORB uses binary descriptors and their similarity is based on their hamming distance (lower distance = closer match). OpenCV has built-in functions for matching features, but further filtering is required to make good data associations. By finding the two best matches for each query descriptor, Lowe's ratio test can be used. The ratio test removes ambiguous matches, but there might still be wrong data associations. Assuming that the point correspondences are predominantly correct,

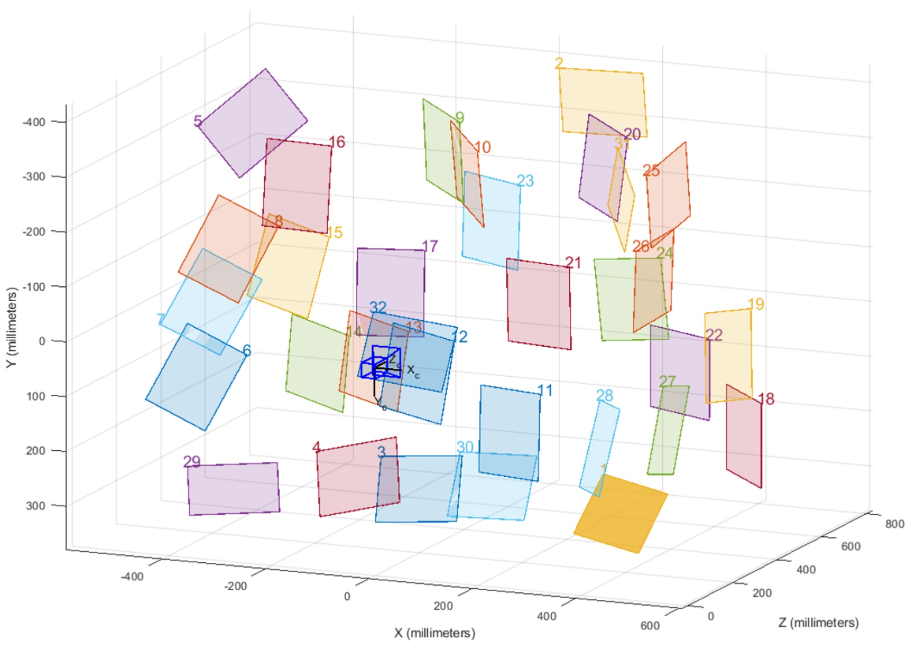


Figure 4.8: Distribution of checkerboards in the images that are used in the calibration

RANSAC can be used to reject the wrong ones. The OpenCV function that estimates the essential matrix applies RANSAC to the given points.

If enough matches remain, estimating the essential matrix can be attempted. The list of matches that OpenCV returns is a list of index pairs of which descriptors correspond to each other. Only the corresponding points are of interest, so these have to be extracted from the lists of detected keypoints. Listing 4.2 shows the code-snippet for doing this. Before these points can be used to estimate the essential matrix, they have to be undistorted. The function that estimates the essential matrix also returns a mask that tells which of the corresponding points were considered outliers.

```
for (size_t i = 0; i < matches.size(); ++i) {
    queryPoints.push_back(currentNodeKeyPoints[ matches[i].queryIdx ].pt);
    trainPoints.push_back(candidateNodeKeyPoints[ matches[i].trainIdx ].pt);
}
```

Listing 4.2: Extracting matched points

OpenCV can then recover the relative camera translation and rotation based on the estimated essential matrix and point correspondences. The outliers are ignored based on the aforementioned mask, so there is no need to manually extract the remaining inliers beforehand. The recovered pose is returned in the form of a 3×3 rotation matrix $\mathbf{R}$ and a 3×1 translation vector $\mathbf{t}$. The translation vector is normalized to a unit vector because epipolar geometry can't recover scale. Section 4.1.7 describes how this scale information is extrapolated from the graph.

The method for matching images that is described here can not reliably tell if the images are of the

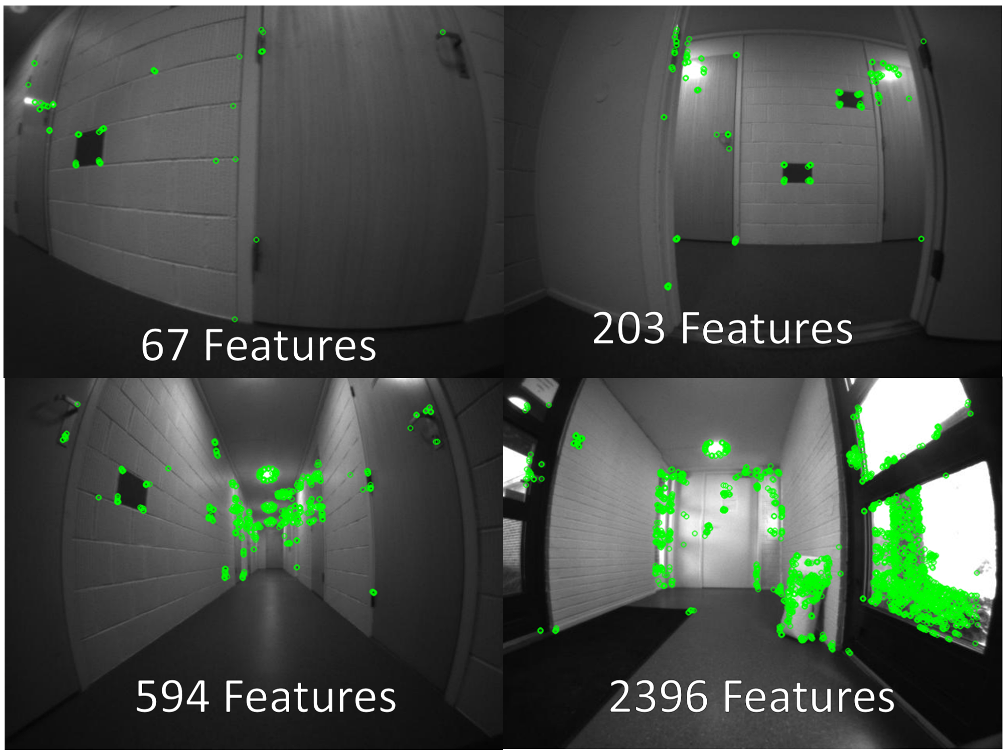


Figure 4.9: Variations in the number of detected features in an environment

same scene. The matching might not succeed, but that alone is not enough information for whether the images corresponded or not. Firstly, most buildings have some frequently appearing features, like doors, light fixtures, etc. Thus, there is a reasonable chance of finding at least some matches regardless of where in the building the query image is taken. Secondly, corresponding images are not expected to overlap by 100%. Two images might have 1/4 overlap, and thus result in relatively few matches (though they might be correct correspondences).

An experiment was conducted to asses whether it is possible to find images of corresponding scenes based on the number of matched features alone. The result of this is presented in section 5.4. A threshold of 50 matched features was able to find most image correspondences with few outliers. The threshold is of course applied after Lowe's ratio test. When searching for loop closures, this threshold is not the only criteria that has to be met.

4.1.7 Adding Virtual Edges to the Graph

Virtual edges are often associated with loop closures, but they do not have to be. The implementation in this report generates virtual edges under two conditions. One is when the Loomo revisits a location, i.e. a loop closure is detected. The other condition is if a new node successfully matches with the previous node. When a new node is added to the graph, a good candidate for matching is the most recent one. It is expected that both nodes are observing the same scene, so the matching criteria can be relaxed. Matching consecutive nodes in this manner can be considered a type of Visual Odometry (VO).

Let's first consider the Visual-Odometry-type of virtual edge. The constraint is expected to be similar to the constraint of the odometry-based edge that is connecting the same two nodes. The odometry-based edge is used to calculate the scale of the translation. The scale is the length of the translation of the constraint. The translation **t** of the VO constraint is thus given by equation 4.6.

$$scale = \sqrt{\hat{x}^2 + \hat{y}^2} \tag{4.5}$$

$$\mathbf{t} \mathrel{*}= scale \tag{4.6}$$

The scale information is only expected to be close, but not exact. Thus the uncertainty of the translation of the Visual Odometry (VO) is higher than that of the wheel-encoder odometry. This implementation used the covariance of the odometry-based edge as a basis for the covariance of the VO-virtual edge. The covariance of the VO translation uses the 2×2 block of the odometry covariance related to translation, and scales it by a factor > 1. The uncertainty of the angle is assumed to be decoupled from the translation. The covariance of the VO-based virual edge, Σ_{VO}, has the form shown in equation 4.7.

$$\Sigma_{VO} = k \cdot \begin{bmatrix} 1.5 \begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & \sigma_y^2 \end{bmatrix} & \mathbf{0} \\ \mathbf{0}^\top & \sigma_\theta^2 \end{bmatrix} \tag{4.7}$$

Since it is expected that two consecutive nodes will match, the only matching criteria is that there are enough point correspondences to estimate the essential matrix and recover the camera pose. There can be as few as 6 matched keypoints. Assuming that the *quality* of the estimated essential matrix is related to the number of point correspondences, k is a scaling factor for the covariance matrix that compensates for this. k is expressed by equation 4.8 and illustrated in figure 4.10. It has a form that rewards higher match counts, but with diminishing returns, and penalizes few matches.

$$k = 4.8 \cdot \exp\left(-0.04 \cdot (n_{match} - 6)\right) + 0.2 \tag{4.8}$$

Now for the loop closure-type of virtual edge. Feature matching alone does not reliably detect loop closures, but it is part of the process of selecting loop closure candidates. First, the search-space should be reduced. This implementation only deals with small graphs, but in large graphs it is necessary. Reducing the search-space also reduced the chance for wrong data associations since potential matches that are far away are not evaluated. Only nodes which are closer than a threshold are considered. The threshold is often three standard deviations (3σ) from the current state estimate. However, as shown in section 3.3, the estimated uncertainty is subject to linearization inaccuracies that accumulate. So instead of using the ellipse as the threshold, a circle with radius r such that the ellipse fits inside it is used:

$$r = 3\sqrt{\max(\sigma_x^2, \sigma_y^2)} \tag{4.9}$$

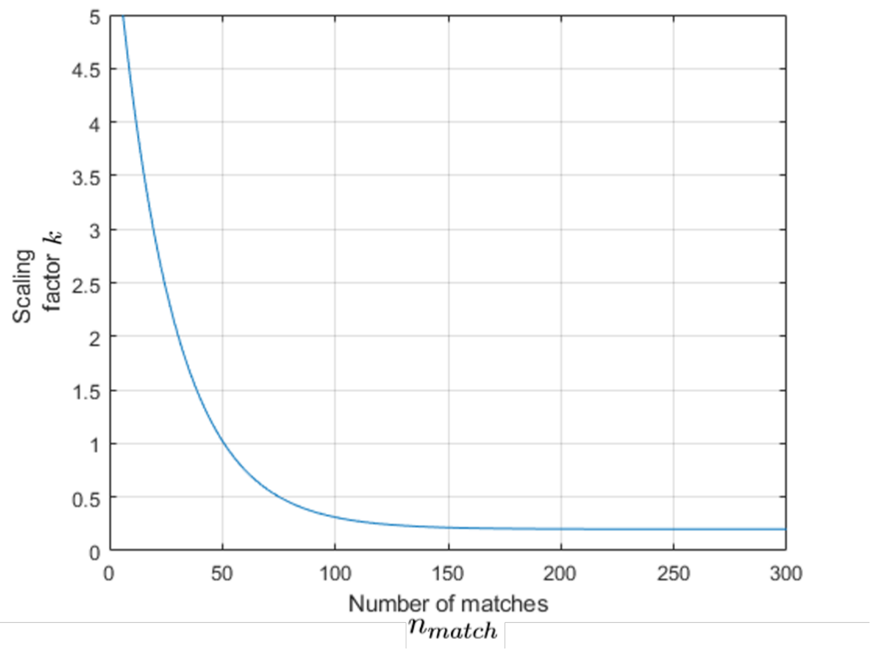


Figure 4.10: Scaling factor for covariance of feature matching based on the number of matches

As described in section 4.1.4, the state estimation is reset for each new node that is added. Thus a second state estimation is done in parallel, which is reset each time the graph is optimized.

If two images have more than 50 matched features it is likely that they are of the same scene. The same scene is often visible from consecutive nodes, which is the premise for the VO-type of virtual edge. Thus, when matching with nodes in the search space, it is expected to get more than 50 matched features with more than one node.

A loop closure is detected if

- the node gets more than 50 matched features with a cluster of at least two connected nodes
- there is only one cluster of connected nodes

A cluster of connected nodes means that all the nodes in the cluster can be reached through the edges provided by the cluster as illustrated in figure 4.11.

Consider a case where a new node $\mathbf{x}_j$ successfully matches with the cluster consisting of $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$. The translation vectors calculated from the estimated essential matrices are unit vectors. Since $\mathbf{x}_j$ is matched with several nodes, the individual edge constraints can be found by triangulation. The unit vectors tell the heading of $\mathbf{x}_j$'s location.

Figure 4.12 shows the theoretical position of $\mathbf{x}_j$ according to the observations. For these calculations, all the translation vectors $\mathbf{t}$, $\mathbf{t}_{ji}$ and $\mathbf{t}_{ji+1}$ must be in the graph coordinate frame. $\mathbf{t}$ is the difference in the positions of $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$ according the the current graph configuration. $\mathbf{t}_{ji}$ and $\mathbf{t}_{ji+1}$ are unit vectors (i.e. the heading of node $\mathbf{x}_j$).

The angles between vectors are found from their dot-product. Once the relevant angles are calcu-

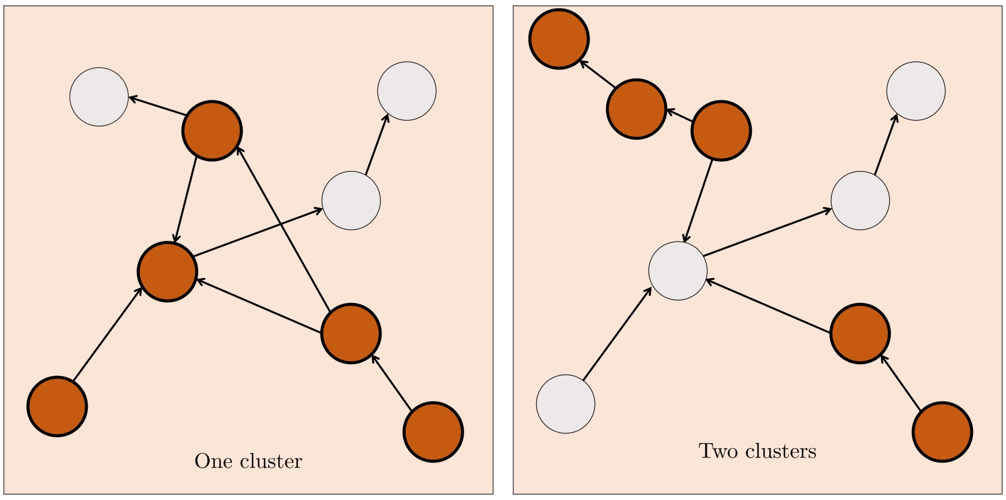


Figure 4.11: Highlighted nodes forms a single cluster if all of them can reach each other

lated (illustrated in figure 4.12), the law of sines is used to solve for the scaling factors k_i and k_{i+1} in equation 4.13. (The scaling factors are the lengths of the green vectors in figure 4.12.)

$$\mathbf{t}_{ji} \cdot \mathbf{t}_{ji+1} = |\mathbf{t}_{ji}| \cdot |\mathbf{t}_{ji+1}| \cdot \cos(\alpha_j) \quad \Rightarrow \quad \alpha_j = \arccos(\mathbf{t}_{ji} \cdot \mathbf{t}_{ji+1}) \tag{4.10}$$

$$\mathbf{t}_{ji} \cdot \mathbf{t} = |\mathbf{t}_{ji}| \cdot |\mathbf{t}| \cdot \cos(\alpha_i) \quad \Rightarrow \quad \alpha_i = \arccos\left(\frac{\mathbf{t}_{ji} \cdot \mathbf{t}}{|\mathbf{t}|}\right) \tag{4.11}$$

$$-\mathbf{t}_{ji+1} \cdot \mathbf{t} = |\mathbf{t}_{ji+1}| \cdot |\mathbf{t}| \cdot \cos(\alpha_{i+1}) \quad \Rightarrow \quad \alpha_{i+1} = \arccos\left(\frac{-\mathbf{t}_{ji+1} \cdot \mathbf{t}}{|\mathbf{t}|}\right) \tag{4.12}$$

$$\frac{|\mathbf{t}|}{\sin(\alpha_j)} = \frac{k_{i+1}}{\sin(\alpha_i)} \qquad \frac{|\mathbf{t}|}{\sin(\alpha_j)} = \frac{k_i}{\sin(\alpha_{i+1})} \tag{4.13}$$

The uncertainties of the new edges are found in a similar matter as with the VO-type of virtual edge. Note that the scale information is extrapolated from the graph configuration, and not the constraint of the odometry-based edge between $\mathbf{x}_i$.

Non of the gathered datasets contain detectable loop closures, so this method has not been properly tested.

4.1.8 SLAM Back-End

The first step in solving the least squares problem is to linearize the error function around $\mathbf{x}$ and compute it for each edge. Equation 4.14 is a general formulation of the error function in a pose graph. It expresses relative poses using homogeneous transformations.

$$\mathbf{e}_{ij}(\mathbf{x}) = \mathrm{t2v}\left(Z_{ij}^{-1}\left(\mathbf{X}_i^{-1}\mathbf{X}_j\right)\right) \tag{4.14}$$

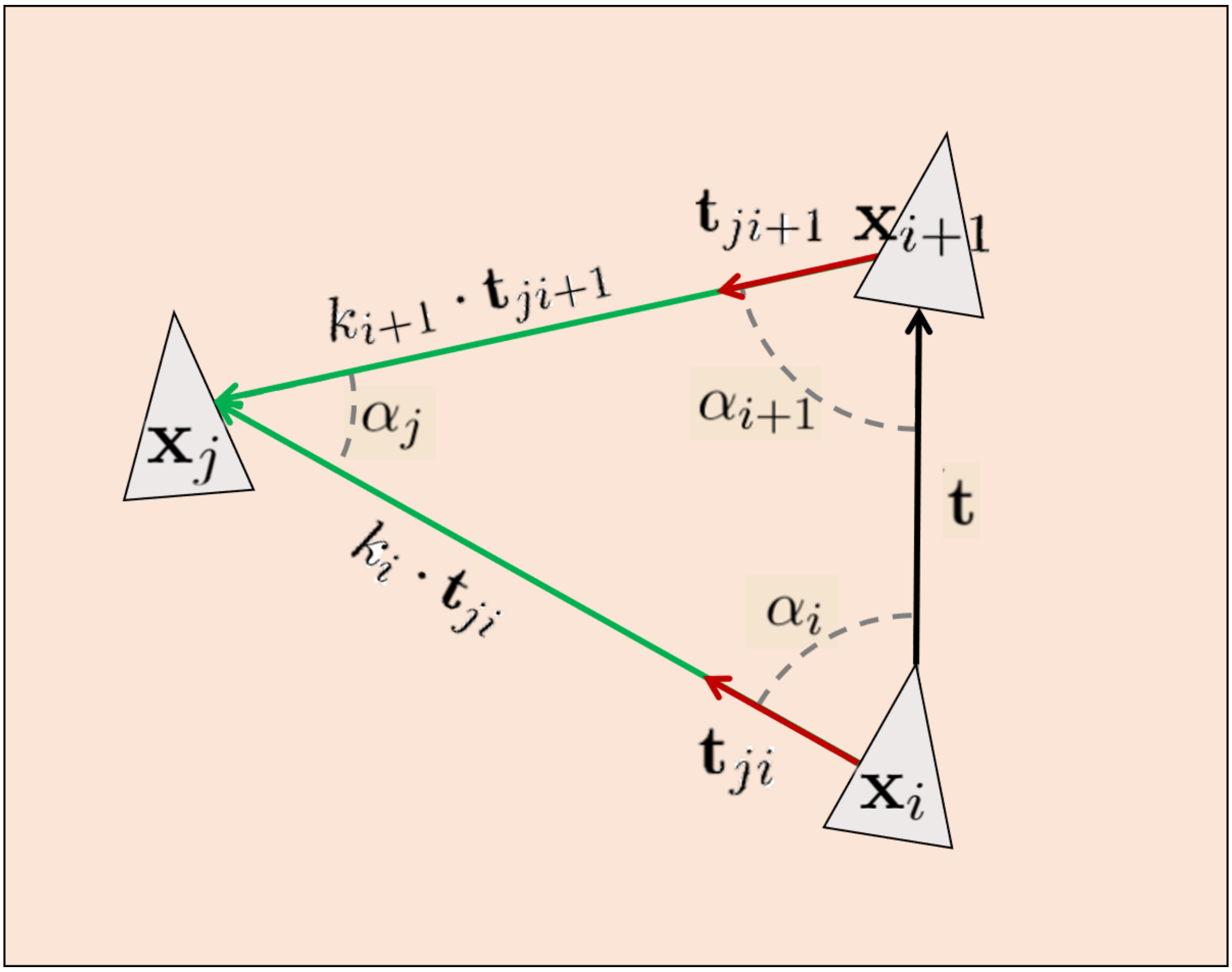


Figure 4.12: Geometric relations for calculating the scale of the translation vectors

For the 2D case, a state vector can be expressed by a translation and an angle

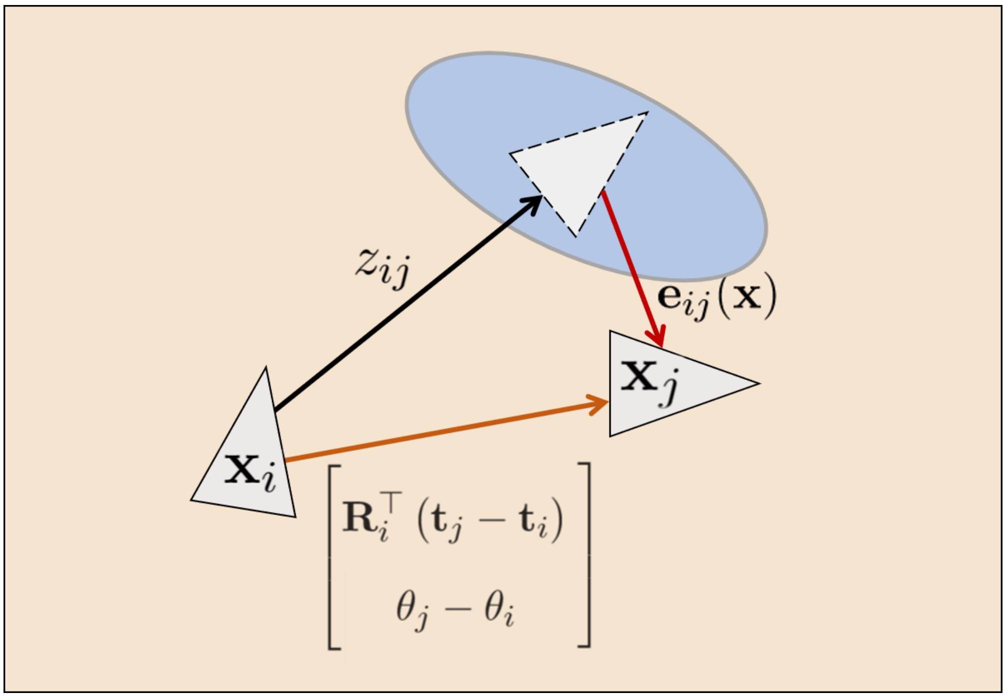
$$\mathbf{x} = \begin{bmatrix} \mathbf{t} \\ \theta \end{bmatrix} \qquad \mathbf{t} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{4.15}$$

Let $\mathbf{R}$ be a 2D rotation matrix. 2D rotation matrices has the convenient property that their inverse is equal their transposed.

$$\mathbf{R} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4.16}$$

$$\mathbf{R}^{-1} = \mathbf{R}^{\top} \tag{4.17}$$

$\mathbf{t}_j$ and $\mathbf{t}_i$ positions of node $\mathbf{x}_j$ and $\mathbf{x}_i$ in the graph. The translation of $\mathbf{x}_j$ relative to $\mathbf{x}_i$ is found by rotating the vector $(\mathbf{t}_j - \mathbf{t}_i)$ to account for the angle of $\mathbf{x}_i$. The angle of $\mathbf{x}_j$ relative to $\mathbf{x}_i$ is simply $\theta_j - \theta_i$. The error is the pose of $\mathbf{x}_j$ relative to the virtual observation of $\mathbf{x}_j$, i.e. relative to z_{ij}. Applying the same method to find the error yields equation 4.18.



z_{ij}
$\mathbf{e}_{ij}(\mathbf{x})$
$\mathbf{x}_j$
$\mathbf{x}_i$
$\begin{bmatrix} \mathbf{R}_i^\top (\mathbf{t}_j - \mathbf{t}_i) \\ \theta_j - \theta_i \end{bmatrix}$

Figure 4.13: Error function in a 2D pose graph

$$\mathbf{e}_{ij}(\mathbf{x}) = \begin{bmatrix} \mathbf{R}_{ij}^\top \left(\mathbf{R}_i^\top (\mathbf{t}_j - \mathbf{t}_i) - \mathbf{t}_{ij} \right) \\ \theta_j - \theta_i - \theta_{ij} \end{bmatrix} \tag{4.18}$$

Further, the blocks of the jacobian, A_{ij} and B_{ij}, can be calculated.

$$A_{ij} = \frac{\partial \mathbf{e}_{ij}(\mathbf{x})}{\partial \mathbf{x}_i} = \begin{bmatrix} -\mathbf{R}_{ij}^\top \mathbf{R}_i^\top & \mathbf{R}_{ij}^\top \frac{\partial \mathbf{R}_i^\top}{\partial \theta_i} (\mathbf{t}_j - \mathbf{t}_i) \\ \mathbf{0}^\top & -1 \end{bmatrix} \tag{4.19}$$

$$B_{ij} = \frac{\partial \mathbf{e}_{ij}(\mathbf{x})}{\partial \mathbf{x}_j} = \begin{bmatrix} -\mathbf{R}_{ij}^\top \mathbf{R}_i^\top & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{4.20}$$

In the C++ implementation, the error function is resolved for all the state variables as done in equation 4.21. This is also done for the functions that calculate the blocks of the Jacobian.

$$\mathbf{e}_{ij}(\mathbf{x}) = \begin{bmatrix} -\cos(\theta_{ij}) \left(\cos(\theta_i)(x_i - x_j) + \sin(\theta_i)(y_i - y_j) \right) - \sin(\theta_{ij}) \left(\cos(\theta_i)(y_i - y_j) - \sin(\theta_i)(x_i - x_j) \right) - x_{ij} \\ \sin(\theta_{ij}) \left(\cos(\theta_i)(x_i - x_j) + \sin(\theta_i)(y_i - y_j) \right) - \cos(\theta_{ij}) \left(\cos(\theta_i)(y_i - y_j) - \sin(\theta_i)(x_i - x_j) \right) - y_{ij} \\ \theta_j - \theta_i - \theta_{ij} \end{bmatrix} \tag{4.21}$$

Now the linearized error function can be computed for each edge in the graph. Building the linear system means populating b and H. If there are n nodes in the graph, then the size of b is $3n \times 1$. b is initialized as a $3n \times 1$ vector of zeros.

H is $3n \times 3n$, but it is sparse and thus not initialized in the same way. The Eigen library [50] has a sparse matrix format. It only holds the non-zero elements in memory, along with information that allows further matrix manipulations (for instance row and column indices of non-zero elements). Eigen has several methods for populating sparse matrices. The recommended method, performance wise, is to first build a list of triplets, and then convert it to a sparse matrix. A triplet is a data type that holds three values. The values in the triplets must be the row index, the column index and the value of the element. If several triplets refer to the same matrix element, their values are summed up when the triplets are converted to a sparse matrix.

Algorithm 2 shows the steps for building the linear system.

Algorithm 2 Build linear system

```
$n$ = number of nodes
$b$ = $3n \times 1$ vector of zeros
H_coeffs = empty list of triplets
for each edge $\in$ graph do
    $\mathbf{e}_{ij}$ = computeError
    $(\mathbf{A}_{ij}, \mathbf{B}_{ij})$ = computeBlocksOfJacobian
    $b(i)$ += $\mathbf{A}_{ij}^\top \Omega_{ij} \mathbf{e}_{ij}$
    $b(j)$ += $\mathbf{B}_{ij}^\top \Omega_{ij} \mathbf{e}_{ij}$
    H_coeffs.push_back$\left(i,\ i,\ \mathbf{A}_{ij}^\top \Omega_{ij} \mathbf{A}_{ij}\right)$
    H_coeffs.push_back$\left(i,\ j,\ \mathbf{A}_{ij}^\top \Omega_{ij} \mathbf{B}_{ij}\right)$
    H_coeffs.push_back$\left(j,\ i,\ \mathbf{B}_{ij}^\top \Omega_{ij} \mathbf{A}_{ij}\right)$
    H_coeffs.push_back$\left(j,\ j,\ \mathbf{B}_{ij}^\top \Omega_{ij} \mathbf{B}_{ij}\right)$
end for
H_coeffs.push_back(0, 0, $\mathbf{I}$) // $\mathbf{I}$ = Identity matrix. This fixes the first node
$H$ = buildSparseFromTriplets(H_coeffs)
return $(b, H)$
```

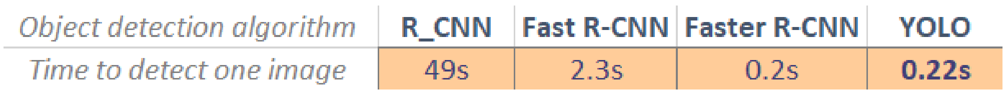
The Eigen library has a sparse solver that uses Clolesky decomposition. Thus the linear system can be solved for the increment $\Delta\mathbf{x}$, and the nodes adjusted accordingly.

4.2 Machine learning

The purpose of the machine learning is to enable the Loomos to detect each other through their on-board fisheye cameras. This section describes the process of collecting training data and training the CNN to achieve this.

The ability to achieve quick inference time is extra important when dealing with real-time collaborate robots. That was taken into consideration when selecting CNN-algorithm. Table 4.2[54] verifies the choice of selecting YOLO.

In this comparison, YOLO has an outstanding inference time of $22ms$, corresponding to a frame rate of 45 FPS. However, such results are computed with powerful GPUs which are not included in the Loomo's hardware.



Object detection algorithm	R_CNN	Fast R-CNN	Faster R-CNN	YOLO
Time to detect one image	49s	2.3s	0.2s	**0.22s**

Table 4.2: Comparing real-time performance of object detection algorithms

4.2.1 Generating Training Data

A robust CNN have been using lots of images while training, leading us to the crucial task of creating a good custom dataset. The Loomo robots contains among other sensors, a fisheye camera. The prominent advantage of a fisheye camera compared to a regular camera is its wide FoV (Spesifications in Ch. 2.2). A fisheye camera utilizes more nearby information, making it easier to spot nearby Loomos. The normal camera would perform better for ”long distance“ detection, but considering a collaborate swarm behaviour, it would be more valuable with wide view rather than long view. Finally the neural network will be implemented on the Loomo itself, and only the Loomo. All detections will appear from a ”Loomo-perspective“, making us shrink the dataset into images that is easy for the Loomo to read. A fisheye camera-app was made to satisfy that need(See code in appendix A), such that all images in the dataset will be gathered from that ”Loomo-perspective“.

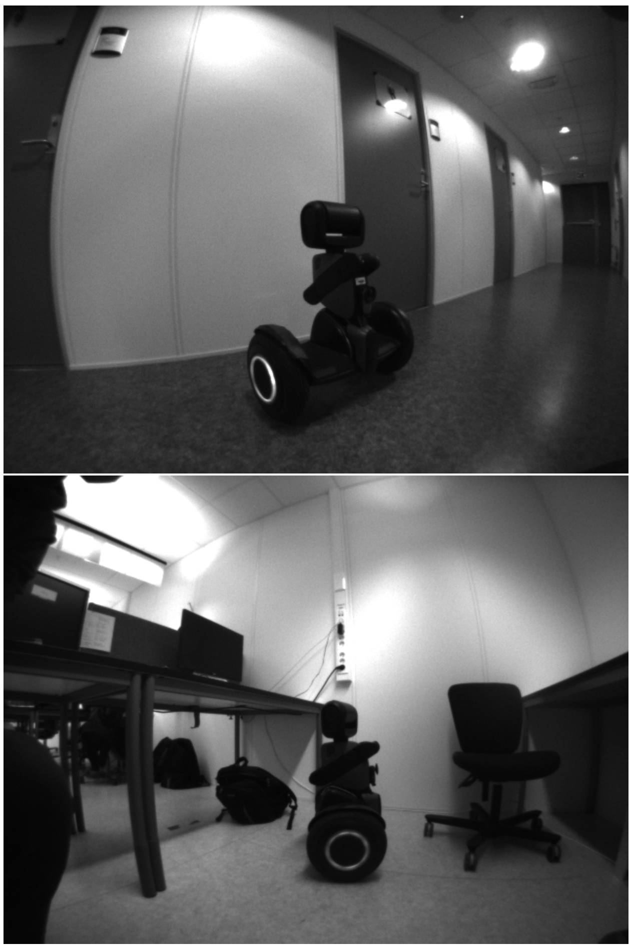


Figure 4.14: Fisheye Images

The labeling strategy was to be precise, always. No image should miss a bounding box, and the deviations between bounding box edges and the Loomo should be pixel-small. An app inside Matlab called ”Image Labeler“ was used to label all the images, and figure 4.15 shows an example. Seven datasets was created, in total 1225 images. Table 4.3 gives a brief overview.

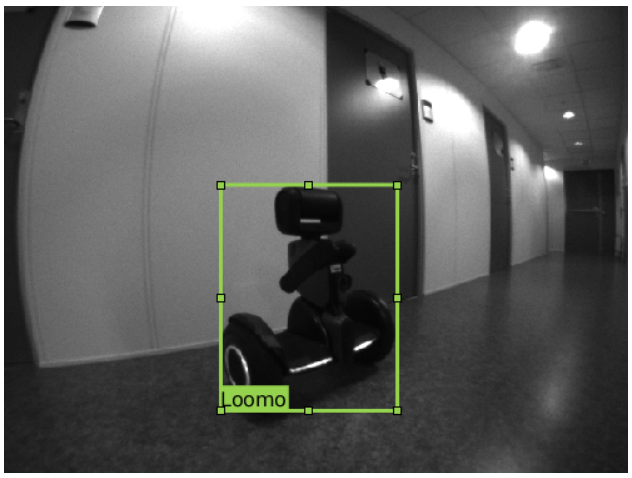


Figure 4.15: Labeling images from a ”Loomo perspective“

	Dataset1	Dataset2	Dataset3	Dataset4
Description	*Varied images*	*Medium range, two Loomos*	*Partly covered*	*Lots of variations*
Number of images	115	179	38	174
	Dataset5	Dataset6	Dataset7	
Description	*Long distance*	*Close up*	*Selective covering*	
Number of images	124	325	270	

Table 4.3: Datasets

The entire labling process was done with the Matlab labeler app, but since Matlab-support for YOLOv3 is still in progress, further training was done with darknet. Matlab and darknet parameterize bounding boxes in different ways, hence every label needed to be converted to the darknet-orientation 4.17. (The script is available in appendix A)

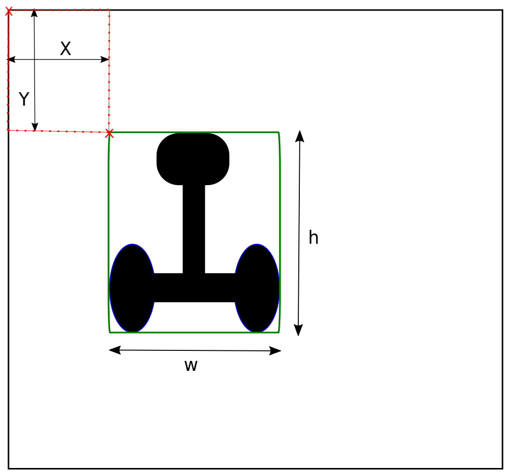


Figure 4.16: Matlab orientation

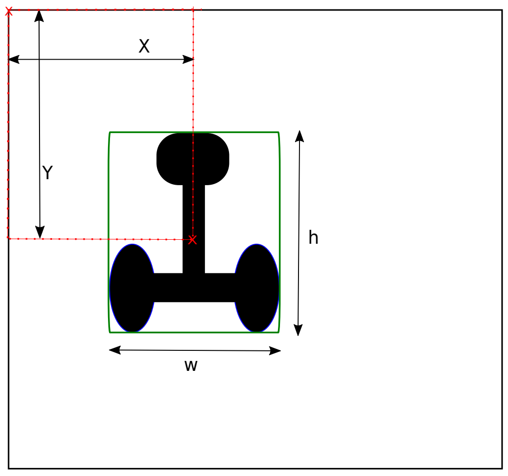


Figure 4.17: Darknet orientation

Usually, an amount of minimum 1000 images is enough to train a neural network from scratch [55]. A dataset of 1225 images is in theory sufficient, but it is a good idea to expand it. Presence of more data results in a more robust, accurate model and helps in handling edge-cases. One effective method to expand the datasets is by augmentations. Augmentation is the process of forcing small changes to an image, for instance blur, scaling, mirroring, adding noise, etc. Figure 4.18 shows an example of augmented images from the original images in the top left corner.

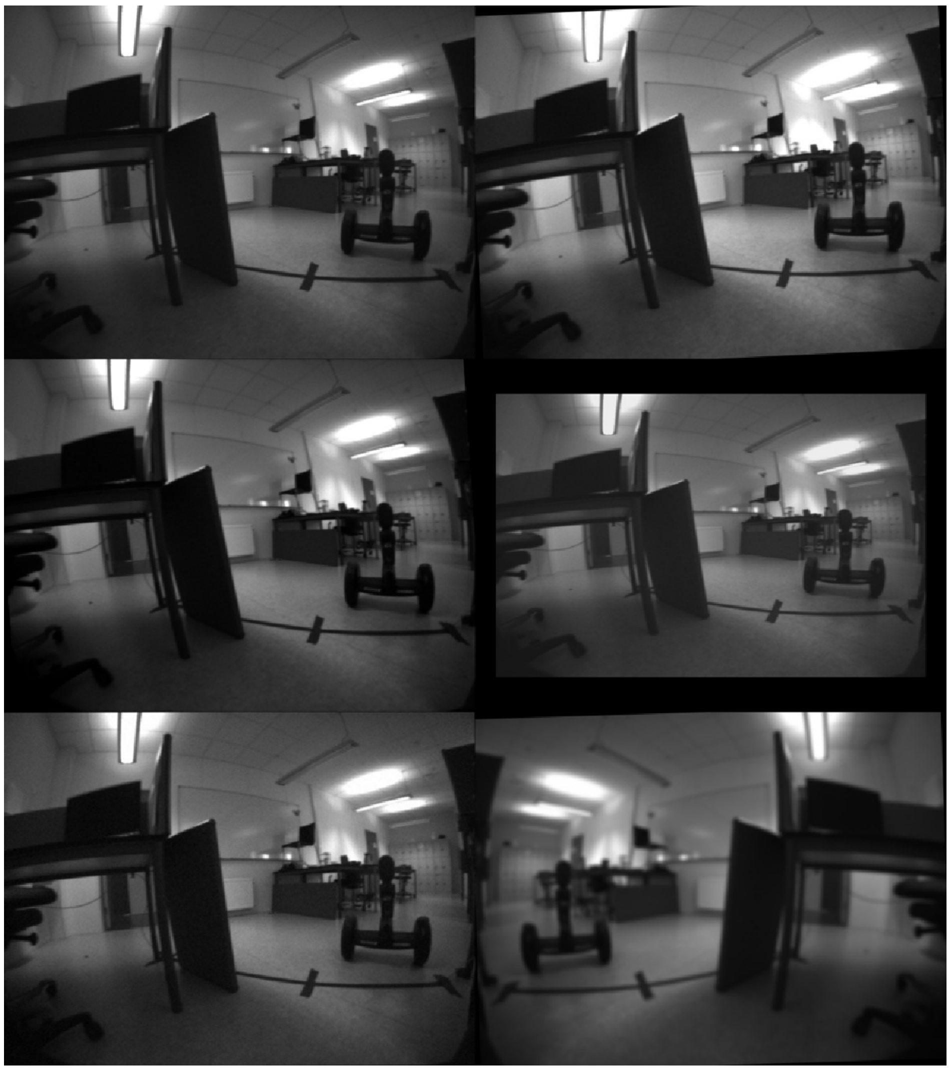


Figure 4.18: Various augmentations that are applied to the top left image

A python script was made to augment the data and also to maintain control of the quantity and amount of the augmentation.(Script available at appendix A). For each original image, 19 new augmented images were created. The original dataset increased in size by a factor of 20. Every image generated is determined by the probability for a specific augmentation to happen. The probability table 4.5 shows the specific augmentation for each dataset.

	GaussianBlur or Defocusblur or MotionBlur	Averagepooling	Fliplr	LinearContrast
AugmentedSet1	30%	5%	5%	33%
AugmentedSet2	-	5%	5%	33%
AugmentedSet3	-	5%	5%	33%
AugmentedSet4	-	5%	5%	33%
AugmentedSet5	-	-	5%	33%
AugmentedSet6	25%	5%	5%	33%
AugmentedSet7	30%	5%	5%	33%

	Multiply	AdditiveGaussianNoise	PiecewiseAffine	ShearX or ShearY
AugmentedSet1	33%	15%	15%	40%
AugmentedSet2	33%	15%	15%	40%
AugmentedSet3	33%	15%	15%	40%
AugmentedSet4	33%	15%	15%	40%
AugmentedSet5	33%	15%	15%	40%
AugmentedSet6	33%	15%	15%	40%
AugmentedSet7	33%	15%	15%	40%

	Scale or Translate or Rotate	ZoomBlur	CoarseSaltAndPepper or Cutout
AugmentedSet1	40%	-	-
AugmentedSet2	40%	-	-
AugmentedSet3	40%	-	-
AugmentedSet4	40%	-	-
AugmentedSet5	40%	-	-
AugmentedSet6	40%	10%	5%
AugmentedSet7	40%	-	-

Table 4.4: Augment the datasets

A rule of thumb when applying augmentation, is to not augment an image so much that the object of interest is unrecognizable for humans. The neural net needs inputs of clear, specific objects to not apply any confusion. Heavier augmentation appear when the object is nearby and easy to recognize. Table 4.5 shows the chosen parameters inside the augmentation. The augmentation parameters that are used were chosen from an augmentation library [56].

Type of Augmentation	GaussianBlur	Defocusblur	MotionBlur
Amount of Augmentation	Sigma = 0.3 – 0.5	Severity = 1 – 2	K = 5 – 10, angle = 60 – 120
Type of Augmentation	LinearContrast	Multiply	AdditiveGaussianNoise
Amount of Augmentation	80 – 110 %	90 – 150 %	Loc = 0, scale = 0 – 7.65
Type of Augmentation	Scale	Translate	Rotate
Amount of Augmentation	x = 80 – 100 %, y = 80 – 100 %	{x , y} = (-5) – 5 %	-4, 4
Type of Augmentation	Averagepooling	Fliplr	PiecewiseAffine
Amount of Augmentation	Severity = 1	(Left to right)	Scale = 0.01 – 0.05
Type of Augmentation	ShearX or ShearY	ZoomBlur	CoarseSaltAndPepper or Cutout
Amount of Augmentation	(-5) – 5 %	Severity = 1	Size = 0.01 – 0.05 %

Table 4.5: Augmentation control

Early attempts at training found that some parts of the dataset noticeably reduced the mAP. To identify and filter out erroneous training data, a Matlab script was made to verify bounding boxes after being augmented.(Script available at appendix A). The script copies the augmented images with the corresponding bounding box, and writes new images with overlapping bounding boxes. The script made it easy to detect if any error occurred.

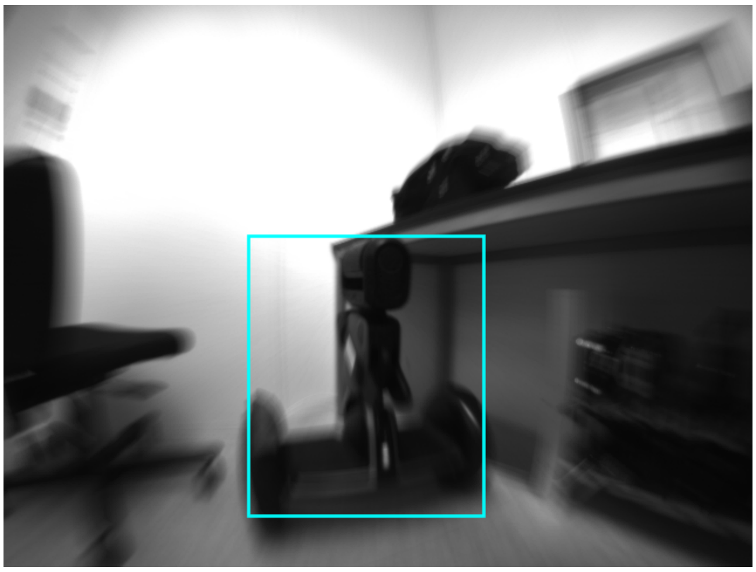


Figure 4.19: Verifying Bounding Boxes

4.2.2 Designing the Neural Network

Getting acquainted with YOLO-architecture, we found that ***YOLO-tiny*** algorithms are specialized for real-time performance for devices without GPUs. Low inference time is crucial when designing a neural network to perform in real-time, and since the Loomo does not have a GPU, the tiny-architecture was reasonable to choose. All versions of the YOLO algorithms are open source, and can be customized to fit for our application. These algorithms have been optimized for accuracy and speed, so any change should preserve this. What we did change was the number of classes into one ”Loomo“ class, and also filters in the pre-outputlayers. The pre-outputlayer is a convolutional layer that takes a filter parameter f which is dependent of the number of classes and anchors. The filter parameter in all pre-outputlayers was changed to 18, according to equation 4.22. N is the number of anchors and C is the number of classes the neural network is trying to detect.

$$f = N \cdot (C + 5) \tag{4.22}$$

The final neural network design of YOLOv3-tiny is visualized in figure 4.20.

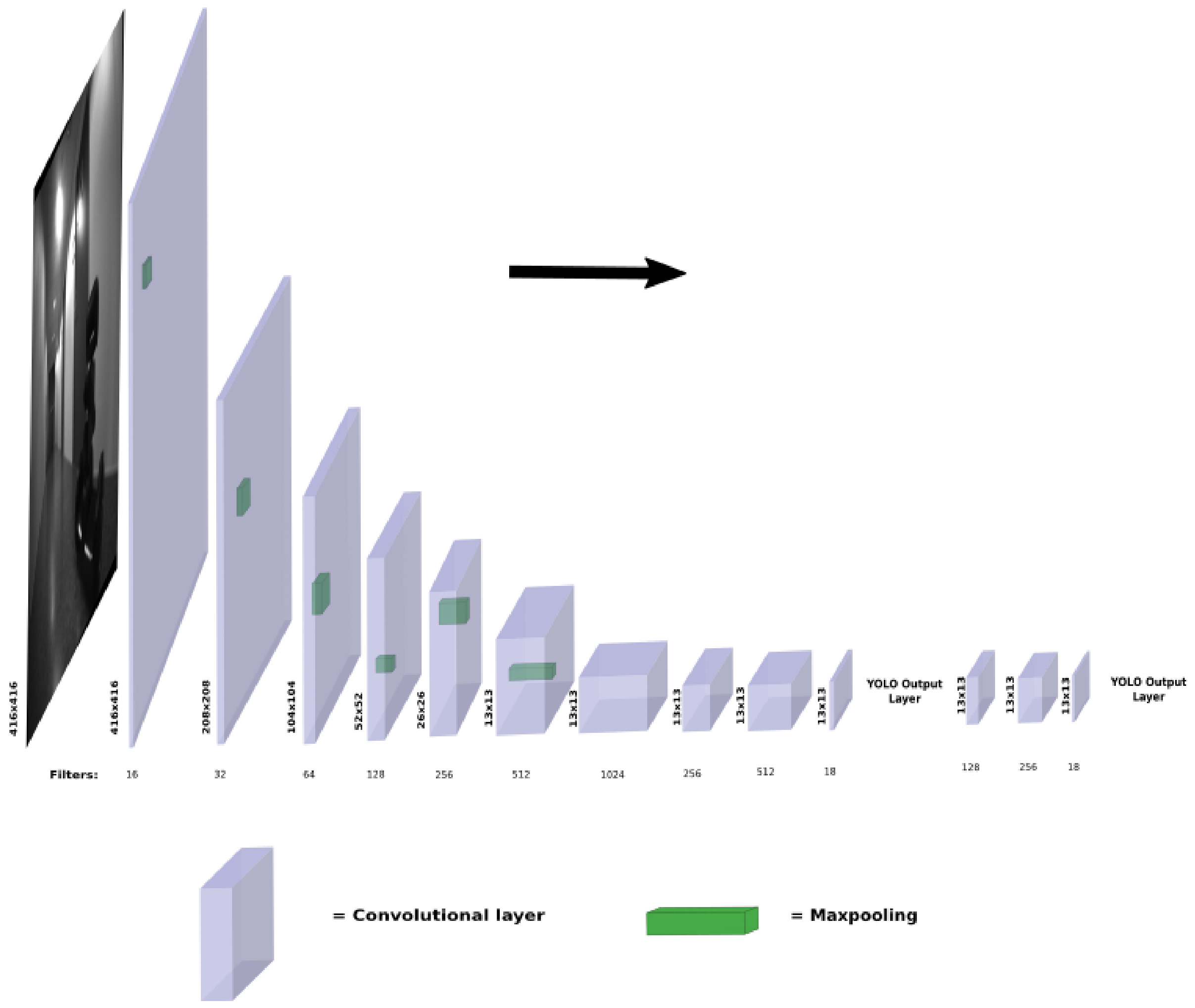
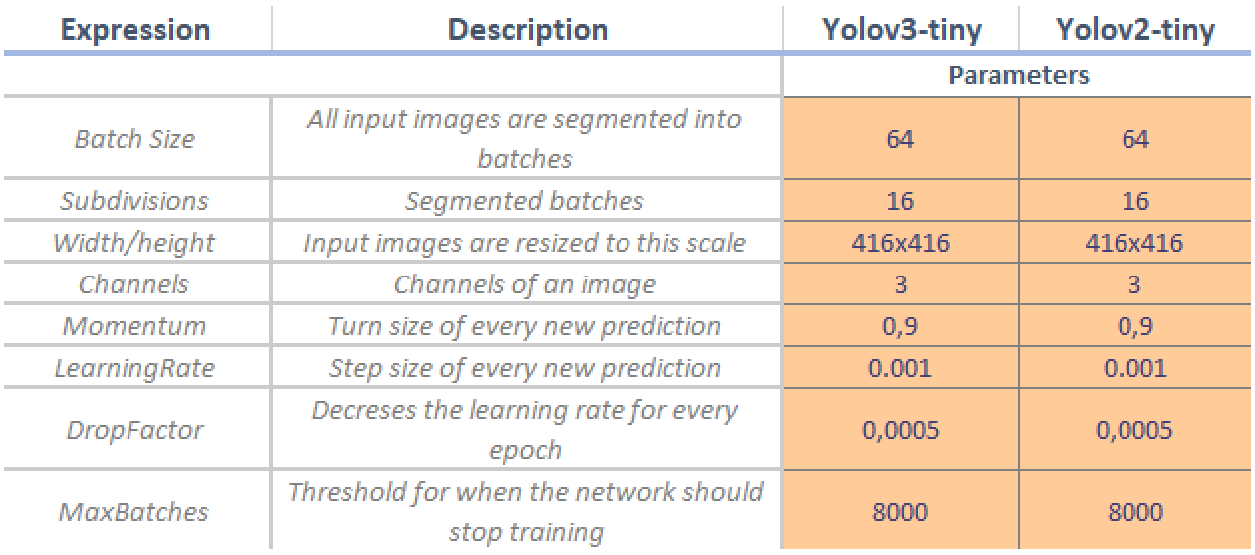


Figure 4.20: YOLOv3-tiny-custom CNN architecture

4.2.3 Training

Several training option parameters affect the training performance, which also affects the final result. A training session in Matlab requires a dataset, training algorithm and training settings. If these conditions are met and one start the training, Matlab performs the training by itself. Time was spent by changing parameters and analyze results. It turned out that crucial parameters are the initial learning rate, dropfactor and batch size. Every parameter change gives different outputs and it is really a neverending story to learn by ”trial and error“ when operating with neural networks. One have to accept that neural networks are complicated, not least hard to track potential improvements.

The group decided to change from the Matlab environment, mainly to apply YOLOv3 and to test tiny-versions. Equal training-settings were transferred from Matlab into darknet, to hopefully achieve good results. Table 4.6 declares the final training settings used for YOLOv2-tiny and YOLOv3-tiny.



Expression	Description	Yolov3-tiny	Yolov2-tiny
		Parameters	
Batch Size	*All input images are segmented into batches*	64	64
Subdivisions	*Segmented batches*	16	16
Width/height	*Input images are resized to this scale*	416x416	416x416
Channels	*Channels of an image*	3	3
Momentum	*Turn size of every new prediction*	0,9	0,9
LearningRate	*Step size of every new prediction*	0.001	0.001
DropFactor	*Decreses the learning rate for every epoch*	0,0005	0,0005
MaxBatches	*Threshold for when the network should stop training*	8000	8000

Table 4.6: Training setting

Training in the darknet environment has three requirements:

- The ***cfg file*** (Available in appendix A) determines training settings and the entire network architecture.
- ***Localize training*** and validation sets. The script "generate_train.py" (available at appendix A) generates a file that writes the path to every labeled image you want to include in the datasets.
- ***Pretrained weights file*** to initialize the network architecture. It works as the neural network template and is obtained from darknet-YOLO official site. The weights-file choice may affect the training, and for YOLOv3-tiny we used the weights file named ”yolov3-tiny.conv.11“ (Available in appendix A).

The training process produces new weights files, specifically one per 1000 iteration. Each weights file represent a trained neural network that can either be used for transfer training or declared as fully trained and applied for its purpose.

4.2.4 Validation

When selecting validation data, we ensured that the content is good, because all further estimations of the network performance use the validation images as basis. It is important that the trained neural network has not seen any images inside the validation set. Producing a proper validation set strengthens the reliability of the validation, hence images with great variations was selected. At last, the validation set was augmented to expand its capacity.

Darknet generates a loss graph during training, seen in figure 4.21. The graph represents the loss function and mAP for our best YOLOv3-tiny model. The loss function should decrease asymptotically to verify a decent training session while high valued mAP verifies decent precision. As the mAP stops to increase, a safe move is to stop training (for instance at 5000 iterations) to prevent any tendency of overfitting.

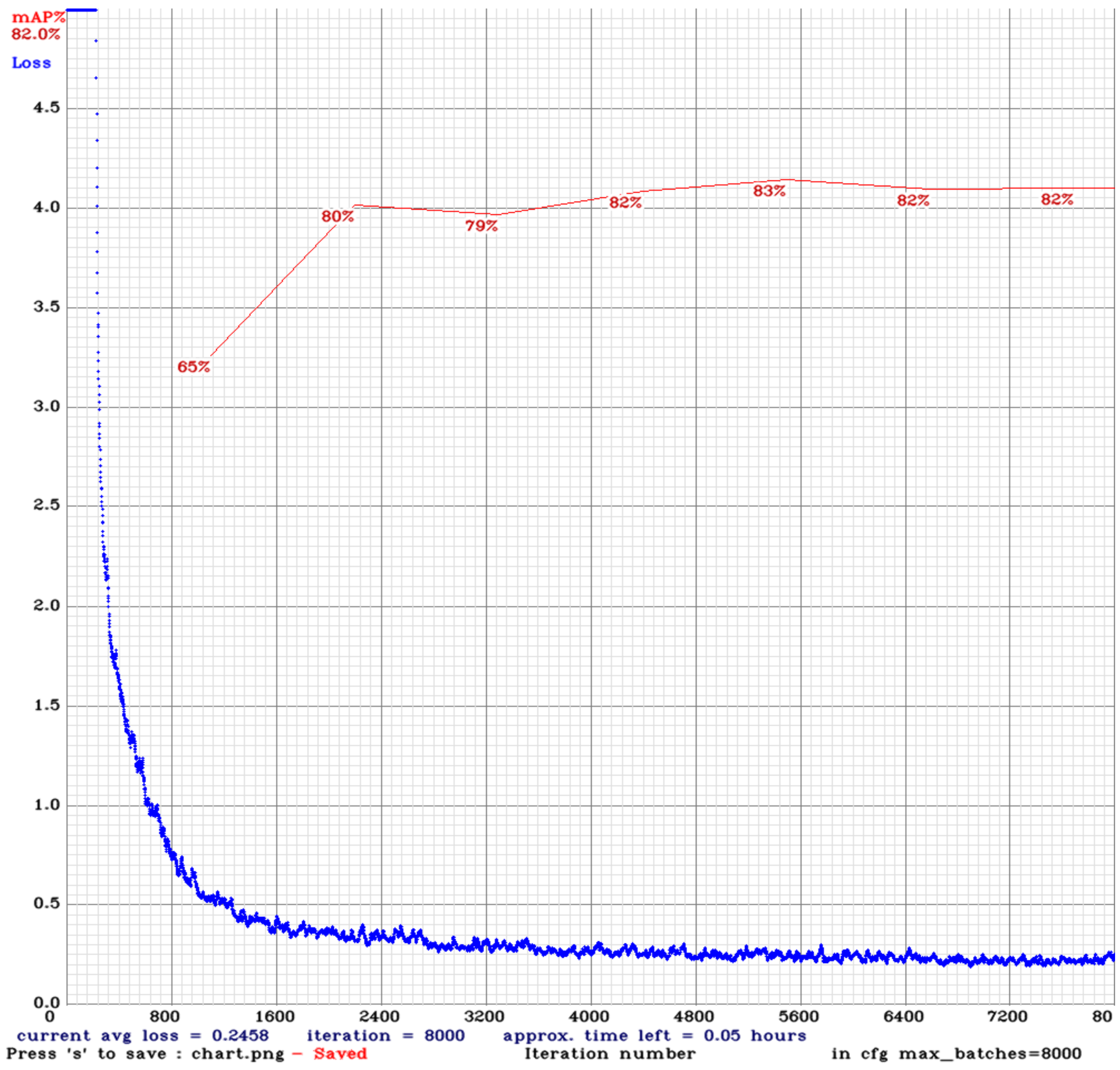


Figure 4.21: Loss Function (blue) and mAP (red) during training with YOLOv3-tiny

4.2.5 Inference

At this point, an approved trained network was ready to be implemented on the Loomo device. The network should cooperate with all other aspects of the app, hence inside an AndroidStudio environment. The valuable training-files to consider are the cfg- and weights file. The weigths file contains all information of the neural network, hence is big and demands much processing consumption. Instead of putting the weights file directly inside the app, they are manually added to the Loomo external memory. Therefore, an important action to remember is to allow permission for both reading and writing from the external memory. These files will be easy to reach, and the startup time is reduced whenever the app is launched. However, one need to implement the files on each specific device to accomplish functional inference.

Inside the app, an inference class(view the code here: Appendix A) was made to actually detect Loomo robots in real-time. Figure 4.22 shows a brief flow-chart of the inference class. At first one need to activate the inference class by pressing a button. A fisheye camera frame is generated, displayed on the device screen. and pushed through the neural network. When the frame has passed all layers, the YOLO-output layers have estimated Loomo positions and confidences. This information is processed and drawn to the device screen, overlapping the camera frame. The process runs inside an infinite loop, where new camera frames are constantly generated.

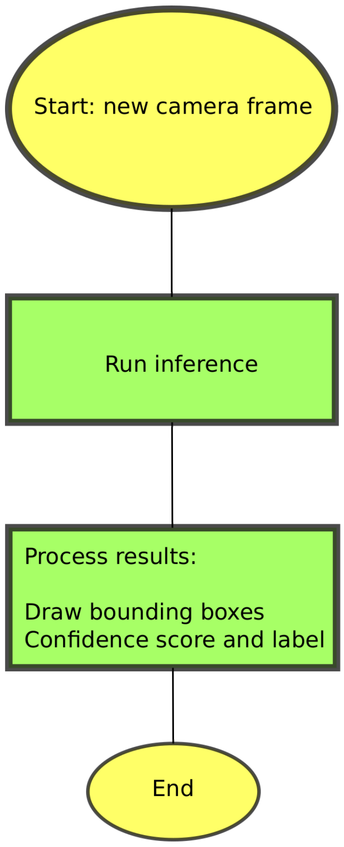


Figure 4.22: Inference

4.3 Simulating Swarm Intelligence for Collaborative Exploration

This chapter explains some concepts that are used in the implementation of the exploration simulation. It is coded in Processing. The source code for the simulation is linked in appendix A

The simulation assumes a functioning SLAM algorithm that is able to maintain an occupancy grid map. It also assumes ideal pose estimation. And that the pose of other bots are shared through the ROS-master.

4.3.1 Map

At the beginning of the simulation, a map data structure is initialized. The map consists of cells. Each cell holds information about their state. The cell data is structured like shown in listing 4.3

```
class Cell {
  float pField       =1.0; //Value of the potential field
  float probability  =0.5; //Value for the occupancy grid map
  float mapValue     =1.0; //Real value the cell, wall or free space
  int[] edge_id      =new int[4]; //Holds information about edges for ray casting
  bool[] edge_exist  =new bool[4]; //Holds information about edges for ray casting
  bool pFieldRendered=false; //cSpace is discovered once, true if rendered
}
```

Listing 4.3: Content of a cell in the map

4.3.2 Room Generator

To give the bots an environment to explore, an unknown building is generated on the map at the initialization of the simulation. The building consists of an array of many rooms with doorways and corridors. The room generator is parameterized which means that the width of the walls and room dimensions can be adjusted. There is also a chance of generating corridors, and the probability can be adjusted. There is a random chance for doors on either wall of the rooms. Figures 4.23 and 4.24 shows some generated buildings.

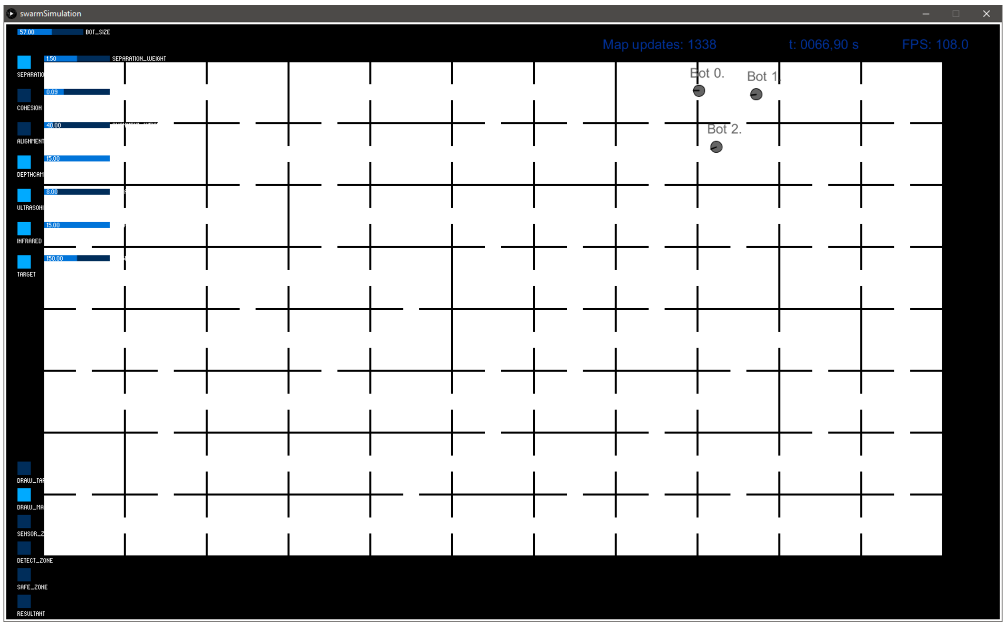


Figure 4.23: Randomly generated map

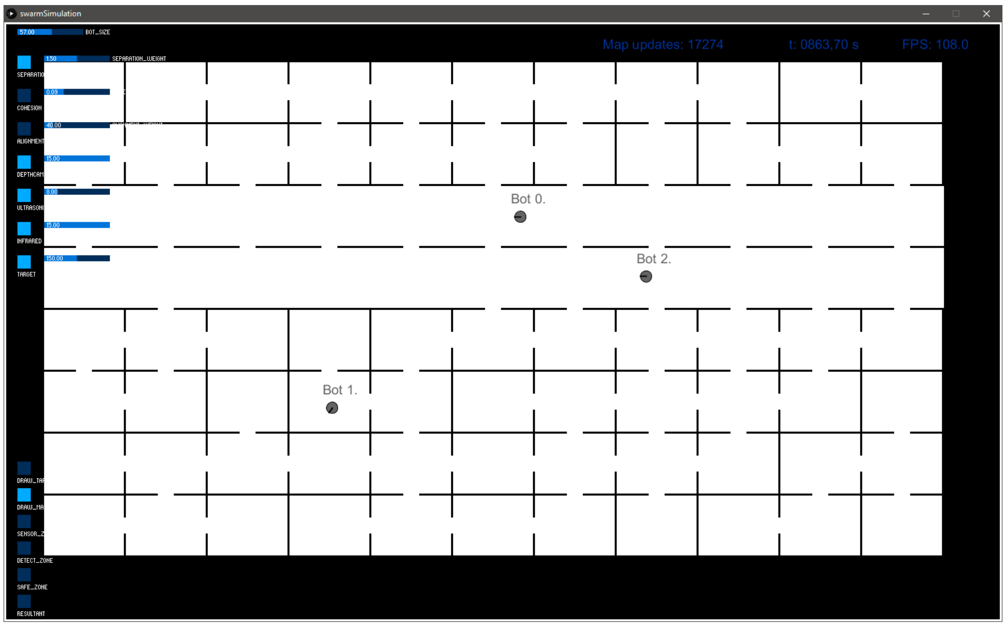


Figure 4.24: Randomly generated map with corridors

4.3.3 Sensors

The distance sensors are simulated by casting rays from the bot in a fan pattern to represent the sensor cone of the real sensors. The rays are used to calculate the distance to objects they interact with. The sensor cones of the real sensors are represented as a discretized number of these rays. For example the depth camera cone is only casting 10 rays in the entire FOV. This is to reduce computational load on the simulation. The ultrasonic sensor is represented by a single ray, and the infrared sensors are represented by 3 rays each. See figure 4.25 for the sensor ray pattern.

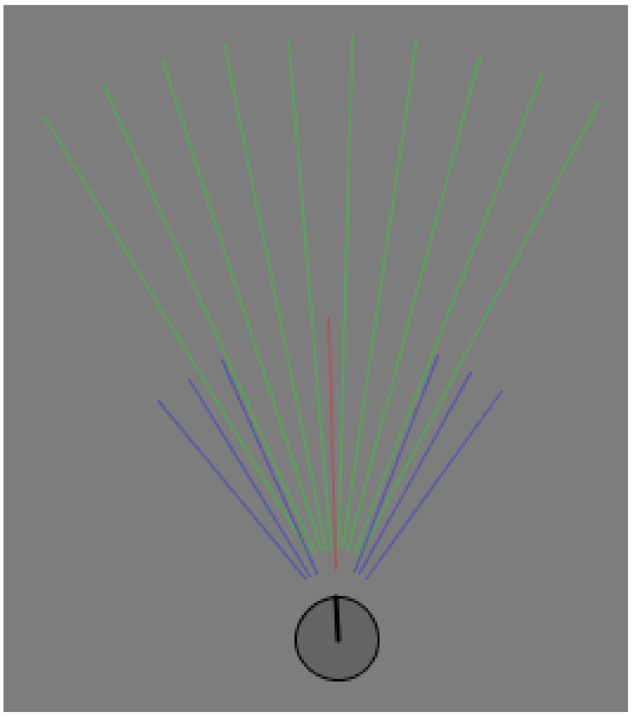
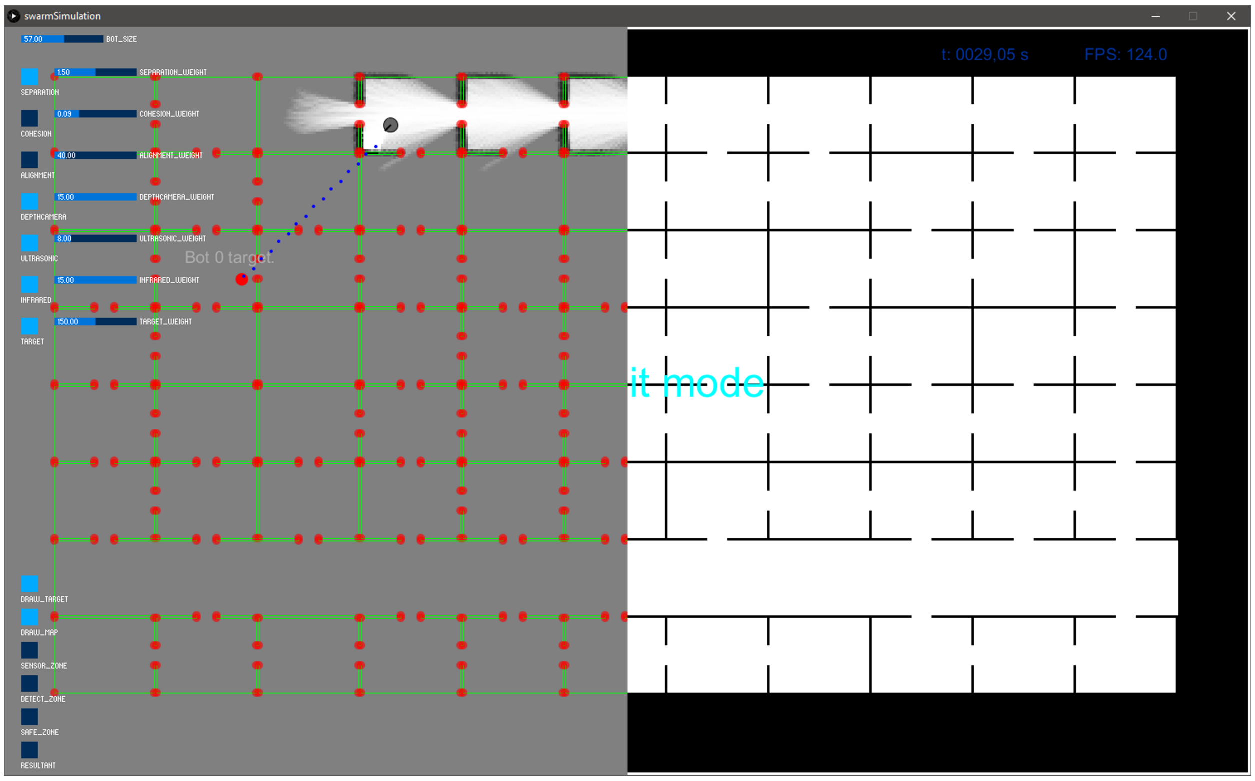


Figure 4.25: Green lines: Depth camera rays. Blue lines: Infrared rays. Red line: Ultrasonic ray

4.3.4 Walls and Obstacles

To properly simulate real life behaviour of the distance sensors, the rays from the bot sensors needs to intersect with objects. This is achieved by generating line vectors on the outer edge of the black cells (walls). After the map generation, an edge detection algorithm iterates through all of the cells in the map, and creates starting- and end point vectors for the edges. The red dots in figure 4.26 represent these points. Walls are generated by calculating the vector between the starting- and end points, and then drawing the vector as a green line. These are the vectors the sensor rays can interact with.


swarmSimulation

57.00 BOT_SIZE
1.50 SEPARATION_WEIGHT
0.09 COHESION_WEIGHT
40.00 ALIGNMENT_WEIGHT
15.00 DEPTHCAMERA_WEIGHT
8.00 ULTRASONIC_WEIGHT
15.00 INFRARED_WEIGHT
150.00 TARGET_WEIGHT

Figure 4.26: Visualization of wall edges. The left half visualize the edges with green lines and edge ends with red dots. The right half shows the map.

Figure 4.27 shows an example where the sensor rays collide and stop when the rays interacts with walls. This is only possible because of the vector lines representing the walls. 4.26

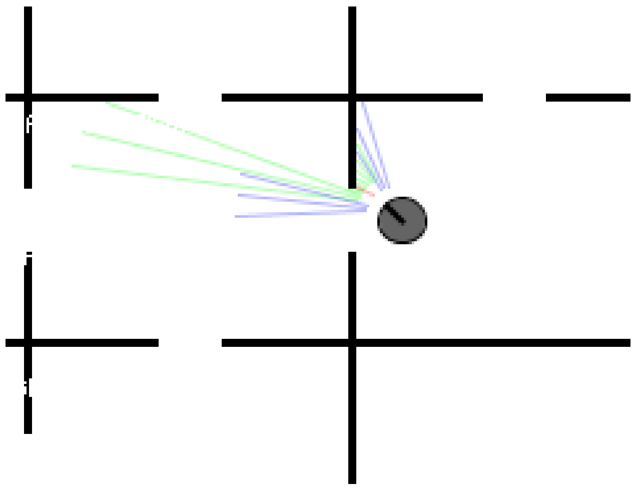


Figure 4.27: Ray casting; Sensor rays interacting with the map

4.3.5 Exploration

A bot will request a new target when it reaches it's current target. Upon request, the exploration algorithm will scan the area around the bot to find the target position which has the closest most undiscovered area. It does so by placing multiple ROIs in a circular pattern, and counting the amount of undiscovered cells within that ROI. see figure 4.28 for an illustration of the pattern. If the cells in the ROI is undiscovered the roiSum is increased by 1.

$$\text{roiSum} \mathrel{+}= \begin{cases} 1 & \text{if } 0.9 > \text{cell.probability} > 0.1 \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

The center coordinate of the ROI with the highest amount of undiscovered cells will become the new target position. For the ROI to be a valid target, the roiSum need to be above an adjustable amount of undiscovered cells. If there are no valid ROIs after the pattern has completed one rotation around the bot, the radius of the circular pattern is increased, and the scan repeats. The search runs until a valid ROI has been selected.

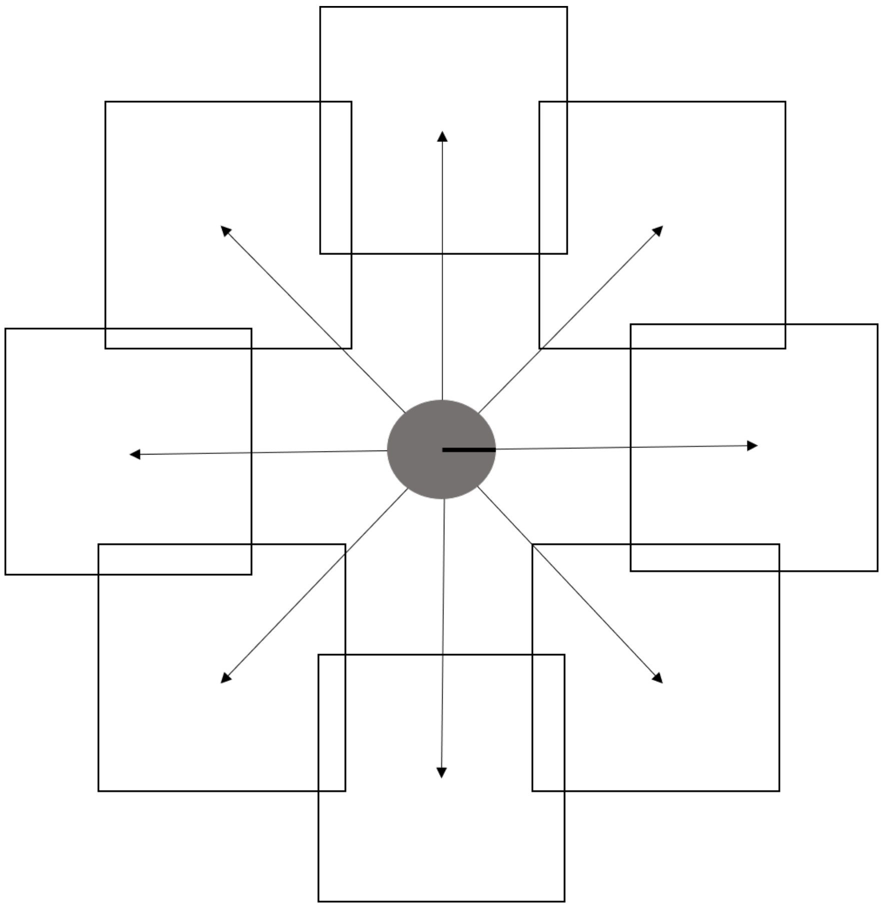


Figure 4.28: Illustration of exploration search pattern

4.3.6 Potential Field

As mentioned in the theory chapter, the potential field is used as an additional cost in the implementation of A*. When a wall is observed by a bot, the potential field is generated. The field is extended out from the wall, and decrease in intensity as the distance from the wall increases. The intensity of white means a higher cost for the path planner. Black cells give no extra cost. Figure 4.29 shows a fully generated potential field. By using this additional cost, the pathfinder can create a path which ensures a safe passage through doors, and help the robots keep sufficient distance to walls. Figure 3.17 show the potential field and some planned paths visualized by blue dots.

When a wall is observed, its potential field becomes active

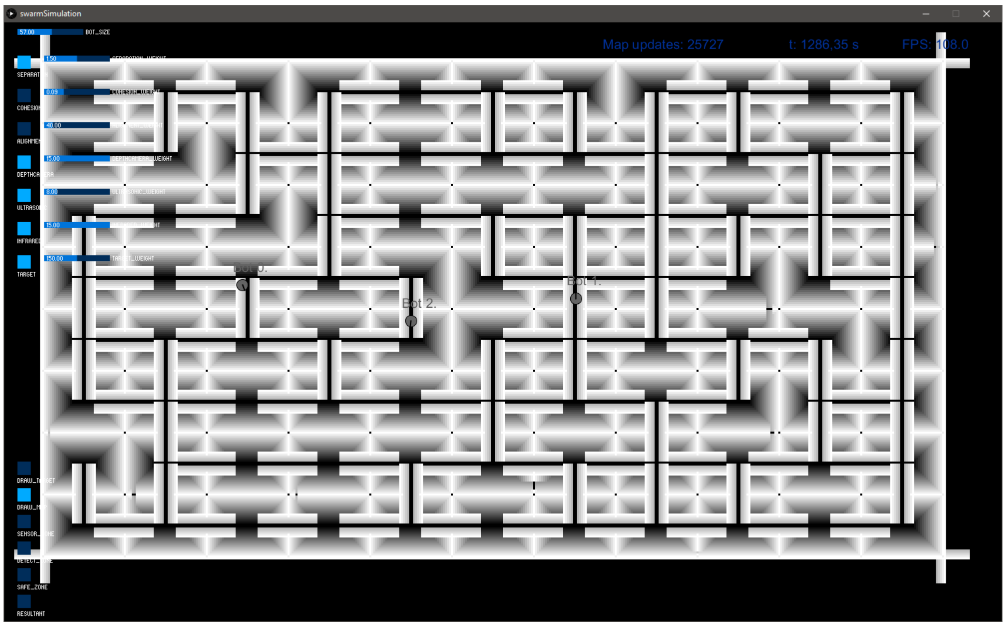


Figure 4.29: Visualization of the potential field

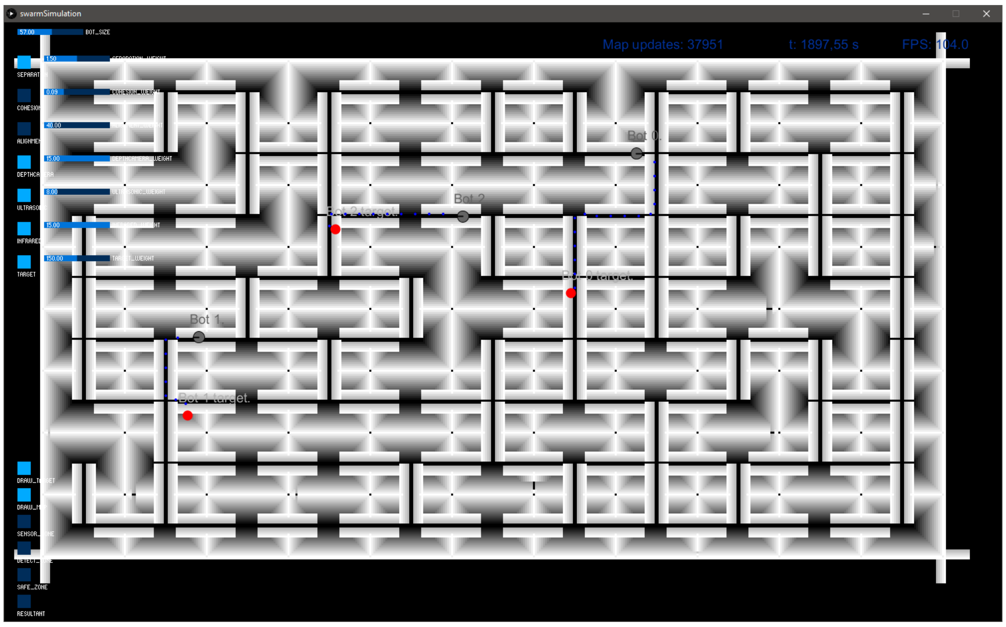


Figure 4.30: Visualization of path planning in the potential field

4.3.7 Pathfinding

This section explains the implementation of the A* pathfinding algorithm. The pathfinder is used when a bot receives a new target from the exploration algorithm, or when a bot is stuck. To keep track of which nodes to check, a stack called *nodesToCheck* is initialized. This stack enables the prioritizing of the best candidate node to explore.

Translating the cost equation shown in section 3.4.5 to variables used in the implementation:

$$f(n) = g(n) + h(n) \tag{4.24}$$

$$f(n) = globalValue \tag{4.25}$$

$$h(n) = \left|\vec{V_n} - \vec{V_g}\right| \tag{4.26}$$

Where

Symbol	Description	Unit
$\vec{V_n}$	Neighbor node position	m
$\vec{V_g}$	Goal position	m

The value of $g(n)$ is based on the current probability state, and the potential field value of the target node.

Step 1, Initialize:

Each cell in the map is initialized with a node, it holds information needed to perform the pathfinding. The node data is structured like shown in listing 4.4

```
class Node {
    int     id;         //Index of the node
    int     parent;     //Index of the parent node
    PVector pos;        //Position of the node
    boolean visited;    //Has the node been visited?
    float   globalValue;//f(n) in the A* cost function
    float   localValue; //g(n) cost of traveling to the node
}
```

Listing 4.4: Content of a node in the map

The *globalValue* and *localValue* of all nodes except the start node are initialized to ∞. The start node is initialized with a *localValue* of 0, and a *globalValue* of the distance from the node to the goal position.

The start and goal positions are initialized by getting the position of the bot, and the target from the exploration algorithm. This is shown in figure 4.31

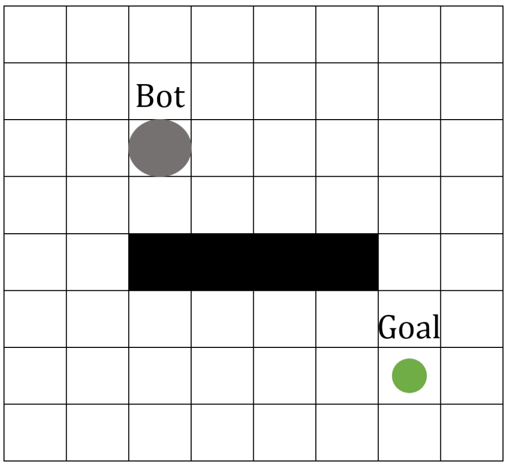


Figure 4.31: Start- and goal position

Step 2, Select current node:

Select the current node from the *nodesToCheck* stack. On the first loop, the start position node is selected (it is the only node in the list). On all consecutive loops, the node with the lowest *globalValue* is selected.

Step 3, Check neighbors:

The northern, southern, eastern and western neighbors of the current node is checked. See figure 4.32 for an illustration.

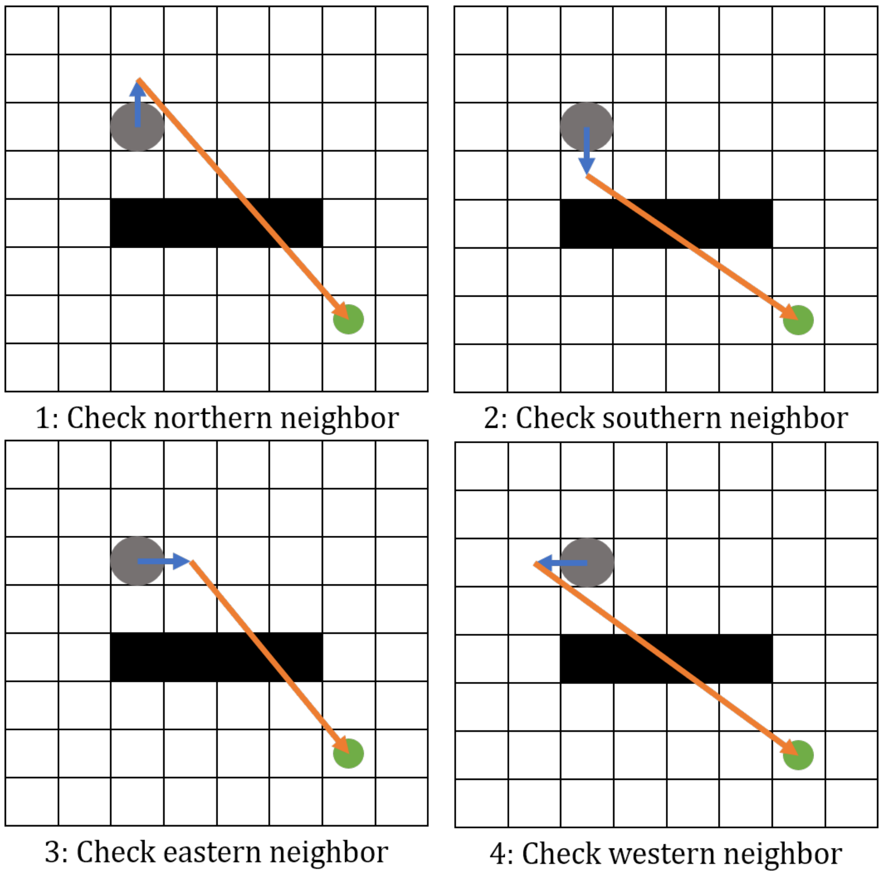

1: Check northern neighbor
2: Check southern neighbor
3: Check eastern neighbor
4: Check western neighbor

Figure 4.32: Evaluate neighbor cells

If the neighbor node is the goal position, Stop checking neighbors, and go to step 4.

If not, Evaluate the current node's *localValue* + cost of traveling to neighbor node, $g(n)$. If that value is lower than the current localValue of the neighbor node, Set current node as neighbor node's parent, and update the global- and localValue. Remove the current node from the *nodesToCheck* stack, and mark it as visited. If the neighbor node has not been visited, add it to the *nodesToCheck* stack.

Repeat step 2-3 until the goal position in reached.

Step 4, Create the path:

Follow the parent nodes backwards from the goal position until the start position is reached. The nodes visited is the path the bot has to follow to reach the goal.

Figure 4.33 show the results of one run of the pathfinder. The pink circles are the nodes that were checked, and the blue dots are the final path.

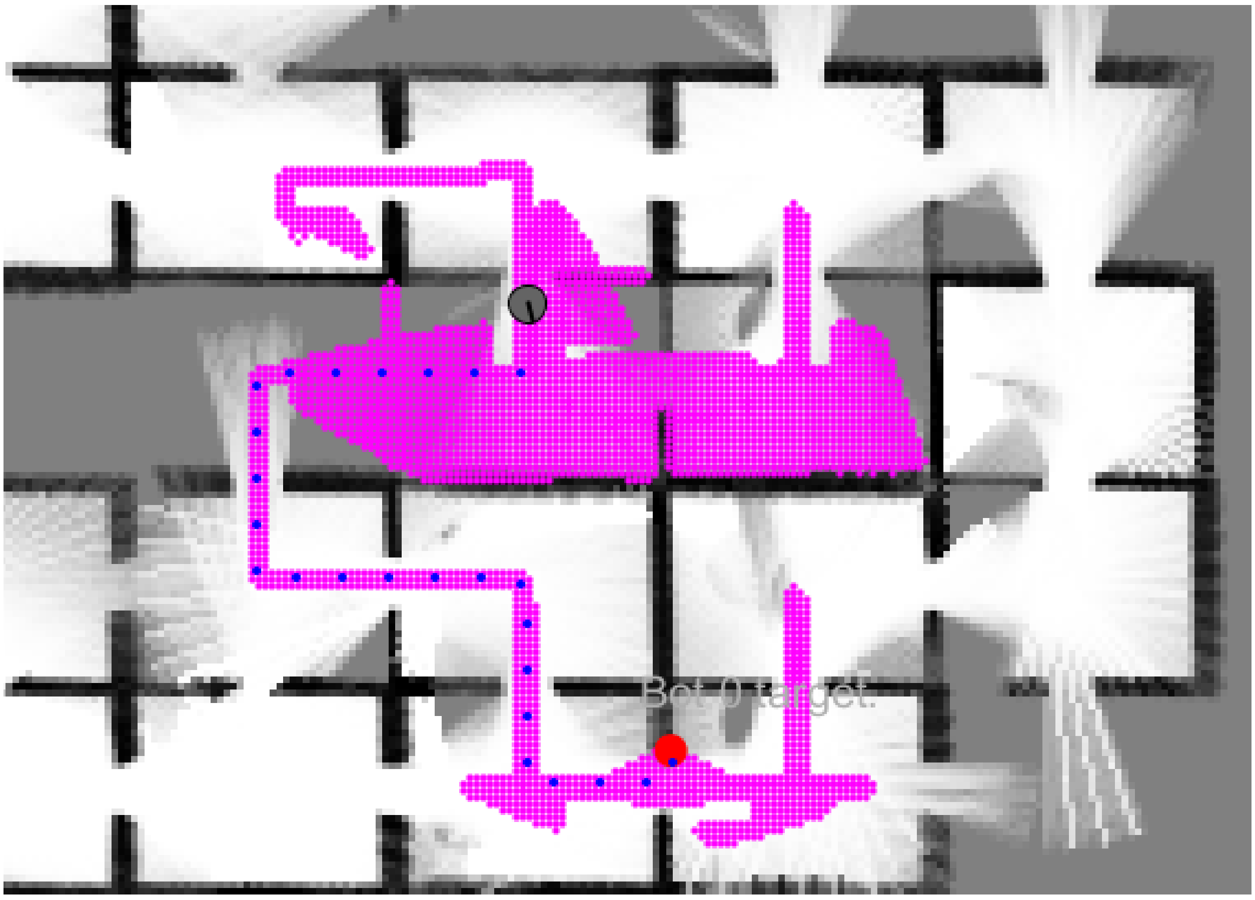


Figure 4.33: Pink dots represent evaluated cells. Blue dots represent the calculated path

The pathfinding algorithm will take the most direct route when there are undiscovered areas between the bot and the target position.

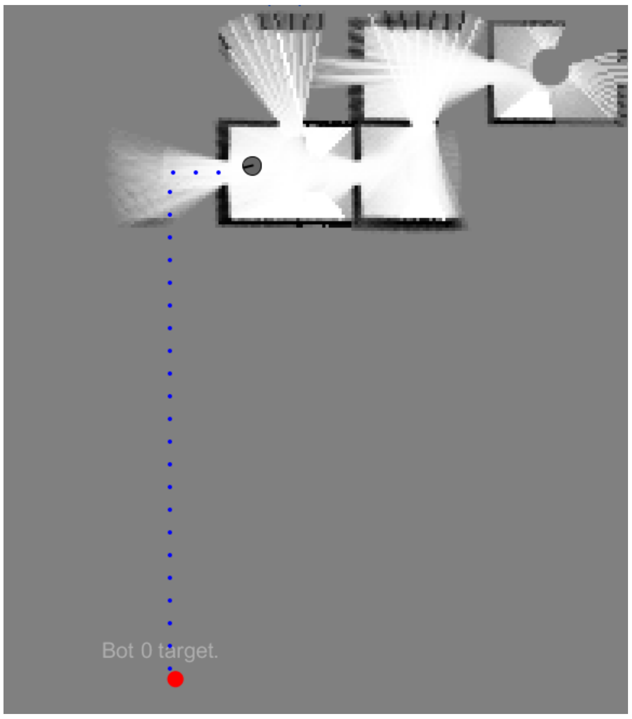


Figure 4.34: Example of pathfinding through undiscovered areas

This direct route approach will often result in the bot getting stuck. When that happens, the bot will request a new path. Normally more information has become available between each time the bot gets stuck. The pathfinding algorithm will use the new information to calculate a more accurate path. Figure 4.35 illustrate an example of this. This will result in a natural exploration behavior, and the bots will reach their targets eventually.

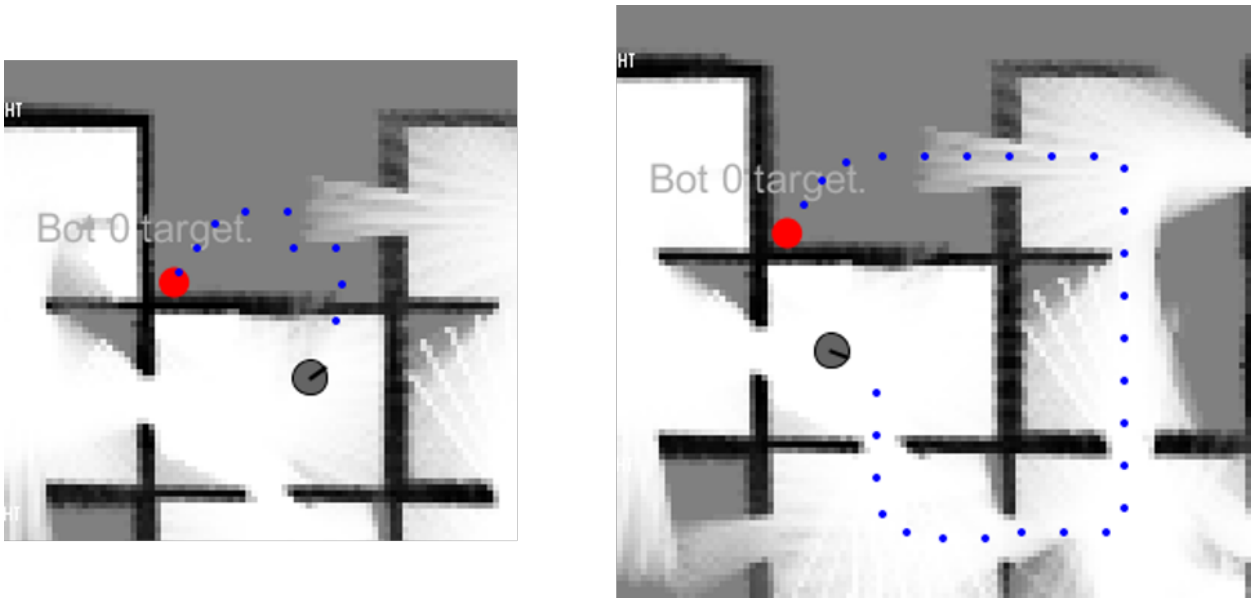


Figure 4.35: Example of rerouting after more information is available

4.3.8 Swarm Intelligence Rules

This is the implementation of the rule principles discussed in section 3.4.1. This simulation does not use the cohesion or alignment rule, but separation is very useful. All of the distance sensors have their own rule and there is a rule for following a target, this is used to follow the path generated by the exploration algorithm.

Separation

This rule results in a steering vector which will direct the bots away from each other if they are too close to each other.

$$\vec{V_d} = \vec{V_b} - \vec{V_t} \tag{4.27}$$

$$\hat{V_d} = \frac{\vec{V_d}}{\left|\vec{V_d}\right|} \tag{4.28}$$

$$\vec{V_{r_i}} = \begin{cases} \hat{V_d} \cdot k_a \cdot w \cdot \tanh\left(\left(R_\text{s} - \left|\vec{V_d}\right|\right) \cdot k_b\right) & \text{if } \left|\vec{V_d}\right| < R_\text{s} \\ \vec{0} & \text{otherwise} \end{cases} \tag{4.29}$$

Where

Symbol	Description	Unit
$\vec{V_d}$	Vector between bot and target bot	m
$\hat{V_d}$	Unit Vector between bot and target bot	m
$\vec{V_b}$	Bot position	m
$\vec{V_t}$	Target bot position	m
$\vec{V_{r_i}}$	Rule resultant vector	m
R_s	Radius of safe zone	m
w	Rule weight	-
k_a	Scaling factor	-
k_b	Scaling factor	-

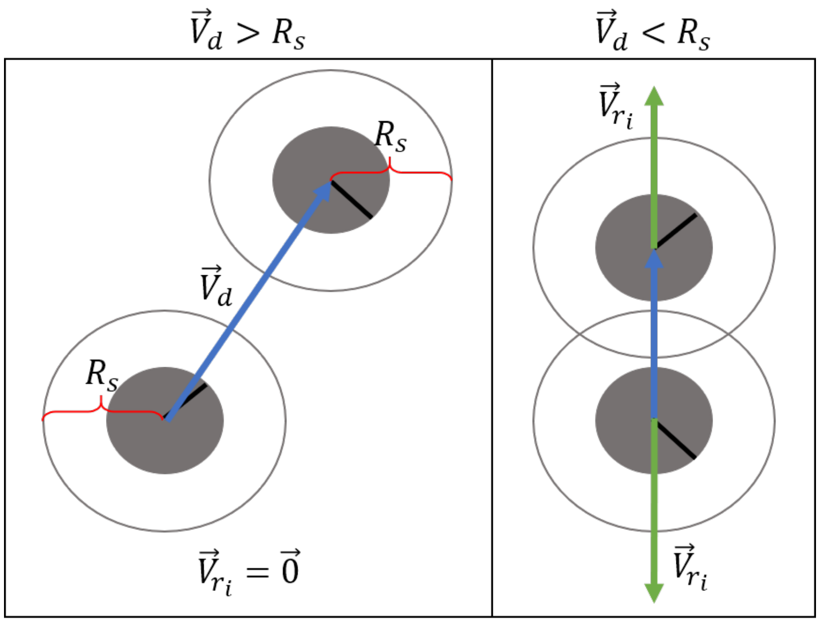


Figure 4.36: Visualization of the separation rule

Depth Camera

This rule creates a steering vector which will increase the turn rate of the bot such that it will steer away from obstacles. The simulated depth camera projects n_{r} rays in a fan pattern. A for-loop iterates through each ray and performs these calculations.

$$\theta_{\mathrm{r}_i} = \theta_b - \frac{\mathrm{FOV}}{2} + \left(i * \frac{\mathrm{FOV}}{n_{\mathrm{r}} - 1} \right) \tag{4.30}$$

$$L_{r_i} = \left| \vec{R_{e_i}} - \vec{R_{s_i}} \right| \tag{4.31}$$

$$\mathrm{l}_r = (L_{r_i} - l_s) * w \tag{4.32}$$

$$\alpha_r = \theta_b + \frac{\pi}{2} * \mathrm{sign}(\theta_b - \theta_{\mathrm{r}_i}) \tag{4.33}$$

$$\vec{V_{r_i}} = \begin{cases} \begin{bmatrix} \mathrm{l}_r \cdot \cos(\alpha_r) \\ \mathrm{l}_r \cdot \sin(\alpha_r) \end{bmatrix} & \text{if } L_{r_i} < l_s \\ \vec{0} & \text{otherwise} \end{cases} \tag{4.34}$$

Where

Symbol	Description	Unit
θ_{r_i}	Angle of ray	rad
θ_b	Angle of bot	rad
FOV	Field of view of depth camera	rad
i	Iterator	-
n_r	Number of rays projected from the camera	-
L_{r_i}	Length of ray	m
$\vec{R_{e_i}}$	End position of of ray	m
$\vec{R_{s_i}}$	Start position of ray	m
l_r	Resultant vector length	m
l_s	Span: length of ray if there are no interactions	m
w	Rule weight	-
α_r	Resultant vector direction	rad
$\vec{V_{r_i}}$	Rule resultant vector	m

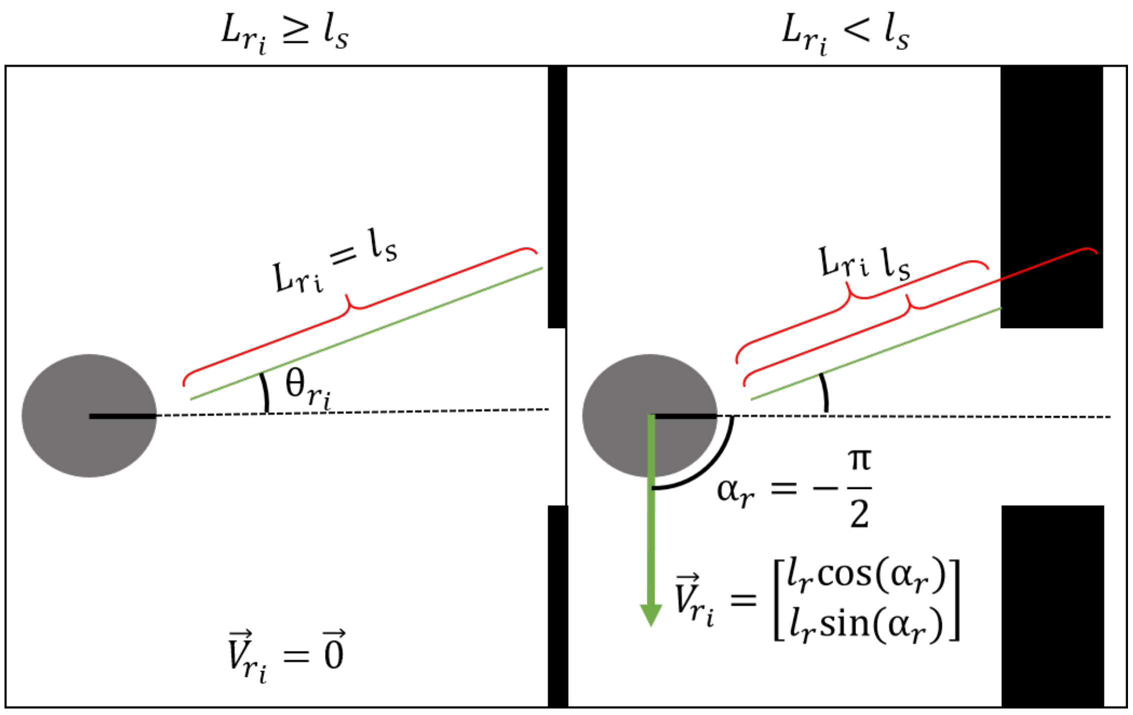

$L_{r_i} \geq l_s$
$L_{r_i} < l_s$
$L_{r_i} = l_s$
θ_{r_i}
$\vec{V}_{r_i} = \vec{0}$
L_{r_i} l_s
$\alpha_r = -\frac{\pi}{2}$
$\vec{V}_{r_i} = \begin{bmatrix} l_r\cos(\alpha_r) \\ l_r\sin(\alpha_r) \end{bmatrix}$

Figure 4.37: Visualization of a single ray in the depth camera rule

Ultrasonic

The ultrasonic sensor is represented by a single ray pointing forward. The steering vector will always point backward and this will make sure the bot stop its forward velocity if it is close to obstacles.

$$\theta_{\mathrm{r}_i} = \theta_b \tag{4.35}$$

$$L_{r_i} = \left|\vec{R_{e_i}} - \vec{R_{s_i}}\right| \tag{4.36}$$

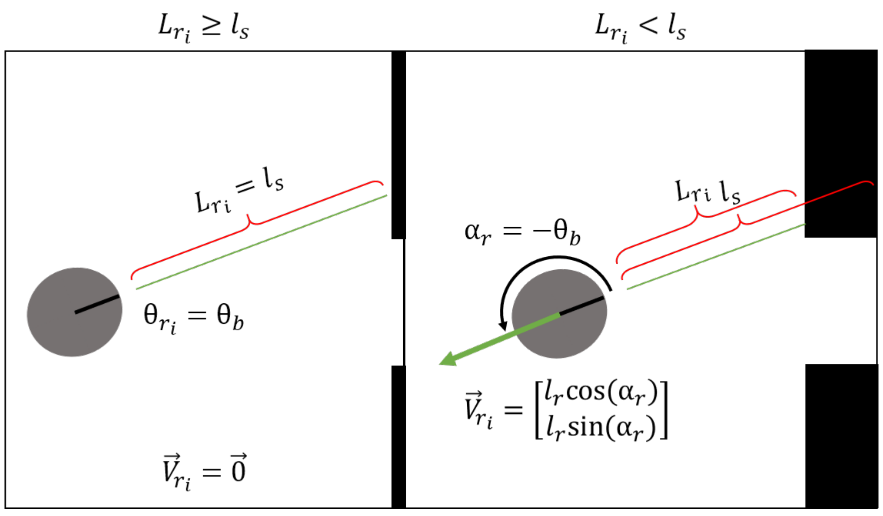
$$\mathrm{l}_r = (L_{r_i} - l_s) * w \tag{4.37}$$

$$\alpha_r = -\theta_b \tag{4.38}$$

$$\vec{V_{r_i}} = \begin{cases} \begin{bmatrix} \mathrm{l}_r \cdot \cos(\alpha_r) \\ \mathrm{l}_r \cdot \sin(\alpha_r) \end{bmatrix} & \text{if } L_{r_i} < l_s \\ \vec{0} & \text{otherwise} \end{cases} \tag{4.39}$$

Where

Symbol	Description	Unit
θ_{r_i}	Angle of ray	rad
θ_b	Angle of bot	rad
L_{r_i}	Length of ray	m
$\vec{R_{e_i}}$	End position of ray	m
$\vec{R_{s_i}}$	Start position of ray	m
l_r	Resultant vector length	m
l_s	Length of ray if there are no interactions	m
w	Rule Weight	-
α_r	Resultant vector direction	rad
$\vec{V_{r_i}}$	Rule resultant vector	m



$L_{r_i} \geq l_s$
$L_{r_i} < l_s$
$L_{r_i} = l_s$
$\theta_{r_i} = \theta_b$
$\vec{V}_{r_i} = \vec{0}$
L_{r_i} l_s
$\alpha_r = -\theta_b$
$\vec{V}_{r_i} = \begin{bmatrix} l_r \cos(\alpha_r) \\ l_r \sin(\alpha_r) \end{bmatrix}$

Figure 4.38: Visualization of the ultrasonic rule

Infrared

This rule creates a steering vector which will increase the turn rate of the bot such that it will steer away from objects. This rule is very similar to the depth camera rule, except for determining the angle of the resultant vector. This rule simply adds the vector on the opposite side of the triggered sensor. i.e. Left sensor triggers, steer right.

$$L_{r_i} = \left|\vec{R_{e_i}} - \vec{R_{s_i}}\right| \tag{4.40}$$

$$\mathrm{l}_r = (L_{r_i} - l_s) * w \tag{4.41}$$

$$\alpha_r = \begin{cases} \theta_b - \frac{\pi}{2} & \text{if left sensor} \\ \theta_b + \frac{\pi}{2} & \text{if right sensor} \end{cases} \tag{4.42}$$

$$\vec{V_{r_i}} = \begin{cases} \begin{bmatrix} \mathrm{l}_r \cdot \cos(\alpha_r) \\ \mathrm{l}_r \cdot \sin(\alpha_r) \end{bmatrix} & \text{if } L_{r_i} < l_s \\ \vec{0} & \text{otherwise} \end{cases} \tag{4.43}$$

Where

Symbol	Description	Unit
θ_b	Angle of bot	rad
i	Iterator	-
L_{r_i}	Length of ray	m
$\vec{R_{e_i}}$	End position of of ray	m
$\vec{R_{s_i}}$	Start position of ray	m
l_r	Resultant vector length	m
l_s	Span: length of ray if there are no interactions	m
w	Rule weight	-
α_r	Resultant vector direction	rad
$\vec{V_{r_i}}$	Rule resultant vector	m

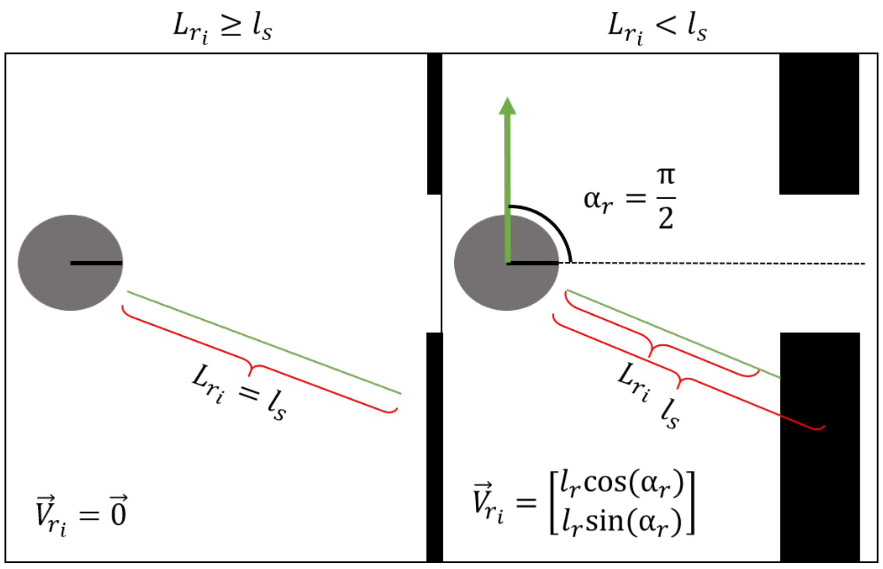

$L_{r_i} \geq l_s$
$L_{r_i} < l_s$
$L_{r_i} = l_s$
$\vec{V}_{r_i} = \vec{0}$
$\alpha_r = \frac{\pi}{2}$
L_{r_i}
l_s
$\vec{V}_{r_i} = \begin{bmatrix} l_r\cos(\alpha_r) \\ l_r\sin(\alpha_r) \end{bmatrix}$

Figure 4.39: Visualization of a single ray in the infrared rule (right sensor)

Target

This rule make the bots drive towards a target position. It is used to make the bots follow the path set by the pathfinder algorithm. The path is stored as an array of waypoints. When the bot gets within a certain distance to a waypoint, the current waypoint is deleted, and the next waypoint in the path becomes the new target. This process continues until the bot reaches the end of the path, which is the goal position.

$$\vec{V_d} = \vec{V_b} - \vec{V_w} \tag{4.44}$$

$$\hat{V_d} = \frac{\vec{V_d}}{\left|\vec{V_d}\right|} \tag{4.45}$$

$$\vec{V_{r_i}} = \hat{V_d} \cdot w \tag{4.46}$$

Where

Symbol	Description	Unit
$\vec{V_d}$	Vector between bot and current waypoint	m
$\hat{V_d}$	Unit Vector between bot and current waypoint	m
$\vec{V_b}$	Bot position	m
$\vec{V_w}$	Current Waypoint position	m
$\vec{V_{r_i}}$	Rule resultant vector	m
R_w	Radius of waypoint zone	m
w	Rule weight	-

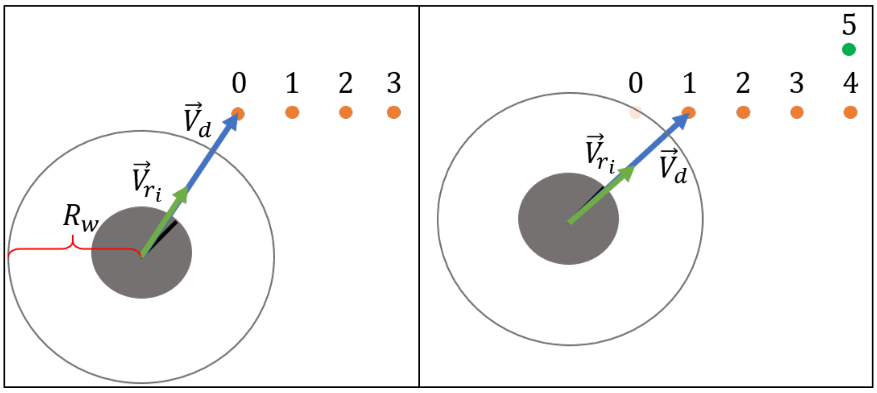


Figure 4.40: Visualization of the target rule

Combining the resultant vectors

As mentioned in the theory section 3.4.6. The main resultant vector is simply the average of all the rule resultant vectors. Zero vectors do not contribute to the average vector. n is increased by 1 for each valid rule. The linear- and angular velocity, v and ω, is the decomposition of $\vec{V_c}$

$$\vec{V_c} = \frac{1}{n}\sum_i \vec{V_{r_i}} \tag{4.47}$$

$$v = \left|\vec{V_c}\right| \cdot \cos(\theta_r - \theta_L) \tag{4.48}$$

$$\omega = \left|\vec{V_c}\right| \cdot \sin(\theta_r - \theta_L) \tag{4.49}$$

Where

Symbol	Description	Unit
$\vec{V_c}$	Resultant vector of combined rules	m/s
v	Linear velocity setpoint	m/s
ω	Angular velocity setpoint	m/s
n	Total number of resultant vectors	-
$\vec{V_{r_i}}$	Resultant vector of individual rule	m/s
θ_r	Angle of resultant vector in global space	rad
θ_L	Angle of Loomo in global space	rad

Chapter 5

Results

5.1 Neural Network Performance

A neural network has been created to identify and detect Loomo robots through their fisheye camera. The neural network is implemented inside the Loomo app, and this section will validate the performance.

5.1.1 Validation

YOLOv2-tiny and YOLOv3-tiny are both strong candidates to be chosen. A performance comparison was made from the validation-dataset, and table 5.1 shows that YOLOv3-tiny has significant better precision. The validation is obtained from the validation set, containing 3800 images. If the image contains equal or more than one loomo, the total IoU score must be above 0.5 to count as a true positive.

	YOLOv3-tiny	**YOLOv2-tiny**
True Positives	2861	2073
False Positives	198	1377
False Negatives	979	1767
Recall	0.75	0.54
mAP(%)	**82.81**	51.44
IoU(%)	**73.54**	44.28

Table 5.1: Validation comparison between YOLOv3-tiny and YOLOv2-tiny

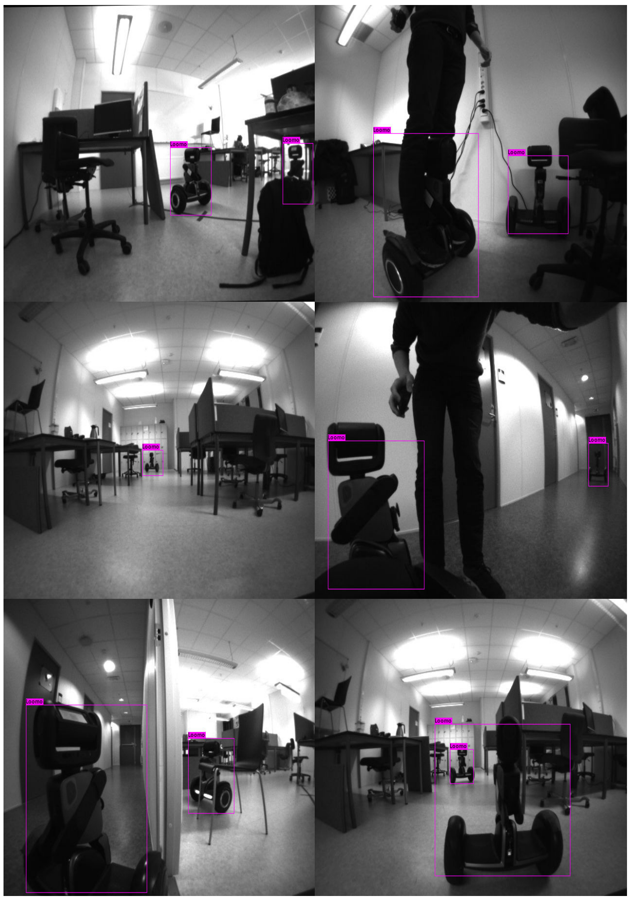


Figure 5.1: YOLOv3-tiny predictions

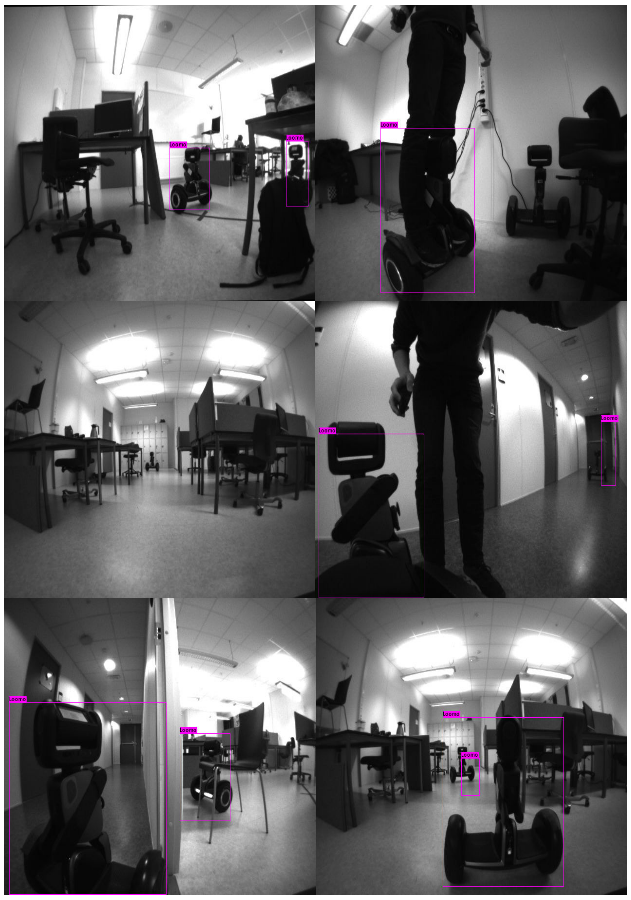


Figure 5.2: YOLOv2-tiny predictions

Inference

Table 5.2 shows that the the time for the Loomo to detect next image is barely beneath one second. YOLOv2-tiny are not much, but significantly faster

	YOLOv3-tiny	**YOLOv2-tiny**
Inference	0.92s	**0.80s**

Table 5.2: Inference

Score analysis

The two current neural networks beats each other in the characteristics we value the most. A simple score analysis was made, shown in table 5.3. Max individual score is 10, the lowest is 0. YOLOv3-tiny has the best score and will be the selected default neural network on the Loomo-app.

	Inference score	**Precision score**	**Total score**
YOLOv3-tiny	2	8	**10**
YOLOv2-tiny	3	5	8

Table 5.3: Performance score

5.2 Exploration Simulation

These are the results of simulating different number of bots in 5 different random buildings. The goal is to determine effectiveness of exploration vs number of bots used. A cell is considered discovered when the probability value of that cell is >0.8 or <0.2. The simulation finishes when 90% of the cells in the building are discovered

Parameter	**Value**
Building size	$50m \times 30m$
Room size	$4m \times 5m$
Door width	$80cm$
Cell Size	$10cm \times 10cm$
bot Size	$57cm$
bot max linear speed	$0.8\ m/s$
bot max angular speed	$0.5\ rad/s$

Table 5.4: Simulation parameters

Number of bots	**Building 1**	**Building 2**	**Building 3**	**Building 4**	**Building 5**	**Average**
1	2675	2128	2160	2013	2782	2352
2	1221	1099	1025	1108	1653	1221
3	863	759	959	721	1131	887
4	600	558	903	539	786	677
5	544	509	676	513	659	582
7	445	419	746	380	526	503
9	-	355	608	395	426	446
11	-	378	614	329	434	438

Table 5.5: Simulation results. Numbers in columns 2-7 are time in seconds

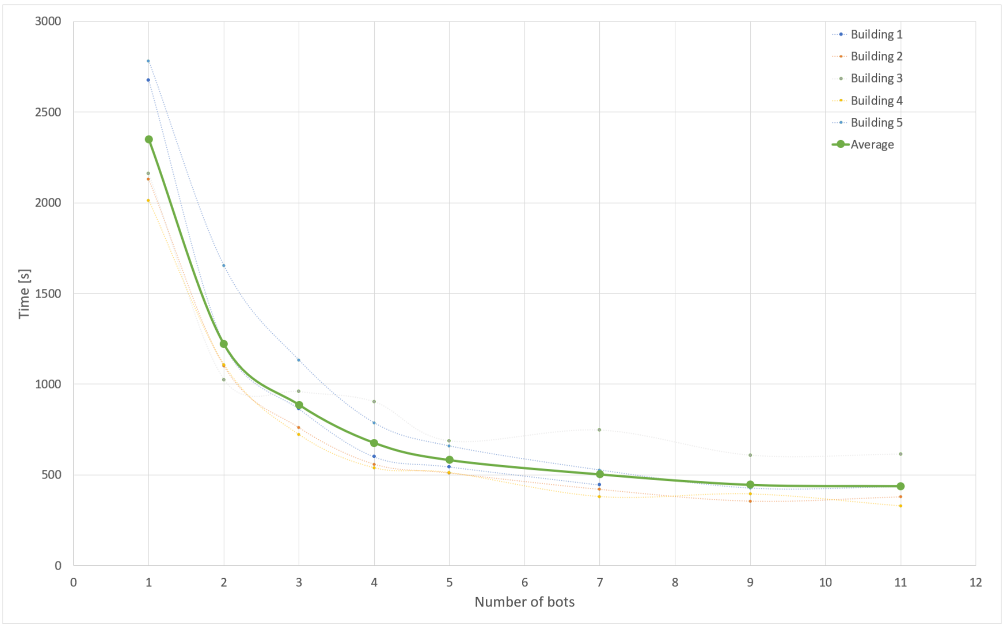


Figure 5.3: Simulation results. Graph of table 5.4

There is a clear trend for diminishing returns on time efficiency with increasing amount of bots. 4-5 bots seems to be the optimal amount for this configuration.

Equation 5.1 is the curve fitting of the average series from table 5.4. The function can be used to estimate the time needed to explore a building with this configuration of the simulation.

$$T(n_{bot}) = 2035 \cdot {n_{bot}}^{-1.065} + 285.4 \tag{5.1}$$

5.3 SLAM Results

Figure 5.4 shows a baseline graph. It is based only on odometry. In it, the Loomo started and stopped in the same position, but faces opposite directions in the start and end. The Loomo drives in a hallway, turns 90° to the right, drives to the end of the hallway, turns 180° and follows the same path back to start. The odometry trajectory *misses* by $6.8m$. This number by itself does not give a good indication of the overall performance of the state estimation, but it illustrates how errors in the heading accumulate and is detrimental to the lateral position estimation. The total length of the trajectory according the odometry is $42.2m$.

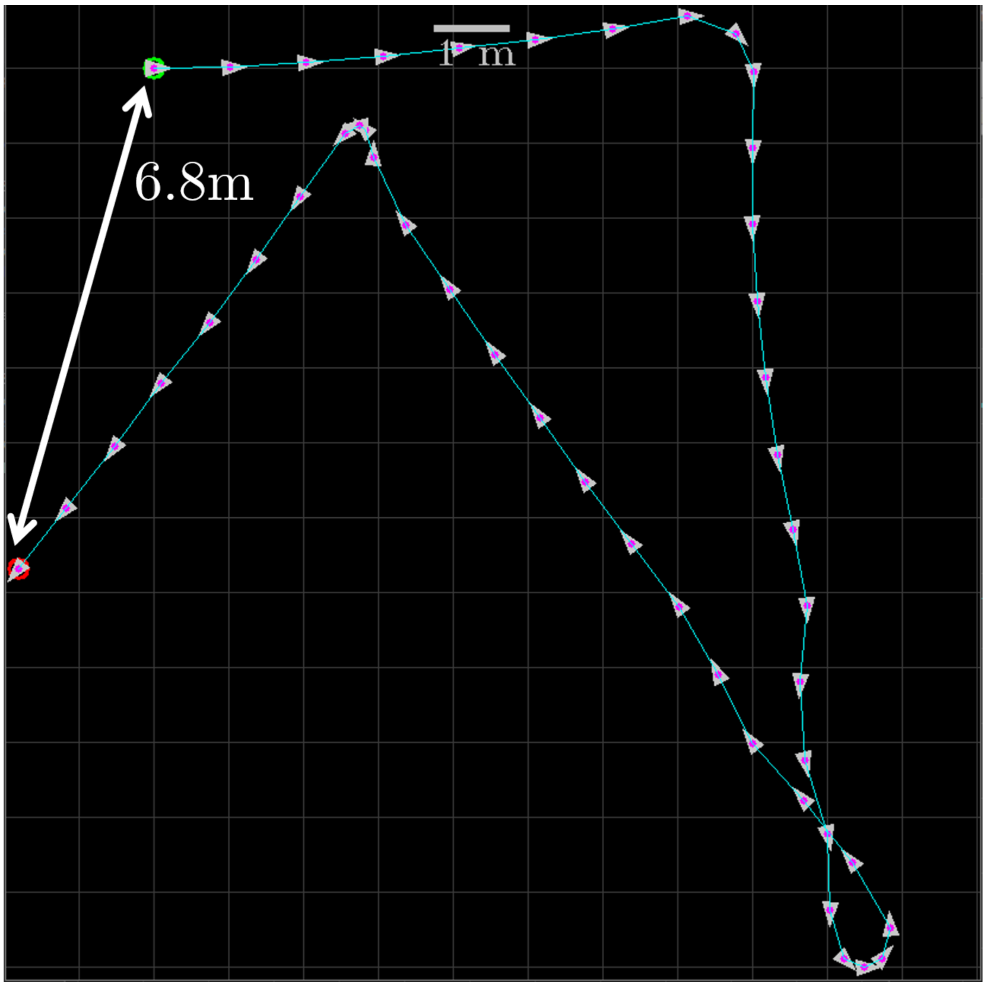


Figure 5.4: Baseline graph. No graph optimization has been done

Figure 5.5 shows a graph that uses visual-odometry-type of virtual edges as described in section

4.1.7. The graph *misses* by $0.45m$. That is considerably better than mere odometry. The graph did not estimate 90° turns though. The graph is however locally consistent. The parts of the trajectory that are supposed to be parallel and overlayed on each other are exactly that.

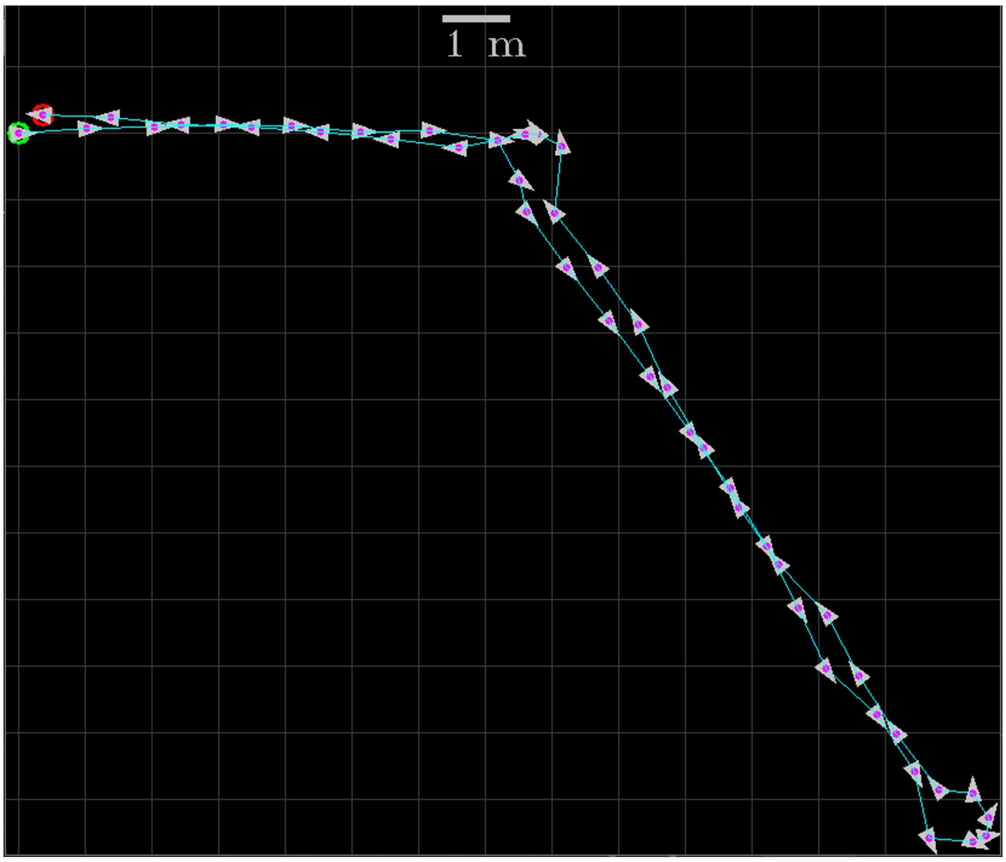


Figure 5.5: Graph using odometry and Visual Odometry. There are no loop closures in the graph.

There are no detectable loop closures in the dataset that has been displayed here. However, since the start and end pose is known, a single loop closure can be artificially added to the graph. Figure 5.6 shows the graph using only the odometry based edges, and an single virtual edge between the first and last node.

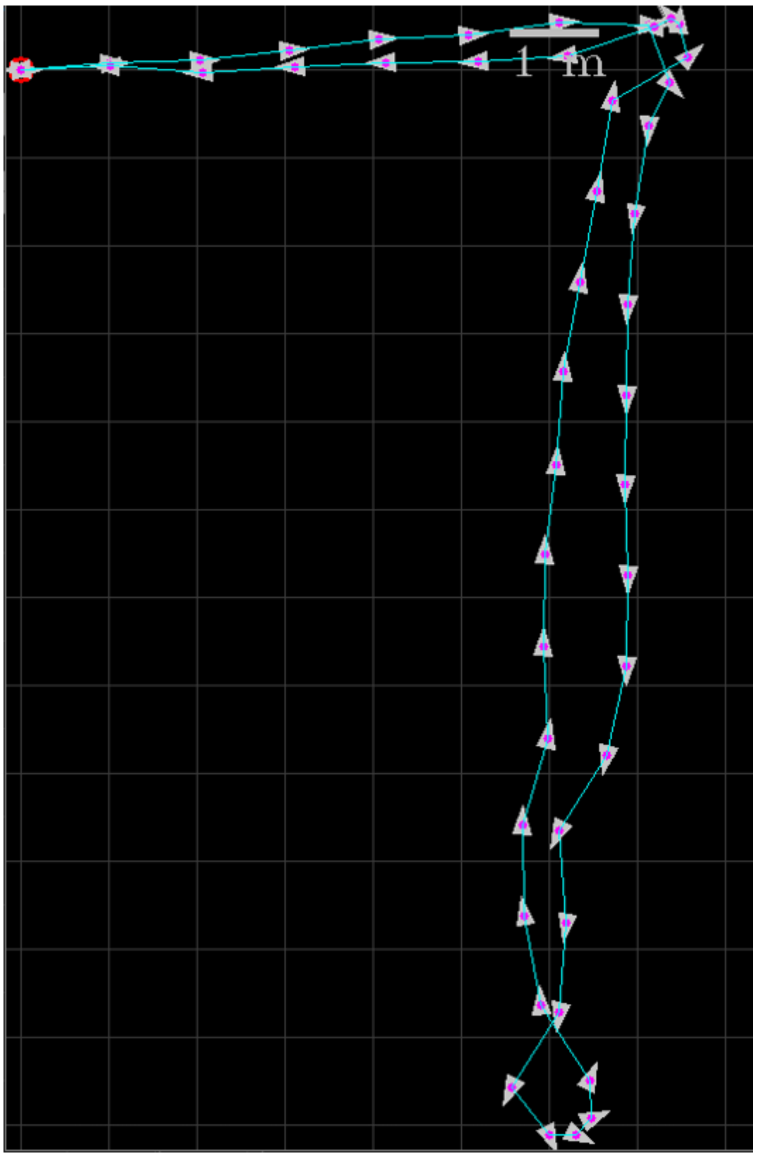


Figure 5.6: Graph with only odometry based edges and a single loop closure

5.4 Computer Vision Performance

A simple benchmarking test was done to asses how the number of images features affects computation time. The ORB detector in OpenCV has a parameter for how many features it should retain, so it was useful to evaluate whether there was a benefit to adjusting this value. The test was done by timing how long it took to detect features and compute their descriptors. This was done for each frame in two Loomo video recordings. One of the recordings was in an environment with few features, and the other was in a feature dense environment. The results are shown in figure 5.7, and they show that there is a constant time penalty for performing the detection, plus a linear increase in time for detecting more features.

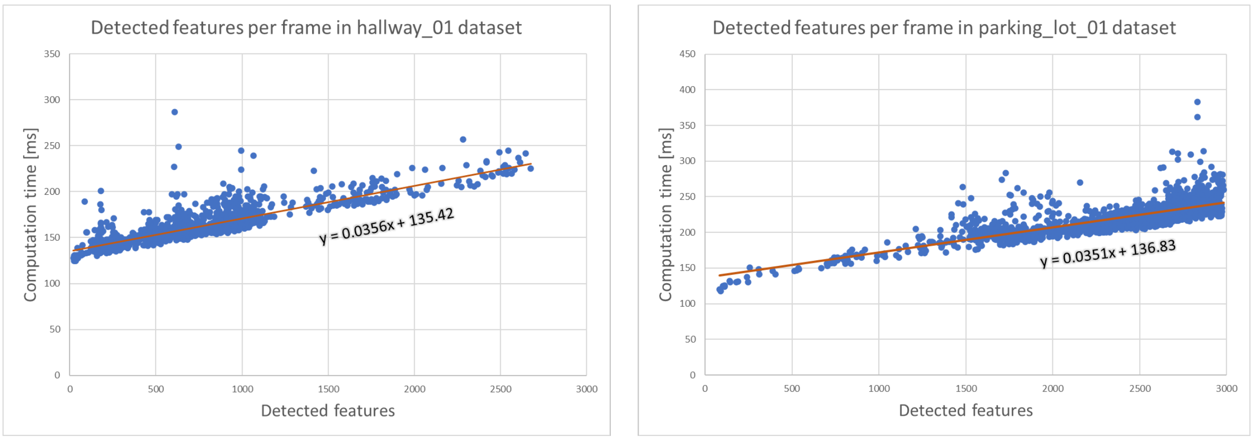


Figure 5.7: Computation time for detecting features and computing their descriptors vs. how many features were detected. The left graph is for a feature poor environment, and the right graph is for a feature rich environment.

Figure 5.8 demonstrates the effect that different lighting conditions has on feature detection. Outdoors is a particularly challenging environment. The two images to the left are of the same scene, but the top one is during a sunny afternoon and the bottom is during twilight. The shadow of the person driving the Loomo is being detected in the top left image. Barely any features that lie in shadow are detected in the top right and bottom left image. Notice also that in the indoor image (bottom right) some light reflections are being detected on the doors.

Figures 5.9 and 5.10 shows matching results when no additional filtering is applied. In figure 5.9 the images are of the same scene, but there are many wrong data associations that are made. In figure 5.10 the images are of different scenes, but there is still a high number of matches.

Figure 5.11 and 5.12 shows the effect of applying Lowe's ratio test. In figure 5.11 the images are of the same scene, and there are much fewer data associations that are wrong. In figure 5.12 the images are of different scenes, and the number of matches is substantially lower.

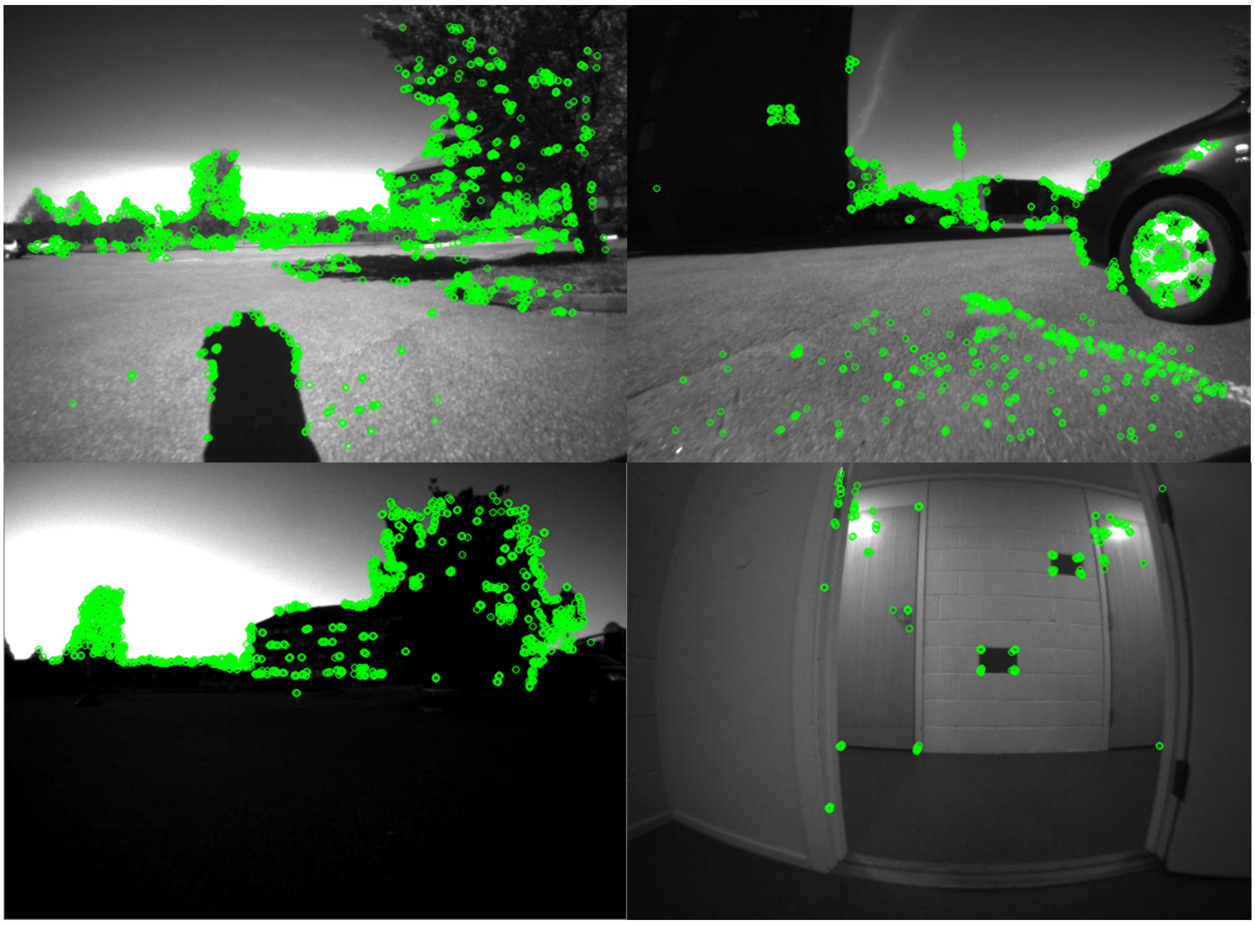


Figure 5.8: Effect of lighting for feature detection

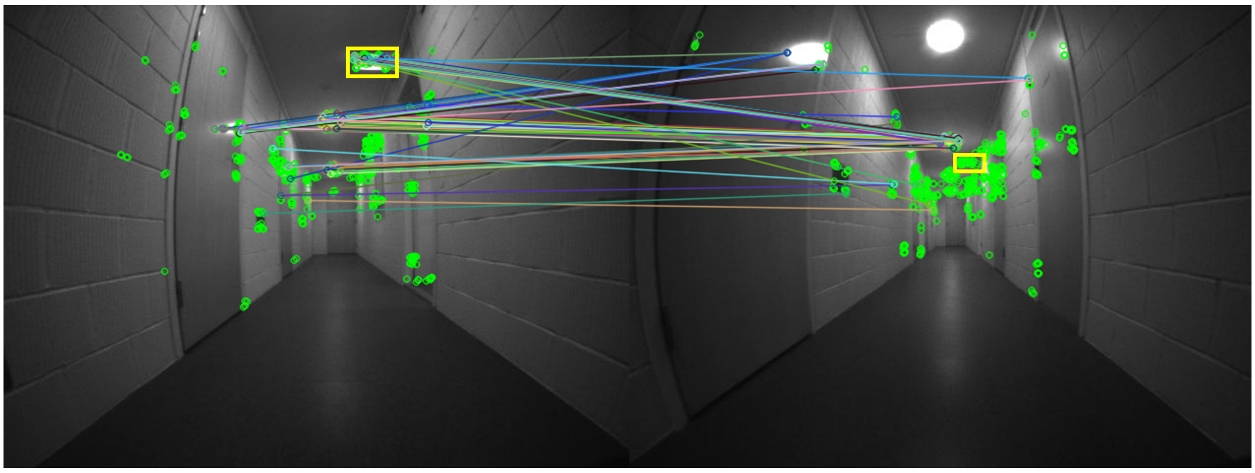


Figure 5.9: Images of the same scene. The yellow rectangles are corresponding locations in both images. None of the features in the yellow rectangle in the left image matches with the correct features in the right image

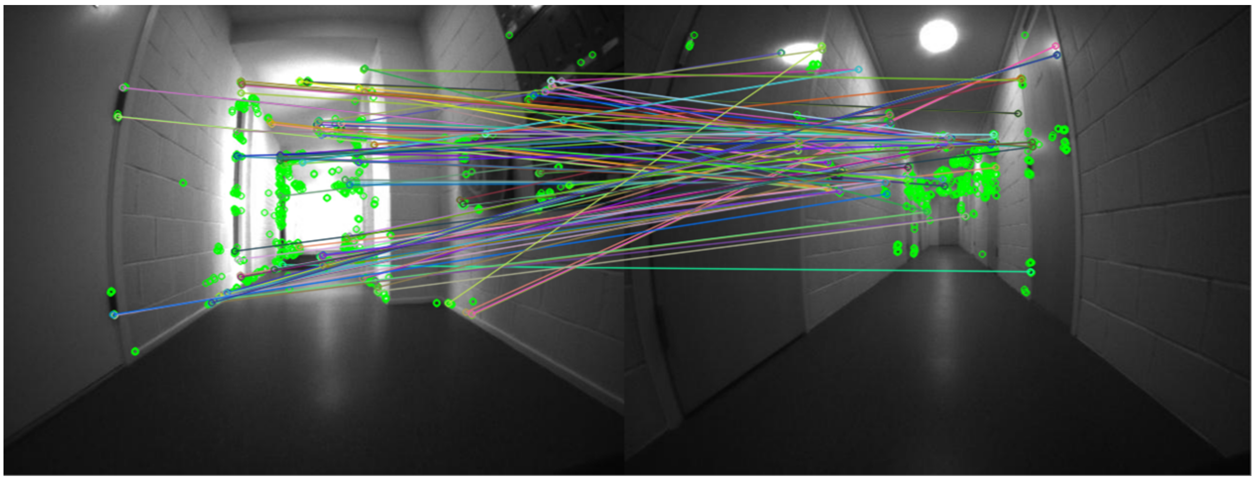


Figure 5.10: Different scenes that has a high amount of matches

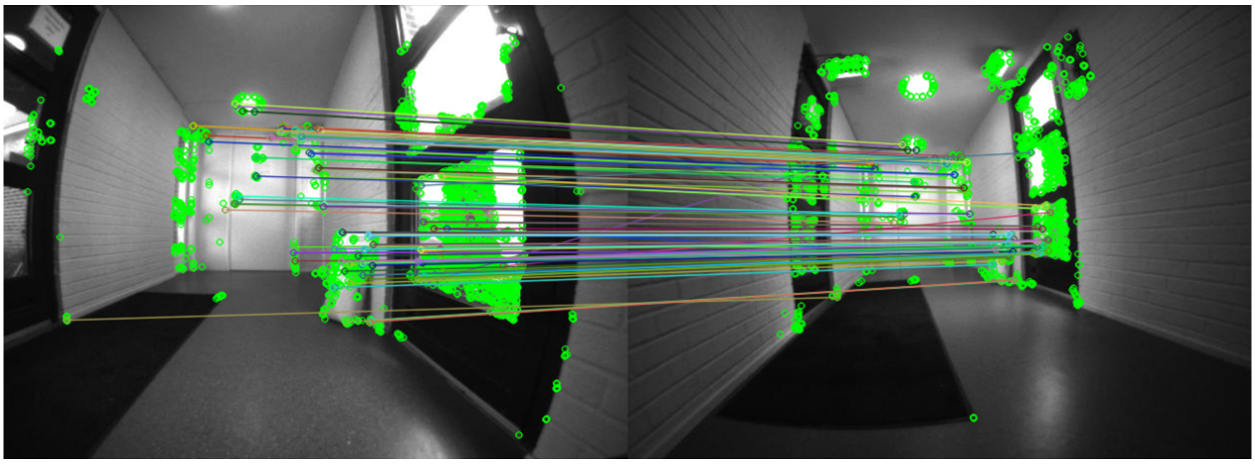


Figure 5.11: Effect of Lowe's ratio test when the images are of the same scene

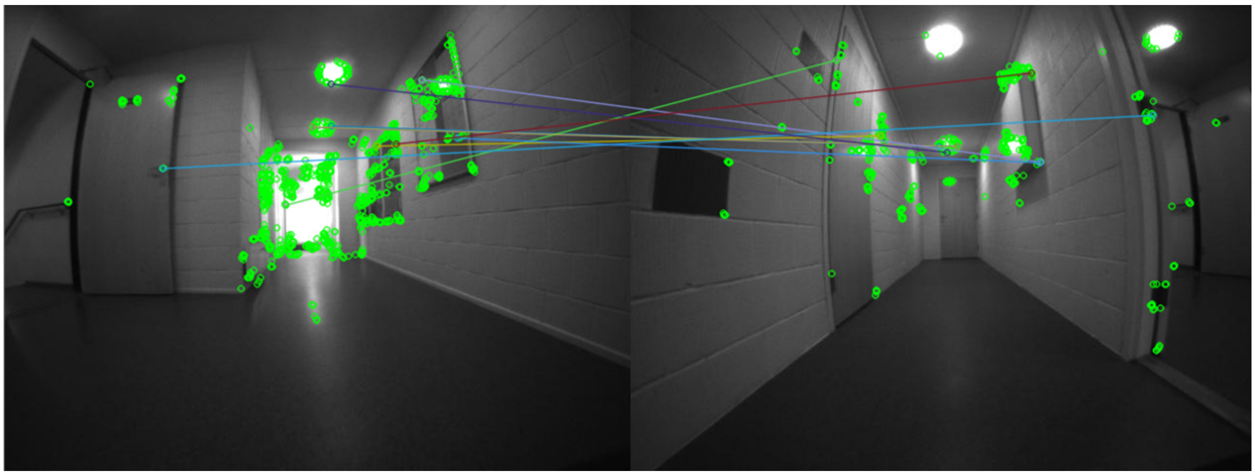
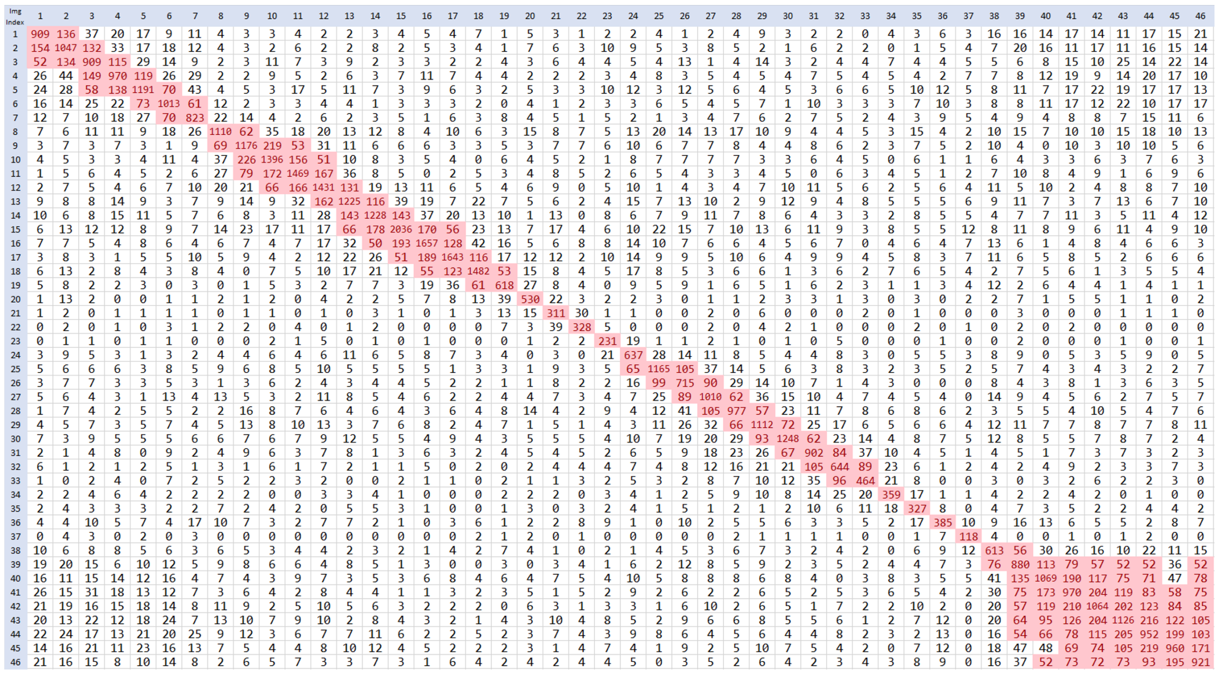


Figure 5.12: Effect of Lowe's ratio test when the images are of different scenes

Lowe's ratio test can't eliminate all wrong data associations. The number of matched features is higher when the images are of the same scene. It was necessary to find a threshold for how many matches were required for two images to be considered to be of the same scene. A method for finding this threshold is to attempt to match a query node with every node in the graph and count the number of matches it has with each node. This is similar to searching for loop closures, except here the query node is attempted matched with the entire graph, including itself. Some ground truth is needed about the graph, namely which nodes are observing the same scene. Doing this for every node in the graph yields table 5.6. A row corresponds to a query node, and each value in that row is the number of matched features that query node had with the node given by the column index. As expected, a query node will have a high match count with itself, and thus there are high values along the diagonal of the table. A query node will have a high match count with a cluster of nodes if those nodes are observing the same scene. Table 5.6 does not use a dataset where there are detectable loop closures, and thus most of the high values are grouped close to the diagonal.

Train node index

Query node index



Img Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
1	909	136	37	20	17	9	11	4	3	3	4	2	2	3	4	5	4	7	1	5	3	1	2	2	4	1	2	4	9	3	2	2	0	4	3	6	3	16	16	14	17	14	11	17	15	21
2	154	1047	132	33	17	18	12	4	3	2	6	2	2	8	2	5	3	4	1	7	6	3	10	9	5	3	8	5	2	1	6	2	2	0	1	5	4	7	20	16	11	17	11	16	15	14
3	52	134	909	115	29	14	9	2	3	11	7	3	9	2	3	3	2	2	4	3	6	4	4	5	4	13	1	4	14	3	2	4	4	7	4	4	5	5	6	8	15	10	25	14	22	14
4	26	44	149	970	119	26	29	2	2	9	5	2	6	3	7	11	7	4	4	2	2	2	3	4	8	3	5	4	5	4	7	5	4	5	4	2	7	7	8	12	19	9	14	20	17	10
5	24	28	58	138	1191	70	43	4	5	3	17	5	11	7	3	9	6	3	2	5	3	3	10	12	3	12	5	6	4	5	3	6	6	5	10	12	5	8	11	7	17	22	19	17	17	13
6	16	14	25	22	73	1013	61	12	2	3	3	4	4	1	3	3	3	2	0	4	1	2	3	3	6	5	4	5	7	1	10	3	3	3	7	10	3	8	8	11	17	12	22	10	17	17
7	12	7	10	18	27	70	823	22	14	4	2	6	2	3	5	1	6	3	8	4	5	1	5	2	1	3	4	7	6	2	7	5	2	4	3	9	5	4	9	4	8	8	7	15	11	6
8	7	6	11	11	9	18	26	1110	62	35	18	20	13	12	8	4	10	6	3	15	8	7	5	13	20	14	13	17	10	9	4	4	5	3	15	4	2	10	15	7	10	10	15	18	10	13
9	3	7	3	7	3	1	9	69	1176	219	53	31	11	6	6	6	3	3	5	3	7	7	6	10	6	7	7	8	4	4	8	6	2	3	7	5	2	10	4	0	10	3	10	10	5	6
10	4	5	3	3	4	11	4	37	226	1396	156	51	10	8	3	5	4	0	6	4	5	2	1	8	7	7	7	7	3	3	6	4	5	0	6	1	1	6	4	3	3	6	3	7	6	3
11	1	5	6	4	5	2	6	27	79	172	1469	167	36	8	5	0	2	5	3	4	8	5	2	6	5	4	3	3	4	5	0	6	3	4	5	1	2	7	10	8	4	9	1	6	9	6
12	2	7	5	4	6	7	10	20	21	66	166	1431	131	19	13	11	6	5	4	6	9	0	5	10	1	4	3	4	7	10	11	5	6	2	5	6	4	11	5	10	2	4	8	8	7	10
13	9	8	8	14	9	3	7	9	14	9	32	162	1225	116	39	19	7	22	7	5	6	2	4	15	7	13	10	2	9	12	9	4	8	5	5	5	6	9	11	7	3	7	13	6	7	10
14	10	6	8	15	11	5	7	6	8	3	11	28	143	1228	143	37	20	13	10	1	13	0	8	6	7	9	11	7	8	6	4	3	3	2	8	5	5	4	7	7	11	3	5	11	4	12
15	6	13	12	12	8	9	7	14	23	17	11	17	66	178	2036	170	56	23	13	7	17	4	6	10	22	15	7	10	13	6	11	9	3	8	5	5	12	8	11	8	9	6	11	4	9	10
16	7	7	5	4	8	6	4	6	7	4	7	17	32	50	193	1657	128	42	16	5	6	8	8	14	10	7	6	6	4	5	6	7	0	4	6	4	7	13	6	1	4	8	4	6	6	3
17	3	8	3	1	5	5	10	5	9	4	2	12	22	26	51	189	1643	116	17	12	12	2	10	14	9	9	5	10	6	4	9	9	4	5	8	3	7	11	6	5	8	5	2	6	6	6
18	6	13	2	8	4	3	8	4	0	7	5	10	17	21	12	55	123	1482	53	15	8	4	5	17	8	5	3	6	6	1	3	6	2	7	6	5	4	2	7	5	6	1	3	5	5	4
19	5	8	2	2	3	0	3	0	1	5	3	2	7	7	3	19	36	61	618	27	8	4	0	9	5	9	1	6	5	1	6	2	3	1	1	3	4	12	2	6	4	4	1	4	1	1
20	1	13	2	0	0	1	1	2	1	2	0	4	2	2	5	7	8	13	39	530	22	3	2	2	3	0	1	1	2	3	3	1	3	0	3	0	2	2	7	1	5	5	1	1	0	2
21	1	2	0	1	1	1	1	0	1	1	0	1	0	3	1	0	1	3	13	15	311	30	1	1	0	0	2	0	6	0	0	0	2	0	1	0	0	0	3	0	0	0	1	1	1	0
22	0	2	0	1	0	3	1	2	2	0	4	0	1	2	0	0	0	0	7	3	39	328	5	0	0	0	2	0	4	2	1	0	0	0	2	0	1	0	2	0	2	0	0	0	0	0
23	0	1	1	0	1	1	0	0	0	2	1	5	0	1	0	1	0	0	0	1	2	2	231	19	1	1	2	1	0	1	0	5	0	0	0	1	0	0	2	0	0	0	1	0	0	1
24	3	9	5	3	1	3	2	4	4	6	4	6	11	6	5	8	7	3	4	0	3	0	21	637	28	14	11	8	5	4	4	8	3	0	5	5	3	8	9	0	5	3	5	9	0	5
25	5	6	6	6	3	8	5	9	6	8	5	10	5	5	5	5	1	3	3	1	9	3	5	65	1165	105	37	14	5	6	3	8	3	2	3	5	2	5	7	4	3	4	3	2	2	7
26	3	7	7	3	3	5	3	1	3	6	2	4	3	4	4	5	2	2	1	1	8	2	2	16	99	715	90	29	14	10	7	1	4	3	0	0	0	8	4	3	8	1	3	6	3	5
27	5	6	4	3	1	13	4	13	5	3	2	11	8	5	4	6	2	2	4	4	7	3	4	7	25	89	1010	62	36	15	10	4	7	4	5	4	0	14	9	4	5	6	2	7	5	7
28	1	7	4	2	5	5	2	2	16	8	7	6	4	6	4	3	6	4	8	14	4	2	9	4	12	41	105	977	57	23	11	7	8	6	8	6	2	3	5	5	4	10	5	4	7	6
29	4	5	7	3	5	7	4	5	13	8	10	13	3	7	6	8	2	4	7	1	5	1	4	3	11	26	32	66	1112	72	25	17	6	5	6	6	4	12	11	7	7	8	7	7	8	11
30	7	3	9	5	5	5	6	6	7	6	7	9	12	5	5	4	9	4	3	5	5	5	4	10	7	19	20	29	93	1248	62	23	14	4	8	7	5	12	8	5	5	7	8	7	2	4
31	2	1	4	8	0	9	2	4	9	6	3	7	8	1	3	6	3	2	4	5	4	5	2	6	5	9	18	23	26	67	902	84	37	10	4	5	1	4	5	1	7	3	7	3	2	3
32	6	1	2	1	2	5	1	3	1	6	1	7	2	1	1	5	0	2	0	2	4	4	4	7	4	8	12	16	21	21	105	644	89	23	6	1	2	4	2	4	9	2	3	3	7	3
33	1	0	2	4	0	7	2	5	2	2	3	2	0	0	2	1	1	0	2	1	1	3	2	5	3	2	8	7	10	12	35	96	464	21	8	0	0	3	0	3	2	6	2	2	3	0
34	2	2	4	6	4	2	2	2	2	0	0	3	3	4	1	0	0	0	2	2	2	0	3	4	1	2	5	9	10	8	14	25	20	359	17	1	1	4	2	2	4	2	0	1	0	0
35	2	4	3	3	3	2	2	7	2	4	2	0	5	5	3	1	0	0	1	3	0	3	2	4	1	5	1	2	1	2	10	6	11	18	327	8	0	4	7	3	5	2	2	4	4	2
36	4	4	10	5	7	4	17	10	7	3	2	7	7	2	1	0	3	6	1	2	2	8	9	1	0	10	2	5	5	6	3	3	5	2	17	385	10	9	16	13	6	5	5	2	8	7
37	0	4	3	0	2	0	3	0	0	0	0	0	0	0	0	0	0	2	1	2	0	1	0	0	0	0	0	2	1	1	1	1	0	0	1	7	118	4	0	0	1	0	1	2	0	0
38	10	6	8	8	5	6	3	6	5	3	4	4	2	3	2	1	4	2	7	4	1	0	2	1	4	5	3	6	7	3	2	4	2	0	6	9	12	613	56	30	26	16	10	22	11	15
39	19	20	15	6	10	12	5	9	8	6	6	4	8	5	1	3	0	0	3	0	3	4	1	6	2	12	8	5	9	2	3	5	2	4	4	7	3	76	880	113	79	57	52	52	36	52
40	16	11	15	14	12	16	4	7	4	3	9	7	3	5	3	6	8	4	6	4	7	5	4	10	5	8	8	8	6	8	4	0	3	8	3	5	5	41	135	1069	190	117	75	71	47	78
41	26	15	31	18	13	12	7	3	6	4	2	8	4	4	1	1	3	2	3	5	1	5	2	9	2	6	2	2	6	5	2	5	3	6	5	4	2	30	75	173	970	204	119	83	58	75
42	21	19	16	15	18	14	8	11	9	2	5	10	5	6	3	2	2	2	0	6	3	1	3	3	1	6	10	2	6	5	1	7	2	2	10	2	0	20	57	119	210	1064	202	123	84	85
43	20	13	22	12	18	24	7	13	10	7	9	10	2	8	4	3	2	1	4	3	10	4	8	5	2	9	6	6	8	5	5	6	1	2	7	12	0	20	64	95	126	204	1126	216	122	105
44	22	24	17	13	21	20	25	9	12	3	6	7	7	11	6	2	2	5	2	3	7	4	3	9	8	6	4	5	6	4	4	8	2	3	2	13	0	16	54	66	78	115	205	952	199	103
45	14	16	21	11	23	16	13	7	5	4	4	8	10	12	4	5	2	2	2	3	1	4	7	4	1	9	2	5	10	7	5	4	2	0	7	12	0	18	47	48	69	74	105	219	960	171
46	21	16	15	8	10	14	8	2	6	5	7	3	3	7	3	1	6	4	2	4	2	4	4	5	0	3	5	2	6	4	2	3	4	3	8	9	0	16	37	52	73	72	73	93	195	921

Table 5.6: The number of matched features each query node has with every other node in the graph

The highlighted values in table 5.6 are those that are ≥ 50, which was found to be a reasonable threshold. Table 5.7 shows the same procedure done to a bigger graph. The table only includes the highlighting because the numbers would not be readable. If a query node passes the threshold with several consecutive nodes it has matched with a cluster of nodes (if there are previously detected loop closures, a cluster can also contain non-consecutive nodes as described in section 4.1.7). The clusters have been highlighted in a region of the table. Note that the top right and bottom left corners are highlighted. This is because in this dataset the Loomo starts and stops in approximately the same pose.

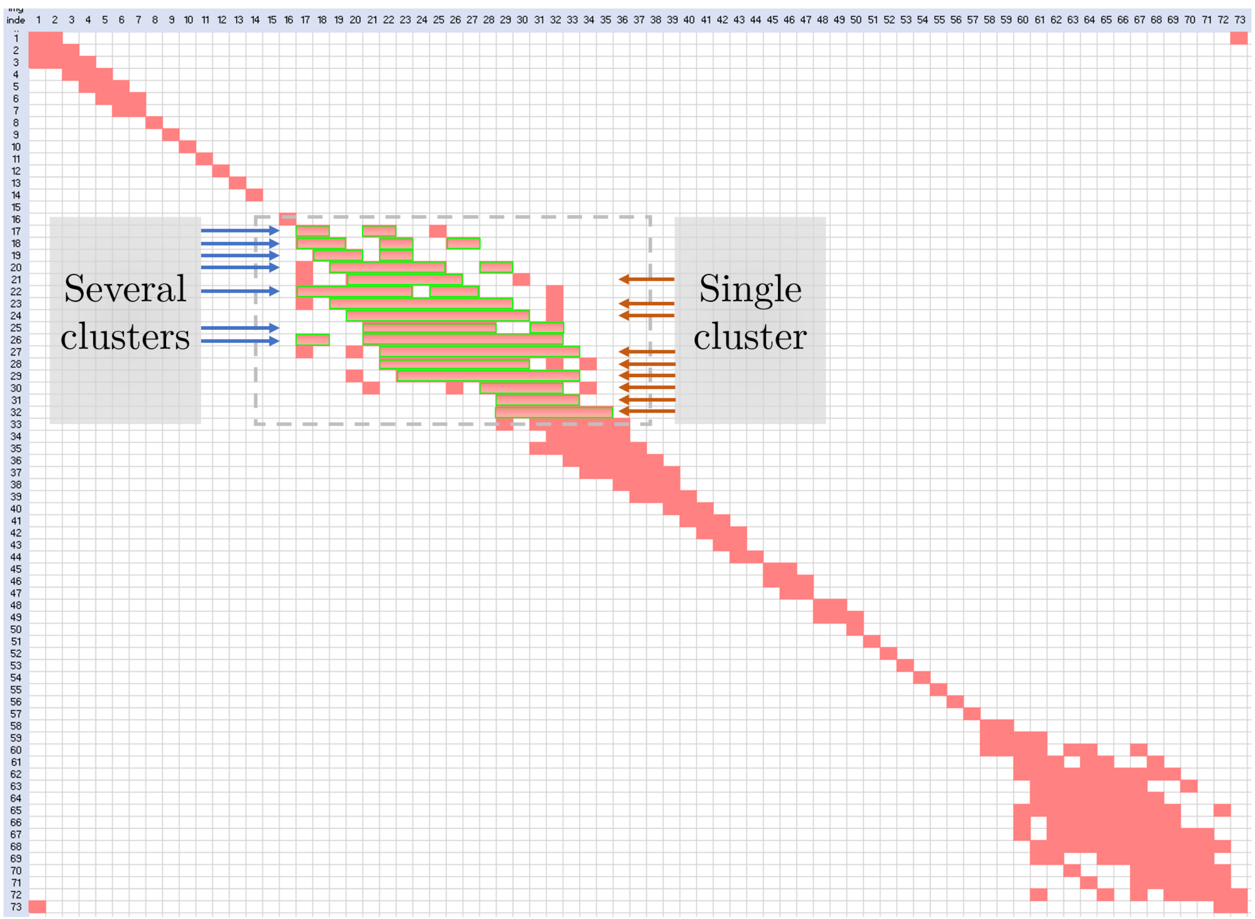


Table 5.7: Highlighting of nodes that passed the matching criteria of 50. The clusters have also been highlighted in the region marked by the dotted rectangle.

Chapter 6

Discussion

6.1 SLAM

A pose-graph solution to SLAM was developed from scratch. The back-end works successfully, but there are some remarks to be made about the front-end, with some points for improvements.

When matched features are used to extrapolate the relative poses between nodes, the translation and rotation are projected to the 2D odometry frame in an inaccurate manner. The front-end should be able to take into consideration the transformation between the odometry frame and camera frame. This information is available through the Loomo SDK. An improvement is then to make the Loomo turn its head back and forth during operation in a scanning motion to increase the FoV of the sensors.

Another improvement is to incorporate more sensors into the SLAM algorithm. One of the natural choices would be to incorporate the depth camera. One way to build an occupancy grid map from a pose graph is to build a local grid map for each node, and have these local maps move with the nodes as the graph is optimized. The local maps can be built using the depth camera. The local maps will probably overlap, so the state estimation can be further improved by optimizing the overlap. The depth camera returns a 3D point cloud, so the Iterative Closest Point (ICP) can be applied to this data.

6.2 Machine Learning

The neural network is successfully designed and implemented on the Loomo device. Deliberate evaluations have lead to good machine learning results, with some discussable remarks:

The neural network validation was conducted by images looking similar to the ones used in the training sets, making it hard to assure the reliability of the validation. However, the real-time performance handles realistic situations with good precision. Remaining arenas could test different lightning conditions or higher velocities. Quick inference time would probably taken care of velocity issues, but such task is hard to achieve without a GPU. According to the Loomo processor (See specifications at 2.1), the final inference time of 0.92 seconds is acceptable. The outputs of this neural network are reliable.

6.3 Exploration Simulation

The simulation was build as a platform for testing different strategies for bots to collaboratively explore an unknown environment. This potential has not been explored fully in this thesis. The most interesting methods to pursue is Dynamic path planning, like D* Lite. Or multi-robot path planning.

In the presented results, there is a clear trend for diminishing returns on the time efficiency vs the amount of bots used. This is because the bots often mass up in rooms, and gets in each other's way as number of bots increase. It would be highly interesting to compare these results if methods for multi-robot path planning were used.

Chapter 7

Further Work

Thoughts and recommendations for further development related to this thesis.

These are the tasks that needs to be performed for the system to function as stated in the original thesis goals. The SLAM, exploration and pathfinding algorithms needs to be implemented on the Loomo robots. The ROS-master needs to be developed. It is responsible for the SLAM back-end, and combining the graphs from the Loomos and redistributing the optimized graph. The university provided a Jetson Xavier for this purpose, but we never got to a state where development of the ROS-master began. The robots also need to communicate with the ROS-master. This was partially completed in the early phase of the thesis, but the communication needs to be expanded further to handle all of the information required for the systems and Loomos to work together.

Further work is also needed to utilize the Loomo recognition system. It can be used as an observation to improve state estimation. Also, detecting other Loomos helps filtering out the parts of the observations that are not part of the static environment.

Chapter 8

Conclusion

This thesis covered the process of developing a SLAM algorithm using a pose graph approach, and the training of a Convolutional Neural Network (CNN) for detecting Loomo robots. Further, a simulation environment was developed for experimenting with collaborative exploration.

The SLAM algorithm was developed offline using recorded datasets from a Loomo. The SLAM algorithm is an implementation of a pose graph. Each node in the graph represents a pose estimation, and the edges are the constraints for the nodes. A least squares solver was implemented to optimize the graph based on the observations and inputs. The least squares solver constitutes the SLAM back-end. The front end processes the inputs and observations and constructs the graph. The Loomo's wheel encoders provides the input, and the fisheye camera provides the observations. The ORB feature detector is used to extract image features. Matched features between nodes is used to extrapolate relative transformations between nodes. The front-end has room for improvements, either by incorporating additional sensors that are available on the Loomo, or by improving how the matched image features are used.

The CNN is designed and trained using the YOLO framework, specifically YOLOv3-tiny. A fully trained CNN is implemented on the Loomo, interconnected with OpenCV. The CNN was able to run on the Loomo, and reliably detect other Loomos in its Field of View. Validations refers to mAP of 82% along with an inference time of 0.92 Frames Per Second.

The simulation is built as a platform to benchmark different exploration strategies and pathfinding algorithms. To give the bots an environment to explore, a map of a random unknown building is generated. The building consists of rooms and corridors. The layout of the building can be adjusted by changing the size of the rooms and doors, and the probability of spawning doors and corridors. It is also possible to edit the map manually. The bots use ray casting to simulate the real-life behavior of distance sensors. The swarm intelligence rules used to control the bots is inspired by the BOID algorithm presented in the 'Flocks, Herds and Schools: A Distributed Behavioral Model' [8] paper. A* is used for pathfinding for bots to reach the target position given by the exploration algoritm. The results of multiple simulations show that with the current exploration and pathfinding algorithm there is a diminishing return on time efficiency of using more bots. The optimal amount seems to be 4-5 bots.

Bibliography

[1] D. Zou and P. Tan. 'CoSLAM: Collaborative Visual SLAM in Dynamic Environments'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.2 (2013), pp. 354–366.

[2] Lingkang Zhang. 'Orbiting a Moving Target with Multi-Robot Collaborative Visual SLAM'. In: July 2015.

[3] L. Riazuelo, Javier Civera and J.M.M. Montiel. 'C2TAM: A Cloud framework for cooperative tracking and mapping'. In: *Robotics and Autonomous Systems* 62.4 (2014), pp. 401–413. ISSN: 0921-8890. DOI: `https://doi.org/10.1016/j.robot.2013.11.007`. URL: `http://www.sciencedirect.com/science/article/pii/S0921889013002248`.

[4] Patrik Schmuck and Margarita Chli. 'CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams'. In: *Journal of Field Robotics* 36 (Dec. 2018). DOI: `10.1002/rob.21854`.

[5] C. Forster et al. 'Collaborative monocular SLAM with multiple Micro Aerial Vehicles'. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 3962–3970.

[6] T. Luo et al. 'Multi-Agent Collaborative Exploration through Graph-based Deep Reinforcement Learning'. In: *2019 IEEE International Conference on Agents (ICA)*. 2019, pp. 2–7.

[7] M. Ossenkopf et al. 'Long-Horizon Active SLAM system for multi-agent coordinated exploration'. In: *2019 European Conference on Mobile Robots (ECMR)*. 2019, pp. 1–6.

[8] Craig W. Reynolds. 'Flocks, Herds and Schools: A Distributed Behavioral Model'. In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 25–34. ISSN: 0097-8930. DOI: `10.1145/37402.37406`. URL: `https://doi.org/10.1145/37402.37406`.

[9] *About ROS*. [Online; accessed 08-Jun-2020]. Open Robotics. URL: `https://www.ros.org/about-ros/`.

[10] *Loomo user manual*. Rev. 1.0. Online; downloaded 13-January-2020. Segway. URL: `https://store.segway.com/pub/media/wysiwyg/warranty/LOOMO-user-manual.pdf`.

[11] *Datasheet for the Intel RealSense Camera ZR300*. CDI/IBP: 565287. Rev. 1.0. Online; downloaded 13-January-2020. Intel. Jan. 2017. URL: `https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/ZR300-Product-Datasheet-Public.pdf`.

[12] Cyrill Stachniss, John J. Leonard and Sebastian Thrun. 'Simultaneous Localization and Mapping'. In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Cham: Springer International Publishing, 2016, pp. 1153–1176. ISBN: 978-3-319-32552-1. DOI: 10.1007/978-3-319-32552-1_46. URL: https://doi.org/10.1007/978-3-319-32552-1_46.

[13] G. Grisetti et al. 'Hierarchical optimization on manifolds for online 2D and 3D mapping'. In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 273–278.

[14] M. Kaess, A. Ranganathan and F. Dellaert. 'iSAM: Incremental Smoothing and Mapping'. In: *IEEE Transactions on Robotics* 24.6 (2008), pp. 1365–1378.

[15] Michael Kaess et al. 'iSAM2: Incremental Smoothing and Mapping Using the Bayes Tree'. In: *International Journal of Robotic Research - IJRR* 31 (May 2012), pp. 216–235. DOI: 10.1177/0278364911430419.

[16] D. Meyer-Delius et al. 'Temporary maps for robust localization in semi-static environments'. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010, pp. 5750–5755.

[17] Dirk Hähnel, Dirk Schulz and Wolfram Burgard. 'Mobile robot mapping in populated environments'. In: *Advanced Robotics* 17 (Jan. 2003), pp. 579–597. DOI: 10.1163/156855303769156965.

[18] Chieh-Chih Wang, C. Thorpe and S. Thrun. 'Online simultaneous localization and mapping with detection and tracking of moving objects: Theory and results from a ground vehicle in crowded urhan areas'. In: vol. 1. Oct. 2003, 842–849 vol.1. ISBN: 0-7803-7736-2. DOI: 10.1109/ROBOT.2003.1241698.

[19] Tom Duckett. 'Dynamic maps for long-term operation of mobile service robots'. In: (Jan. 2005).

[20] Cyrill Stachniss and Wolfram Burgard. 'Mobile Robot Mapping and Localization in Non-Static Environments.' In: Jan. 2005, pp. 1324–1329.

[21] Dirk Hhnel et al. 'Map Building with Mobile Robots in Dynamic Environments'. In: *Proceedings - IEEE International Conference on Robotics and Automation* (Nov. 2002). DOI: 10.1109/ROBOT.2003.1241816.

[22] E. Rublee et al. 'ORB: An efficient alternative to SIFT or SURF'. In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571.

[23] Shinya Sumikura, Mikiya Shibuya and Ken Sakurada. 'OpenVSLAM: A Versatile Visual SLAM Framework'. In: *Proceedings of the 27th ACM International Conference on Multimedia*. MM '19. Nice, France: ACM, 2019, pp. 2292–2295. ISBN: 978-1-4503-6889-6. DOI: 10.1145/3343031.3350539. URL: http://doi.acm.org/10.1145/3343031.3350539.

[24] Raúl Mur-Artal and Juan D. Tardós. 'ORB-SLAM: A Versatile and Accurate Monocular SLAM System'. In: *IEEE Transactions on Robotics* 31.5 (2015), pp. 1147–1163. DOI: 10.1109/TRO.2015.2463671.

[25] Raúl Mur-Artal and Juan D. Tardós. 'ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras'. In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: `10.1109/TRO.2017.2705103`.

[26] D. G. Lowe. 'Object recognition from local scale-invariant features'. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision.* Vol. 2. 1999, 1150–1157 vol.2.

[27] Herbert Bay et al. 'Speeded-Up Robust Features (SURF)'. In: *Computer Vision and Image Understanding* 110.3 (2008). Similarity Matching in Computer Vision and Multimedia, pp. 346–359. ISSN: 1077-3142. DOI: `https://doi.org/10.1016/j.cviu.2007.09.014`. URL: `http://www.sciencedirect.com/science/article/pii/S1077314207001555`.

[28] W. Förstner and E. Gülch. 'A Fast Operator for Detection and Precise Location of Distinct Points, Corners and Centres of Circular Features'. In: *ISPRS Intercommission Workshop.* Jan. 1987.

[29] Hans Moravec. *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover.* Tech. rep. CMU-RI-TR-80-03. Pittsburgh, PA: Carnegie Mellon University, Sept. 1980.

[30] C. Harris and M. Stephens. 'A combined corner and edge detector'. In: *Alvey Vision Conference* (1998), pp. 147–151.

[31] Jianbo Shi and Tomasi. 'Good features to track'. In: *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition.* 1994, pp. 593–600.

[32] Edward Rosten and Tom Drummond. 'Machine Learning for High-Speed Corner Detection'. In: *Computer Vision – ECCV 2006.* Ed. by Aleš Leonardis, Horst Bischof and Axel Pinz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443. ISBN: 978-3-540-33833-8.

[33] Wikipedia contributors. *Midpoint circle algorithm — Wikipedia, The Free Encyclopedia.* [Online; accessed 2-May-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Midpoint_circle_algorithm&oldid=952371625`.

[34] Michael Calonder et al. 'BRIEF: Binary Robust Independent Elementary Features'. In: *Computer Vision – ECCV 2010.* Ed. by Kostas Daniilidis, Petros Maragos and Nikos Paragios. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792. ISBN: 978-3-642-15561-1.

[35] Mathworks. *Introduction to Machine Learning.* `https://www.mathworks.com/content/dam/mathworks/ebook/gated/machine-Learning-ebook.pdf`. Online; accessed 05-May-2020. 2020.

[36] Sergios Sundaram SureshNarasimhan SundararajanRamasamy Savitha. *Supervised Learning with Complex-valued Neural Networks.* Springer, Berlin, Heidelberg, 2013.

[37] Mathworks. *What is Deep Learning.* `https://se.mathworks.com/discovery/deep-learning.html?s_tid=srchtitle`. Online; accessed 06-May-2020. 2020.

[38] *Neural Networks and Deep Learning.* Determination Press, 2015.

[39] Taweh Beysolow II. *Introduction to Deep Learning Using R. A Step-by-Step Guide to Learning and Implementing Deep Learning Models Using Re.* Apress, Berkeley, CA, 2017.

[40] Sik-Ho Tsang. *Review: ResNet — Winner of ILSVRC 2015 (Image Classification, Localization, Detection).* Online; accessed 15-May-2020. 2018. URL: `https://towardsdatascience.com/review-resnet-winner-of-ilsvrc-2015-image-classification-localization-detection-e39402bfa5d8`.

[41] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement.* 2018. arXiv: `1804.02767 [cs.CV]`.

[42] Ross Girshick Joseph Redmon Santosh Divvala and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.* 2016. arXiv: `1506.02640 [cs.CV]`.

[43] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation.* 2013. arXiv: `1311.2524 [cs.CV]`.

[44] Ross Girshick. *Fast R-CNN.* 2015. arXiv: `1504.08083 [cs.CV]`.

[45] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* 2015. arXiv: `1506.01497 [cs.CV]`.

[46] Wei Liu et al. 'SSD: Single Shot MultiBox Detector'. In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: `10.1007/978-3-319-46448-0_2`. URL: `http://dx.doi.org/10.1007/978-3-319-46448-0_2`.

[47] Joseph Redmon and Ali Farhadi. *YOLO9000: Better, Faster, Stronger.* 2016. arXiv: `1612.08242 [cs.CV]`.

[48] Alexey Bochkovskiy, Chien-Yao Wang and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection.* 2020. arXiv: `2004.10934 [cs.CV]`.

[49] G. Bradski. 'The OpenCV Library'. In: *Dr. Dobb's Journal of Software Tools* (2000).

[50] Gaël Guennebaud, Benoît Jacob et al. *Eigen v3.* http://eigen.tuxfamily.org. 2010.

[51] *Opencsv - A simple library for reading and writing CSV in Java.* http://opencsv.sf.net/. [Online; accessed 25-May-2020]. 2020.

[52] ISO/IEC. *International Standard ISO/IEC 14882:2017(E) – Programming Language C++.* Geneva, Switzerland, 2017.

[53] Zhengyou Zhang. 'Camera Calibration'. In: *Computer Vision: A Reference Guide.* Ed. by Katsushi Ikeuchi. Boston, MA: Springer US, 2014, pp. 76–77. ISBN: 978-0-387-31439-6. DOI: `10.1007/978-0-387-31439-6_164`. URL: `https://doi.org/10.1007/978-0-387-31439-6_164`.

[54] Rohith Gandhi. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms.* Online; accessed 27-Feb-2020. 2018. URL: `https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e`.

[55] Pete Warden. *How many images do you need to train a neural network?)* Online; accessed 04-Jun-2020. 2017. URL: `https://petewarden.com/2017/12/14/how-many-images-do-you-need-to-train-a-neural-network/`.

[56] Alexander Jung Revision. *Overview of Augmenters.* Online; accessed 22-Apr-2020. 2020. URL: `https://imgaug.readthedocs.io/en/latest/source/overview_of_augmenters.html`.

Appendix A

Source Code

Exploration Simulation

Repository available at: `https://github.com/Stedd/swarmSimulation`
Branch: master
Commit: 4ccf697157ed12720c741b72354ada6f7ed44809

Machine learning

Repository at: `https://github.com/wiggleif/MAS500`
Branch: master
Commit: fe1adeb675cbb9c64728ebb913a310cf85a18f30

Augmentation

Script: ”AugmentedSet1.py“
`https://github.com/wiggleif/MAS500/blob/master/AugmentedSet1.py`

Matlab to Darknet Conversion

Script: mat2darkOri.m
`https://github.com/wiggleif/MAS500/blob/master/mat2darkOri.m`

Verify Bounding Boxes

Script: ”verifybboxes.m“
`https://github.com/wiggleif/MAS500/blob/master/verifybboxes.m`

Cfg-File

Script: ”yolov3-tiny-custom.cfg“

`https://github.com/wiggleif/MAS500/blob/master/yolov3-tiny-custom.cfg`

Generate Training Data

Script: ”generate_train.py“

`https://github.com/wiggleif/MAS500/blob/master/generate_train.py`

Pretrained Weights File

Script: ”yolov3-tiny.conv.11”

`https://github.com/wiggleif/MAS500/blob/master/yolov3-tiny.conv.11`

Inference

Repository available at: `https://github.com/Stedd/LoomoApp/tree/YOLO`
Branch: YOLO
Commit: 738b7f7a9dfebadf1b9818b32061d86349dee62a

SLAM

Loomo Recorder

Repository available at: `https://github.com/Jakob1-5/Loomo_Recorder`
Branch: Save_raw_vid
Commit: c8854044b9408ec59cb4b8dfed3cc7a45ec2c248

SLAM implementation

Repository available at: `https://github.com/Jakob1-5/LoomoSLAM`
Branch: master
Commit: e907a11b1c01f426ccbc3884ed6175ec1d113760

RealSense Camera App

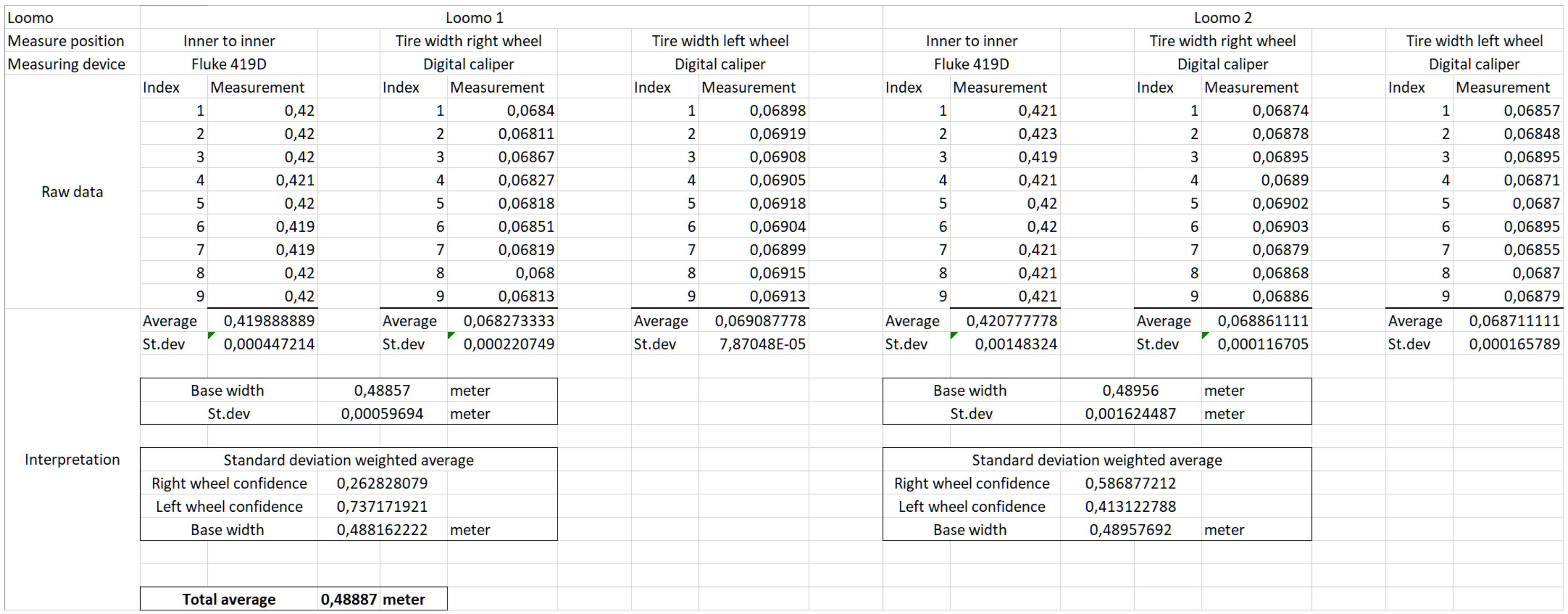
Repository available at: `https://github.com/Jakob1-5/RealSensecamera`

Branch: master

Commit: a51dceb772a23d20ef6b957222803aeea424ca17

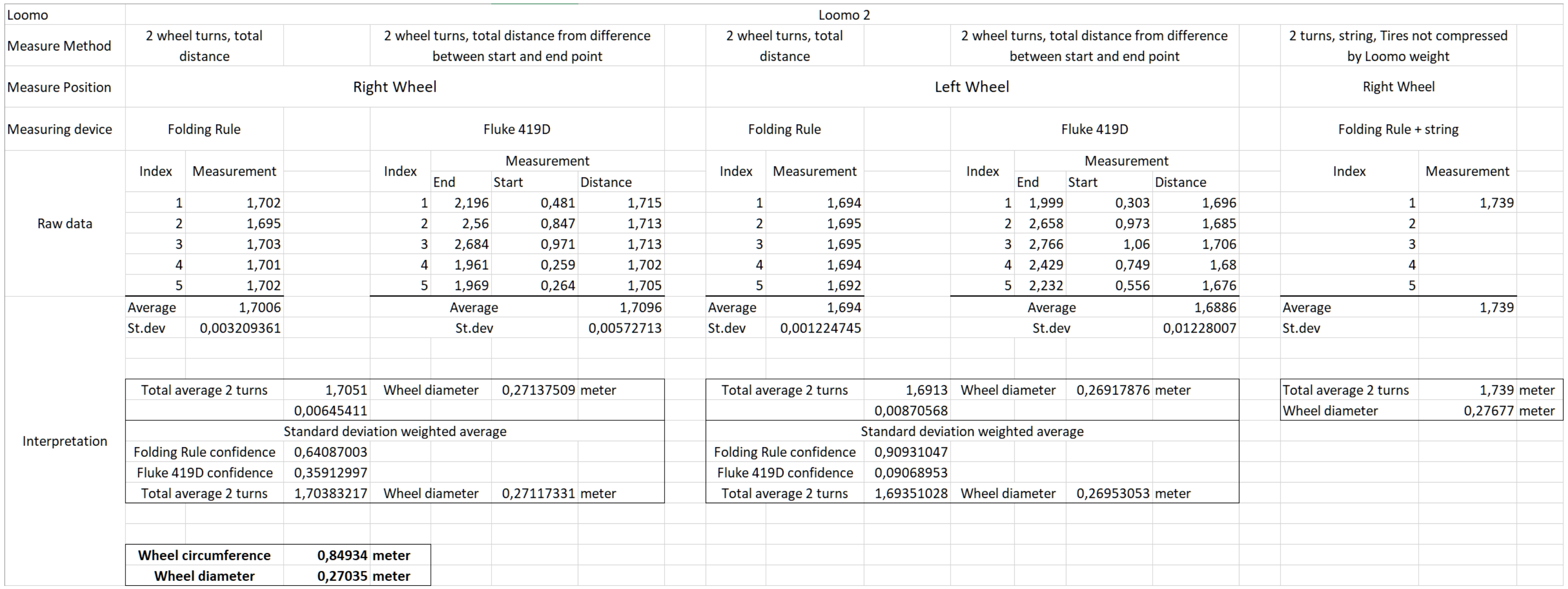
Appendix B

Sensor Calibration



Loomo	Loomo 1						Loomo 2					
Measure position	Inner to inner		Tire width right wheel		Tire width left wheel		Inner to inner		Tire width right wheel		Tire width left wheel	
Measuring device	Fluke 419D		Digital caliper		Digital caliper		Fluke 419D		Digital caliper		Digital caliper	
Raw data	Index	Measurement	Index	Measurement	Index	Measurement	Index	Measurement	Index	Measurement	Index	Measurement
	1	0,42	1	0,0684	1	0,06898	1	0,421	1	0,06874	1	0,06857
	2	0,42	2	0,06811	2	0,06919	2	0,423	2	0,06878	2	0,06848
	3	0,42	3	0,06867	3	0,06908	3	0,419	3	0,06895	3	0,06895
	4	0,421	4	0,06827	4	0,06905	4	0,421	4	0,0689	4	0,06871
	5	0,42	5	0,06818	5	0,06918	5	0,42	5	0,06902	5	0,0687
	6	0,419	6	0,06851	6	0,06904	6	0,42	6	0,06903	6	0,06895
	7	0,419	7	0,06819	7	0,06899	7	0,421	7	0,06879	7	0,06855
	8	0,42	8	0,068	8	0,06915	8	0,421	8	0,06868	8	0,0687
	9	0,42	9	0,06813	9	0,06913	9	0,421	9	0,06886	9	0,06879
Interpretation	Average	0,419888889	Average	0,068273333	Average	0,069087778	Average	0,420777778	Average	0,068861111	Average	0,068711111
	St.dev	0,000447214	St.dev	0,000220749	St.dev	7,87048E-05	St.dev	0,00148324	St.dev	0,000116705	St.dev	0,000165789
	Base width		0,48857	meter			Base width		0,48956	meter		
	St.dev		0,00059694	meter			St.dev		0,001624487	meter		
	Standard deviation weighted average						Standard deviation weighted average					
	Right wheel confidence		0,262828079				Right wheel confidence		0,586877212			
	Left wheel confidence		0,737171921				Left wheel confidence		0,413122788			
	Base width		0,488162222	meter			Base width		0,48957692	meter		
	Total average		**0,48887**	**meter**								

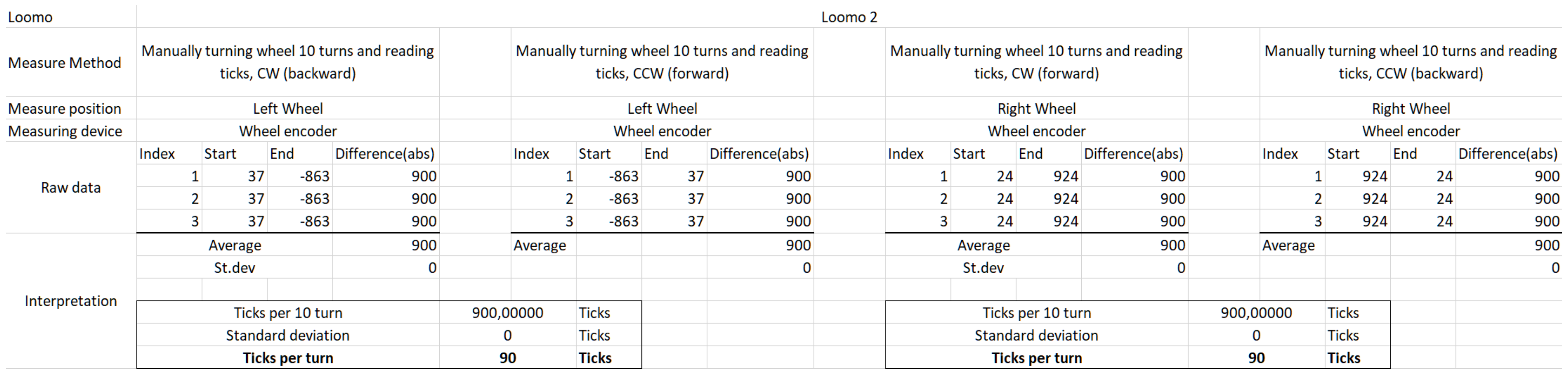
Base width calculations



Loomo								
Measure Method	2 wheel turns, total distance			2 wheel turns, total distance from difference between start and end point				
Measure Position	Right Wheel							
Measuring device	Folding Rule			Fluke 419D				
Raw data	Index	Measurement		Index	Measurement			
					End	Start	Distance	
	1	1,702		1	2,196	0,481	1,715	
	2	1,695		2	2,56	0,847	1,713	
	3	1,703		3	2,684	0,971	1,713	
	4	1,701		4	1,961	0,259	1,702	
	5	1,702		5	1,969	0,264	1,705	
Interpretation	Average	1,7006		Average			1,7096	
	St.dev	0,003209361		St.dev			0,00572713	
	Total average 2 turns		1,7051	Wheel diameter		0,27137509	meter	
			0,00645411					
	Standard deviation weighted average							
	Folding Rule confidence		0,64087003					
	Fluke 419D confidence		0,35912997					
	Total average 2 turns		1,70383217	Wheel diameter		0,27117331	meter	
	Wheel circumference		**0,84934**	**meter**				
	Wheel diameter		**0,27035**	**meter**				

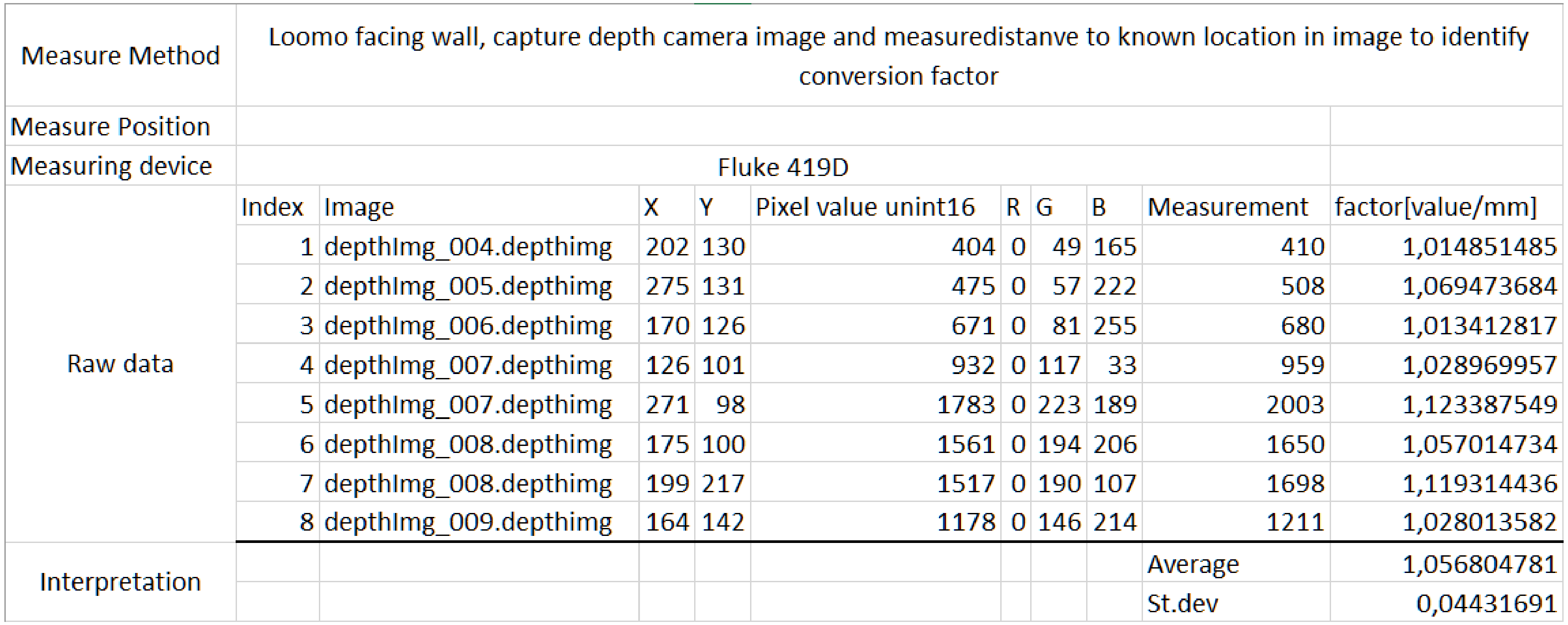
Loomo	Loomo 2											
Measure Method	2 wheel turns, total distance			2 wheel turns, total distance from difference between start and end point					2 turns, string, Tires not compressed by Loomo weight		
Measure Position	Left Wheel								Right Wheel		
Measuring device	Folding Rule			Fluke 419D					Folding Rule + string		
Raw data	Index	Measurement		Index	Measurement				Index	Measurement	
					End	Start	Distance				
	1	1,694		1	1,999	0,303	1,696		1	1,739	
	2	1,695		2	2,658	0,973	1,685		2		
	3	1,695		3	2,766	1,06	1,706		3		
	4	1,694		4	2,429	0,749	1,68		4		
	5	1,692		5	2,232	0,556	1,676		5		
Interpretation	Average	1,694		Average			1,6886		Average	1,739	
	St.dev	0,001224745		St.dev			0,01228007		St.dev		
	Total average 2 turns		1,6913	Wheel diameter		0,26917876	meter		Total average 2 turns	1,739	meter
			0,00870568						Wheel diameter	0,27677	meter
	Standard deviation weighted average										
	Folding Rule confidence		0,90931047								
	Fluke 419D confidence		0,09068953								
	Total average 2 turns		1,69351028	Wheel diameter		0,26953053	meter				

Wheel diameter calculations



Loomo	Loomo 2															
Measure Method	Manually turning wheel 10 turns and reading ticks, CW (backward)				Manually turning wheel 10 turns and reading ticks, CCW (forward)				Manually turning wheel 10 turns and reading ticks, CW (forward)				Manually turning wheel 10 turns and reading ticks, CCW (backward)			
Measure position	Left Wheel				Left Wheel				Right Wheel				Right Wheel			
Measuring device	Wheel encoder				Wheel encoder				Wheel encoder				Wheel encoder			
Raw data	Index	Start	End	Difference(abs)	Index	Start	End	Difference(abs)	Index	Start	End	Difference(abs)	Index	Start	End	Difference(abs)
	1	37	-863	900	1	-863	37	900	1	24	924	900	1	924	24	900
	2	37	-863	900	2	-863	37	900	2	24	924	900	2	924	24	900
	3	37	-863	900	3	-863	37	900	3	24	924	900	3	924	24	900
Interpretation	Average			900	Average			900	Average			900	Average			900
	St.dev			0				0	St.dev			0				0
	Ticks per 10 turn				900,00000	Ticks			Ticks per 10 turn				900,00000	Ticks		
	Standard deviation				0	Ticks			Standard deviation				0	Ticks		
	Ticks per turn				**90**	**Ticks**			**Ticks per turn**				**90**	**Ticks**		

Wheel encoder calculations



Measure Method	Loomo facing wall, capture depth camera image and measuredistanve to known location in image to identify conversion factor									
Measure Position										
Measuring device	Fluke 419D									
Raw data	Index	Image	X	Y	Pixel value unint16	R	G	B	Measurement	factor[value/mm]
	1	depthImg_004.depthimg	202	130	404	0	49	165	410	1,014851485
	2	depthImg_005.depthimg	275	131	475	0	57	222	508	1,069473684
	3	depthImg_006.depthimg	170	126	671	0	81	255	680	1,013412817
	4	depthImg_007.depthimg	126	101	932	0	117	33	959	1,028969957
	5	depthImg_007.depthimg	271	98	1783	0	223	189	2003	1,123387549
	6	depthImg_008.depthimg	175	100	1561	0	194	206	1650	1,057014734
	7	depthImg_008.depthimg	199	217	1517	0	190	107	1698	1,119314436
	8	depthImg_009.depthimg	164	142	1178	0	146	214	1211	1,028013582
Interpretation									Average	1,056804781
									St.dev	0,04431691

Depth camera calculations