December 12, 2016

MASTER THESIS

Analysis, optimization, and design of a SLAM solution for an implementation on reconfigurable hardware (FPGA) using CλaSH

Authors:
Robin Appel
Hendrik Folmer

Faculty:
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Chair:
Computer Architecture for Embedded Systems (CAES)

Exam committee:
Dr. ir. J. Kuper
Dr. ir. C.P.R. Baaij
Dr. ir. J.F. Broenink
Dr. ir. R. Wester

UNIVERSITY OF TWENTE.

Abstract

SLAM stands for simultaneous localization and mapping and is a mathematical problem with a potentially heavy computational load which is usually solved on a large computational system. Large computational systems consume a lot of energy which is not ideal for mobile robots. A field programmable gate array (FPGA) allow developers to create reconfigurable hardware architectures which could be more efficient in terms of energy compared to standard computer systems because it has parallel capabilities. The goal of this project is to develop a SLAM implementation on an FPGA to be more efficient in terms of energy and computation time. SLAM is realized without the use of external systems like GPS and uses a laser range finder (LRF) + odometry as sensor input. CλaSH is developed by the CAES-group and allows the developer to specify strongly typed mathematical descriptions, which can be translated automatically to an hardware description language (HDL). Due to the mathematical nature of the SLAM problem, CλaSH is suitable for implementation. This report describes the choices, realization, and results of a SLAM solution on an FPGA and has two main subjects: Graph-based SLAM and iterative closest point (ICP) which are discussed below.

Graph-based SLAM

Graph-based SLAM is a method to describe the SLAM problem as a graph. The main purpose of the algorithm is to correct noise that is present in sensor data. The errors that need to be corrected can be described by a linear system of equations which has the size of the number of poses that the robot has visited. The number of poses a robot visits can be large and the computational complexity of the algorithms used to solve large matrices increases quadratically. By restricting the algorithm on the amount of loop closing the complexity has been reduced to only grow linear.

Since the linear system of equations contains many zeroes, sparse matrices and vectors can be used to describe the system. However, sparse matrices in hardware create a lot of overhead and additional complexity. Using the sparse notation for sparse matrices and vectors and the ordinary notation for dense data, the best of two worlds is combined. Since the notation is different, operators that combine sparse and non-sparse data is one operation are implemented. The conjugate gradient algorithm is an iterative linear system solver that does not alter the structure of the matrix, which is an important property with sparse matrices. By using the sparse notations, the amount of memory and computations is reduced. A feasible hardware structure for the conjugate gradient algorithm is proposed which has the sequential structure which performs parallel computations.

ICP

ICP is used to find the transformation between two sensor observations (laser range scans) to determine the movement of the robot. The ICP algorithm realized in this project contains two parts; constructing correspondences without outlier rejection, and minimizing the error between

correspondences. A point-to-line metric with normal vectors are used for correspondence creation. Normal vectors are calculated using the fast inverse square root. For error minimization a least square solution is constructed and solved using QR decomposition. To construct the orthonormal matrix **Q** for QR decomposition the Gram-Schmidt method is used. Laser range scanner data and the algorithms used to determine the transformation between two observations have a regular structure and a vector ALU architecture is designed as a trade-off between resource usage (chip area) and execution time. Data is stored into blockRAMs and the control "streams" the data through the vector ALU.

The calculated transformation by the FPGA is slightly less accurate compared to a reference solution in MATLAB but has a performance increase per Joule of three orders of magnitude. Due to the regular structure of the algorithms and the realized architecture the solution of the ICP problem is suitable for an FPGA.

Contents

1 — Introduction

1.1 Context

Intelligent robots play an important role amongst mobile robots these days. An intelligent feature a robot can have is the ability to learn. A disadvantage of learning in computer terms is the amount of *computations* and *memory* it requires. A widely used set of learning algorithms for robots can be used to make a mobile robot learn about the environment. Without knowledge of an environment a robot is unable to perform tasks within this environment. How well the robot represents the environment is called *quality*. The better the quality of the representation, the easier it is for a robot to use the representation for navigation.However, this quality comes at the cost of heavier algorithms. To execute these heavy algorithms, computers with a large computational capacity can be used. These computers are *heavy* and demand a lot of *energy* and are often not suitable to mount on a robot, especially when they need to be as light as possible and run on batteries.

To reduce the energy usage and weight of the computer, smaller computers can be used. Despite that these computers use far less energy, they use the same computing structure as larger systems and are optimized for using less energy for the same amount of work. Another type of hardware that can be used to do computations such as algorithms is called a *field programmable gate array* (FPGA). An FPGA is a chip that has predefined *configurable logic blocks* that can be configured to act as digital circuits. The large advantage over other chips is the fact that the *behaviour* of an FPGA is not static, but can be changed by the user. By reconfiguring an FPGA it can act as any type of hardware that can also be created in digital chips.

The complexity of an algorithm can be described as the amount of computational power that is needed to execute the algorithm. An algorithm on a normal computer is described by *software*. Software divides the total complexity of the algorithm into small parts which are called *intructions*. Instructions are executed over time and if the complexity of software increases so increases the number of instructions which increases the execution time.

FPGAs can be configured to to execute computations in parallel which require hardware resources. *Physical boundaries* of FPGAs restrict the amount of hardware resources and thus the amount of parallelism. When computations become to complex and require more hardware resources than available fully parallel solutions are not possible. If computations do not fit in the *area* of an FPGA it becomes necessary to execute computation after each other over *time*. A developer has to make a trade-off between resource usage (chip area) and execution time.

Implementation of heavy algorithms on FPGAs can be considered as a trade-off between the use of time and area. Because FPGAs can be configured to be applications specific, the available FPGA area can be used efficiently which will result in a fast and energy efficient solution. Robot

algorithms are very suitable for such an implementation because of the following properties:

- The algorithms are commonly heavy (have large computational complexity)
- Many of the computations can potentially be done in parallel
- FPGAs have deterministic behaviour which means computation times can be guaranteed

As mentioned before, in robotics a fundamental problem is the awareness of the environment. For robots to perform complex tasks autonomous it is often required that a robot needs to navigate through an environment. For navigation a mobile robot needs to know something about the environment, for instance, if there are obstacles in the way. The robot gathers data about the environment using sensors and with this data it can create a *map* and find out its own *location* in the environment. Creating a map and finding the robot position in that map is called *SLAM*. SLAM stands for *simultaneous localization and mapping*. Mapping is a trivial process when the robot is aware of its location. However, performing both localization and mapping simultaneously introduces interesting new challenges, because the two processes are completely dependent of each other. Sensors provide information about the environment or movement of the robot but sensors always have some inaccuracy and noise which makes the SLAM process more complex. When navigating through an environment a robot must be able to avoid obstacles, this puts a timing constraint on the SLAM algorithm because if the SLAM algorithm is not fast enough the robot is not able to avoid the obstacle. Because of the simultaneous nature, inaccuracy and noise of sensors, and timing constrains SLAM is considered to be a complex mathematical problem with a heavy computational load. As mentioned before complex algorithms are often executed on systems with a high computational power which are not desirable to mount on mobile robots. A method used in robotics is to use external systems for calculating complex algorithms. In that case the robot sends its sensor data to this external system (base station). The communication is often wireless and therefore prone to bad quality signals or signal loss. If, for some reason, the communication between the base station and the robot is not available, the robot can not perform its tasks.

Because SLAM algorithms have high complexity, they often require many computational resources. FPGAs can function as dedicated hardware created with an HDL (hardware description language). Because of the lower frequencies and potentially parallel structures, it is possible to obtain a much lower overall energy usage and computation time, therefore realizing a SLAM on an FPGA can have advantages.

Hardware architectures on an FPGA are usually described with languages like VHDL or Verilog. These languages requires the developer a lot of manual work. CλaSH is developed by the CAES-group and allows the developer to specify strongly typed mathematical descriptions, which can be translated automatically to an HDL. Because of the mathematical nature of SLAM, CλaSH is suitable tool for realizing a SLAM solution on an FPGA.

1.2 Problem statement

Realizing a SLAM solution on an FPGA has potential advantages in terms of speed, computational load, and energy. To analyse these advantages a SLAM solution has to be realized on an FPGA. The problem definition can be summarized into the following:

- How can a SLAM solution be realized into a feasible hardware architecture
- Does a hardware architecture have the potential to be more efficient in terms of performance per joule compared to the commonly used computational systems

1.3 Approach and outline

SLAM can be solved in different ways and to realize one solution on an FPGA one must first determine which SLAM solutions are suitable for implementation on hardware. Because CλaSH is suitable for describing complex mathematical problems and creating hardware to solve these problems. Therefore CλaSH will be used for the implementations in this thesis. FPGAs come in different size and forms and the target FPGA used for this project is described in Appendix A. Part I, Background, contains the research on the different aspects of SLAM. Chapter 2 introduces the basic concepts, terminology and mathematics regarding robotics and SLAM. Chapter 3 describes the two main data representations used in SLAM. After these introducing chapters the project is split into two subjects; *Graph-based SLAM* and *ICP* (iterative closest point). ICP is method to extract information about the robot's position from sensor data. The main focus of the authors can also be split up in these parts. R. Appel is responsible for the Graph-based SLAM part and H. Folmer for the ICP part. Both of these subjects have the same approach which has the following structure:

- Background, contains research and explanations of the working and structure of the algorithms and concepts, Part I
- Design Space Exploration, shows and explains the different choices made, Part II
- Realisation and results, shows the realised architecture and results, Part III
- Conclusions and future work, Part IV

	Graph-based SLAM	ICP
Background	Chapter 4	Chapter 5
Design Space Exploration	Chapter 9	Chapter 10
Realisation	Section 11.1	Section 12.2
Results	Section 11.2	Section 12.3
Conclusions	Section 13.2	Section 13.3
Future work	Section 14.2	Section 14.3

Table 1.1: Structure of the report, reader can choose to read this report "vertically" or "horizontally"

Table 1.1 shows an overview of the different parts of this report. The reader can chose to read this report in two different ways. One way is to read each part of the report for both Graph-based SLAM and ICP, which means that one goes through the table "horizontally". Another way is to read every part for one subject, which means that the reader goes through the table "vertically".

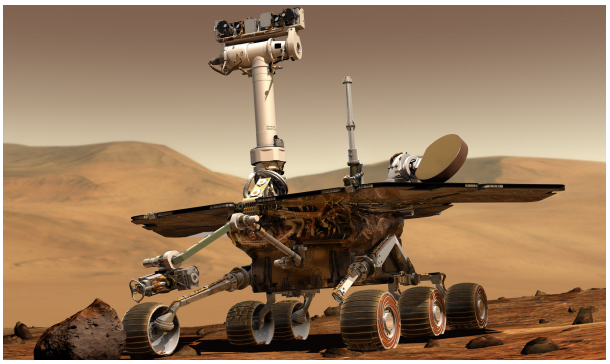
Part I

Background

2 — Robotics and SLAM

Robots are systems that perform *tasks* using *robotics*. These tasks can be various from cleaning a house(figure 2.1a) to the exploration of a planet(figure 2.1b). Robotics are science and engineering aspects that are necessary to technically enable a robot to work. Robotics consists of mechanical, electrical, and control engineering of the physical structure, the electronic hardware and control software.

(a) Picture of a Samsung robotic vacuum cleaner[56]



(b) Illustration of the NASA's Mars Exploration Rover[60]

Figure 2.1

A separation can be made between autonomous and non-autonomous robots, most of the time these have very different tasks to fulfill. Autonomous robots have the feature they do not need any additional input from a user to perform a given task. For autonomous robots it is very important that they are able to learn about their environment and react accordingly to the changes the environment undergoes. Autonomous robots often have a significantly higher complexity than non-autonomous robots, which makes autonomous robots more interesting for scientific research.

Throughout this report a driving robot in an indoor office is considered the target. Figure 2.2a shows a two-wheeled robot with encoders and a scanning laser range finder (LRF) (also known as laser range scanner) mounted on top. Encoders are sensors that measure the rotations of the wheels by counting the amount of rotational steps, encoders have limited resolution and do not take physical effects like slip into account. A laser range scanner is a sensor that creates environment data by measuring the distance to the nearest object under multiple angles using a single rotating laser range finder. Figure 2.2b shows a two-dimensional drawing of the target robot and the beams around it are laser range measurements. The robot has a position in the environment, which is also called state or pose. A pose in the two-dimensional case is expressed in the coordinates x, y and an orientation θ. The orientation is the angle the robot is facing, similar to north on compass, there is a reference orientation.

A control command is a command describing where the robot needs to go in the next timestep relative to where it is now. Therefore, the robot can be moved by giving it these control commands, which are inputs by the user or input the robot makes up itself by a planning algorithm. The control commands used in this thesis should not be confused with control theory which is the algorithm that controls the motors. Instead, the change of the *state vector* that the robot has measured after a certain movement is called the control command. The state vector is a vector that described the poses the robot has been. The control command is equal to the actual change of the pose assuming the control loop that moves the robot based on the encoders and motors is perfect. The error in this case will be the error introduced by the encoders or whatever other sensor that is used for positioning.

In many cases, autonomous robots need to be able to navigate through an environment by themselves. Without navigation a robot does not know where it needs to go and can therefore not perform tasks autonomously. Whether a robot needs to deliver a package on the other side of the street or inspect the inside of a tunnel, navigation is key.

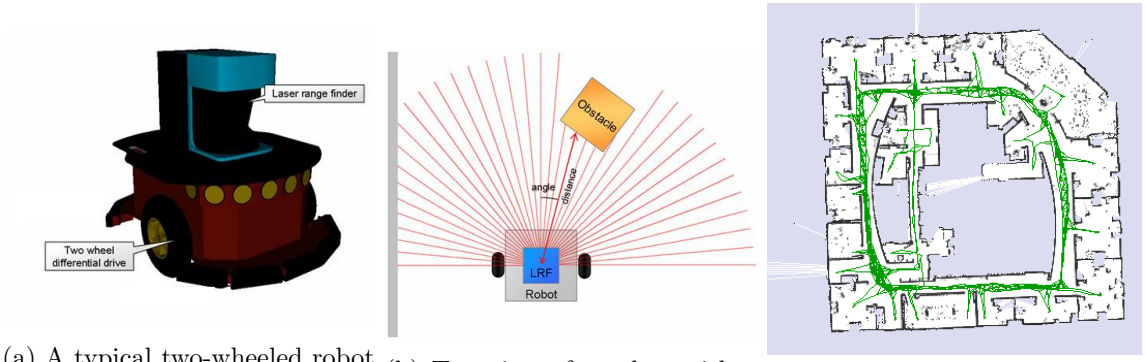
Figure 2.2c shows an example of a map with the green line representing the path the robot has driven, the map has been constructed by drawing the laser scans around the poses in the green path. Later will be shown that this is only one representation of a map and other types of representations are possible. In robotics the map is an important object, since the map describes the environment. If a robot would like to navigate from one end of the environment to the other, the map will describe where the robot is able to go safely.

Apart from the structure of the map, for navigation the robot needs to be aware of its position within this map. Localization can be a difficult task for a robot, depending on what system is behind the localization. Localization in a square building can be *undetermined* because despite the robot knows it is in a corner, it can still be in any of the four corners of the building if there is no other data that depicts the specific corner.

Whenever a robot is placed in an environment without knowing its position and without knowing the structure of the map, the task of navigation becomes even more complex. In this case the robot really needs to learn its environment by exploration and stitch the measurements that it does together to construct a good representation of the environment in the form of a map. Whenever a robot visits one place multiple times it will have multiple measurements of that part of the environment and the quality of that part of the map will increase. This feature is fundamental and will be extensively used throughout this thesis. This phenomenon is called loop closing and will in this chapter be further discussed in Section 2.2.1.

The process of creating a map of the environment using sensors and at the same time correcting the robot path using the map is called simultaneous localization and mapping (SLAM). Because the map can only be correctly constructed when knowing the pose of the robot, and the pose of the robot is based on what the map looks like, SLAM can be seen as a chicken and egg problem. The SLAM problem has however been solved by different strategies but still is an important research topic for many researchers. SLAM research focuses on alternative algorithms and filtering techniques to create robuster and faster SLAM algorithms. SLAM is considered to be a fundamental and complex mathematical problem, its complexity comes mainly from the fact that sensors are not perfect and are subject to noise and inaccuracy.
Another challenge of solving the SLAM problem is the computational complexity, because the

complexity grows over time, eventually a the system that runs the algorithm will run out of computational or storage resources.



(a) A typical two-wheeled robot with a laser-range finder (LRF) and wheel encoders

(b) Top view of a robot with a laser-range finder (LRF)

(c) Map of the environment, the green line is the path that the robot has driven

Figure 2.2: Robot illustrations, found in [2] and [5]

2.1 Mathematics of SLAM

SLAM is a mathematical problem and therefore variables are used to describe the elements within the SLAM algorithm. Each term is used to express a set of data that can later be used to alter one or more other sets of data by using algorithms. Several terms of the SLAM algorithm are used in every type of SLAM and are not solution dependent, these terms are described below.

The sets of data can be split up in two groups, the first group are the sets of data that are already available once the robot has moved to a pose, the other sets are the sets that need to be discovered by using known data within the algorithms:

Given

- The robot's controls: $u_{1:T} = u_1, u_2, u_3, ..., u_T$
- Observations from a sensor: $z_{1:T} = z_1, z_2, z_3, ..., z_T$

Wanted

- Map of the environment m
- Path (state) of the robot $s_{0:T} = s_0, s_1, s_2, ..., s_T$

A mathematical model to describe control commands and observations can be found in Appendix B. Control commands and obeservations can have different forms. In this thesis, control commands will consist of a state vector change that describes a translation and a rotation $(x, y, \ and \ \theta)$ and the observations will consist of laser scans described by angles and distances. Due to inaccuracy and noise in sensors data from the controls (u_t) and observations (z_t), SLAM algorithms are probabilistic problem [58][55][34]. In SLAM, a probabilistic approach means that data has a probability of the data being correct. The higher the probability, the smaller the maximum error taken into calculations. Noise sensors introduce is often non-Gaussian, therefore approximation techniques are used to approach the sensor noise as Gaussian. A Gaussian

is a probabilistic distribution for which many of analysis techniques are known. A probability distribution is stated as follows:

$$p(wanted|given) \tag{2.1}$$

Which means there is a certain distribution that describes a wanted set by a given set. In this case the probability function of the state vector and the map can be found by given input of control commands and observations. This is called the full SLAM problem. The following expression is a mathematical description of the full SLAM problem:

$$p(s_{0:T}, m|z_{1:T}, u_{1:T}) \tag{2.2}$$

where:

- p = distribution
- $s_{0:T}$ = path
- m = map
- $z_{1:T}$ = observations
- $u_{1:T}$ = controls

In full SLAM the robot keeps track of all the previous poses including the current one. A different approach from full SLAM is online SLAM where the robot is only interested in the current pose. The probability distribution of online SLAM is shown in equation 2.3.

$$p(s_t, m|z_{1:t}, u_{1:t}) \tag{2.3}$$

The wanted part now only contains s_t instead of $s_{0:T}$ which means only the current pose is wanted instead of the complete state vector.

2.2 Datatypes and filtering techniques for SLAM

Within SLAM, two ways data representation can be used, the first is feature-based, described in Section 3.2. The second way is volumetric, described in Section 3.1. In feature-based, algorithms are used on sensor data to find distinguishable properties (features) of the environment which are stored as landmarks. Using landmarks the robot determines its position and path and the collection of the landmarks can be used to construct a map of the environment and localise the robot. Volumetric SLAM only stores volumetric data of the environment. The environment is represented as a volume filled with cells. Each cell can be free, which means the robot can access that cell, or occupied, which means the robot can not go there. The distinction between feature-based and volumetric SLAM influences the way data is stored and handled and therefore the entire SLAM algorithm. Each SLAM system is essentially a *filtering problem*, which means data that comes in is corrected with other data in order to increase the quality. Multiple filters are known to be used for SLAM:

- Kalman filters
- Information filters
- Particle filters
- Graph-based filters

Each of this filters has advantages and disadvantages which will be discussed in Chapter 8. The math used for this filters has been analysed in order to make a decision of which algorithm to use. However, the choice will be made for graph-based filtering which means the other filters are not used. Therefore the mathematics of the alternative filters can be found in Appendix D.

2.2.1 Loop closing

One problem in SLAM is that the probability of the robot position over time decreases due to accumulating uncertainty of long path. Loop closing is a way to increase the probability of the position due to the recognition of an area visited before. Loop closing means that the data from the current robot position is somehow associated with the data from a previous position. Because of new information of the closed loop the certainty and state of the poses between the so called "loop closing poses" can be corrected and improved. The way loop closing is applied is different for each variant of SLAM. A correct loop closing will improve the constructed map drastically, however an incorrect loop closing could be catastrophic for reconstruction of the map as can be seen in figure 2.3 where the left map is likely unusable for navigation.

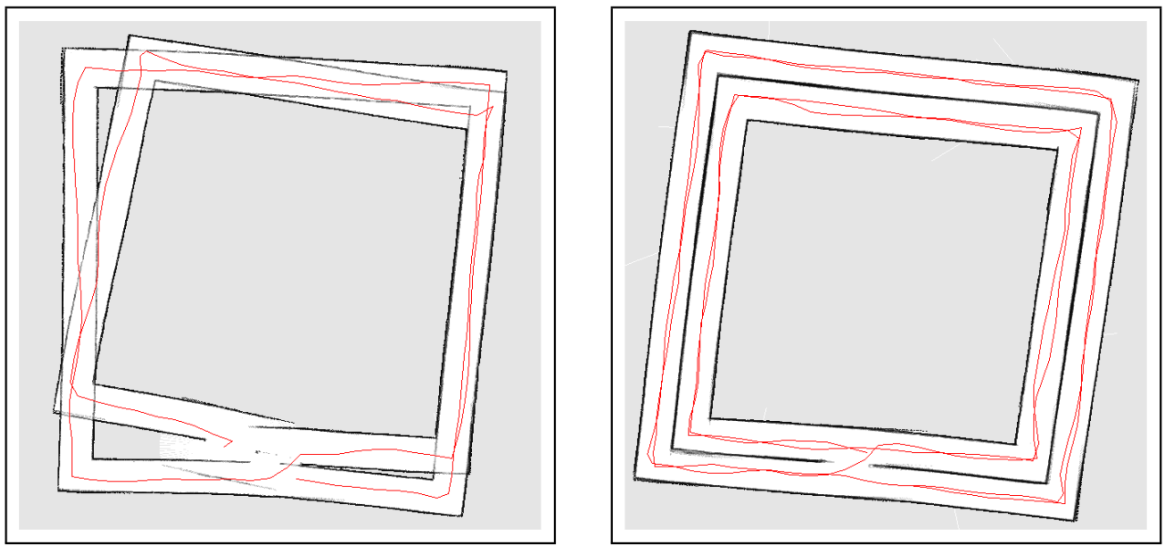


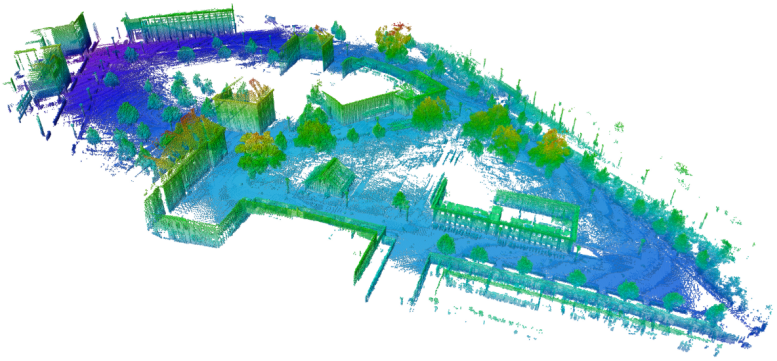
Figure 2.3: The effects of wrong loop closing (left) and correct loop closing (right)

3 — Data formats in SLAM

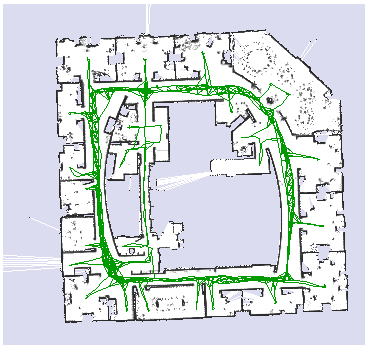
3.1 Volumetric SLAM

One way to represent the world is using a *volumetric* representation. Volumetric represents physical structure of the environment. Physical representation, or map (m), could either be in 3D like in figure 3.1a or 2D like in figure 3.1b. A map is constructed using sensor observations ($z_{1:T}$) and the path of the robot ($s_{1:T}$). A probabilistic mathematical model of the map is given in Equation (3.1).

$$p(m|z_{1:t}, s_{1:t}) \tag{3.1}$$



(a) Freiburg outdoor 3D map[32]



(b) Intel research lab occupancy grid[33]

3.1.1 Occupancy grid

One way of implementing 2D SLAM is by using of *occupancy grids* . The map is divided into a *grid* with *cells*. Each cell of the grid has an *occupancy probability*. Initially this probability is set to neither occupied or free. As soon as the robot starts detecting obstacles it will start changing the probabilities of the cells that correspond with the detected obstacles. If a distance to an object has been measured, the probability of occupancy of the points between the robot and the object will decrease and the grid points of the detected object will increase. In other situations the probabilities are left unchanged. The complete map can be considered a sum of the individual cells (m_i):

$$m = \sum_i m_i \tag{3.2}$$

This factorization leads to the property that m_i can be represented as a binary probability if a cell is occupied ($p(m_i) \approx 1$) or free ($p(m_i) \approx 0$). Each grid cell is calculated independently and

is does not have correlation to its neighbours. The distribution of the map becomes a product of the grid cells:

$$p(m|z_{1:t}, u_{1:t}) = \prod_i p(m_i|z_{1:t}, u_{1:t}) \tag{3.3}$$

Thrun et al. [58] describes a log odds notation to avoid numerical instabilities for probabilities close to zero or one. The log-odds representation of occupancy grids is shown in Equation (3.4)

$$l_{t,i} = \log \frac{p(m_i|z_{1:t}, s_{1:t})}{1 - p(m_i|z_{1:t}, s_{1:t})} \tag{3.4}$$

The probabilities can be recovered using the following:

$$p(m_i|, z_{1:t}, s_{1:t}) = 1 - \frac{1}{1 + e^{l_{t,i}}} \tag{3.5}$$

Thrun et al. [58] suggest to use the log-odds notation in combination with the **inverse_sensor_model** to update the probabilities of the occupancy grid. The **inverse_sensor_model** for proximity sensors is described in Appendix B. The algorithm for simple occupancy grid mapping is shown in Algorithm 1. The algorithm checks whether the cell m_i falls into the current sensor observation z_t. If this is the case then in line 3 the observation is added to the cell probability. The value l_0 represents the p_{prior} as described in Appendix B, and can be stated as follows:

$$l_0 = \log \frac{p(m_i = 1)}{p(m_i = 0)} = \log \frac{p(m_i)}{1 - p(m_i)} \tag{3.6}$$

Algorithm 1 Occupancy grid mapping

```
1: for all cells m_i do
2:     if m_i in z_t then
3:         l_{t,i} = l_{t-1,i} + inverse_sensor_model − l_0
4:     else
5:         l_{t,i} = l_{t-1,i}
6:     end if
7: end for
8: return l_{t,i}
```

Figure 3.2 shows a sample of a environment with black shapes represented by an occupancy grid. A grey cell represents an occupied cell and each white cell means the cell is free. The representation is somewhat pessimistic because the resolution of the cell is roughly the same as the size of the objects, which results in oversized shapes in the occupancy grid.

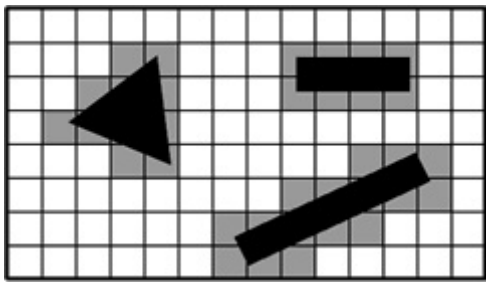


Figure 3.2: An example of an occupancy grid[7]

3.2 Feature-based SLAM

Feature-based SLAM is an approach to solve the SLAM problem using features for both the localization and mapping part. Features are *distinguishable properties extracted from sensor data*. Due to sensor noise and inaccuracy the position of the feature is stored with a probability, which could modelled using a Gaussian distribution. Many SLAM algorithm support feature-based data representation,i.e. Gaussian approaches, particle filters, and graph-based solutions. Figure 3.3 shows an implementation of feature-based SLAM in MATLAB. In this figure the blue arrow represents a robot at its' state estimation with Gaussian distribution drawn as red circle around it. The landmarks which are shown as blue asterisks have green circles showing Gaussian distribution.

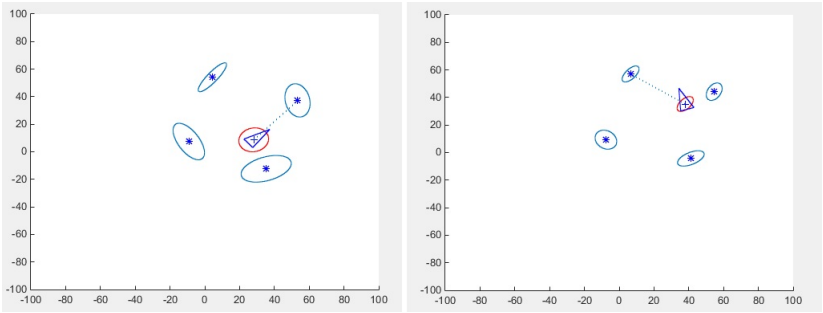


Figure 3.3: Feature-based slam example

The features that are used for feature-based SLAM are sometimes called *landmarks* . A problem with landmarks is that, in a normal environment, landmarks differ in size and shape. Therefore, the algorithms that are used to determine and distinguish landmarks are not completely part of the SLAM problem. The landmarks are distinguished by a separate vision algorithm which should be able to recognize those differences in size and shapes. The SLAM algorithm gets the feature locations and descriptions and compares them to the features that have a likelihood to be seen considering the proposal distribution. Figure 3.4 shows a feature-based SLAM algorithm which was applied to existing data set from the Victoria park located in Sydney. The landmarks in this case are trees which are quite easy to detect with a laser range scanner. It can be seen that each tree has a ellipse drawn around it which is actually Gaussian distribution. The red line is the trajectory of the vehicle during the data acquisition.

Kalman- and Information filters are suitable for feature-based SLAM. These filtering techniques are discussed in Appendix D.

3.2.1 Feature extraction

For feature-based SLAM, feature recognition and association is essential in order to determine the correct translations and rotations relative to previous poses. A feature is known as a strong detectable point in the data. In most SLAM problems, features are detectable independent of the orientation. This way, the features' relative position to the robot can be used to the determine the robot's pose in the global map. Feature recognition is, in most cases, a computer vision problem for which exists a large amount of solutions. Which algorithm to use highly depends on the sensors that are used and which hardware is available. There are also algorithms that are robust at the cost of more computational complexity. An image acquired from a camera requires a different amount of processing than an image acquired by a laser scanner. A small amount of feature extraction techniques are briefly discussed in Appendix F.

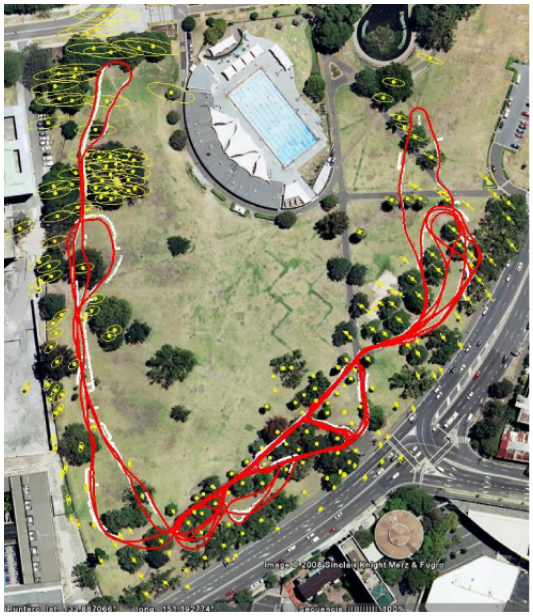


Figure 3.4: An example of feature-based SLAM on the Victoria Park dataset[47]

3.2.2 Loop closing

Loop closing in feature-based SLAM is done by matching the expected observation to the real observation. If the robot has actually completed a loop, the object will be recognized and the errors can be corrected by recalculating all of the robot poses given the new data. The loop closing gives the robot a high payload but will result in a better state estimate. The loop can only be closed if associated data is within the robot's state distribution and is indeed recognized as being the same features.

4 — Graph-based SLAM

4.1 Graph-based SLAM

4.1.1 Introduction

The SLAM problem can be solved by different filtering techniques, in on of them the system is represented by a graph. This method is called *graph-based SLAM* The graph consists of robot poses represented by the *nodes* in the graph. The *edges* between the nodes represent the relation between two poses and denoted by Ω. A robot *pose* exists of a position and an orientation. In the 2-dimensional world, this pose contains three degrees of freedom: (x,y,θ). In the 3-dimensional world, a robot pose has six degrees of freedom: three translational and three rotational. In this thesis only the 2-dimensional case is discussed. The mathematical rules are the same for the three dimensional case. The *edges* do not represent positions, they represent the probability between two poses. Similar to other SLAM approaches, the positions and their probabilities can be interpreted as Gaussian probabilities. The mean of the Gaussian is the position of a node and the probability (Ω) is the inverse of the sigma, which determines the width of the Gaussian. The Gaussian can have different sigma's in different directions and in a two dimensional representation the Gaussian will be represented by an ellipse around the position of the robot. Each position within the ellipse has a probability probability to be at that position. In this report, the mathematical proof behind graph-based SLAM is not discussed, it can be found in detail in [25].
For clarification, vector notations will be depicted by an arrow above the symbol and matrices by a bold capital letter.

Position and orientation data combined are called robot poses. Each node contains a robot pose, there is only one robot pose at each point in time which is denoted by:

$$\vec{s}_i \;=\; (\vec{s}_{i,x} \;,\; \vec{s}_{i,y)} \;,\; \vec{s}_{i,\theta})$$

The state vector is a set of the current pose and all previous poses:

$$\vec{s} \;=\; \{\vec{s}_0, \vec{s}_1, .., \vec{s}_t\}$$

Every pose in the graph has a relation with every previous pose, which is called $\Omega_{i,j}$. Each relation is non-directional, which means the relation between pose i and j is the same as the relation between j and i. This relation is described by the probability of correctness of the pose relative to another. Every probability is initially set to zero for each pose, only when poses are connected by edges, the probability will become non-zero.

Edges are created when a sensor detects a relation between two poses. It is possible to distinguish two different types of edges:

1. Edges created by consecutive odometry commands
2. Edges created by loop closing

Edges created by consecutive odometry commands always describes the probability of the current node (pose) relative to the next one.

$$\vec{s}_i \rightarrow \vec{s}_{i+1}$$

Edges created by loop closing are not consecutive. Loop closing is found at random and can virtually appear between every two poses in a graph. A state vector can contain many or no loops and therefore a lot of loop closing or no loop closing at all. The indices of the poses that contain loop closing have no relation as they have with odometry edges. The loop closing indices are described by i and j:

$$\vec{s}_i \rightarrow \vec{s}_j,\ i > j$$

Important is that these relations do not describe positions, they describe the probability of correctness of one pose relative to another. Before the first loop closing has occurred, the only information about the poses of the nodes comes from odometry sensors. Once loop closing has occurred, there will be conflicting information about the poses. The algorithm of graph-based SLAM is used to combine the conflicting information to find an error corrected version of the graph where the sum of all errors is as little as possible.

4.1.2 Graph construction

The nodes from the graph can be seen as a vector of tuples where each tuple consists of three components: x, y and θ. This state vector is constructed from odometry data which consists of angles and distances, which is actually the polar coordinate system. Each angle in the odometry is the angle that the robot rotates before it starts translating. After the rotation the robot translates in the direction of the new angle. Thus, in this model, there is only one translation instead of two as described in the odometry motion model described in Appendix B. The translations are denoted by δ. Odometry commands in datasets are in polar coordinates which means conversion to Cartesian coordinates is necessary if the Cartesian coordinate system is used. This conversion can be performed using trigonometric functions:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\delta}_{x,i} * cos(\theta_{i+1}) \tag{4.1}$$

and:

$$\vec{y}_{i+1} = \vec{y}_i + \vec{\delta}_{y,i} * sin(\theta_{i+1}) \tag{4.2}$$

In the above equations $\vec{x}_i$ is $\vec{s}_{i,x}$ and $\vec{x}_{i+1}$ is $\vec{s}_{i+1,x}$, δ_i is the translation that the robot will make, which for most robots will only be in one direction. The action performed with this equations is actually a frame conversion, and for robots that translate in more than one axis, the equations are not yet complete. The complete equations with two translation axis also take translation in the y direction into account as follows:

$$\vec{x}_{i+1} = \vec{x}_i + \vec{\delta}_{i,x} * cos(\theta_{i+1}) + \vec{\delta}_{i,y} * sin(\theta_{i+1}) \tag{4.3}$$

and:

$$\vec{y}_{i+1} = \vec{y}_i + \vec{\delta}_{i,x} * sin(\theta_{i+1}) + \vec{\delta}_{i,y} * cos(\theta_{i+1}) \tag{4.4}$$

The frame conversion is needed because the map and the robot have different frames. The robot's x-axis is aligned with the direction the nose of the robot is facing. Figure 4.1 shows the movement of a robot with an angle and translation as odometry command. This odometry command seems to be in polar coordinates, but a translation in y-direction in the robot's frame could also be possible. The figure only shows a translation in the x-direction with the

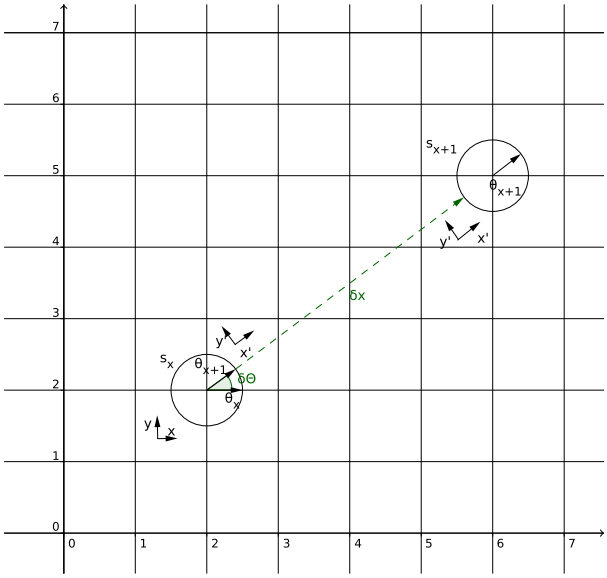


Figure 4.1: A robot that moves from position (2,2) with $\theta = 0$ with odometry command $\delta_x = 5$, $\delta_y = 0$ and $\delta_\theta = atan\left(\frac{3}{4}\right)$ to position (6,5) with $\theta = atan\left(\frac{3}{4}\right)$

corresponding robot frames.

Figure 4.2 shows an example of a graph created by odometry. At this moment in time, the best graph estimate that can be obtained is the raw odometry data since no other information (such as loop closing) is available. This graph will later be used execute the graph SLAM algorithm.

4.1.3 Loop closing

Loop closing has been discussed earlier to be one of the main components for SLAM and can best be explained by the recognition of features or area that a robot has observed before, and using this information to improve the state estimate. Loop closing in graph-based SLAM is performed when the robot's pose estimate is within a given range of an earlier pose. The new pose estimate will create an edge between the earlier pose and itself. Loop closing happens between two nodes that are already in the graph, which means their positions and also their difference in position is stored. The loop closing calculates a new relative position and the difference with the previous position is called the error, which also has an x, a y and a θ component. The error can be calculated by the error function, which will be explained in the next section. An error is used to improve the pose-graph estimate.

4.1.4 The error function

Odometry commands will create a state vector $\vec{s}$. This state vector consists of the poses the robot has been. Additional edges can be formed by loop closing which means that there is a new position relation found between two poses. The loop closing (laser) data is of better quality than odometry since odometry data contains accumulated errors. The laser data is accurate and contains a direct relation between the two poses.

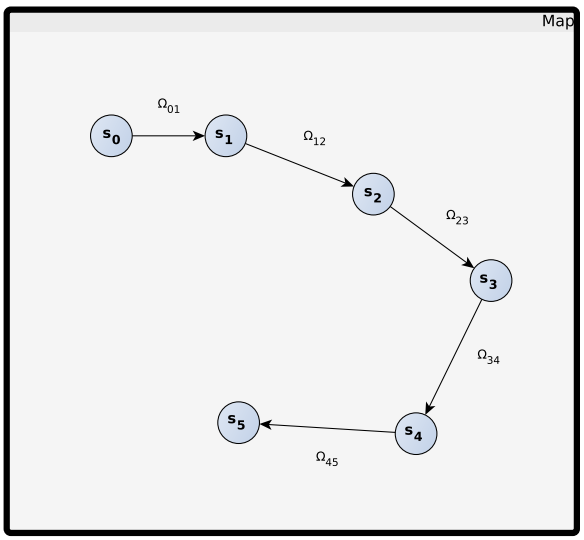


Figure 4.2: Example of a 2-dimensional graph created from odometry commands with six poses. The blue nodes are robot poses and the errors are edges. The black square is considered to be the map.

The difference between the same poses defined by other information is called an error. In this case the error is the difference between the pose according to the graph, and the pose according to the laser scan observation.

To calculate the error, two poses that an error is calculated from, have to be aligned. Alignment of the poses is performed by taking the observation into account. When a pose i closes a loop with another pose j, there must be observational sensor data that makes loop closing possible. Since the poses that enable loop closing are often not the same, the transformation between them must be taken into account as well. This transformation is known by z_{ij} and will, similar to poses, consist of a tuple of three items for a two-dimensional problem.

If the poses that close the loop are exactly the same, which means $z_{ij} = (0, 0, 0)$, the error in the loop can simply be calculated by subtracting one pose from another. If alignment z_{ij} is not a zero vector, the difference must be subtracted from this alignment. This equation is called the error function and is shown in figure 4.5.

The error function:

$$e_{ij} = z_{ij} - (\vec{s}_j - \vec{s}_i) \tag{4.5}$$

4.1.5 The information matrix

Nodes are connected by edges which contain the probability of correctness between one pose relative to another. Edges are not used when only adding poses to the graph from odometry. Until loop closing the edges will be stored in a matrix which is called the information matrix. After loop closing the information matrix will be used to correct the state vector to obtain a better estimate.

Because it is possible for every pose to create an edge to any other pose, the number of possible edges is the number of nodes squared, hence the data will become a matrix. The

matrix will be symmetrical since the probability of pose i relative to pose j is the same as the probability of pose j relative to pose i.

To find the places in the matrix to write, the partial derivatives of the appropriate error function need to be found, the vector of partial derivatives is called a *Jacobian*. The error function was defined by equation 4.5. The partial derivates can only have three values: 1, -1 and 0. The rules that are used to find the partial derivatives are as simple as:

$$\frac{x_i}{dx_i} = 1$$

and:

$$\frac{-x_i}{dx_i} = -1$$

Other partial derivatives become zero.

The partial derivatives of the error function will become:

$$\mathbf{J}_{ij} = \begin{pmatrix} \frac{\delta e_{ij}}{\delta \vec{s}_i} & \frac{\delta e_{ij}}{\delta \vec{s}_i} \end{pmatrix} = (1 \quad -1)$$

where:

$$\frac{\delta e_{ij}}{\delta \vec{s}_i} = \frac{\delta z_{ij}}{\delta \vec{s}_i} - \left(\frac{\delta \vec{s}_j}{\delta \vec{s}_i} - \frac{\delta \vec{s}_i}{\delta \vec{s}_j} \right) = 1$$

and :

$$\frac{\delta e_{ij}}{\delta \vec{s}_j} = \frac{\delta z_{ij}}{\delta \vec{s}_j} - \left(\frac{\delta \vec{s}_j}{\delta \vec{s}_j} - \frac{\delta \vec{s}_i}{\delta \vec{s}_j} \right) = -1$$

In theory, there are more partial derivatives to find as the full Jacobian would look as follows, given $(i < j)$ and N being the number of poses (size of the system):

$$J_{ij} = \begin{pmatrix} \frac{\delta e_{ij}}{\delta \vec{s}_0} & \frac{\delta e_{ij}}{\delta \vec{s}_1} & \dots & \frac{\delta e_{ij}}{\delta \vec{s}_i} & \dots & \frac{\delta e_{ij}}{\delta \vec{s}_j} & \dots & \frac{\delta e_{ij}}{\delta \vec{s}_{N-1}} & \frac{\delta e_{ij}}{\delta \vec{s}_N} \end{pmatrix} = (0 \;\; 0 \;\dots\; 1 \;\dots\; -1 \;\dots\; 0 \;\; 0)$$

Since there are no non-zero values rather than the derivatives to $\delta \vec{s}_i$ and $\delta \vec{s}_j$, the Jacobian becomes a vector with only a 1 at the $i-th$ position and a -1 at the $j-th$ position. The Jacobian can also be represented in matrix form, which can be calculated by the outer product of the Jacobian transposed as a column-vector and the same Jacobian as row-vector:

$$\mathbf{J}_{ij}^T * \mathbf{J}_{ij} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{4.6}$$

In the information matrix, the Jacobian in matrix form is multiplied with the probability of the edge:

$$\mathbf{H}_{ij} = \mathbf{J}_{ij}^T \Omega_{ij} \mathbf{J}_{ij} \tag{4.7}$$

In this equation the $\mathbf{H}_{ij}$ matrices are parts of the information matrix and represent a matrix in which only four values that are non-zero representing the probabilities between two poses in matrix form with positive values on the diagonal and negative values on the off-diagonal. Each edge creates a matrix which can be add to create the information matrix:

$$\mathbf{H} = \sum_{ij} \mathbf{H}_{ij} \;\; \forall ij$$

In the constructed matrix, the diagonal elements are an accumulated value of different edges, which becomes the total probability of correctness of that node in the graph. Each off-diagonal element tells the probability of correctness of two poses relative to each other.

For the graph in figure 4.2 the matrix would consist of 5 edges which in matrix form will consist of 5 partial matrices. The addition of partial matrices is shown below in equation 4.8.

$$
\begin{aligned}
\mathbf{H} = \mathbf{H}_{01} + \mathbf{H}_{12} + \mathbf{H}_{23} + \mathbf{H}_{34} + \mathbf{H}_{45} = \\
&\begin{bmatrix} \Omega_{01} & -\Omega_{01} & 0 & 0 & 0 & 0 \\ -\Omega_{01} & \Omega_{01} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Omega_{12} & -\Omega_{12} & 0 & 0 & 0 \\ 0 & -\Omega_{12} & \Omega_{12} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \\
&\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Omega_{23} & -\Omega_{23} & 0 & 0 \\ 0 & 0 & -\Omega_{23} & \Omega_{23} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Omega_{34} & -\Omega_{34} & 0 \\ 0 & 0 & 0 & -\Omega_{34} & \Omega_{34} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \\
\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \Omega_{45} & -\Omega_{45} \\ 0 & 0 & 0 & 0 & -\Omega_{45} & \Omega_{45} \end{bmatrix} &= \begin{bmatrix} \Omega_{01} & -\Omega_{01} & 0 & 0 & 0 & 0 \\ -\Omega_{01} & \Omega_{01}+\Omega_{12} & -\Omega_{12} & 0 & 0 & 0 \\ 0 & -\Omega_{12} & \Omega_{12}+\Omega_{23} & -\Omega_{23} & 0 & 0 \\ 0 & 0 & -\Omega_{23} & \Omega_{23}+\Omega_{34} & -\Omega_{34} & 0 \\ 0 & 0 & 0 & -\Omega_{34} & \Omega_{34}+\Omega_{45} & -\Omega_{45} \\ 0 & 0 & 0 & 0 & -\Omega_{45} & \Omega_{45} \end{bmatrix}
\end{aligned}
\tag{4.8}
$$

Whenever loop closing occurs, the size of matrix **H** does not change, because the loop closure occurs between already existing poses. Continuing on the previous graph and matrix, loop closing will occur between the next pose ($\vec{s}_6$) and the first pose ($\vec{s}_0$), which is shown in figure 4.3

The graph shown in figure 4.3 is an extension of the graph in figure 4.2. The matrix found in equation 4.8 gets extended with a sixth odometry edge, but also with a loop closing edge between pose $\vec{s}_6$ and pose $\vec{s}_0$, which is shown in equation 4.9.

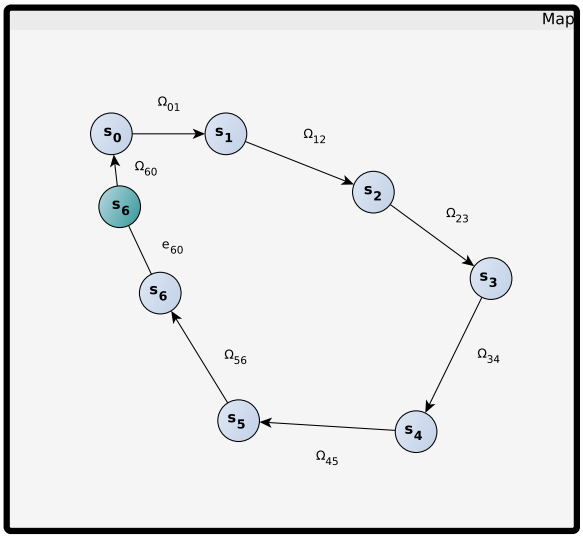


Figure 4.3: Example of a graph created from odometry commands with seven poses and one loop closing occurrence

$$\mathbf{H} = \mathbf{H}_{previous} + \mathbf{H}_{56} + \mathbf{H}_{60} =$$

$$\begin{bmatrix} \Omega_{01} & -\Omega_{01} & 0 & 0 & 0 & 0 \\ -\Omega_{01} & \Omega_{01}+\Omega_{12} & -\Omega_{12} & 0 & 0 & 0 \\ 0 & -\Omega_{12} & \Omega_{12}+\Omega_{23} & -\Omega_{23} & 0 & 0 \\ 0 & 0 & -\Omega_{23} & \Omega_{23}+\Omega_{34} & -\Omega_{34} & 0 \\ 0 & 0 & 0 & -\Omega_{34} & \Omega_{34}+\Omega_{45} & -\Omega_{45} \\ 0 & 0 & 0 & 0 & -\Omega_{45} & \Omega_{45} \end{bmatrix} +$$

$$\begin{bmatrix} 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&\Omega_{56}&-\Omega_{56} \\ 0&0&0&0&0&-\Omega_{56}&\Omega_{56} \end{bmatrix} + \begin{bmatrix} \Omega_{60}&0&0&0&0&0&-\Omega_{60} \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0 \\ -\Omega_{60}&0&0&0&0&0&\Omega_{60} \end{bmatrix} =$$

$$\begin{bmatrix} \Omega_{01}+\Omega_{60} & -\Omega_{01} & 0 & 0 & 0 & 0 & -\Omega_{60} \\ -\Omega_{01} & \Omega_{01}+\Omega_{12} & -\Omega_{12} & 0 & 0 & 0 & 0 \\ 0 & -\Omega_{12} & \Omega_{12}+\Omega_{23} & -\Omega_{23} & 0 & 0 & 0 \\ 0 & 0 & -\Omega_{23} & \Omega_{23}+\Omega_{34} & -\Omega_{34} & 0 & 0 \\ 0 & 0 & 0 & -\Omega_{34} & \Omega_{34}+\Omega_{45} & -\Omega_{45} & 0 \\ 0 & 0 & 0 & 0 & -\Omega_{45} & \Omega_{45}+\Omega_{56} & -\Omega_{56} \\ -\Omega_{60} & 0 & 0 & 0 & 0 & -\Omega_{56} & \Omega_{56}+\Omega_{60} \end{bmatrix} \quad (4.9)$$

The graph in figure 4.3 shows a graph in which loop closing has occurred. There are two instances of the $\vec{s}_6$ drawn in the graph, the light blue one is the one that has been constructed from odometry and the green one is the position of $\vec{s}_6$ according to the observation sensor. The green node is not present in the state vector, it only shows there is a mismatch in the information about that node. There is a line drawn between the two instances of $\vec{s}_6$, this line is not an edge, it represents the error which also implies the position of the green $\vec{s}_6$ node relative to the blue version of $\vec{s}_6$. Because the edge that represents the loop closure ($\mathbf{\Omega}_{60}$) is added to

the graph, conflicting data is present in the system. The found error and information matrix can now be used to change the graph in a weighted manner.

4.1.6 Minimization of errors and correction of the graph

To find the optimal version of the graph with the available data, the state vector needs to be corrected. The weighted error is the error multiplied with the probability. Equation 4.10 shows that a state vector needs to be found for which the weighted error is minimal where $\vec{s}^*$ is the best state estimate.

$$\vec{s}^* = argmin_x \sum_{ij} \vec{e}_{ij}^T \mathbf{\Omega}_{ij} \vec{e}_{ij} \tag{4.10}$$

Equation 4.10 shows that a value of the state vector $\vec{s}$ needs to be found where the sum of the squared errors is minimal. The corresponding values of matrix $\mathbf{\Omega}$ contains the probabilities of distances between the nodes and acts like a scalar on the error values.

To find values for $\vec{s}$ for which the error is small, the state vector must be changed. Just changing the pose of one node is not a sufficient method since the relation with other nodes is in that way neglected. The change of the state vector can be calculated by spreading the error over the nodes while by their probabilities. If the relation between two nodes has a low probability, it will absorb a larger part of the error than a relation with a high probability.

The update of the state vector can be defined as $\Delta\vec{s}$, which has the same structure as the state vector itself. $\Delta\vec{s}$ can be found by solving the linear system of equation 4.11. In this equation $\mathbf{H}$ is the Information matrix as shown above, and $\vec{b}$ is a vector in which the errors are taken into account.

$$\mathbf{H}\ \Delta\vec{s} = -\vec{b} \tag{4.11}$$

Looking at the graph with loop closure in figure 4.4 and the corresponding $\mathbf{H}$-matrix found in equation 4.9 the complete system will look like the system in equation 4.12.

$$\begin{bmatrix} \mathbf{\Omega}_{01}+\mathbf{\Omega}_{60} & -\mathbf{\Omega}_{01} & 0 & 0 & 0 & 0 & -\mathbf{\Omega}_{60} \\ -\mathbf{\Omega}_{01} & \mathbf{\Omega}_{01}+\mathbf{\Omega}_{12} & -\mathbf{\Omega}_{12} & 0 & 0 & 0 & 0 \\ 0 & -\mathbf{\Omega}_{12} & \mathbf{\Omega}_{12}+\mathbf{\Omega}_{23} & -\mathbf{\Omega}_{23} & 0 & 0 & 0 \\ 0 & 0 & -\mathbf{\Omega}_{23} & \mathbf{\Omega}_{23}+\mathbf{\Omega}_{34} & -\mathbf{\Omega}_{34} & 0 & 0 \\ 0 & 0 & 0 & -\mathbf{\Omega}_{34} & \mathbf{\Omega}_{34}+\mathbf{\Omega}_{45} & -\mathbf{\Omega}_{45} & 0 \\ 0 & 0 & 0 & 0 & -\mathbf{\Omega}_{45} & \mathbf{\Omega}_{45}+\mathbf{\Omega}_{56} & -\mathbf{\Omega}_{56} \\ -\mathbf{\Omega}_{60} & 0 & 0 & 0 & 0 & -\mathbf{\Omega}_{56} & \mathbf{\Omega}_{56}+\mathbf{\Omega}_{60} \end{bmatrix} \begin{bmatrix} \Delta\vec{s}_0 \\ \Delta\vec{s}_1 \\ \Delta\vec{s}_2 \\ \Delta\vec{s}_3 \\ \Delta\vec{s}_4 \\ \Delta\vec{s}_5 \\ \Delta\vec{s}_6 \end{bmatrix} = \begin{bmatrix} \vec{e}_{ij}\mathbf{\Omega}_{61} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -\vec{e}_{ij}\mathbf{\Omega}_{61} \end{bmatrix} \tag{4.12}$$

Vector b is a vector in which the weighted errors are accumulated to find the total set of errors:

$$\vec{b} = \sum_{ij} \vec{b}_{ij}$$

Each part of $\vec{b}$ consists of an error and a probability of correctness of the observation edge that created the error. Again the value of the multiplied values is multiplied with the Jacobian which will again be a vector with a 1 at position i and a -1 at position j. The construction of $\vec{b}$ is shown in equation 4.13.

$$\vec{b}_{ij} = \vec{e}_{ij}\mathbf{\Omega}_{ij}\mathbf{J}_{ij} \tag{4.13}$$

A possible way to solve the system is by inverting the inverting matrix in order to obtain $\Delta\vec{s} = \mathbf{H}^{-1}\vec{b}$. Inversion is a computationally heavy operation. Hence, other solving methods need to be considered.

The found vector $\Delta\vec{s}$ can be added to the state vector to alter the graph for the Cartesian part (x and y). The changes of θ can be added as well, but applying the changes to the state vector is performed differently as described in the next subsection.

4.1.7 Correction of errors of angles

Each pose within a two-dimensional graph consists of 3 values: x, y and θ and in state vector form one could write: $\vec{x}$, $\vec{y}$ and $\vec{\theta}$. Solving the linear systems as described in equation 4.11, results in values for $\Delta\vec{x}$, $\Delta\vec{y}$ and $\Delta\vec{\theta}$ which can be used to change the graph in order to decrease the total error. The calculated values for $\Delta\vec{x}$ and $\Delta\vec{y}$ are already in Cartesian coordinates, therefore the updated values of the $\vec{x}$ and $\vec{y}$ coordinates can be calculated by vector addition.

Each value of θ represents the angle the robot is facing at that position, and therefore the angle in which it has moved from the previous position. In a constructed graph, the x and y are independent of the values of θ since the graph is in Cartesian coordinates. Applying a change of θ can initially also be performed using a vector addition to find the new values of θ in the graph. After applying the calculated correction, the direction of the robot in the graph does not match the theta values. In this case, the direction the robot is facing does not match the direction the robot has driven. Applying the updated values of θ can only be performed by converting the graph into polar coordinates and reconstructing the graph with the updated values of θ.

Converting the graph to polar coordinates is performed by calculating the distances between the nodes in the graph. This can be performed using Pythagoras' theorem:

$$d_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \tag{4.14}$$

Each value of d_i represents a distance between two nodes from the updated graph. The updated values of θ can be used to reconstruct the graph with the same trigonometric functions as used for graph construction earlier in this chapter:

$$x_{i+1} = x_i + d_i * cos(\theta_{i+1}) \tag{4.15}$$

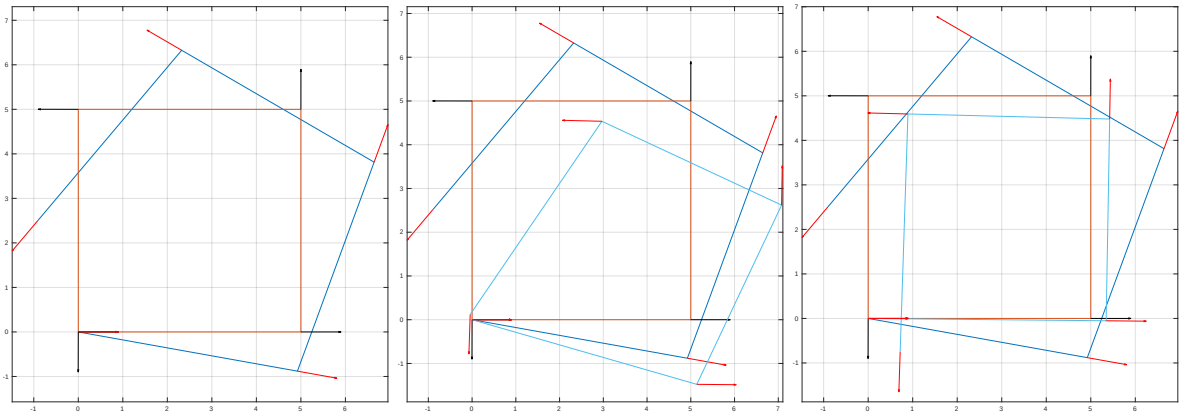
and:

$$y_{i+1} = y_i + d_i * sin(\theta_{i+1}) \tag{4.16}$$

The disadvantage of this method is that applying changes in the θ direction, does also make changes in the x and y coordinates of the graph. If error correction for x and y is done after error correction for θ, the euclidean distance between the two nodes is larger than before θ corrections, however, the sum of all three errors will have decreased.

4.1.8 Error convergence

Graphs can contain multiple errors because it is possible to have multiple loop closings. When the state vector changes, the errors change accordingly. Because $\Delta\vec{s}$ is still a weighted combination between odometry and observations, the sum of all errors will not be minimal after



(a) Ground truth (orange) and odometry sensor data (blue)

(b) Graph corrected with updated values without theta correction applied(light blue)

(c) Graph corrected and theta correction done (light blue)

Figure 4.4: Different corrected graph plots to show difference in resulting graphs.

one iteration. When a graph is also corrected in θ-direction, the graph will become over corrected. Multiple errors and an overshoot in correction introduces the need for iterations. The sum of errors will converge after multiple iterations. The amount of iterations can be fixed or determined by the total resulting error.

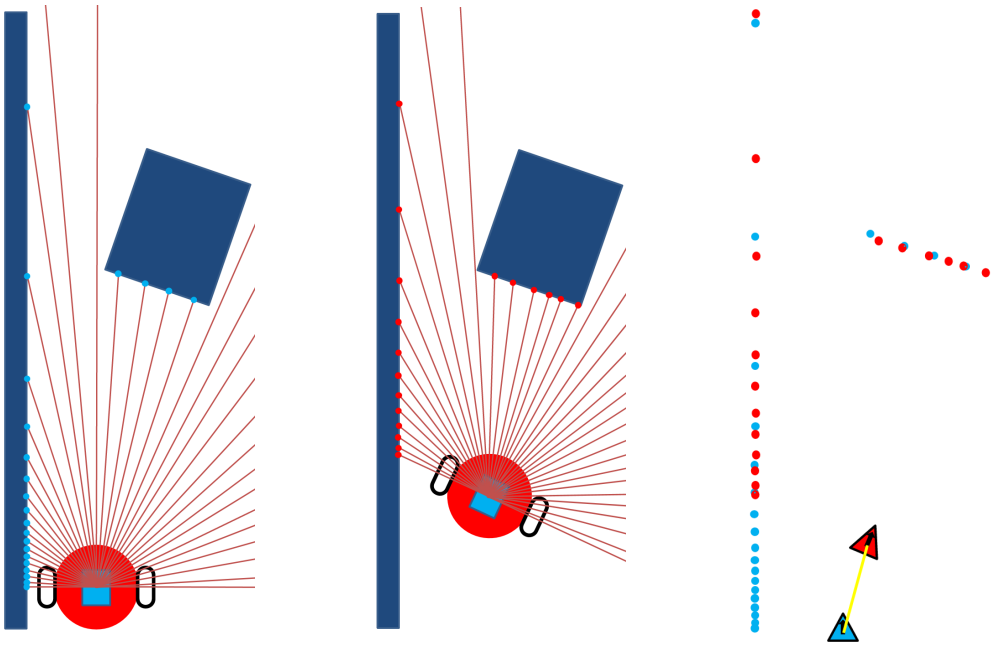
4.1.9 Hierarchical Optimization

Over time the dimensions of the state vector and information matrix will increase. The growth of the state vector and information matrix can be limited by Hierarchical Optimization for Graph-based slam (HOG)[27]. HOG is a method to decrease the number of poses in the state vector based on the distance between them. Each node after the certain distance becomes the first node of the new group. Removed poses are used to draw a resulting map, but they are not taken into account in error convergence. Using the HOG algorithm, the computational load reduces. Poses that are not inside the pose graph are still corrected by the correction that is applied on the remaining higher level poses. This method can be seen as an extension of the normal graph-based SLAM algorithm and can play a large role in implementation where the use of resources should be limited. In hardware design, a limited amount of memory is available which can not be exceeded, HOG could potentially be a good method to limit the amount of used memory. If an algorithm restricts the amount of memory usage by abstracting the map while keeping a sufficient quality of the map, the maximum length of the state vector can increase.

4.1.10 Graph pruning: Removing non informative poses

Another method to decrease the complexity of the graph is called graph pruning[35]. The approach is based on selecting laser scans that are most informative with respect to the map estimate. Graph pruning aims at minimizing the expected loss of information in the resulting map without introducing a bias during the selection of the laser scans. Graph pruning can be a method to allow for long-term robot mapping since a robot that keeps all scans will run out of resources earlier. In addition to that, the method can be used to directly implement SLAM system with an environment of any given size.

5 — Scan-matching



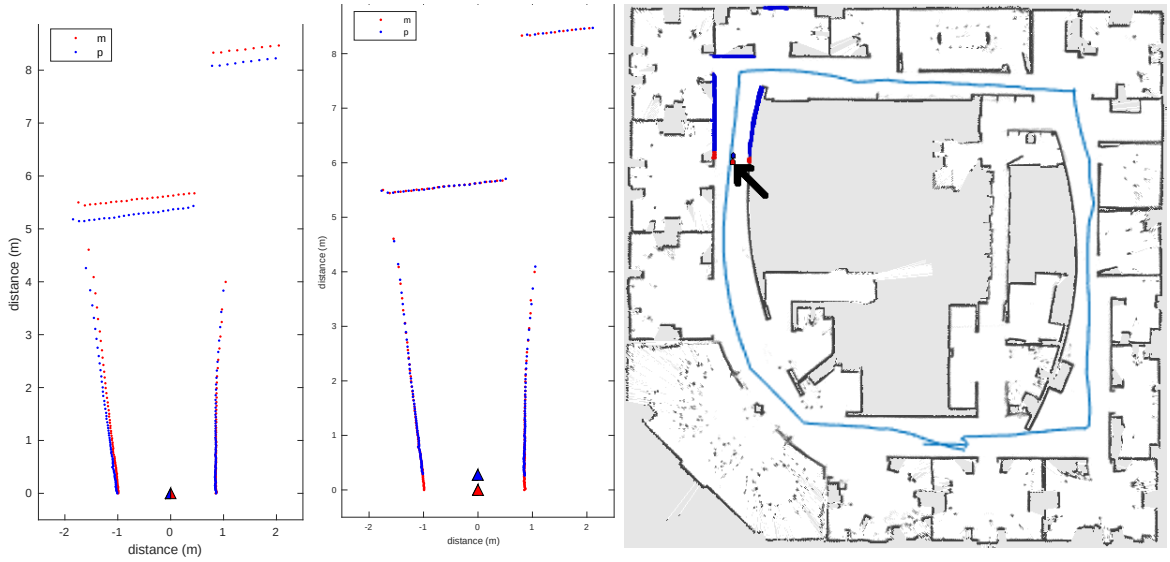
(a) Robot with a laser range scanner taking a scan of the environment

(b) Robot with a laser range scanner taking a scan of the environment

(c) Two scans lying on top of each other with a yellow line showing the transformation

Figure 5.1: Proximity scan used to determine robot movement (transformation)

To determine the robot movement it needs to know which paths it has travelled for which odometry sensors can be used. However, odemetry sensors are know to be inaccurate because of slippage and drift, therefore other sensors, like a *laser range scanner* (LRS) , can be used to determine the movement of the robot. Proximity scans provide environmental data using a beam-format, described in Appendix B. A beam-format means that the sensor determines the range to an object in combination with the angle with respect to the sensor. Figure 5.1 shows a simple view of this principle where the red lines represent the beams. Figure 5.1a shows the initial robot position, the robot takes a scan of the environment and the laser beams register objects at the blue dots of the end of the red lines. The coordinates of the blue dots are stored and the robot drives a bit forward with a slight rotation to the right, as can be seen in Figure 5.1b. In the second position the robot takes another scan of the environment with the red dots at the end of the red lines representing the detected obstacles (Figure 5.1b). Once the robots has the two observations it determines the *movement* (*transformation*) between pose 1 and pose 2 by aligning the two scans (Figure 5.1c). The process of aligning two scans is called *scan-matching* . The objective of scan-matching is locating the current robot position relative



(a) Two consecutive scans without transformation

(b) Two consecutive scans with transformation applied

(c) Robot path in the Intel data set where the black arrow indicate the location where the consecutive scans are taken.

Figure 5.2: Example of scan data before and after an ICP algorithm

to a previous one [41]. Scan-matching can be done by matching locally (current observation with previous one) or globally (current observation with the already known map). A lot of research is done in scan-matching [45], and several solutions to solve the problem are available but only the ICP (Iterative Closest Point) is described below, other solutions are described in Appendix C.

5.1 ICP: Iterative Closest Point

The *ICP* (*Iterative Closest Point*) algorithm has become the dominant method for aligning multi-dimensional models based purely on the geometry from sensor data [51]. Sensor data for ICP must be a set of one or more points in a one or more dimensional space. In the world of SLAM ICP is used to find the *rotation* and *translation* between two sensor observations. Figure 5.2 illustrates an example case of the ICP algorithm results applied on two consecutive scans taken from the location indicated by the black arrow in Figure 5.2c. Figure 5.2a shows the non transformed laser scans of the environment plotted over each other with both the robot pose (triangles) at $(0, 0)$. Figure 5.2b shows the result after applying an ICP algorithm, one can see that the current observation ($\vec{p}$, blue dots) is aligned onto the previous observation ($\vec{m}$, red dots) to know the movement of the robot. The key concept of the standard ICP can be summarized into two steps [52]:

1. Construct *correspondences* between two scans
2. Construct *transformation* which minimizes the distances (errors) between the correspondences

A correspondence is a match from point p_i from the current observation $\vec{p}$ to a point m_j from the previous observation $\vec{m}$. Figure 5.3 shows two laser scans, with correspondence variables, shown in the zoomed box and, depending on the implementation, contains the following variables:

- Point (p_{ix}, p_{iy}) from the new observation $(p_i \in \vec{p})$

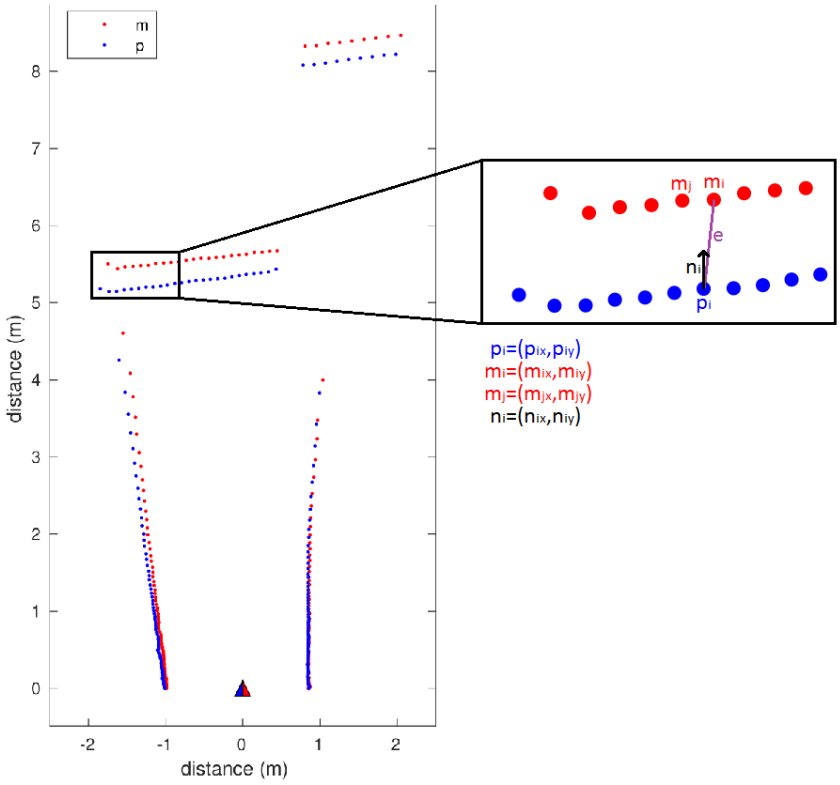


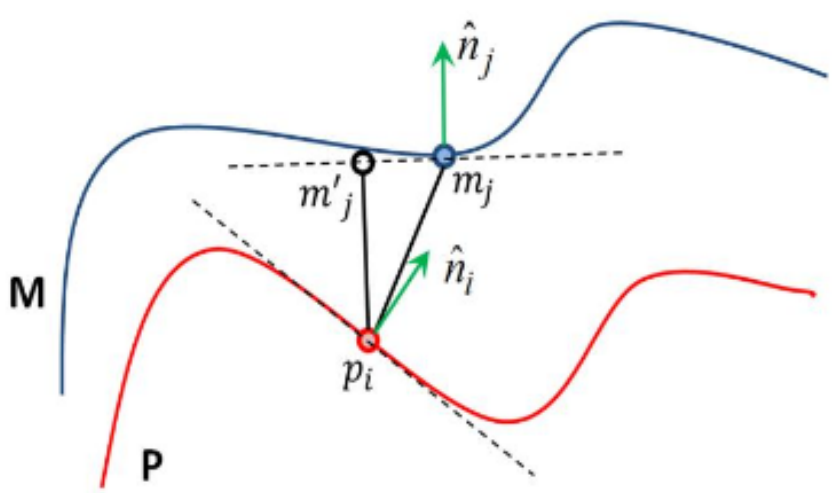
Figure 5.3: Laser scans with correspondence variables

- Closest point (m_{ix}, m_{iy}) to p_i, from the previous observation $(m_i \in \vec{m})$
- Normal vector (n_{ix}, n_{iy}) from p_i to m_i
- Error (e_i) which is the euclidean distance between p_i and m_i

The ICP algorithm always *converges* monotonically to the nearest local minimum of a mean-square distance metric, and experience shows that the rate of convergence is rapid during the first few iterations [13]. An more elaborate explanation of the ICP algorithm is given in Rusinkiewicz and Levoy [51]. The basic algorithm can be divided into three main stages, divided into five sub stages:

1. **Finding**: Constructing correspondence pairs
   - **Selection**: Selecting points in both sensor observations
   - **Matching**: Finding a correspondence between the selected points
   - **Weighting**: Weighting the correspondence pairs appropriately
2. **Rejecting**: Removing pairs that are less likely to be correct correspondences
3. **Minimizing**: Finding the rotation and translation which minimizes the error between the correspondences.

Each stage with different options is explained below into more detail.



(a) Distance to tangent line, taken from [63]

(b) Point to line using two closest points, taken from [16]

Figure 5.4: Point-to-line correspondences

5.1.1 Finding correspondences

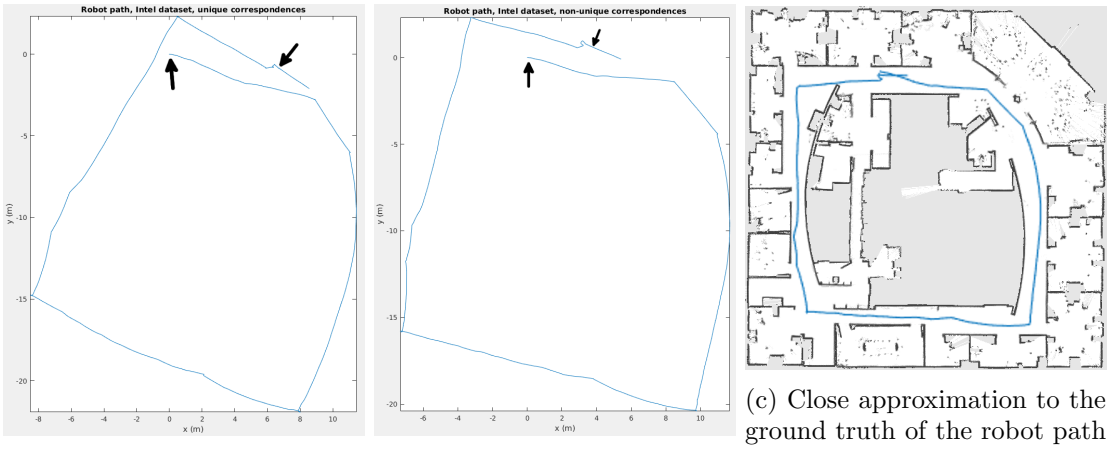
There are different ways of generating correspondence pairs. Rusinkiewicz and Levoy [51] explain several methods for selection and *rejection* of pairs. For selection, one can choose to match all available points from the current observation to the reference observation, or only a subset. It is possible to match multiple points to a single point in the reference observation, vice versa, or to only allow a unique correspondence.

5.1.1.1 Point-to-Point matching

The standard ICP algorithm uses *Point-to-point* correspondence. A correspondence is a match between a point p_i from the current observation $\vec{p}$ and a point m_i from the previous observation $\vec{m}$. All correspondences are all the matches from $\vec{p}$ to $\vec{m}$, which allows for multiple correspondences for each point. If all correspondences are found it is then up to the error minimization part to find the best transformation such that the error between the matches is minimized. Lu and Milios [39] propose two methods for scan-matching, both methods use odomotry as starting estimation for the robot orientation. One method uses the same Point-to-point matching as the standard ICP, but in addition, the squared distance to the origin is taken into account. Using two correspondence sets allows to estimate the registration (rotation and translation) accurately and it converges significantly faster than the standard ICP algorithm. The second method of [39] is matching the tangent directions of two scans. The idea is to compute the tangent directions on both scans. Then associate correspondence of scan points, assuming a known robot orientation. From these associations a translation T is extracted using a least square method (only translation not rotation). The current observation is then translated with the translation T and the rotation is estimated by minimizing the distances. An error minimization approach is described in Section 5.1.4.

5.1.1.2 Point-to-Line matching

Point-to-line matching was proposed by Chen and Medioni [17]. And it is proven that point-to-line matching has quadratic convergence [16]. The standard point-to-line matching algorithm uses the *normal vector* on the tangent line as correspondence, See Figure 5.4a. Figure 5.4b



(a) Robot path on dataset with unique correspondences

(b) Robot path on dataset with non-unique correspondences

(c) Close approximation to the ground truth of the robot path plotted width the map of the environment

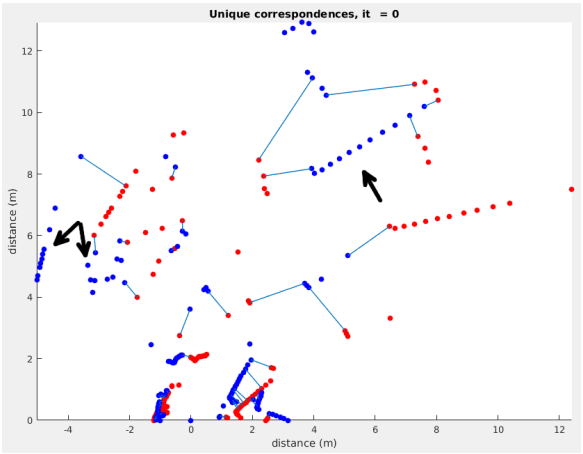
Figure 5.5: Robot path of the Intel dataset without loopclosing. Based on ICP with odometry input + outlier rejection

shows an approach that uses the two closest points in the reference observation without the tangent.

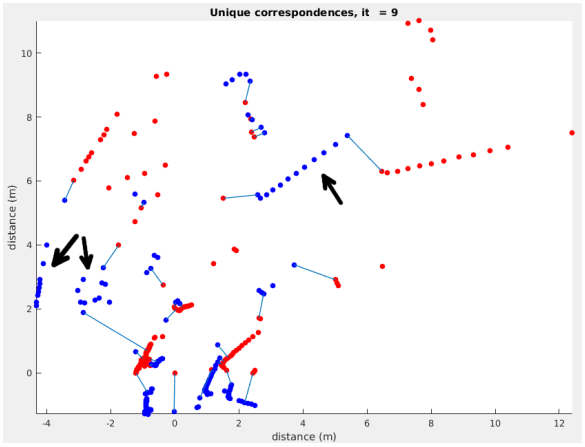
5.1.2 Unique vs non-unique correspondences

For selecting correspondence pairs one can choose to only allow *unique* pairs or allow *multiple* correspondences to a point. In most cases the choice between this does not have a huge impact on the resulting image and iteration steps needed. However, in some cases it has some influence which is shown in Figure 5.5 where the robot path from both *unique* and *non-unique* correspondence selection is shown compared to a close approximation of the ground truth of the Intel dataset. Figure 5.5a shows the result using unique correspondences. The black arrow indicate the area where the robot is in roughly the same position. Most of the time the algorithm performs good enough, but at some keypoints it misses some transformation. Figure 5.5b shows the calculated robot path with the same dataset using non-unique correspondences. In general the algorithm with non-unique correspondences has a more accurate result.

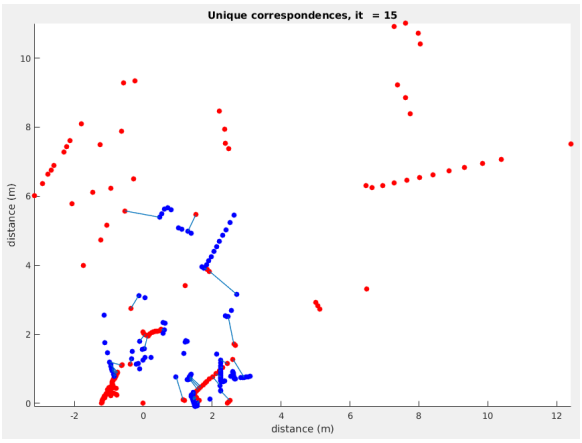
A more detailed look at a specific case shows why the non-unique preforms better in most cases. Figure 5.6 shows a ICP algorithm performed on two observations from a laser range scanner of a specific case where the robot is just driving around a corner. The blue dots represent the new observation data, the red dots represent the previous observation data. Figure 5.6a shows the initial correspondence selction. The blue lines represent a correspondence between two points. It can be seen that there are a lot of points not matched because only one unique pair may exist. Especially the areas pointed out by the black arrows contain few to non correspondences. With very few correspondences each correspondence has a significant impact on the transformation. This is why in the 9th iteration, Figure 5.6b, the the convergence of the points are heading towards a wrong local minimum. The black arrows indicate again the areas where a lot of correspondences are left out because of the unique requirement. After 15 iterations, Figure 5.6c, the solution has converged into a local minimum and the result is a bad transformation. Comparing this same exact case with an ICP algorithm that does not require the correspondences to be unique can be seen in Figure 5.7. Figure 5.7a shows the initial correspondence pairs. Comparing this to the unique case, Figure 5.6a, one can see that the areas around the black arrows are now filled with correspondences. After 4 iterations the points are heading toward the



(a) Initial unique correspondences

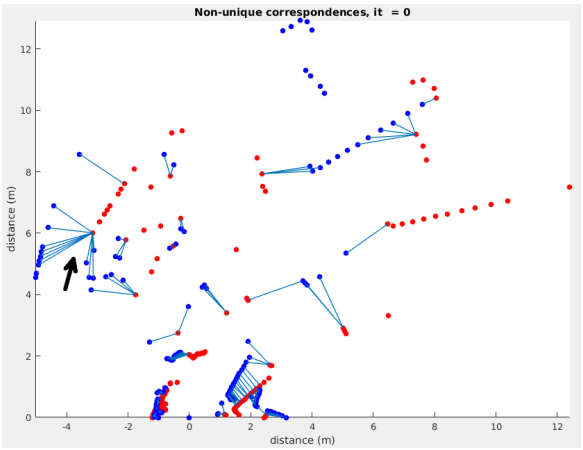


(b) Heading towards local minimum with unique correspondences

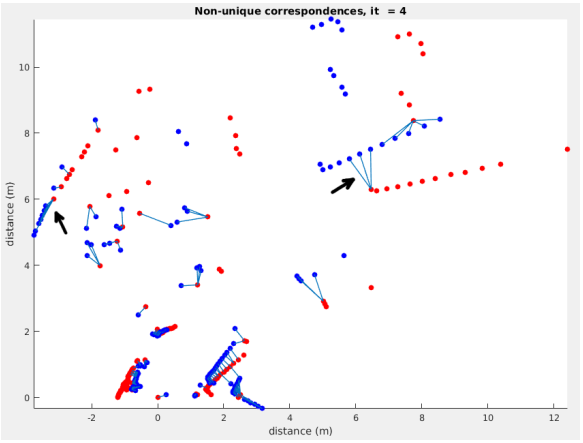


(c) Converged to local minimum after 15 iterations

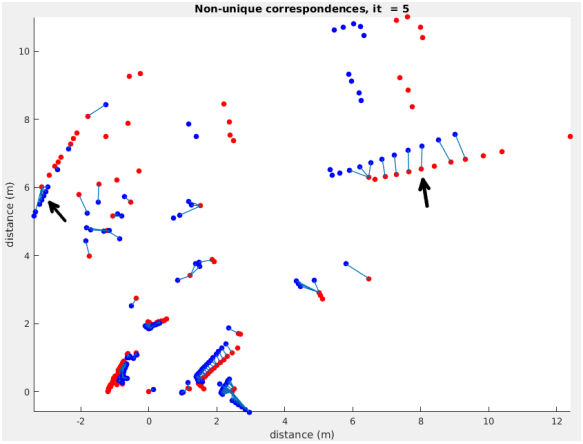
Figure 5.6: ICP iterations for a unique correspondence case



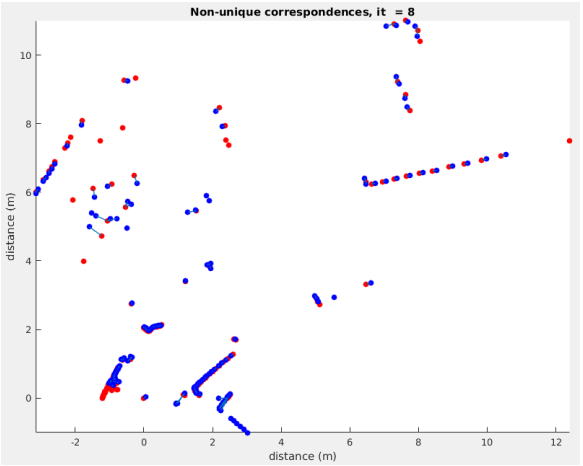
(a) Initual non-unique correspondence pairs



(b) Multiple non-unqiue correspondences are directed into the correct way



(c) New correspondences are close to the correct ones



(d) Converged after 8 iterations

Figure 5.7: ICP iterations for for a non-unique correspondence case

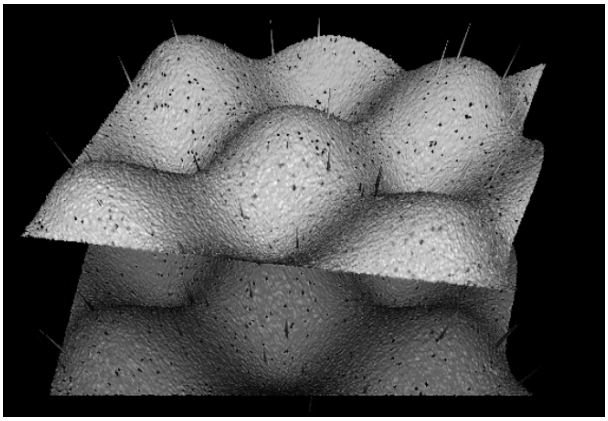


Figure 5.8: "wave" image used in outlier rejection of Rusinkiewicz and Levoy [51]

correct solution. The correspondences arround the black arrows may not be the correct ones but they are pointing in the right direction. Therefore in the following iteration, Figure 5.7c, multiple correspondences are connected to the right, or the neighbor of the right point. After 8 iterations the solution is converged into the correct one, see Figure 5.7d.

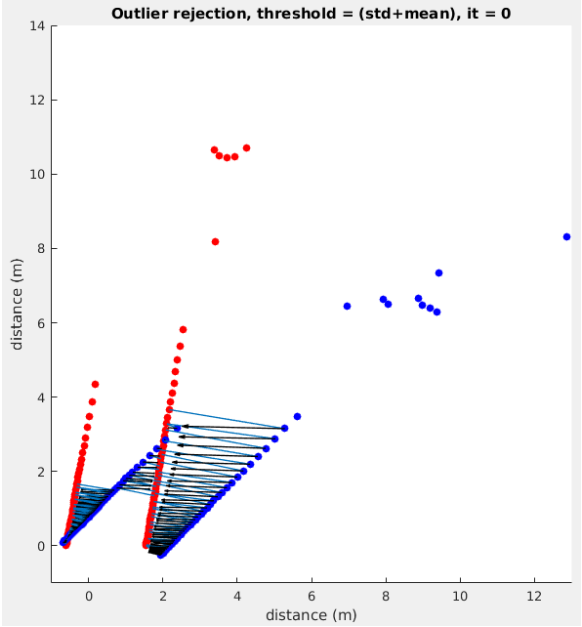
5.1.3 Rejecting correspondences

For *rejection* of pairs a few methods are discussed in Rusinkiewicz and Levoy [51] and in Pomerleau et al. [48]. The methods proposed are the following

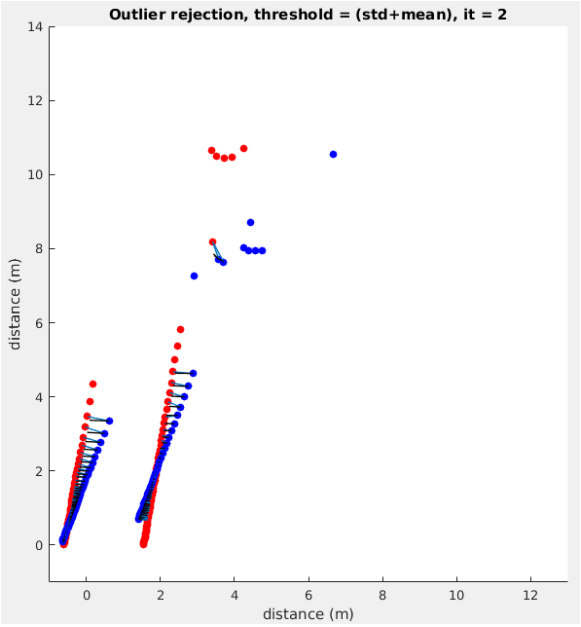
- Rejection of pairs if the distance between them is above a certain user specified threshold
- Rejection of the worst n % of the pairs as suggested in Pulli [50]
- Rejection of pairs whose point-to-point distance is larger than some multiple of the standard deviation of distances [40][16].
- Rejection of pairs that are not consistent with neighbouring pairs, assuming surfaces move rigidly [21].
- Rejection of pairs containing points on mesh boundaries [59].
- Rejection of pairs with a distance larger than the standard deviation + mean of the errors [22].

Rusinkiewicz and Levoy [51] states that, though it may have effects on the accuracy and stability with which the correct alignment is determined, in general does not improve the speed of convergence. However, Rusinkiewicz and Levoy [51] draw their conclusions based on applying an ICP algorithm with different outlier rejection techniques on a 3D image of a wave shown in Figure 5.8, but results on 2D scanner data sometimes differ in accuracy as explained below.

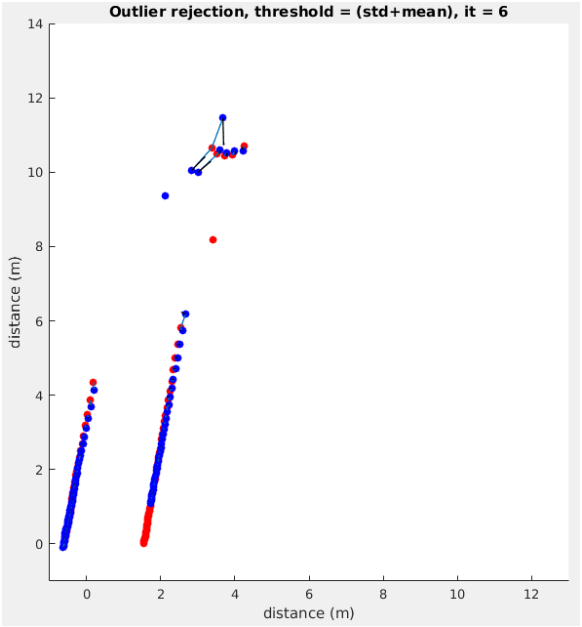
The accuracy difference is in most cases very small but sometimes there can be a noticeable difference. On example of this is given in Figure 5.9 and Figure 5.10. Both figures represent the same input observations. The blue line represents the correspondence to the closest point from observation $\vec{p}$ to $\vec{m}$. The black arrows represents the dot product of the correspondence with the normal vector. One can see the black arrows as pulling vectors on each point. Figure 5.9 shows the case with outlier rejection. correspondences with an euclidean distance between the points that are larger than the threshold are considered outliers. The threshold taken here is the standard deviation together with the mean. One can see that after 6 iterations the solution



(a) Outlier rejection, threshold (std+mean), it = 0

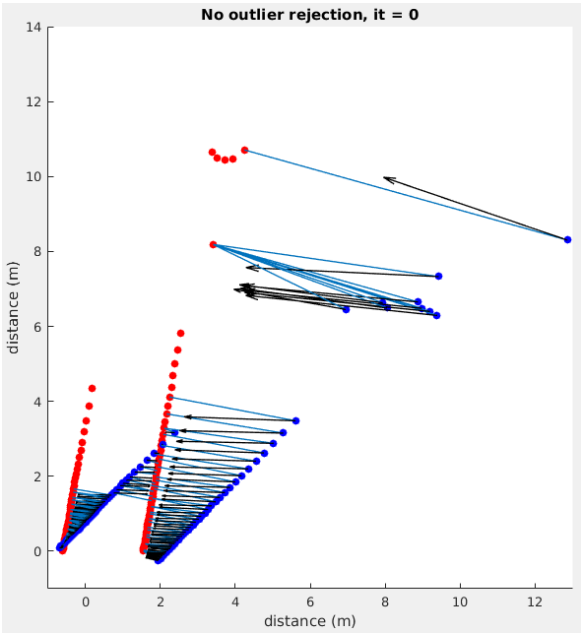


(b) Outlier rejection, threshold (std+mean), it = 2

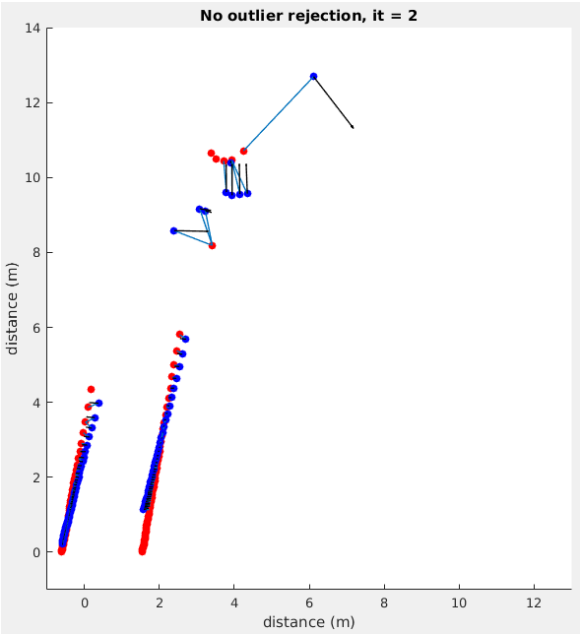


(c) Outlier rejection, threshold (std+mean), it = 6

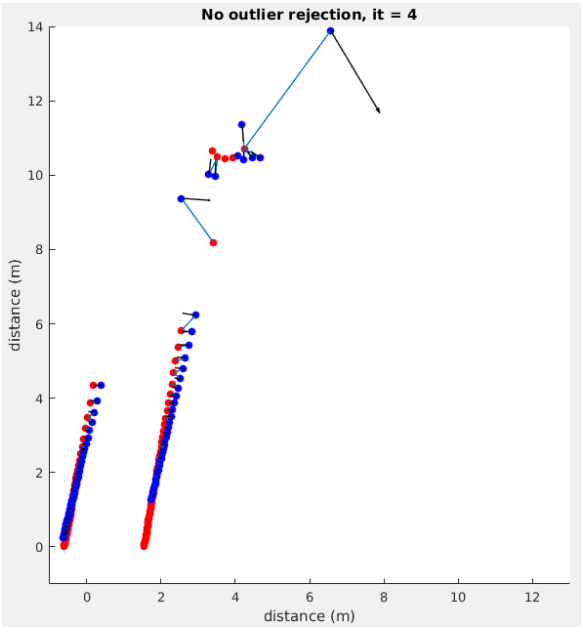
Figure 5.9: ICP iterations with outlier rejection on correspondences



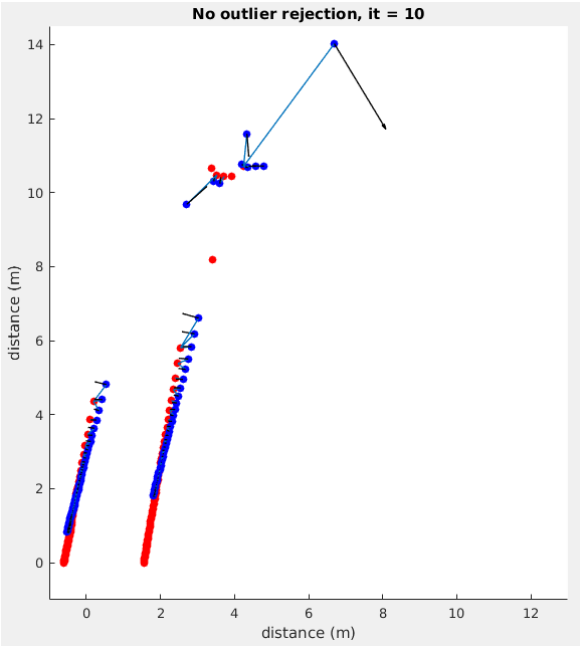
(a) No outlier rejection, it = 0



(b) No outlier rejection, it = 2



(c) No outlier rejection, it = 4



(d) No outlier rejection, it = 10

Figure 5.10: ICP iterations without outlier rejection on correspondences

has converged. Figure 5.10 shows the case of the ICP algorithm without outlier rejection. In the first couple of iterations the results are positive (Figure 5.10a, Figure 5.10b). However in iterations 4 (Figure 5.10c) and 10 (Figure 5.10d) a large outliers is responsible for an inaccurate result.

5.1.4 Minimizing the error

Once correspondences are found, *minimizing the error* can be done in multiple ways. Rusinkiewicz and Levoy [51] mention several methods for error minimization. Solutions based on single value decomposition [9], quaternions [30], and orthonormal matrices [31]. The numerical accuracy and stability are all evaluated by Eggert et al. [23], concluding that the differences among them are small. This chapter describes the following two methods into more detail.

- Closed form solution by reducing the problem to quadratic form by Censi [16]
- Singular Value Decomposition used in the linear least square approach by Low [37]

5.1.4.1 Closed form solution by reduction to quadratic form

In APPENDIX I of Censi [16] a *closed form* solution for the 2D point-to-line metric is given. It assumes that there is already point correspondence, (does not have to be the correct one) and then it calculates the rotation and translation needed in order to minimize the error between the point correspondences. This can be formulated into the following non-linear minimization problem:

$$\sum_i ||(R(\theta)p_i + t) - \pi_i||^2_{C_i} \tag{5.1}$$

where p_i is a point in the current observation, and π_i is the correspondence point in the reference frame. $R(\theta)$ is a 2×2 rotation matrix. t is the translation vector. $p_i, \pi_i, t \in \mathbb{R}^2$. $C_i = w_i I_{2\times 2}$ for point-to-point metric, $C_i = w_i n_i n_i^T$ for point-to-line metric, where w_i is a weight and n_i is the normal vector to the line between $p_{j_1^i}$ and $p_{j_2^i}$. The three dimensional solution (t_x, t_y, θ) is calculated using a four dimensional space, $x = [x_1, x_2, x_3, x_4] = [t_x, t_y, \cos\theta, \sin\theta]$, by imposing the constraint $x_3^2 + x_4^2 = 1$. Using Lagrange multipliers (λ) the following equation represents the rotation and translation:

$$x = -(2M + 2\lambda W)^{-T} g \tag{5.2}$$

where

$$M = \sum_i M_i^T C_i M_i \tag{5.3}$$

$$g = \sum_i -2\pi_i^T C_i M_i \tag{5.4}$$

$$M_i = \begin{bmatrix} 1 & 0 & p_{ix} & -p_{iy} \\ 0 & 1 & p_{iy} & p_{ix} \end{bmatrix} \tag{5.5}$$

However, solving the Lagrange multipliers comes down to finding the roots of an quartic function (4th order polynomial). A detailed mathematical explanation and proof can be found in APPENDIX I of Censi [16].

5.1.4.2 Linear least squares by approximating $\theta \approx 0$

Low [37] proposes a method for minimization using a linear system using *least squares* approach. The object of minimization is to reduce the sum of the squared distance between each source point and the tangent plane at its corresponding destination point (see Figure 5.11). More specifically, if $s_i = (s_{ix}, s_{iy}, s_{iz}, 1)^T$ is a source point, $d_i = (d_{ix}, d_{iy}, d_{iz}, 1)^T$ is the corresponding destination point, and $n_i = (n_{ix}, n_{iy}, n_{iz}, 0)^T$ is the unit normal vector at d_i , then the goal of each ICP iteration is to find M_{opt} such that

$$M_{opt} = \underset{M}{\operatorname{argmin}} \sum_i ((Ms_i - d_i) \bullet n_i)^2 \tag{5.6}$$

M is a 3D rigid-body transformation composed of a rotation and translation matrix shown below:

$$(M)(\alpha, \beta, \gamma, t_x, t_y, t_z) = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

With

$$\begin{aligned} r_{00} &= \cos\gamma\cos\beta \\ r_{01} &= -\sin\gamma\cos\alpha + \cos\gamma\sin\beta\sin\alpha \\ r_{02} &= \sin\gamma\sin\alpha + \cos\gamma\sin\beta\cos\alpha \\ r_{10} &= \sin\gamma\cos\beta \\ r_{11} &= \cos\gamma\cos\alpha + \sin\gamma\sin\beta\sin\alpha \\ r_{12} &= -\cos\gamma\sin\alpha + \sin\gamma\sin\beta\cos\alpha \\ r_{20} &= -\sin\beta \\ r_{21} &= \cos\beta\sin\alpha \\ r_{22} &= \cos\beta\cos\alpha \end{aligned} \tag{5.8}$$

By using a *linear approximation* that $\theta \approx 0$, the following can be approximated: $\sin\theta \approx \theta$ and $\cos\theta \approx 1$. Using the approximation $\alpha, \beta, \gamma \approx 0$, the following transformation matrix can be derived:

$$(M)(\alpha, \beta, \gamma, t_x, t_y, t_z) \approx \begin{bmatrix} 1 & \alpha\beta - \gamma & \alpha\gamma + \beta & t_x \\ \gamma & \alpha\beta\gamma + 1 & \beta\gamma - \alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 1 & -\gamma & \beta & t_x \\ \gamma & 1 & -\alpha & t_y \\ -\beta & \alpha & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

Using the approximated transformation matrix, Equation (5.6) can be rewritten into a least square problem.

$$x_{opt} = \underset{x}{\operatorname{argmin}} |Ax - b|^2 \tag{5.10}$$

This is a standard linear least-squares problem and they propose to use *singular value decomposition* (SVD) for solving. Further mathematical derivation and explanation why the approximation is needed, and how the matrices are constructed can be found in Low [37].

SVD requires one to break down the A matrix into the product of three matrices [11]. An orthogonal matrix U, and a diagonal matrix S, and the transpose of an orthogonal matrix V.

$$A = USV^T \tag{5.11}$$

where $U^TU = I, V^TV = I$, the columns of U are orthonormal eigenvectors of AA^T, the columns of V are orthonormal eigenvectors of A^TA, and S is a diagonal matrix with the square roots of

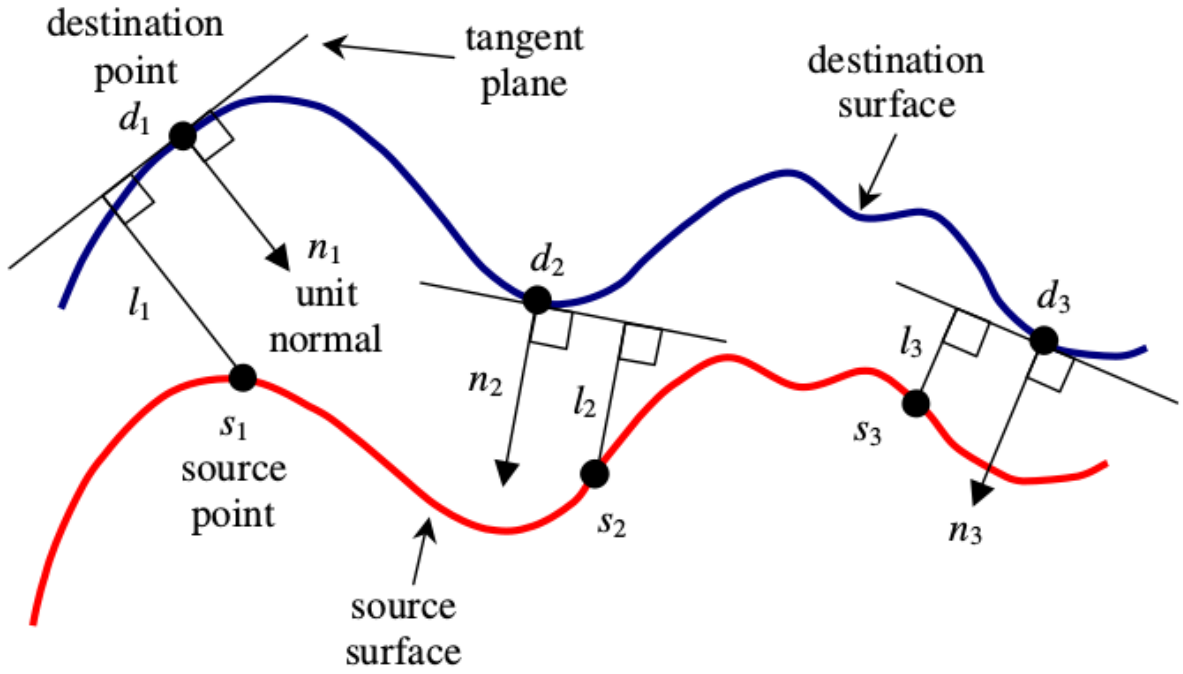


Figure 5.11: Point-to-plane error between two surfaces, taken from [37]

the eigenvalues from U or V in descending order. Solving a linear least square problem with SVD goes as follows [36]:
Given the linear least squares as:

$$\underset{x}{\operatorname{argmin}} |Ax - b|^2 \tag{5.12}$$

Using the orthogonality of U and V, $r = rank(A)$, and the size of $A = m \times n$, the system can be written down as follows:

$$|Ax - b|^2 = |U^T(AVV^T x - b)|^2 = |S\underbrace{V^T x}_{=z} - U^T b|^2 = \sum_{i=1}^{r}(s_i z_i - u_i^T b)^2 + \sum_{i=r+1}^{m}(u_i^T b) \tag{5.13}$$

The minimum norm solution with $i = 1, ..., r$ is

$$x_{opt} = \sum_{i=1}^{r} \frac{u_i^T b}{s_i} v_i \tag{5.14}$$

Further mathematical derivation and proof can be found in Leykekhman [36], Golub and Reinsch [24], and Baker [11].

5.1.5 Scoring the outcome

5.1.5.1 Separate scoring for key and non-key scans

Not every scan-match has the same certainty or is converged to zero error. In constructing a plot one might want to keep track of the quality of a scan-match. Guo et al. [28] suggest two scoring methods. One for key scans and one for non key scans. A scan is considered a key scan if they are for example used in loop closing. Non key scans are most of the time consecutive scans.

$$Score1 = \omega_1 \cos(m_e) + \omega_2 \sin(c_r) + \omega_3 \sin(v_r) \tag{5.15}$$

$$Score2 = \omega_4 \sin(Score1) + \omega_5 \sin(k_i) + \omega_6 \sin(s_i) \tag{5.16}$$

where:

- m_e = Mean correspondence error enlarged by a limit value.
- c_r = Ratio of correspondences to valid points in scans
- v_r = Ratio of visible points from current scan and matching observation
- k_i = Reciprocal of key scan index (0 if not key scan)
- s_i = Reciprocal of scan index
- ω_i = Weights ($\omega_1 + \omega_2 + \omega_3 = 1; \omega_4 + \omega_5 + \omega_6 = 1$)

Score1 is used to score the match for every scan, regardless of whether it is a key scan or not. *Score2* is used in matches where a key scan is also taken into account.

5.1.5.2 Generalized score

Using the ideas of Guo et al. [28] and the pair rejecting techniques described in Section 5.1.3 an own scoring metric was created using the following formula:

$$score = \sin(c_r \frac{pi}{2}) \frac{1}{1 + \tau \frac{\sum e_i}{||e_i||}} \tag{5.17}$$

where:

- c_r = Ratio of correspondences to valid points in scans
- τ = Factor to scale the mean error
- e_i = Vector containing all the errors

6 — CλaSH

The number of transistors on a chip has been rapidly growing for several decades as predicted by Moore's Law. Because of physical limits, the size and therefore the number of transistors per area is starting to reach limitations. Many techniques have been introduced to increase the number of transistors on a chip by putting multiple instances of chips into one package, or using alternatives to normal processors. The Field Programmable Gate Array (FPGA) is a reconfigurable chip which allows the developer to create dedicated architectures for specific applications. Because FPGAs can be used to do computations in parallel, FPGAs potentially have better performance compared to standard processors in terms of time and energy.

Hardware descriptions for FPGAs are often created in low-level hardware description languages such as (Very high speed itegrated circuit) Hardware Description Language (VHDL) and Verilog. Although HDLs have been the standard for years, these languages are complex and specific. Research focuses on finding alternatives that enable designers to create hardware descriptions by using higher level languages. The problem with most program languages however, is that they do not have (enough) separation between time and area, which happen to be the most important factors in hardware design. By (mis-)using programming languages for hardware design, the programming language is used as description languages which means they become indirect and have high levels of abstraction, which makes making changes at lower levels difficult.

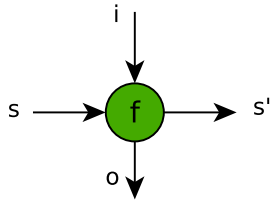
CλaSH[10] is a programming language, developed by the Computer Architecture for Embedded Systems group, which is based on the functional programming language Haskell[6]. CλaSH allows developers to transform strongly typed mathematical specifications to low-level hardware descriptions. Because it is possible to describe structures in a mathematical way and also in terms of time and area, CλaSH is suitable for implementing mathematical algorithms such as SLAM. Within the CλaSH language, parallel operations on data can be performed using *higher-order functions*(HOF's). An example of a higher order function is the map-accumulate function. The map-accumulate function can be used to find a parallel outcome of operations of a list on one output, and an accumulated version of a list on another.

A basic example of the map-accumulate function is shown in figure 6.2a. Each higher-order function consists of a structure and a function. The functions contains two inputs and two outputs which work independent of each other. An example of such a function is shown in Figure 6.1. Horizontally the inputs are added and vertically each stage produces a product of the input i and state s.

Timing in hardware is defined by a *clock signal*, the clock signal is a signal that consists of transitions with a static period. The clock signal can be used in circuits and a *clock-triggered system* is called *synchronous*. An example of the clock-triggered system is the *register*. A register has one input and one output. The register copies the input to the output when the clock

Listing (6.1) Function with two inputs and outputs

```
f s i = (s', o)
  where
    s' = s+i
    o = s*i
```

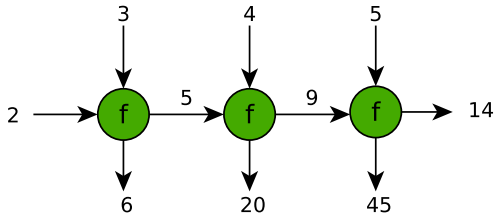


(a) Function adding the inputs horizontally and multiplies vertically

Figure 6.1: Basic function with two inputs and two outputs and corresponding code

Listing (6.2) Map accumulate function

```
f s i = (s', o)
  where
    s' = s+i
    o = s*i
mapAccum f 2 [3,4,5]
```



(a) Example of the map accumulate function with values filled in

Figure 6.2: Map accumulate example and corresponding code

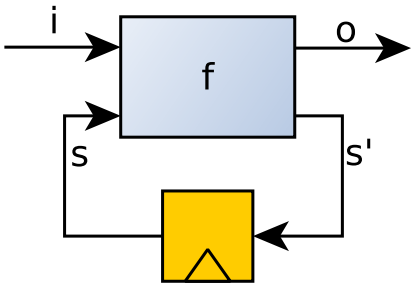
makes a transition. The output is stored until the next transition of the clock. A register can be used to store variables over time. The combination of a combinational path that changes the output of the register into another input for the register combined with the register itself makes a complete synchronous system.

The functionality of the map accumulate can also be described a synchronous system. A system with the same inputs and outputs as the map accumulate function is called a *Mealy machine* . A mealy machine contains a combinational path which creates a next state and an output from the previous state and input. Figure 6.3a shows a basic mealy machine with input i, output o, current state s, and next state s'. The CλaSH code is shown in Listing 6.3 where the function f could be the function from the previous example or any other mathematical specification.

With f being a more complex function which requires control (for reusing hardware for example) a mealy machine can be used to control the input. The mealy machine does also control the datapath so it performs the required operation. An example of mealy machine which controls memory and a datapath is shown in Figure 6.4a and the corresponding code in Listing 6.4. *BlockRAM* is a type of synchronous memory that is available on an FPGA. The width of the data that is written to and read from the blockRAM is configurable, the depth of a blockRAM is FPGA specific.

Listing (6.3) Mealy machine

```
f s i = (s', o)
  where
    s'= ..
    o = ..
mealy f s0 i0
```



(a) Mealy machine with input i, output o, current state s, and next state s'

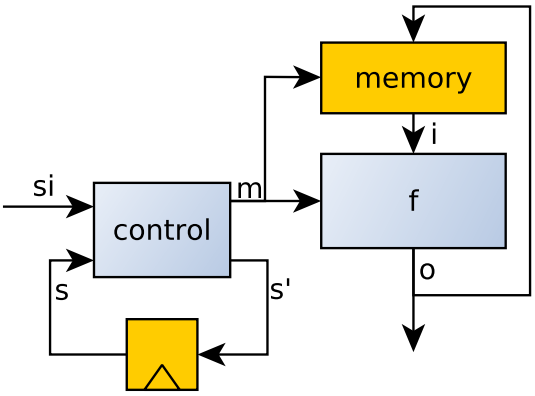
Figure 6.3: Mealy machine and corresponding code

Listing (6.4) Mealy machine for control

```
topEntity = o
  where
    m = mealy control s0 si0
    i = blockram o
    o = f i m

f i m = o
  where
    o = ..

control si s = (s', m)
  where
    s'= ..
    m = ..
```



(a) Mealy machine controlling f and memory

Figure 6.4: Mealy machine controlling external memory with corresponding code

Part II

Design Space Exploration

7 — Introduction to decision trees

The research part of this document discusses different approaches to realize a SLAM solution. Many choices must be made to be able to implement a solution feasible for an FPGA. To document the fundamental choices of this project a tree structured representation of design space exploration (DSE) approach is used. Every choice made opens up new choices and every solution found provides new problems to be solved. A tree structure can represent these choices and influences on different abstraction levels where the level of detail increases from top to bottom. Since the size of the entire decision tree is too large to fit onto a single page the tree is split up into three parts. The first, shown in Figure 7.1, represents the initial choices which are split up into two parts, Figure 7.3, and Figure 7.2 representing the scan-matching and Graph-based SLAM parts respectively. Note that the tree contains references to the different chapters in this section where choices are explained. The ellipse shaped nodes represent choices, the rectangular shaped nodes represent the different solutions, and the green colored rectangular nodes show the chosen solution among its alternatives. The tree can be used as an index and overview of how the different sections of this document part can be linked together.

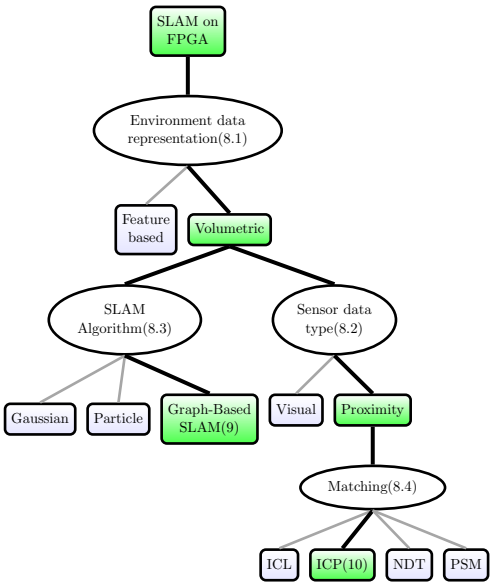


Figure 7.1: DSE tree global

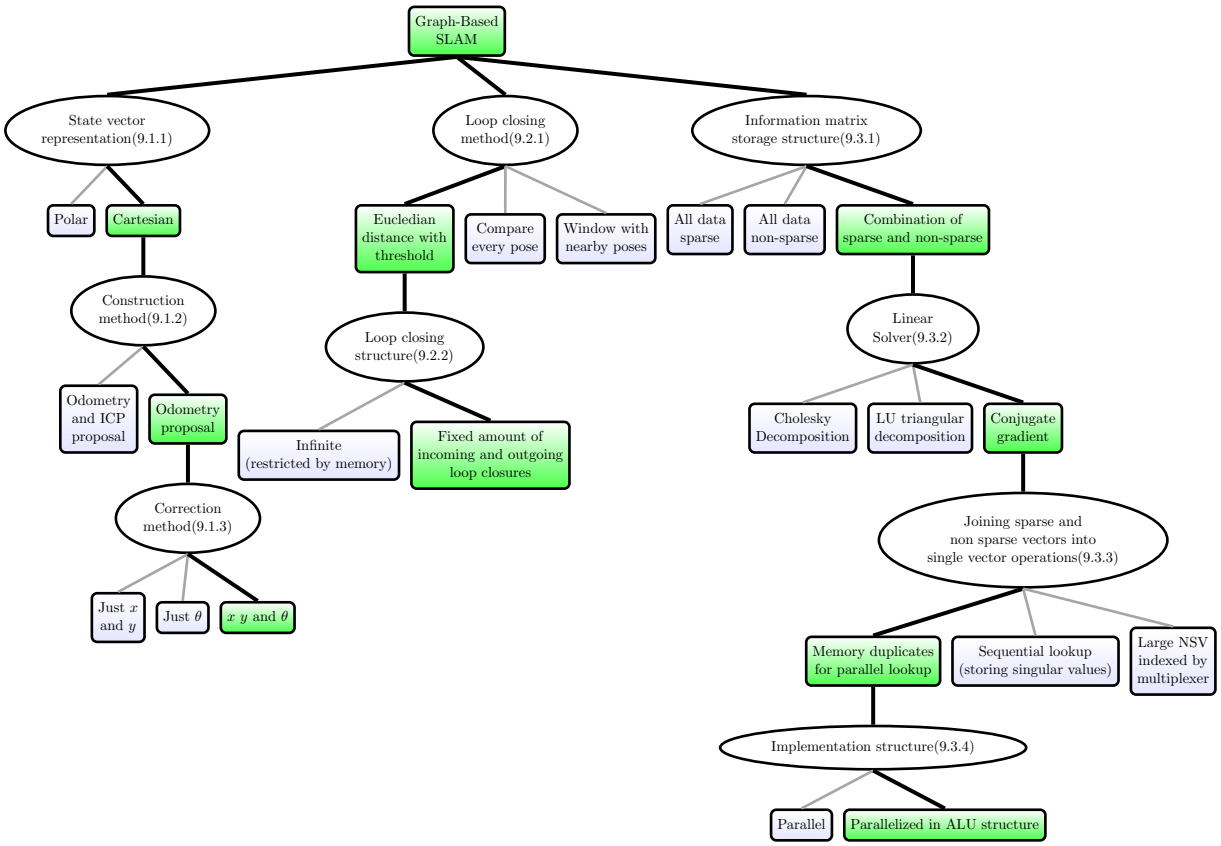


Figure 7.2: DSE tree Graph-based SLAM

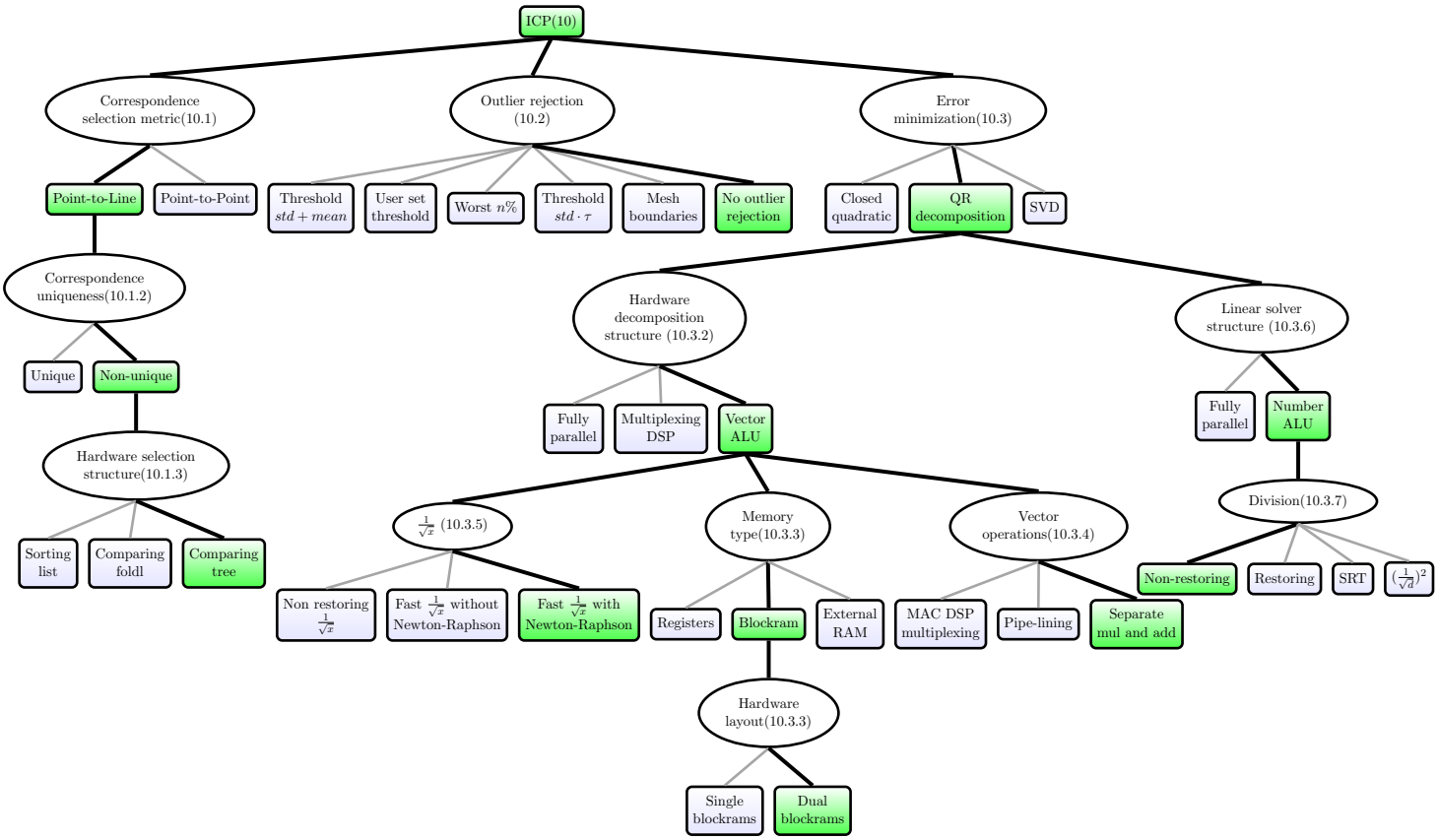


Figure 7.3: DSE tree scan-matching

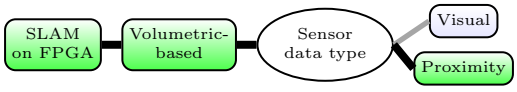
8 — Exploration on SLAM properties

8.1 Environment data representation

One of the goals of the project is to have an environment representation that is usable for navigation, for navigation a robot needs to know whether a certain place is occupied or not. Volumetric representation does this by definition. If the map produced by the SLAM solution has a high enough resolution it is suitable for navigation. Feature-based approaches extract sparse information form a sensor stream [34]. The map produced consist only of extracted features and thus some information from the sensor is discarded.

The choice was made to use the volumetric data representation because it represents the physical structure of the real world and thus better suitable for navigation purposes.

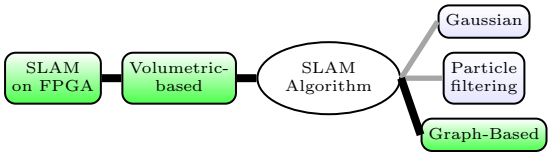
8.2 Sensor data



The choice for using the volumetric approach also effects the sensor choice. Proximity sensors are a good choice because the sensor data already represents physical distances which have a volumetric representation. Camera sensors could be used but this introduces the problem of data association. Appendix F gives short introduction to some of the vision solutions for data association but this is an extensive field of research. Most data association algorithms also require a lot of computational resources. So the choices for proximity sensor in combination with volumetric based representations are a matter of time, complexity and resources. In robotics the two popular sensors are the ultrasonic sensor and laser-range scanner. Laser range scanners are known for their high accuracy but are unable to detect transparent and highly reflective objects. In full sunlight the laser range scanner is subject to noisy data. An advantage of the laser range scanner is that a lot of the current available datasets also use the laser-range data [3]. The ultrasonic sensors are able to detect transparent and reflective objects but are know for there unreliable and inaccurate measurements.

The choice was made to use the laser range scanner due to accuracy, reliability, and dataset availability. Proximity data from either a dataset or laser range scanner has the beam-format.

8.3 Choice of filtering technique



As described in previous chapters, there are three different approaches to solve the SLAM problem. To choose between the different approaches, an overview of the advantages and disadvantages of the different algorithms is show below. The Gaussian based approaches have a lot of the advantages and disadvantages in common and are treated as one approach.
Gaussian based SLAM (dis)advantages:

- ⊕ Few landmarks gives a good result
- ⊖ Data association with two methods: point for point, or feature detection
- ⊖ All of the probabilities are Gaussian distributions
- ⊖ Matrices grow quadratically with the number of landmarks
- ⊖ Laser-data represents points and not specific landmark types, so conversion is needed

Particle filter based SLAM (dis)advantages:

- ⊕ No state explosion
- ⊕ Easy quality improvement by adding particles
- ⊕ Complexity $\mathcal{O}(nm)$ or with tree based data storage $\mathcal{O}(n \log m)$ (better than Gaussian or graph-based)
- ⊕ With good proposal functions few particles will give good result
- ⊖ Multiple storage of same data
- ⊖ Environment without loop closing requires significant more particles

Graph-based SLAM (dis)advantages:

- ⊕ Quality of the map increases over time due to error convergence
- ⊕ Separation of concerns in back-end and front-end
- ⊕ Full slam, complete path gets optimized, when error is found
- ⊕ HOG for efficient computations and hierarchical memory usage
- ⊕ Information-Theoretic Graph pruning for efficient memory usage
- ⊕ Matrix equation can be solved with other techniques than inversion
- ⊖ Error is converged by iteration, which makes the algorithm less deterministic

These advantages and disadvantages give a global overview of the possibilities. The following section gives a more detailed description and argumentation for a choice. For an implementation on a FPGA a couple of criteria are established in order to choose one of the SLAM solution approaches. The following criteria are defined and explained below:

- Memory/Computational scalability
- Non-linearity
- Easy of implementation
- Easy and effect of optimization of parameters and implementation
- Quality of the map

On an FPGA there is always a trade-off between time and area. Multiple data dependent operations will result in a longer combinational path and will reduce the clock frequency. One might choose to have intermediate states (pipe-lining) to reduce the longest combinational path and improve the clock frequency. To store intermediate states memory is needed, be it external or on-chip. The criteria: memory scalability, and computational scalability represents these constraints.

Some SLAM algorithms handle non-linearities better than others. This criterion is added because the environment and robot movements are often non-linear.

The main goal of this project is not to find the best SLAM solution and implementation but to realize a solution on an FPGA. This means that ease of implementation is also a criterion.

In most SLAM solutions parameters must be determined in order to get a better result, also implementation can sometimes be optimized in terms of area and time. The criterion "optimization of parameters and implementation" represents this aspect.

As stated in the problem definition, the produced map must be usable for navigation. Some algorithms have a trade-off between complexity (memory and computation) and quality of the map. Therefore the quality criterion is added.

Table 8.1 shows the score of each SLAM solution, the following section gives a description of the weighted scores. Note that the scores are relative to each other and are based on volumetric based SLAM using a proximity sensor.

8.3.0.1 Non-linearity

Kalman filter and information filters are designed to solve linear problems and are by definition not suitable for non-linear situations. The extended versions of these filters use linearisation to approximate. The UKF does not use approximations and with enough Sigma points the performance on the non-linearity is better than EKF or EIF. The particle filters are by definition insensitive to non-linearities. Even proposal functions do not have to be linear. In graph-based SLAM is also not subject to errors with non-linearities because it uses error approximations between poses. Appendix D describes the process of robots taking over the world. Also, the proposal function operates under the same principle as the particle filtering version.

8.3.0.2 Memory and computational scalability

For volumetric based approaches the Gaussian-based SLAM solutions, except the SEIF, score worse on scalability because without data association the covariance matrices grow very fast. Operations, like inverting, and storage of these large matrices involve both a lot of computations and memory (grows quadratically). Because SEIF uses sparse matrices both the memory usage

and computations can be optimized. In case of particle filtering there is a trade-off between memory and computational resources. One could choose to realize a fully parallel particle filter at the cost of a lot of area for computation. If particles are calculated over time it takes more memory to store the entire map for each particle for each iteration. Because particle filtering only represents the last known position and the entire map for each particle it is more efficient, in both memory and computation, compared to the Gaussian-based solution. The trade-off between memory and computational scalability is also visible with the graph-based solutions because HOG has an increased computational complexity but reduced memory usage with its hierarchical representation of its data. The graph-based solution does not store the map, but all the poses with corresponding proximity data. Depending on the environment and the path of the robot this method could be more efficient than particle filtering. Intuitively, if the robot continuously sees an already known world then the accuracy will improve, but the memory efficiency in graph-based approaches is worse than particle filtering. This is due to the fact that with graph-based approaches the new poses will be added to the pose graph, while in particle filtering the memory usage of each particle will stay roughly the same.

8.3.0.3 Ease and effect of optimization of parameters and implementation

For Gaussian filters the parameters are relatively easy to find. In most cases it comes down to finding the right matrices which describe the state transitions of the robot. However in UKF and SEIF it is harder to determine the right sigma points and sparsifications which leads to a more efficient storage and computation. For particle filtering the optimization step is very important. Parameters like: the number of particles, the weight calculations of particles, and the re-sampling threshold are hard to determine but has a significant impact on the quality, and resource usage.

8.3.0.4 Ease of implementation

Standard KF and IF are easy to implement because they deal with linear situations with constant matrices. EKF and EIF are harder to implement because the linearisations use matrices containing approximation functions. SEIF is harder to implement because of the sparsification using manipulators to filter out passive landmarks. FastSLAM1.0 is easier than FastSLAM2.0 because it only uses odometry as proposal function for its particles. FastSLAM2.0 also uses its observations which means that it contains scan-matching. Graph-based SLAM also uses scan-matching for its loop closing. For HOG optimization, multiple layers of poses must be constructed.

8.3.0.5 Quality of map

Quality of the resulting map is a difficult thing to measure, as a constraint the map should be usable for navigation within the environment. Wrong loop closing is the main reason for errors that cause the map to be unusable for navigation. The quality of the EKF and EIF is higher than the quality of the UKF with an environment that can be linearised correctly. The quality of the particle filter does highly depend on the number of particles and the parameter optimization, but the quality of FastSLAM1.0 is usually worse than FastSLAM2.0 due to the less accurate proposal function. Because graph-based solutions address the full SLAM problem, the errors will converge over time.

8.3.0.6 Conclusion: Graph-based SLAM

Computational complexity, separation of concerns, scalability, and optimization methods are the main reasons why graph-based SLAM is the most promising method. The computational complexity of the graph-based approach depends on the number of poses while in other methods it depends on the number of landmarks. In most cases there are more landmarks than poses. The separation of the front-end and back-end in the algorithm allows for outsourcing or even excluding computations. One could choose to do the map construction based on the known poses on external hardware. Another advantage of graph-based SLAM are the various ways of optimizing for memory and computational resources. HOG, and the Graph pruning of Kretzschmar et al. [35], allow very efficient optimizations and putting constraints on resources without significant loss of quality and accuracy.

		Complexity	Non-linearity	Scalability		Optimization of parameters and implementation		Ease of implementation	Quality of the map
				Computatinal	Memory	Ease	Effect		
Gaussian	KF	$\mathcal{O}(n^{2.4})$	- -	- -	- -	+	-	+	- -
	EKF	$\mathcal{O}(n^{2.4})$	- +	- -	- -	+	-	+	+
	UKF	$\mathcal{O}(n^{2.4})$	+	- -	- -	-	+	- +	- -
	IF	$\mathcal{O}(n^{2.4})$	- -	- -	- -	+	-	+	- -
	EIF	$\mathcal{O}(n^{2.4})$	- +	- -	- -	+	-	+	+
	SEIF	$\mathcal{O}(n^2)$	- +	- +	- -	-	+	-	-
Particle	FastSLAM1.0	$\mathcal{O}(n \log m)$	+ +	- +	- +	- -	+	+	+
	FastSLAM2.0	$\mathcal{O}(n \log m)$	+ +	- +	- +	- -	+	+	+ +
Graph	Standard	$\mathcal{O}(k^2)$	+ +	+	+	- -	+	+	+ +
	HOG	$\mathcal{O}(k^2)$	+ +	+	+ +	- -	+	+	+

Table 8.1: An overview of different properties of SLAM n = number of landmarks, m = number of particles, k = number of poses

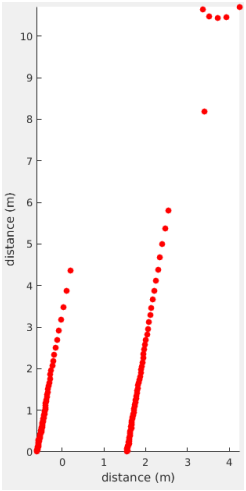
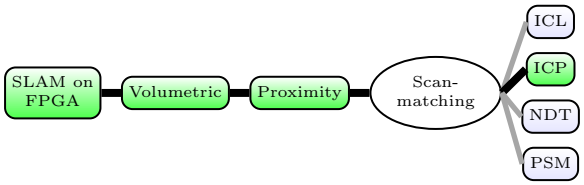


Figure 8.1: A common hallway sensor observation from a laser scan

8.4 Scan-matching



Scan-matching is the process of determining robot *movement* using *observations*. The common use of scan-matching is matching the current sensor observation with the previous one in order to determine the transformation between those two. This transformation is the distance travelled by the robot. Different ways of scan-matching are discussed in Chapter 5 and Appendix C, below follows an explanation of the choice for a scan-matching algorithm.

8.4.1 ICL

Iterative Closest Line requires one to extract features like corners and lines from a data set. ICL is most of the time applied on large datasets like pictures or 3D models. Extracting features reduces a stream of data into sparse information. Laser scans often consist of 180, 270, or 360 points which is already sparse information about the environment and therefore extracting features could likely end up with to little information for determining the translation.

8.4.2 NDT

Normal Distribution Transform has a similar problem as ICL. The world around the robot is discretized into a grid, and then the information from the laser scan is projected into the grid. If a grid cell contains at least three scan points the occupancy probability distribution is constructed. Many laser scanners return a distance with a step of 1 degree, which means that points that are far away from the robot are less clustered compared to the points that are close to the robot. A hallway for example contains a lot of points on the side walls but not in the middle, see Figure 8.1. The points on the side wall are good for calculating the rotation. However, for determining the translation the points in the middle are important. In the NDT it is likely that the points to determine the translation are left out because it requires multiple points in a grid cell.

8.4.3 ICP

Given two datasets of points the Iterative Closest Point algorithms will find the transformation between the datasets and will handle every datapoint given by the beam sensor model as an individual point without sparsifying the information (as done in ICL), so no data association, extraction, or other processing is needed. There are a lot of different ICP algorithms available each with its mathematical structure.

8.4.4 Conclusion: ICP

Because there is no need to process the sensor observations and the different mathematical structures possible to solve the scan-matching problem the choice was made to create the ICP and to test different rejection and error minimization techniques. The easy of implementation and the large amount of research already available in this area also contribute to this choice.

9 — Graph-based SLAM

Mathematical definitions and the most important aspects of graph-based SLAM have been discussed in Chapter 4. Implementation of these mathematical definitions can not be carried out without exploration of the possible approaches. The following chapters will describe the exploration for the important choices that have been made before starting an implementation. The design choices at higher levels have large influence on the complete implementation, while the lower level choices barely have influence on the final design. However, it can be the case that a higher level choice has influence on the implementability of the a lower level choice, and has an indirect large effect when it would be implemented in a way that is out of the scope of the design space.

9.1 Graph representation

9.1.1 State vector representation

The two possible coordinate systems to represent the state vector in are the Cartesian coordinate system where each point is described by an distance that it is away from the base along two or three axes that are orthogonal to each other. The state vector can also be represented in the polar coordinate system. The polar coordinate system describes absolute angles and relative distances instead of absolute position coordinates. The angle is absolute which means there is a global frame in which the angles are expressed. The distances are relative which means the distance between the current robot pose and the next robot pose is expressed. The choice of the state vector mostly depends on the level of difficulty to work with them. The coordinate system which demands the least amount of computational load to solve the SLAM problem can be considered the best choice.

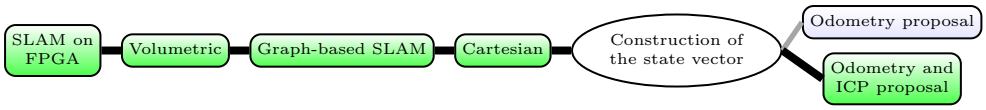
The format of the state vector matters whenever the state vector is used for computations. This happens when the state vector is created, but also when it is updated during convergence of the error in the SLAM algorithm. Creation of the state vector in Cartesian coordinates has been discussed in Section 4.1.2. However, creation of a state vector in polar coordinates is much easier since it is very closely related to the format odometry data gets presented. The difference however is that odometry data will present relative angles that the robot has moved, while the state vector needs to represent angles in the reference frame (2-Dimensional xy-frame). Summing the angles will directly result in the state vector that represents the robot path.

Considering the way error convergence is carried out described in Section 4.1.6, it is only possible to correct an error based on absolute values, which are always x, y and θ relative to the reference frame. The number of dimensions in the state vector that can not immediately

be corrected by a vector addition is only one (θ) when working with Cartesian coordinates and two when working in polar coordinates (x and y). However, the amount of conversions is the same since the graph needs to be converted to polar and then back to Cartesian or the other way around. The conversions will be done as described in Section 4.1.7.

The map that is constructed at the end of the algorithm can not be represented in polar coordinates the same way as the state vector. The map should always be represented in a global coordinate frame which the Cartesian state vector is already in. Also for human readability, showing the map or a path in Cartesian coordinates is much more intuitive, since it directly represents a 2-D environment in meters instead of angles and meters relative to the base of a coordinate system.

9.1.2 Construction of the state vector

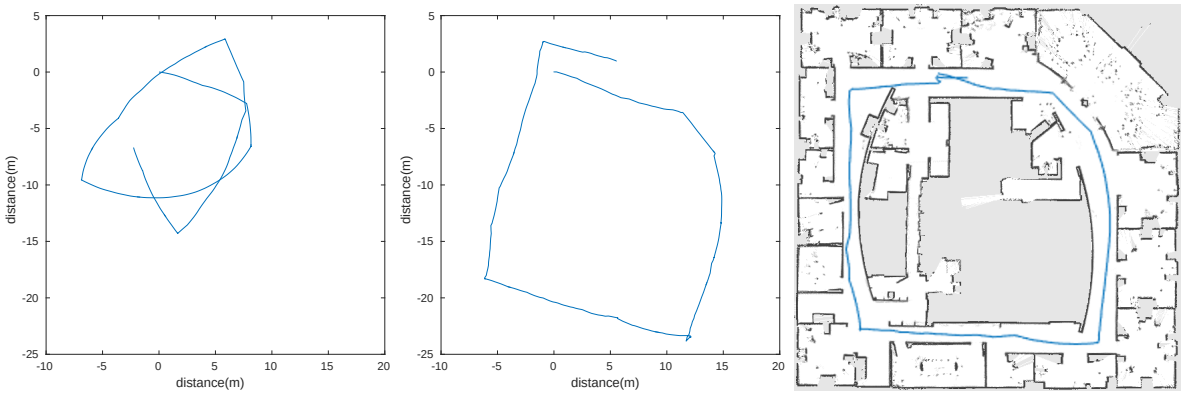


The state vector represents the poses the robot has been and will eventually be used as a basis for mapping. The state vector is constructed with sensor data between each two sequential poses which has already been discussed in Section 4.1.2. Odometry sensors introduce noise which results in bad proposals for loop closing. An observation sensor can not only be used when loop closing has occurred, but also to improve the initial path created by odometry. The transformation between two poses was defined by a translation and a rotation. Using the combination of odometry and a scan matching algorithm, in this case the ICP algorithm, a transformation which is created by combining odometry and ICP can be created. The laser scans that will be passed to the ICP algorithm are represented by m as the model and p as the (current) data. The data is matched with the model to obtain a translation and rotation, called a transformation. By first applying the transformation found by odometry on the laser scan data, the transformation that the ICP algorithm needs to find becomes smaller. The laser scan data transformed by odometry is called p'. The transformation the ICP algorithm returns can be added to the transformation. Using this method the transformation between two poses can be calculated in an accurate and computationally light way. The transformation between two poses ($\vec{s}_i$ *and* $\vec{s}_{i+1}$) in a graph can be written down in a mathematical way:

Algorithm 2 Combination of odometry and icp

1: $\mathbf{H}_{odo} = \begin{bmatrix} cos\theta & -sin\theta & x \\ sin\theta & cos\theta & y \\ 0 & 0 & 1 \end{bmatrix}$
2: $p' = \mathbf{H}_{odo} p$
3: $\mathbf{H}_{icp} = icp(m, p')$
4: $\delta s_{\theta,(i,i+1)} = atan2(sin\theta, cos\theta)$

The above algorithm consists of four steps, in the first step a transformation matrix is constructed from the odometry command which can be used in the second step. In the second step the transformation matrix can be multiplied with the model which is the oldest of two the scans. Because of this multiplication, the model is already translated and rotated to easily match the new scan. The error that remains between the scans is the error of odometry, but since it is only one step of odometry, this error is rather small and because it is corrected by the ICP algorithm every step, it becomes even smaller. The third step is running the actual ICP



(a) Pose graph created by odometry

(b) Pose graph created by odometry combined with ICP

(c) Map of the corresponding environment and highly corrected path

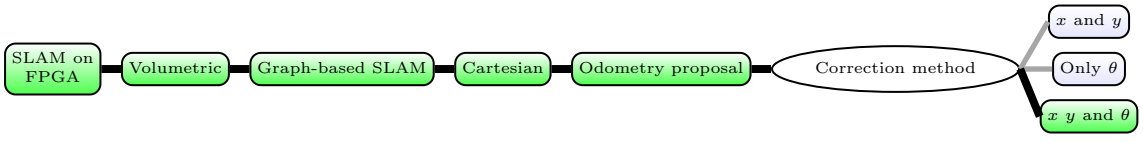
Figure 9.1: Initial pose graph without loop closing created by odometry and odometry combined with ICP

algorithm in which the transformation between the odometry transformed model and the new laser scan is calculated. From this transformation, the difference in θ can be calculated with the $atan2$ function, which is the a normal tangent function which also considers the quadrant of the angle based one both the sine and the cosine of an angle. The robot path with ICP assisted odometry has far less error than the version created with odometry only.

Figure 9.1 shows resulting robot paths based on odometry and ICP assisted odometry. The map is included to show that the ICP assisted odometry is a lot better. The robot path is rotated, this is not an error. Because the mapping will also happen in the same skewed way, the map is essentially the same. The pose graph that is merely based on odometry does show artifects in rotation, but also in translation, which can be seen by the length of the lines, which are significantly shorter than the lines of the ICP assisted odometry.

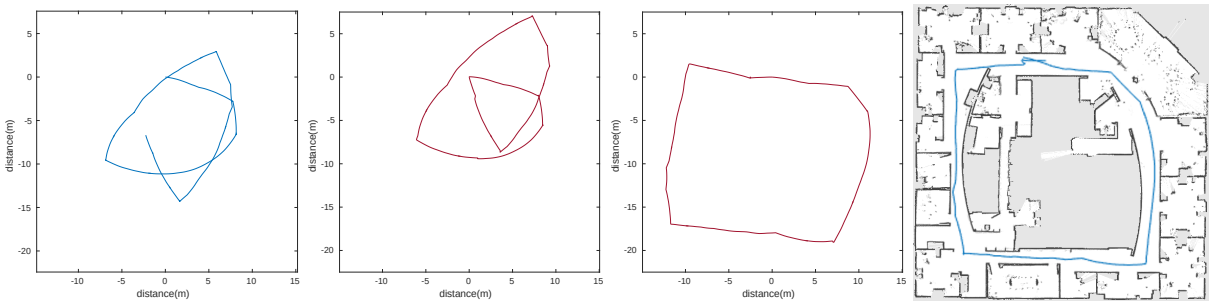
Although the ICP assisted odometry robot paths are more accurate the paths constructed only with odometry, the SLAM algorithm does effectively not change. The state vector that needs to be corrected being different is the only thing that differs. To show the functionality of the SLAM algorithm it would be more interesting to show the correction and convergence of a bad proposal path, rather than a path that is good. Development time plays a role in this choice. The SLAM algorithm is separated from the ICP algorithm while they need to be in the same implementation to combine their transformations.

9.1.3 Correction of the state vector



The linear equation that needs to be solved to optimize the graph when loop closing occurs is of the standard linear form. The system that needs to be solved is described in Equation 4.11 which was denoted by:

$$\mathbf{H}\Delta\vec{s} = -\vec{b}$$



(a) Intel dataset odometry not corrected

(b) Intel dataset odometry corrected in x and y after loop closing

(c) Intel dataset odometry corrected in x, y and θ after loop closing

(d) Map of the corresponding environment and highly corrected path

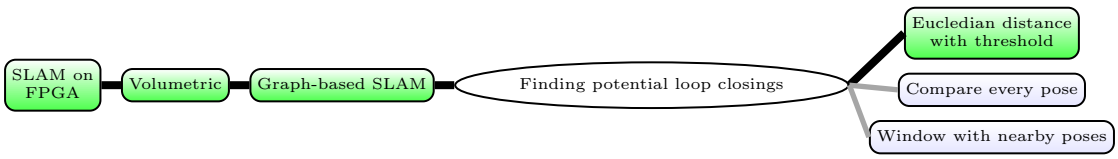
Figure 9.2: Odometry data, two different correction methods which give different results and a corresponding map to show the possible geometry of a robot path.

This system needs to be solved for each every dimensional vector in the system, which for the 2-D case are $\vec{x}$, $\vec{y}$ and $\vec{\theta}$. Each linear system can be solved independent of the corrections in other dimensions. For convergence of errors in a graph, it is possible to correct the graph in one coordinate system. For the polar coordinate system this convergence can be done for θ and in the Cartesian coordinate system correction for x and y can be calculated. The quality of resulting maps heavily depend on the correctness of the sensor and in which directions the sensors are most prone to errors.

The algorithm can be carried out without correcting the theta direction, this would however result in bad maps for a robot path that contains a lot of error in the angle. An example of this difference is shown in figure 9.2. For this example one complete loop of the Intel research labs SLAM dataset has been taken and the error minimization method of graph based SLAM has been used to find the corrected robot path. The robot path that is only corrected in x and y directions shows a minimized error but does not take the error of θ into account. The error in rotation is so high, that it the final path shows two loops instead of single loop it has actually travelled. This method is therefore not sufficient. The third robot path has been corrected in rotation as well showing a very similar geometry to the robot path drawn on the right showing a much further developed and corrected robot path which matches the map of the environment well.

9.2 Loop closing

9.2.1 Finding potential loop closings



Loop closing is the name for a robot visiting a pose that it has visited before and uses this event to correct historical data with accumulated errors with more direct data. Before a robot path can be corrected, it needs to correspond the current measurements to historical measurements to confirm a correct loop closing. There are several methods to link measurements, every

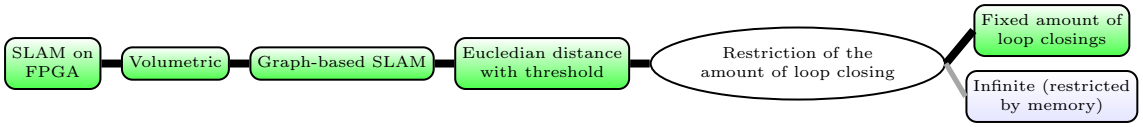
method having its own advantages and disadvantages. The algorithm used to correspond the data is a scan matching algorithm that detects the transformation between two laser scans with a corresponding score. If the score of the scan match is high enough, the loop closing can be used within the graph.

The easiest way to find corresponding data is by checking the laser scan of the current pose of the robot against all the laser scans of previous poses. This simple approach works for small robot paths, but the amount of laser scan matches the robot needs to execute in order to detect correct loop closings grows over time. Laser scans are considered heavy operations, which means that other proposals with less computational complexity are preferred.

When it is not desired that every pose gets matched to every previous pose, a loop closing proposal method is needed. The simplest method to find loop closing is by performing scan matching between the current pose and every other pose to find a high scored match. This method would have a high computational complexity which is not wanted. Another possible method to find corresponding poses is the calculation of euclidean distances between the poses according to the pose graph. If the distance is below a threshold, the scan matching algorithm is run with the two poses. The threshold can be a fixed pessimistic value, but it can also be a value dependent on the certainty of the pose. If the threshold is a fixed value, there is still a large amount of scan matches that need to be carried out, which potentially takes a long time.

Calculation of the euclidean distances between poses will be used to propose loop closings. By using a threshold on these distance, it is only necessary to carry out ICP for the poses that are within a given distance and angle from each other. The computational load will be reduced because the ICP algorithm is used less.

9.2.2 Restriction of the amount of loop closing

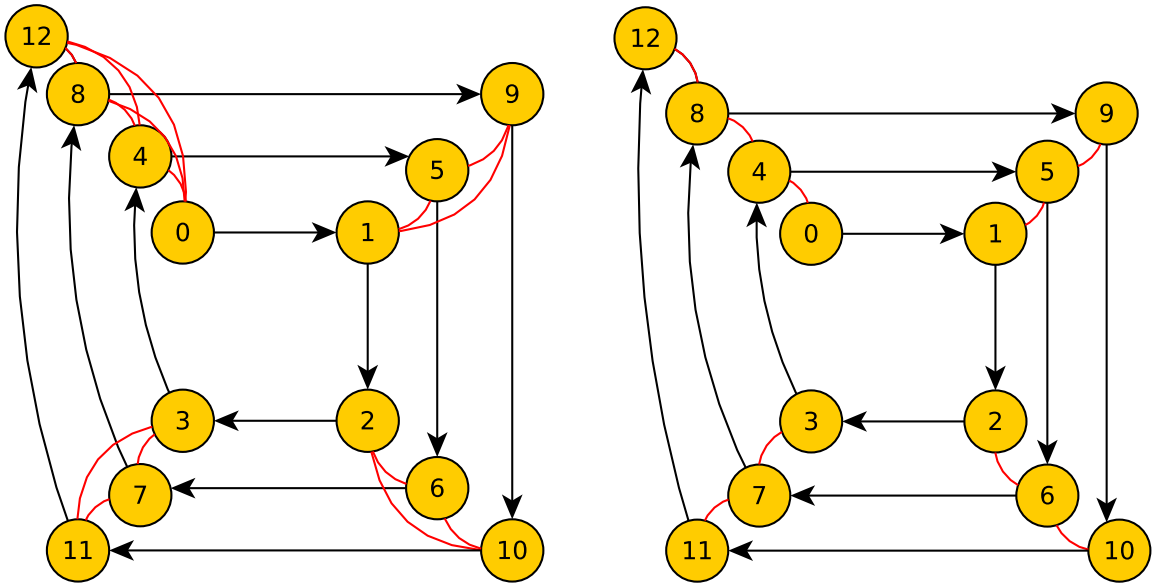


Because loop closing is found between the current pose and one or more previous poses, the amount of loop closing can become very large. However, the number of actual loop closings is most of the time not even close to the number of loop closings that can theoretically occur. Normally a minimal number of poses should pass before the robot would have made a complete loop trough an environment. The probability of each loop closing is stored in the information matrix, position of the loop closing (which is the error) is stored in a separate vector.

Because there can be a large amount of loop closing, the information matrix can become dense and the vector containing the errors can be as large $\frac{1}{2}n^2$ where n is the length of the state vector. Allowing this many loop closings results in a large amount of memory being used on the FPGA, which for large systems will not be available. The amount of memory that is used can be reduced if there are fewer loop closings.

Every loop closing has a pose which is the current pose of the robot (s_i) and a pose with which the loop closing occurs (s_j). The loop closing for the pose which creates the loop closing is called outgoing loop closing. The loop closing for the pose that gets loop closed with is called incoming loop closing. The number of outgoing and incoming loop closings per pose can be fixed to one. Essentially fixing the amount of loop closing means that every current pose can only close the loop with one other pose, and every pose that already has an registered an incoming

loop closing can not again be used for loop closing. Each pose contains a boolean that expresses whether the incoming loop closing port is still available. If the incoming loop closing port is already used, an alternative pose to close the loop with needs to be looked for. If there are no other loop closing proposals, the loop closing can be neglected.



(a) Graph with 13 poses showing multiple loop closings between one pose and other poses. (pose s_i is pose s_{i+4} for all i)

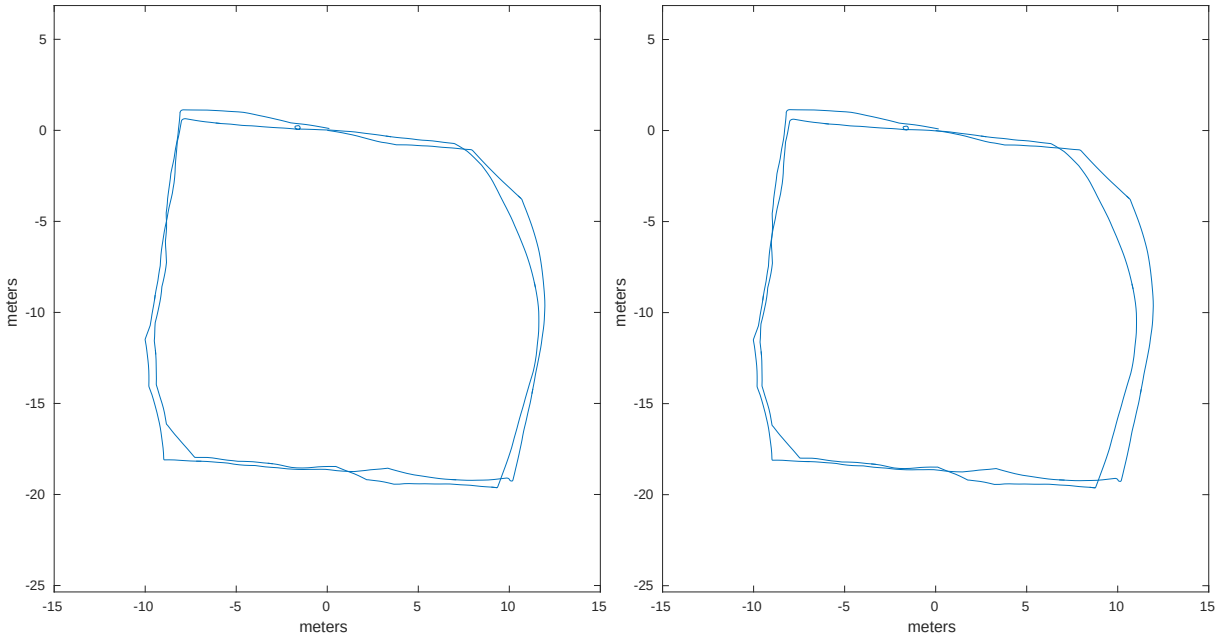
(b) The same as figure on the left with amount of loop closing restricted to one per pose.

Figure 9.3: Unrestricted amount of loop closing and restricted amount of loop closing

An important factor to consider when restricting the amount of loop closing is the effect the restriction has on the quality of the resulting map the SLAM algorithm produces. Hence, the more successful loop closings a graph contains, the better the resulting robot path and thus the map will be. However, if loop closing occurs ambiguously between poses, the loop closings will correct smaller errors. A loop closing after a long period of time without loop closing has a lot more effect on the resulting graph than loop closings right after another.

Figure 9.4 shows an example of the difference in quality between graphs that performs loop closing to the previous loop on the left. The image also shows a figure where the loop closing is allowed to each previous loop on the right. This is a small example but it is clearly visible that the difference between the two graphs is minimal. Because the amount of poses between each loop closing is high, the probability of correctness will be low after closing the loop. Whether the second loop only closes to the end of the first loop or also to the first pose of the complete graph has little influence because the probability of loop closing will be large in comparison to the probability that odometry has introduced. A few small differences between the paths can be detected, which is caused by the even higher probability of multiple loop closings. Without a map or ground truth of the robot path, it can not be told which path is a better representation. The errors that do remain in the graph can be corrected by other strategically chosen loop closing in other positions of the robot path.

Each incoming loop closing can be considered as a port, which means it can be connected by another pose to express the relation between the two poses. Each port can only be used once,



(a) Resulting graph with restricted amount of loop closing

(b) Resulting graph with unrestricted amount of loop closing

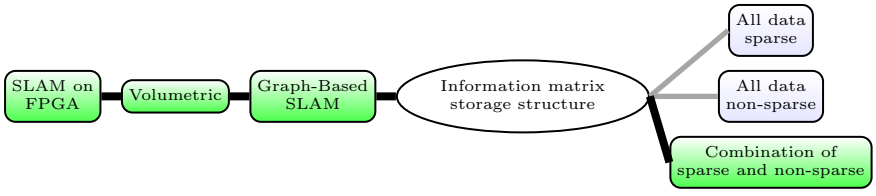
Figure 9.4: Quality difference between restricted and unrestricted loop closing

which means that there is a high probability of all the poses being occupied after each loop. When there is additional functionality added to prevent the robot from occupying the ports in a single loop, the loop closings can later be made between loops that are older than one or two loops ago. Closing the loop with older loops can therefore still be carried out without the need for a quadratically growing memory usage. Considering the example of figure 9.4, the second loop does not implicitly have to relate to the first pose of the previous loop, it can also relate to the second pose. The influence of the loop closing from the second pose will not be different from the first one as soon as the data is good enough. Whether this data is good enough is determined by if it can be correctly expressed in a transformation found by ICP.

By restricting the amount of loop closings, the amount of memory can be reduced. The quality of the graph does not have to suffer from this restriction method as long as the loop closings are strategically choses. When the loop closings are proposed in a strategic way, the quality of the state estimate and therefore the final map will be (almost) just as good, and the restricted amount of ports is not exceeded.

9.3 Linear Solver

9.3.1 Information matrix storage structure



The information matrix contains probabilities of the relations between poses. The size of the

information matrix with size n^2 where n is the number of nodes minus one. Because this quadratic size, the required amount of memory becomes high for large graphs. Looking at the graph SLAM algorithm, it can be seen that the matrix has a high chance of containing a lot of zeros since only the items in the information on the diagonal and a few on loop closing positions are non-zero. However, in a normal environment, the robot would visit a reasonable amount (tens or hundreds) of poses before a loop will be closed. The matrix only contains non-zero values at the positions an edge has been created. For an edge (i,j) this is always at positions:

- (i, i)
- (j, j)
- (i,j)
- (j,i)

Positions (i, i) and (j,j) are always on the main diagonal, odometry edges for which hold $j = x + 1$, the two (diagonal) lines above and below the diagonal contain values. For loop closing there is no fixed relation between i and j. However, one could say $j < i$ because pose s_i is the position that closes the loop with pose s_j, which means pose s_i is created at that moment while s_j already existed, hence $j < i$. Odometry poses will only create 3 items around the diagonal, and additional loop closing in reasonable large environment would result in a very low density of the matrix. With a static loop amount of closing per pose, a larger state vector would results in more sparsity in the corresponding information matrix.

9.3.1.1 Sparse matrices

A matrix with a low density is called a sparse-dense matrix or sparse matrix. The sparsity of a matrix is a property that has to be exploited if possible. Especially whenever a matrix is very large and only few items are non-zero. Normal matrices is built up out of vectors, these vectors contain elements. These matrices will be called dense matrices, ordinary matrices or non-sparse matrices from now on. In normal matrices, each value of each vector of the matrix is stored separately in the memory. Instead of defining the value of each element, it is also possible to generalize the value of all common elements, and only denote the deviations. The common term for this is a sparse matrix. The sparse matrix is different than an ordinary matrix since it only mentions the values of elements that are not zero in an ordered way. This ordered way is often by storing the index of the element that is going to be described, followed by the value of the element.

An ordinary $4 - by - 4$ matrix containing three non-zero elements could look like this:

$$\begin{bmatrix} 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

The same matrix described in sparse form would only describe the three non-zero items leaded by the indices describing first the column, then the row and finally the corresponding value.

$$(0, 1, 1), (2, 0, 2), (3, 3, 3)$$

The amount of items that is needed to describe the matrix is 16 for the ordinary notation and 9 for the sparse notation. The sparser the matrix is, the more memory saved. Also, the

indices in the sparse matrix notation are always values between 0 and 3 for this case, but they will always be integer values between 0 and $n-1$ where n is the system size (assuming describing an $n-by-n$ matrix), while the values can be much more complex values than integers.

Vectors can also be written in the sparse notation. In the case of vectors the place of a non-zero item is described by one index. It is also possible to describe matrices with sparse vectors in one dimension and dense vectors in the other dimension. The matrix could than either be a sparse vector of dense vectors or a dense vector of sparse vectors.

9.3.1.2 Memory

Because the final design is a hardware implementation, the required and used amount of memory should be analysed. The disadvantage of the sparse matrix notation however is that it requires a dynamic amount of memory. A dynamic amount of memory is often not a problem on general platforms like PC's or micro controllers since there is an amount of memory available which can be allocated and freed by a memory controller. There is however still a static amount of memory available to allocate. Exceeding physical available amount amount of memory would cause a slower or unstable system. This memory controller is not standard implemented when making a hardware design. When a memory controller is not available, the designer has to account for the memory-use during design-time.

9.3.1.3 Combination of sparse matrices and static memory use

In the loop closing section(9.2.2) of this chapter, the effect of restricting the amount of loop closing has been explored. It was concluded that the effect of loop closing restriction is small and with proper loop closing proposals can almost be neglected. With a restricted amount of loop closing, the matrix contains a few features that can be exploited for storage of sparse matrices on a static amount of memory. These features are:

- Matrix is symmetrical
- The number of items per vector (column) is static
- The number of vectors is static

The second property only holds if the allowed amount of loop closing is fixed. The amount of memory per column in terms of items would then be: $3 + 2 * n_{lc}$ where n_{lc} is maximum amount of incoming and outgoing loop closing. The third property, that the number of vectors is static holds because the diagonal is always completely used for a graph that is exactly the size of state vector, even when no loop closing has occurred, the matrix will not contain empty column vectors. The first item will be an important property which will be used later in the linear solving algorithm.

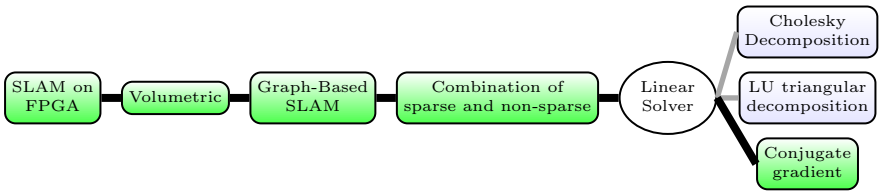
The use of the sparse matrix notation is only useful when the matrix is indeed significantly sparse. Though it could be possible that the amount of memory needed to store the matrix is not that much higher because they are only integer indices, the amount of additional actions that is need to be performed can be large. The reason for this is the unpredictability of a sparse sparse notation. In the normal vector notation, each item can simply be found by looking up the index, while in the sparse notation the value of each index needs to be inspected before looking up the corresponding element.

A sparse notation will even be less predictable when the items are not sorted within a matrix or vector, if this is however the case, a dot product between two sparse vectors will consist of checking the existence of every index within the list of indices of the other vector and fetching the corresponding value, before the actual computation can be carried out. In hardware design, a search through a list and sorting a list are heavy operations, especially for vectors with a large amount of non-zero items.

9.3.1.4 Conclusion on matrices and memory

When sparse-matrices are used to represent the matrix, the amount of memory and computations can be reduced. The choice was made to use the sparse matrix notation for the information matrix, but the ordinary notation for vectors. The matrix can then be described by a non-sparse vector of sparse vectors. The combination of sparse and non-sparse vectors in one system will result in some overhead in the transformations between the two, but would potentially save an enormous amount of memory and computations, especially for large systems. The choice to not only use sparse matrices has not been made because all the vectors in the system (although they might seem sparse sometimes) do not have a fixed amount of items which makes the sparse notation profitable.

9.3.2 Linear solver for graph convergence



The choice of using a(partially) sparse matrix has a lot of impact on the way a linear system containing this matrix is solved. All the systems have in common that the size of the matrix has influence on the time to solve the system. Several algorithms allow for solving systems containing a sparse matrix. In this section reasonable techniques are analysed to find an appropriate method to solve the linear system.

9.3.2.1 Cholesky decomposition

Cholesky decomposition is a matrix-decomposition method which can potentially be used to solve a linear problem with a sparse matrix. Decomposition is a method to find solutions of linear systems without having to invert the matrix or applying row reduction. Cholesky decomposition of a matrix is only possible whenever a matrix is positive definite which means that the equation $\vec{z}^T\mathbf{M}\vec{z}$ is positive for every non-zero vector $\vec{z}$. The **H**-matrix has this property and is therefore suitable for Cholesky decomposition. The first step of solving a system with a decomposition method is finding two matrices from which the product is the decomposed matrix. With this resulting matrices, new equations can be formed which can be solved easily because the matrix in these equations is a triangular matrix. In Cholesky the decomposition of a matrix is given by:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

In this equation **L** is a lower triangular matrix, which means all the items above and right of the diagonal are zero. The lower diagonal matrix can be found by working from left to right and top to bottom and can be found by square roots and subtractions. The standard equation $A\vec{x} = \vec{b}$ can be rewritten as:

$$\mathbf{L}\mathbf{L}^T\vec{x} = b$$

Another vector ($\vec{z}$) can be introduced as a temporary vector, it replaces $L^T\vec{x}$ and can be used to which can be found by forward substitution:

$$\mathbf{L}\vec{z} = \vec{b}$$

Forward substitution means that the equation will be solved from top to bottom, which is a trivial operation since $\mathbf{L}$ is a lower triangular matrix. Every row only contains one unknown. The value for $\vec{z}$ has now been found and the value for x can be solved by backward substitution, which is substitution from the bottom to the top:

$$\mathbf{L}^T\vec{x} = \vec{z}$$

The Cholesky decomposition will be much faster than normal row reduction, because only the non-zero items are taken into account. However, the resulting triangular matrix will end up having one or more non-sparse vectors. This means that the matrix representation needs to be changed in order to support the full version of Cholesky decomposition, which makes implementation in hardware a challenge.

9.3.2.2 LU decomposition

Another decomposition method is LU decomposition, this is also a decomposition method for which the matrix will be decomposed into a Lower and an Upper triangular matrix. However, in Cholesky decomposition, the upper and lower matrix are each others transpose. In LU decomposition, the upper and lower triangular and upper triangular matrix are unique which means the decomposition will be different. The upper and lower triangular matrices which are also called the factors are harder to find because linear equations still need to be solved. However, the linear equations become simple because the matrix that is decomposed is sparse and contains few items.

Once the decomposition is found, the steps to find the actual values for the x vector are the same. First the matrix is decomposed using linear equations which have a maximum fixed amount of unknown variables:

$$\mathbf{A} = \mathbf{LU}$$

The the matrix can be seen as the product of the upper and lower triangular matrix:

$$\mathbf{LU}\vec{x} = \vec{b}$$

The Ux can be substituted with a temporary vector $\vec{z}$ which can be found by triangular substitution, just as $\vec{x}$ can be found by solving it using $\vec{z}$:

$$\mathbf{L}\vec{z} = \vec{b}$$

$$\mathbf{U}\vec{x} = \vec{z}$$

Although LU decomposition does give different resulting matrices than Cholesky decomposition, the matrix also contains vectors that have more non-zero items than the static size sparse matrices support.

9.3.2.3 Conjugate gradient algorithm

Instead of using decomposition, it is also possible to solve sparse systems with iterative algorithms. The property that is required from the solver method is that unlike the decomposition methods, there should be no matrices involved that are dense or have dense vectors. The conjugate gradient algorithm is an algorithm in which the error is determined based on the matrix, the given vector, and the wanted vector. This determined error is in vector form and can be used to manipulate the wanted vector based on a direction found with the error vector.

The conjugate gradient algorithm is an iterative algorithm which will approach the wanted vector by decreasing the total error. The algorithm is shown in 3. An initial $\vec{x}_0$ is chosen to calculate an error that needs to be converged. The error that is calculated is called the residual vector $\vec{r}$. The initial search direction is set to the initial residual vector which is defined by $\vec{p}_0 = \vec{r}_0 = \vec{b} - \mathbf{A}\vec{x}_0$. A scalar α is calculated by the fractional of the squared sum of the errors and the $\mathbf{A}$ matrix scaled with the search direction vector. It can be seen that the result of $\mathbf{A}\vec{p}_k$ is used two times, and is the only sparse matrix to non sparse vector reduction. The rest of the vector calculations are done with non sparse vectors because the resulting vectors are not sparse anyway. The state vector is altered by the found scalar α times the direction vector. The calculated β scalar is calculated by the fractional of the new and current squared errors and is used to alter the search direction. Finally the search direction is altered in step 10.

Algorithm 3 Conjugate gradient algorithm

1: **initialize:**
2: $\vec{r}_0 = \vec{b} - \mathbf{A}\vec{x}_0$
3: $\vec{p}_0 = \vec{r}_0$
4: $k = 0$
5: **repeat until convergence of r:**
6: $\alpha_k = \frac{\vec{r}_k^T \vec{r}_k}{\vec{p}_k^T \mathbf{A}\vec{p}_k}$
7: $\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{p}_k$
8: $\vec{r}_{k+1} = \vec{r}_k - \alpha_k \mathbf{A}\vec{p}_k$
9: $\beta_k = \frac{\vec{r}_{k+1}^T \vec{r}_{k+1}}{\vec{r}_k^T \vec{r}_k}$
10: $\vec{p}_{k+1} = \vec{r}_{k+1} - \beta \vec{p}_k$
11: $k = k + 1$

The computational load of the algorithm can be analysed by looking at the amount of operations needed for each iteration. The operation $\mathbf{A}\vec{x}$ in step 2 is a matrix vector multiplication which is a heavy computation for large matrices. The complexity of this particular operation will become less because matrix $\mathbf{A}$ is a sparse matrix. However, it still requires n dot products of a sparse vector with a non-sparse vector where n is the system size. The division in step 6 is a normal division that is operated on two numbers instead of on a vector or matrix, which is a relatively light operation compared to vector operations. Table 9.1 shows the resources used

	division	scaleNSV	dotNSV	addNSV	subNSV	SMNSV
step 2					1	1
step 6	1		2, (1 shared, id=0)			1, (1 shared, id = 1)
step 7		1		1		
step 8		1			1	1, (1 shared, id = 1)
step 9	1		2, (1 shared, id = 0)			
step 10		1			1	
total 1 iteration	2	3	3	1	3	2
total i iterations	2*i	3*i	3*i	2*i	1+2*i	1+i

Table 9.1: A resource overview for the parallel conjugate gradient algorithm

	scaleNSV	dotNSV	addNSV	subNSV	SMNSV
adders		n-1	n		$\sum_{i=0}^{n}(m_i - 1)$
subtracters				n	
multipliers	n	n			$\sum_{i=0}^{n} m_i$

Table 9.2: Low-level cost of the high-level functions

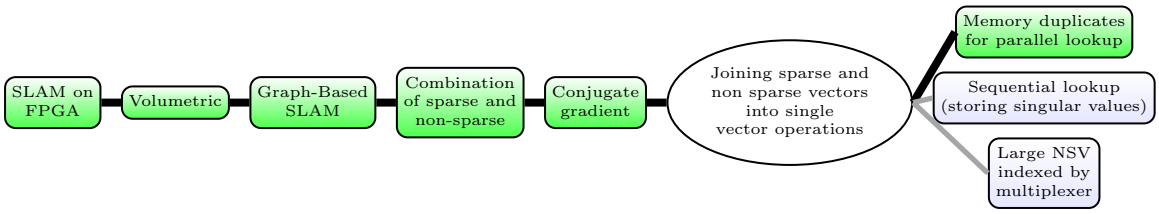
to execute the conjugate gradient algorithm in terms of operations. The amount of cycles will determine the total amount of operations that the algorithm needs to perform. On an FPGA the number of resources is limited, the resources used to perform operations like additions and multiplications are dedicated digital signal processors (DSPs) and look-up tables (LUTs) and performing large vector operations with operators created in LUTs will make the FPGA run out of resources very quickly. A possible solution to this problem will be discussed in the next level of the design space exploration.

Table 9.2 shows the hardware cost of the used vector operations in expressions of low level components, where n is the system size. Only the sparse matrix non-sparse vector multiplication has the variable m which is the amount of active items in the each vector. When $m_i = m_{i+1}$ for each value of i, which means the number of non-zero values in each vector is the same, the total number of adders becomes $n * (m-1)$ and the number of multipliers becomes $n * m$ for the complete matrix vector multiplication. In practice this is the case because the number of active elements in the sparse vectors has been fixed at 5 items. The items are not checked before they are used for calculations so they will be used inside the multipliers and adders.

9.3.2.4 Conclusions on the linear solver

The conjugate gradient algorithm will be implemented to solve the linear systems in hardware for graph-based SLAM. The conjugate gradient algorithm does not alter the sparse matrix and therefore no additional memory is required to store multiple matrices which is the case in the proposed decomposition methods. Because the conjugate gradient algorithm does not change the sparse matrix by making it larger of creating other matrices, and consists of only vector operations, the conjugate gradient algorithm seems like the most promising method for solving linear systems on hardware consisting of sparse matrices with static memory.

9.3.3 Joining sparse and non sparse vectors into single vector operations



The linear solver contains sparse vectors with a fixed size to represent the information matrix and ordinary vectors to represent everything else. The conjugate gradient algorithm does only contain one operation that concerns both these sparse and non-sparse vectors. This operation is present in the initialization phase of the algorithm ($\mathbf{A}\vec{x}$) and in the iterative part of the algorithm ($\mathbf{A}\vec{p}$). Effectively, the matrix vector multiplication consists of the system size amount of dot products. These dot products between ordinary vectors is a trivial problem in math, because it is just a pairwise multiplication of two vectors and the sum of the resulting vector.

9.3.3.1 Coupling sparse vectors

When the dot product is performed on two sparse vectors, the operation is functionally the same. However, i1f two sparse vectors are multiplied, it is not possible to do the multiplications without finding the corresponding indices in the vectors. An example of a dot product of two sparse vectors is shown in equation 9.1. Only the non-zero parts that have corresponding indices will be used in the final answer. Therefore, the dot product can be rewritten into vectors that only contain the items with matching indices.

$$\begin{bmatrix}(0,5)\\(2,5)\\(4,5)\\(6,5)\\(8,5)\end{bmatrix} \bullet \begin{bmatrix}(0,5)\\(1,5)\\(2,5)\\(3,5)\\(4,5)\end{bmatrix} \simeq \begin{bmatrix}(0,5)\\(2,5)\\(4,5)\end{bmatrix} \bullet \begin{bmatrix}(0,5)\\(2,5)\\(4,5)\end{bmatrix} \tag{9.1}$$

The introduction of sparse matrices and doing calculations with them results in less computations, but they create a certain amount of overhead that need to be taken into account before working with them. In this case, the overhead that is introduced is the coupling of the indices of the sparse matrices. In the particular case of a dot product, there is an advantage. Because a dot product is a multiplication of the items corresponding to an index, every index of one sparse vector (of which the item is always non-zero) missing in the other sparse vector will become zero and will not alter the final outcome of the dot product. Therefore, it is only necessary to check the presence of the indices of one vector and multiply these in order to find the final solution of the dot product.

Figure 9.5 shows an abstract structure of the sparse vector coupling. Each index of $\vec{v1}$ will be looked up in the indices of $\vec{v2}$ in the green block. If the index is present in $\vec{v2}$, the corresponding value is returned, otherwise a zero is produced. Producing a zero means the multiplication will be done, but the outcoming value will always be zero, instead of not doing the multiplication at all like in equation 9.1. In a static hardware structure it is not beneficial to skip the multiplication, because the possibility of doing the multiplication means that the

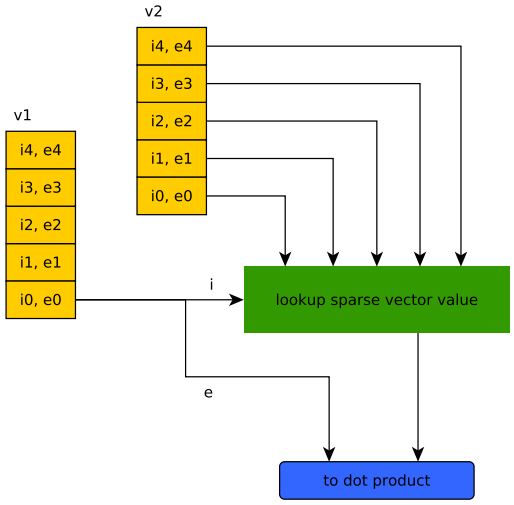


Figure 9.5: Value lookup of sparse vector index to perform a dot product

hardware is already available. In the figure the only one lookup block is shown, in the fully parallel solution, each index will go to a lookup element.

9.3.3.2 Coupling a sparse vector to a non-sparse vector

In the previous paragraph the coupling of two sparse vectors has been discussed. In the conjugate gradient algorithm the the dot products will be performed between a sparse vector and a non-sparse vector. The problem with this different vectors is the fact that the way the items are stored is different. The non-sparse vector is just a plain vector with the system size amount of items. Effectively, each index of the sparse vector needs to be looked up in the non-sparse vector. This lookup can be done by using large multiplexers, but large multiplexer will use more area on the FPGA. Another method is using memories in which the it is easier to find the value without large multiplexers. However, memories are sequential, an address is provided and that address will be fetched or written to in the next clock cycle. A fully parallel implementation does not contain multiple clock cycles since the complete implementation is a single combinatorial path.

If synchronous memories are used, only one block can be fetched out of a memory at a time where each block can contain one or multiple items. If a memory only contains one item per block, the address can be provided at the read-address input of the memory and the correct value will directly be available at the read data output in the next clock cycle. If the block contains multiple items, the required element needs to be extracted from the block it is in with the help of a multiplexer with the size of the block.

The disadvantage of using memories for the storage and lookup of vectors is that they are synchronous, which means they are dependent of a clock input and will only present one block of data at a time. If the sparse matrix has a fixed amount of items which need to be coupled to values from the non-sparse matrix which are all in different blocks, each block needs to be fetched, the correct data should be extracted and stored, after which the next block can be fetched in the next clock cycle. The amount of clock cycles needed to do for example the final complete sparse matrix non-sparse vector multiplication will take five times as long.

Instead of extracting the data with a large multiplexer or fetching it from one memory over

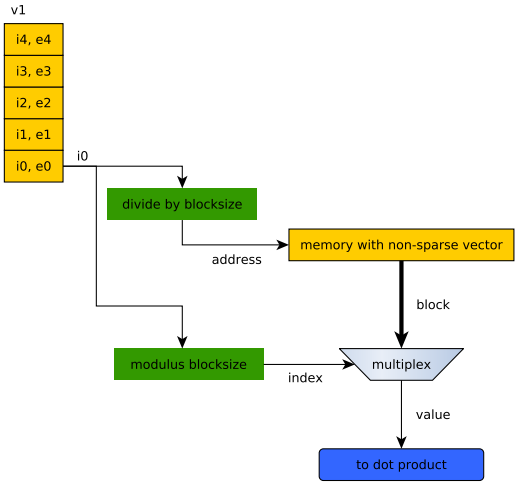


Figure 9.6: Structure of finding and fetching the correct non-sparse vector block and the correct value value within the found block. The architecture that is connected to the first element of the sparse vectors appears for each element.

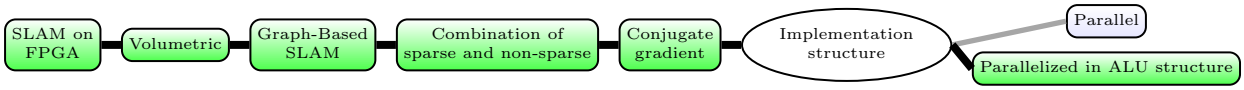
time, it is also possible to use multiple memories, each responsible for fetching one item for from the non-sparse vector corresponding to the index of the sparse vector. It is possible to use the available blockram memory on the FPGA as multiple parallel accessible small memories. Each of these small memories contains a copy of the complete non-sparse vector from which values needs to be coupled to a sparse vector's index.

Figure 9.7 shows the structure sparse vector to non-sparse vector coupling with multiple memory copies holding the non-sparse vector. Because the memory will hold blocks instead of separate values, which will be discussed in the next section, the blocks still need to be multiplexed to find the actual corresponding value.

9.3.3.3 Conclusions on sparse to non-sparse vector coupling

To realize vector operations between sparse and non-sparse vectors, memory duplicates of the non-sparse vectors are used. A parallel lookup of the multiple items in the vector would consume a lot of multiplexers and a sequential look-up would result additional clock cycles which results in a much slower overall computation time. The indices of the sparse vectors are to fetch the correct vector parts in the memory. Once the vector parts have been fetched, the correct values are extracted from the vector parts.

9.3.4 Implementation structure



Implementation of the graph-based SLAM algorithm can be generalized to the following steps:

1. Creation of the state vector
2. Check for loop closing

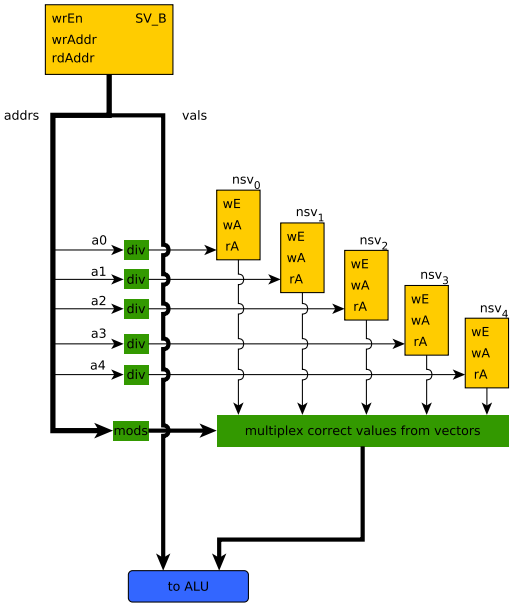


Figure 9.7: Structure of fetching values from a non-sparse vector position, based on the indices of a fixed size sparse vector.

3. Minimize loop closing errors

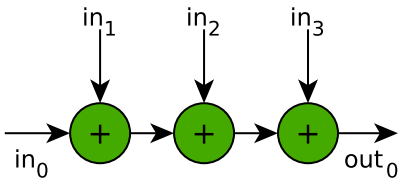
If there is no error to minimize because there are no (new) loop closings, the minimization step can be skipped. The final solution (robot path) is constructed by running these three steps n number of times, where n is the size of the state vector. Which means the total amount of operations can be seen as the sum of the operations of the three steps times n. Considering that creation of the state vector consists of trigonometric functions and multiplications, loop closing checks consist of comparing and selecting Eucledian distances and minimizing errors by solving a linear system using the conjugate gradient algorithm, the total amount of calculations is large.

In hardware, it is possible to spread the computational load over time and over area. Spreading calculations over time is called sequential computing and has been applied in many computing systems such as PC's, cellphones, and micro controllers over the last years. By creating a platform that is able to run simple instructions over time, complex computations can be done. These complex computations are created in software, which is translated into simple instructions so it can run on the hardware platform. The only way to do more instructions in a fixed amount of time is by speeding up the frequency with which the instructions are executed. Because of physical boundaries, the frequency of these system is reaching its limits and the urge to share the workload over area becomes larger. Sharing workload over area is also called parallelization, which requires a somewhat different structure than sequential computing.

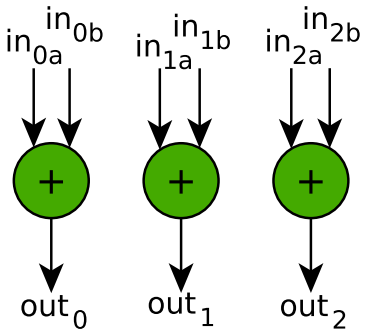
To express the differences between sequential and parallel computations, the conjugate gradient algorithm will further be analysed because of its potential parallelization, the amount of computational load and the significance in the total graph based SLAM implementation.

9.3.4.1 Sequential computations

Sequential computing does have some advantages over parallel computations, as mentioned above. Complex problems can be described in a general software language and translated to



(a) Example of sequential (fold) operation



(b) Example of a parallel (zip) operation

Figure 9.8: Basic operations on vectors, the *fold* operation has sequential nature while the *zip* operation has a parallel nature

sequential instructions to be run on an ordinary processor. Because the problem is spread over time, sequential computing is very scalable. For example will a larger state vector just take more clock cycles before the solution is known. Sequential systems are often also more general which makes them support many different application by changing the software, parallel systems use specific software and are less general.

Looking at the conjugate gradient algorithm it can be seen that it mainly consists of vector operations. It is possible to distinguish two types of operations looking at vectors, the first one is a folding operation which is for example used to sum the values of the vector. The folding operation is naturally a sequential operation because the amount each item needs to be add to the previous found sum. A certain amount of parallelization can be accomplished by dividing the operations into groups and join the groups in later stages. However, this will always restrict amount of parallelization. The second type of vector operations is a mapping or zipping operation. Which means every item from a vector gets used without any correlation between them. Because there is no correlation, the operation can be executed in parallel. The basic sequential and parallel operation are shown in figure 9.8.

Looking at the vector operations within the conjugate gradient algorithm, the possibilities for parallelization are highly present. Because the clock frequency of an FPGA is often much lower than the clock frequency of ordinary processors, there is no reason looking into an only sequential implementation of the graph-based SLAM algorithm on an FPGA.

9.3.4.2 Parallel computations

The fully parallel structure of one iteration of the conjugate gradient algorithm is shown in figure 9.9, the sparse matrix non-sparse vector product is carried out first after which the rest of the vector operations of the algorithm. The amount of functional blocks needed to do one iteration in parallel is very high for large systems. Therefore the algorithm needs to be fit into a sequential structure with parallelized vector operations to make the algorithm fit. Each block of three green dots represents a vector operation of the system size amount of items.

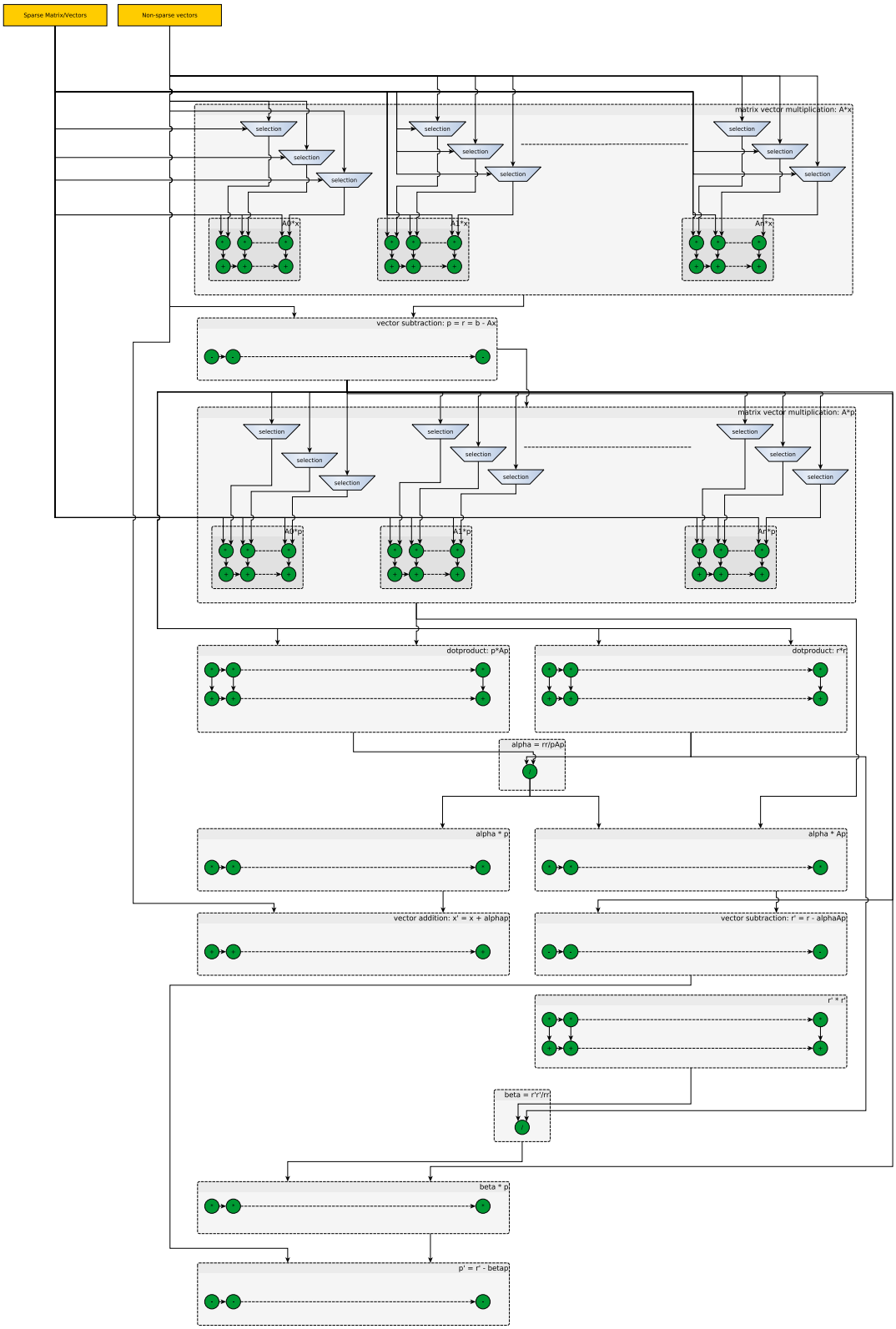
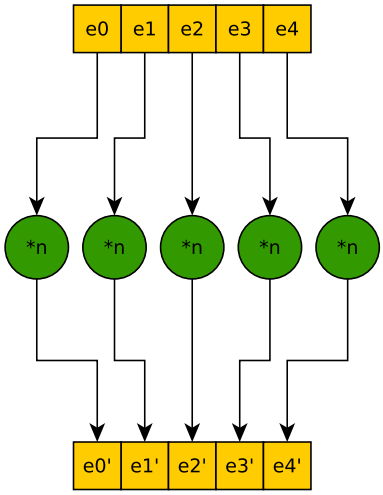
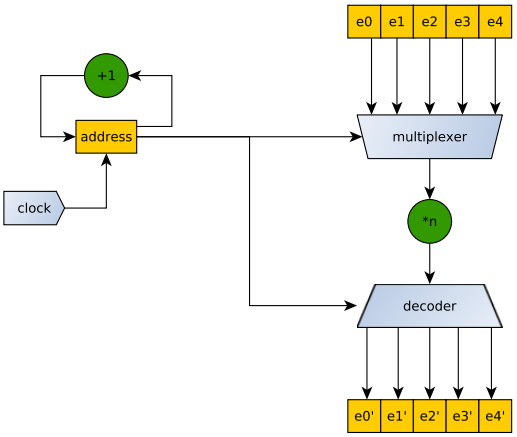


Figure 9.9: One parallel iteration of the conjugate gradient algorithm



(a) Example of a parallel vector scale



(b) Example of a sequential vector scale

Figure 9.10: Sequential and parallel structure of pairwise multiplication

9.3.4.3 Sequentialized parallel computations

The difference between a sequential en parallel implementation of a scalar of a vector is shown in figure 9.10. The important difference is placing of the functional blocks. There is no overhead required in the parallel solution, but there is a large amount of functional blocks. In the sequential solution, there is a certain amount of overhead that reuses the single functional block to do the same computations over time.

However, this is only the difference between sequential and parallel computations while the goal is compare a parallel solution to a sequentialized parallel solution. The reason one does not want a fully parallel solution is because it is simply not possible looking at the physical constraints of an FPGA, doing things parallel will cost resources which are not available. Especially when the system to solve is actually a large one. In the example that was used throughout this thesis a loop closing occurs after 335 odometry commands. The distance it has travelled is in the tens of meters which is a fairly normal environment looking at the resolution of the sensors. To store all of the poses into memory and minimize the error using the conjugate gradient algorithm, a system of 336 poses needs to be solved.

A system size of this amount of poses means that for a single dot product 336 multipliers are needed for pairwise multiplication and the same amount minus one adders for accumulation. The amount of available sufficient sized DSPs on the given FPGA is only 112. About half of the totally available DSPs will be used for the implementation of the scan-matching algorithm of this research. The amount of multiplications that can be done in parallel on the DSPs will be around 50. This amount of DSPs is by far not enough to do even one single dot product of non-sparse vectors in parallel. It is also possible to create additional multipliers in the variable blocks of the FPGA (LUTs) which might add a few extra possible parallel multiplications.

Instead of wanting a fully parallel implementation or a fully sequential one, the algorithm can be implemented in such a way that there are still advantages of working with an FPGA and having the possibility to use parallel components, but also the advantages of the principle of a

sequential processor to make the algorithm fit. This trade-off between parallel and sequential will bring a lot of overhead since the control will needs to be dedicated to the hardware structure. This will make the hardware implementation very application specific but on the other hand optimal for this particular algorithm.

A sequentialized parallel structure can be implemented by using a component that is reused during the algorithm. This reused component has been constructed to do multiple operations at the same time. The operations are of the same type which means the set of operations can be seen as a vector operation. This component can be seen as an arithmetic logical unit (ALU) which is used over time to complete more (time) complex instructions.

10 — ICP

Section 5.1 describes the iterative closest point algorithm into detail and a quick recap is given below. ICP is an algorithm used to find the transformation between two sets of points, the model ($\vec{m} = \sum_k m_i$) and the data ($\vec{p} = \sum_i p_i$). In case of a robot with a laser range scanner ICP can be used to align two sensor observations to determine the movement of the robot. The ICP algorithm can be summarized into two steps:

- Finding correspondences, with or without outlier removal
- Calculating the transformation which minimizes the distances (error) between the correspondences

A *correspondence* is a match from point p_i from the current observation $\vec{p}$ to a point m_j from the previous observation $\vec{m}$ and contains the following variables: contains the following variables:

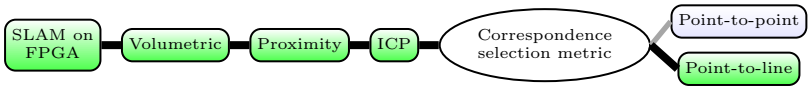
- Point (p_{ix}, p_{iy}) from the new observation $(p_i \in \vec{p})$
- Closest point (m_{ix}, m_{iy}) to p_i, from the previous observation $(m_i \in \vec{m})$
- Normal vector (n_{ix}, n_{iy}) from p_i to m_i
- Error (e_i) which is the euclidean distance between p_i and m_i

The choices of the different solutions for selecting correspondences, rejecting outliers, and minimizing the error to find the transformation, are found in the Sections 10.1, 10.2, 10.3 respectively.

10.1 Construction of correspondences

Section 5.1.1 states several ways of selecting points to form a correspondence. This section will explain the choices made to realize the construction of correspondences. The first choice to make is whether to use *Point-to-point* matching or *Point-to-line* matching (Section 10.1.1). After that is the choice of using *unique* or *non-unique* correspondences (Section 10.1.2), and the different architectural structures that can be used to realize the selection process in hardware (Section 10.1.3).

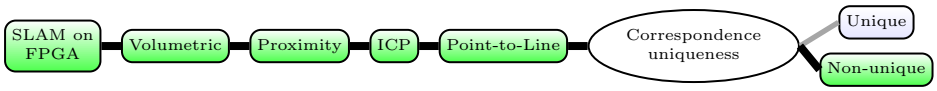
10.1.1 Correspondence selection metric



Section 5.1.1 describes that the Point-to-line matching has a faster convergence rate compared to the Point-to-point matching. Both matching types require finding the closest point. However, Point-to-line matching requires two points from dataset $\vec{m}$ to calculate a normal vector n, which makes the sorting slightly more complicated, because one has to find the closest m_i and second closest point m_j.

The choice was made to use the Point-to-line selection metric for its faster convergence speed.

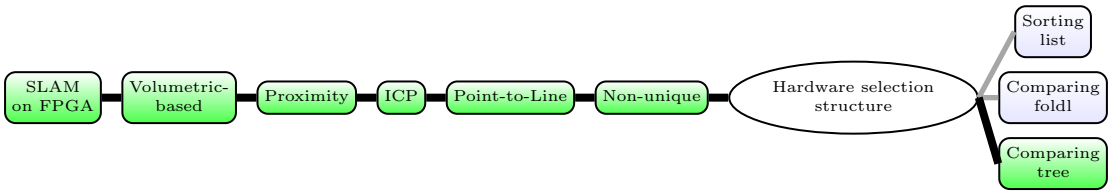
10.1.2 Correspondence uniqueness



Unique correspondence matching one allows only one correspondence for each point, *non-unique* allows for multiple correspondences for each point. Section 5.1.2 describes into more detail the differences between unique or non-unique correspondence selection and the effect it has on the results. Purely based on the quality of the results one might already choose for the non-unique selection criteria but another advantage of this solution is that no hardware is needed for filtering on uniqueness.

Based on the overall better result and the complexity of the selection process the choice was made to go with the non-unique correspondence selection.

10.1.3 Hardware selection structure



Based on the previous choices the selection algorithm must find the *closest* and *second closest* point from the previous observation with respect to the new observation. Since non-unique correspondences are allowed, there is no need to filter out the unique ones. The selection process must be able to find for every point (p_i), from the new observation ($\vec{p}$), the closest (m_j) and second closest point (m_k) from the previous scan ($\vec{m}$). This means that all the distances between all the points in $\vec{p}$ and $\vec{m}$ must be calculated. Looking at a single correspondence one must

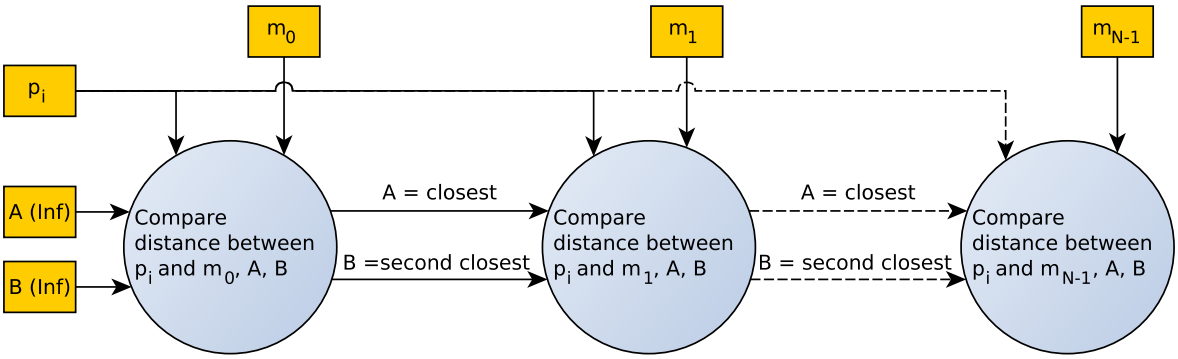


Figure 10.1: Finding the two closest points from the set m to p_i

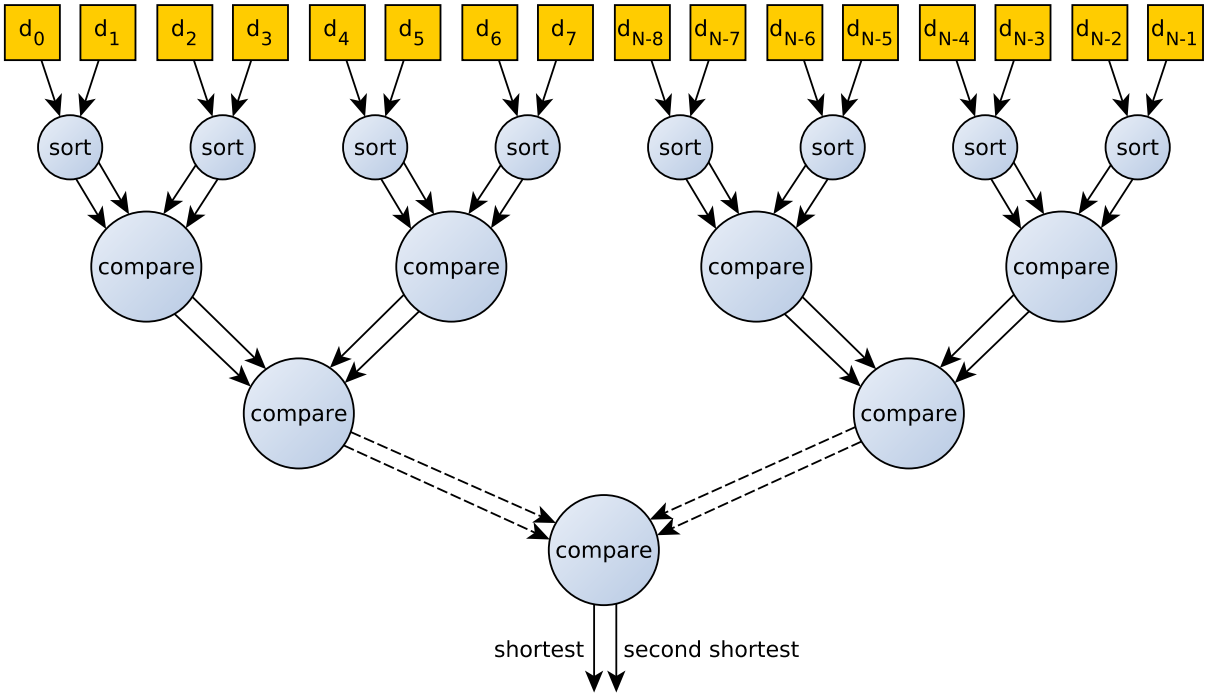


Figure 10.2: Finding the shortest two distances in a tree structure

calculate all the distances from p_i to $\vec{m}$. A single squared distance calculation $d_i = (p_{ix} - m_{ix})^2 + (p_{iy} - m_{iy})^2$ requires 2 multiplications and 3 additions. A single correspondence calculation requires $2N$ multiplications and $3N$ additions. For N correspondences $2N^2$ multiplications and $3N^2$ additions are required. With $N = 180$, synthesizing this on hardware is going to be a problem since an FPGA, as described in Appendix A, does not have that many multipliers. The realization of the QR decomposition, described in Section 10.3.2, faces a similar problem and solved in such a way that the same solution can be applied here. Further explanation of the solution can be found in Section 10.3.2.

From the distances, $\vec{d}$, the *shortest* and *second shortest* distances must be found. In MATLAB and Haskell, finding these two distances is easy because it is a matter of sorting an entire list containing all the distances and taking the first two. But sorting a list containing $N = 180$ items in parallel results in a large sorting structure. This structure can be split into slices and executed over time but it remains inefficient because only the closest and second closest points are needed. Alternative structures can be created which require less hardware

Logic		Truth table					Figure
Possibilities	**Check logic**	$a < x$	$b < x$	$a < y$	out_0	out_1	
$a < b < x < y$	$b < x$	1	1	_	a	b	
$a < x < b < y$	$a < x$	1	0	_	a	x	
$a < x < y < b$							
$x < a < b < y$	$x < a$	0	_	1	x	a	
$x < a < y < b$							
$x < y < a < b$	$a < y$	0	_	0	x	y	

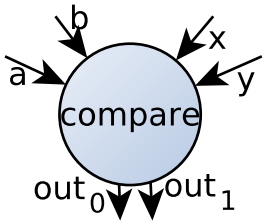


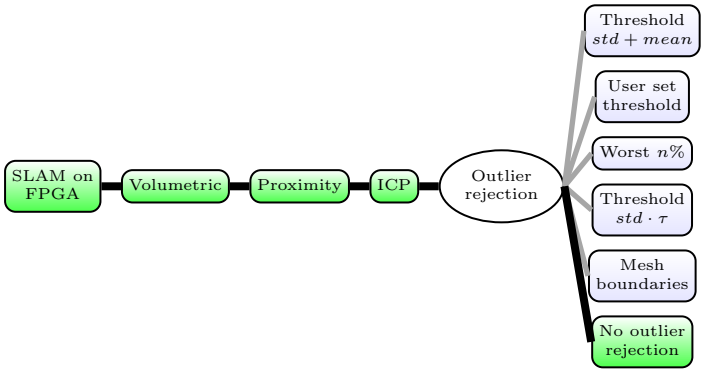
Table 10.1: Logic and truth table of the compare inside the tree

and less execution time. Since the closest and second closest points are the only ones needed the first idea was to generate a fold left (foldl) structure as shown in Figure 10.1. However, this structure ends up in a long combinational path from p_i to the output. A solution for this is a tree structure as shown in Figure 10.2. The input values (distances) d_0 to d_{N-1} are first sorted so that the right and left input of the *compare* are shortest and longest distances respectively. Knowing that the input values are sorted, the *compare* function becomes easier and consist only of three compares which reduces the length of the combinational path. An overview of the checks done in the *compare* function are given in Table 10.1.

10.1.3.1 Conclusion: Tree structure sorting

The choice was made to realize a tree structure because it has a shorter combinational path compared to the foldl architecture and requires less hardware compared to sorting the entire input.

10.2 Outlier rejection



Section 5.1.3 describes several methods and results of *rejecting outliers* concluding that it could in some cases have a minor influence accuracy and reduce iterations needed for the solution to converge. The outlier threshold can be calculated in different ways and the results researched in Section 5.1.3 use the standard deviation and the mean as a threshold. Calculating the standard

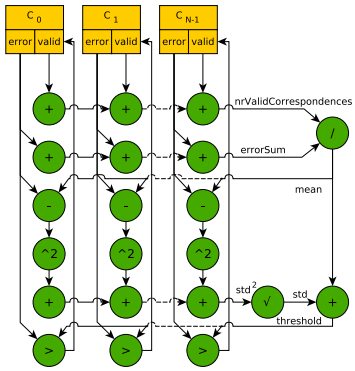


Figure 10.3: Calculation of the standard deviation and mean

deviation σ and mean μ with the error e_i between correspondences is shown in Equation 10.1.

$$\begin{aligned}
\sigma &= \sqrt{\frac{1}{N}((e_0-\mu)^2+(e_1-\mu)^2+...+(e_{N-1}-\mu)^2)} \\
\sigma &= \sqrt{\frac{1}{N}(\sum_{i=0}^{N-1}(e_i-\mu)^2)} \\
\mu &= \frac{1}{N}(e_0+...+e_{N-1}) \\
\mu &= \frac{1}{N}\sum_{i=0}^{N-1} e_i
\end{aligned} \tag{10.1}$$

A structural overview of this calculation is given in Figure 10.3. One can see that apart form the division and square root the algorithm has a regular structure and can be calculated over time in order to save hardware. Section 10.1 and Section 10.3.2 have a similar problem and the architecture to solve the mean + standard deviation can be realized in a similar way, as described in Section 10.3.2. Section 10.3.1.3 describes the QR decomposition method for minimizing the error. To remove outliers from the system created by the QR decomposition one can multiply all the columns of the Q matrix with a vector containing 1's and 0's indicating if a correspondence is valid and create a zero row in the Q matrix to create a zero equation in the system.

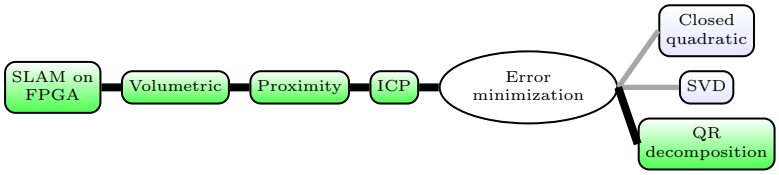
The choice was made to do no outlier rejection. As shown in previous sections the outlier rejection only influences the quality of the result in very specific cases (Figure 5.10, 5.9) and has some impact on the convergence rate of the algorithm. Realizing the outlier rejection including its irregularities of this algorithm (division and square root) requires development time which is not available after realisation of the necessary parts of the ICP algorithm, The structure of the algorithm, and where to split it over time has already been worked out but the realisation of outlier rejection will be future work.

10.3 Error minimization

Once the correspondences are found, the *transformation* must be calculated in order to determine the movement of the robot. The goal of the minimization process is to find the transformation between the current pose of the robot and the previous one. Different algorithms,

discussed in the literature study in Section 5.1.4, are compared in Section 10.3.1 and the choice was made to use a form of QR decomposition. Section 10.3.2 explains the hardware structures that can be used to realize the QR decomposition algorithm, which has further consequences for elements like memory (Section 10.3.3), operations like the inverse square root (Section 10.3.5), vector operations (Section 10.3.4), a linear solver (Section 10.3.6), and division (Section 10.3.7)

10.3.1 Error minimization algorithm



Section 5.1.4 describes several algorithms for minimizing the error and this chapter will contain the choices made and solutions found to do this minimization on hardware.

10.3.1.1 Closed quadratic form

The closed form method proposed by Censi [16] was used in MATLAB. Constructing the quartic function can be done in a regular structure. However, solving the fourth order polynomial in hardware requires a lot of roots, divisions, and power raising [4].

10.3.1.2 Least squares by approximation $\theta \approx 0$ using SVD

The method proposed by Low [37] and described in Section 5.1.4.2 approximates the rotation $\theta \approx 0$ to reduce the number of variables (from 12 to 6) for the linear least square problem for the 3 dimensional case. However, for the 2 dimensional case this is not needed since the number of variables is 4. Further on, Low proposes to use the SVD (singular value decomposition) for solving the linear least square problem. SVD requires one to find two orthonormal matrices and the square roots of the eigenvalues of the $\mathbf{A}$ matrix in the linear least square problem $\mathbf{A}\vec{x} = \vec{b}$. Calculating the orthonormal matrices can be done by using the Gram-Schmidt method. In case of a 2D transformation the $rank(\mathbf{A}) = 4$. Finding the eigenvalues of a 4×4 matrix comes down to solving a fourth order polynomial. As mentioned with the Closed quadratic form (Section 10.3.1.1), solving a fourth order polynomial requires a lot of hardware area/time to do roots, divisions, and power raising.

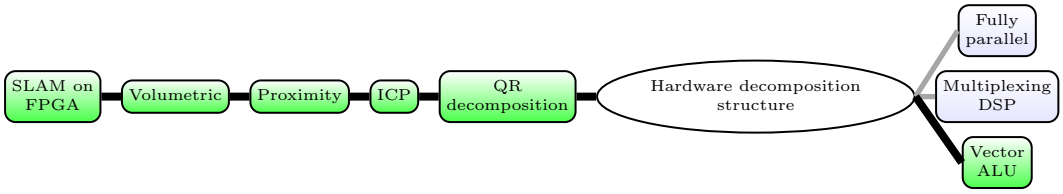
10.3.1.3 Least squares solver using QR decomposition

Section 12.1 explains the mathematical derivation and structure of a minimization method called *QR decomposition*. Advantages of this algorithm are its regular structure and that it mostly uses basic operations (multiplication and addition). The width of the entire system N is determined by the number of correspondences. In case of proximity sensors with a beam model the number of correspondences are equal between every two observation. Outlier correspondences can just be interpreted as a zero equation and will not affect the structure of the system. The algorithm itself mostly contains dot products, both for matrix vector and vector vector multiplications. Looking at the regular structure of the algorithm one can see that reusable hardware can be used. There are two downsides of this algorithm. First there are 4 divisions at the end (for the backward substitution in the row reduction process). Second, 4 inverse square roots are needed in order to normalize a vector (orthonormal vector space).

10.3.1.4 Conclusion: QR decomposition

Both the closed quadratic form and the Singular Value Decomposition method require solving a fourth order polynomial. The SVD method also requires constructing two orthonormal matrices while the QR decomposition method only requires one. Mainly due to its regular algorithm structure, vector dot products with fixed length, and known methods for fast inverse square root (Section 10.3.5) the choice was made to realize the QR decomposition method.

10.3.2 Hardware decomposition structure



Section 10.3 describes that the main reason to choose for QR decomposition is its regular structure of the algorithm and that it mainly consist of basic operations. In this chapter the realization choices of this algorithm will be described in more detail. The first part of this chapter consist of a full parallel algorithm structure. However, due to limited hardware the full parallel solution is not possible to synthesize on hardware and therefore two options are considered, multiplexing DSPs and a vector ALU structure. First the QR decomposition algorithm is explained into three steps, after which the full parallel solution is given with diagrams that show the regular structure. Due to its regular structure the algorithm can be split into parts and executed over time to reuse components with multiplexers and memory which is explained in Section 10.3.2.2. A third option is considered in Section 10.3.2.3 where an ALU structure is proposed.

The QR decomposition can be split up into three parts:

- Constructing the system $\mathbf{A}\vec{x} = \vec{b}$ from correspondences
- Constructing the orthonormal matrix $\mathbf{Q}$
- Constructing the (Echelon from) matrix $\mathbf{R} = \mathbf{Q}^T\mathbf{A}$, and vector $\vec{t} = \mathbf{Q}^T\vec{b}$

Each paragraph below will give a short recap of the parts

Constructing the A matrix To create the matrix $\mathbf{A}$ the following data from the correspondences are needed (Section 10.1):

- Point (p_{ix}, p_{iy}) from the new observation $(p_i \in \vec{p})$
- Closest point (m_{ix}, m_{iy}) to p_i, from the previous observation $(m_i \in \vec{m})$
- Normal vector (n_{ix}, n_{iy}) from p_i to m_i

The calculation for one row of the $\mathbf{A}$ matrix and $\vec{b}$ vector from a correspondence is restated below:

$$\underbrace{\begin{bmatrix} n_{ix} & n_{iy} & (p_{ix}n_{ix} + p_{iy}n_{iy}) & (p_{ix}n_{iy} - p_{iy}n_{ix}) \end{bmatrix}}_{\mathbf{A}_i} \quad \underbrace{\begin{bmatrix} m_{ix}n_{ix} + m_{iy}n_{iy} \end{bmatrix}}_{\mathbf{b_i}} \tag{10.2}$$

Constructing the orthonormal matrix Q In the QR decomposition process the *Gram-Schmidt* method is used to find the orthonormal vector space $\mathbf{Q} = [\vec{u_0}, \vec{u_1}, \vec{u_2}, \vec{u_3}]$. The calculations of these vectors is restated in Equation (10.3). One can see that every vector $\vec{y_i}$ needs to be *normalized*, for normalization a inverse square root is needed which is explained in Section 10.3.5. After the Gram-Schmidt method the **Q** matrix is the orthonormal space of **A**.

$$\begin{aligned}
\vec{u_0} &= \frac{\vec{y_0}}{|\vec{y_0}|} \Rightarrow \vec{y_0} = \vec{v_0} \\
\vec{u_1} &= \frac{\vec{y_1}}{|\vec{y_1}|} \Rightarrow \vec{y_1} = \vec{v_1} - (\vec{v_1} \bullet \vec{u_0})\vec{u_0} \\
\vec{u_2} &= \frac{\vec{y_2}}{|\vec{y_2}|} \Rightarrow \vec{y_2} = \vec{v_2} - (\vec{v_2} \bullet \vec{u_0})\vec{u_0} - (\vec{v_2} \bullet \vec{u_1})\vec{u_1} \\
\vec{u_3} &= \frac{\vec{y_3}}{|\vec{y_3}|} \Rightarrow \vec{y_3} = \vec{v_3} - (\vec{v_3} \bullet \vec{u_0})\vec{u_0} - (\vec{v_3} \bullet \vec{u_1})\vec{u_1} - (\vec{v_3} \bullet \vec{u_2})\vec{u_2}
\end{aligned} \tag{10.3}$$

Constructing the (Echelon form) matrix R and $\vec{t}$

$$\begin{aligned}
\mathbf{R} &= \mathbf{Q}^T\mathbf{A} \\
\vec{t} &= \mathbf{Q}^T\vec{b}
\end{aligned} \tag{10.4}$$

Multiplying $\mathbf{Q}^T$ with **A** will produce an 4×4 matrix **R**, note that the lower triangular is not needed in the **R** matrix, which reduces the number of computations. After the calculation of the **R** matrix and $\vec{t}$ vector for the system $\mathbf{R}\vec{x} = \vec{t}$, finding x_0, x_1, x_2, and x_3 is a matter of *row reduction*. Looking at the dot products needed to create the orthonormal matrix **Q** (see Equation 10.3) one can see that these dot products are also done in the computation of the $\mathbf{Q}^T\mathbf{A} = \mathbf{R}$ matrix. So the results from dot products used in the **Q** construction can be *reused* in the computation of **R**.

10.3.2.1 Fully parallel

This section will show fully parallel architectures for each part of the QR decomposition.

Constructing the A matrix The calculation for one row of the **A** matrix and $\vec{b}$ vector from a correspondence is restated below:

$$\underbrace{\begin{bmatrix} n_{ix} & n_{iy} & (p_{ix}n_{ix} + p_{iy}n_{iy}) & (p_{ix}n_{iy} - p_{iy}n_{ix}) \end{bmatrix}}_{\mathbf{A}_i} \quad \underbrace{\begin{bmatrix} m_{ix}n_{ix} + m_{iy}n_{iy} \end{bmatrix}}_{\mathbf{b_i}} \tag{10.5}$$

A structured view of the construction for the entire matrix **A** end vector $\vec{b}$ from the correspondences $C_0, C_1, ..., C_{N-1}$ can be seen in Figure 10.4.

Constructing the orthonormal matrix Q In the QR decomposition process the Gram-Schmidt method is used to find the orthonormal vector space **Q**. Figure 10.5 shows the structure one iteration of the Gram-Schmidt method. In this figure the previous normalized vector, $\vec{u_0} = \vec{v_0}/||\vec{v_0}||$, is used in a dot product with the vector, $\vec{v_1}$, and scaled accordingly. The outcome of this dot product is subtracted from $\vec{v_1}$. To use the outcome $\vec{y_1}$ in the next Gram-Schmidt iteration one only needs to normalize : $\vec{u_1} = \vec{y_1}/||\vec{y_1}||$. In a full parallel architecture one can put 4 of the same structures of Figure 10.5 after each other to create the orthonormal space **Q**.

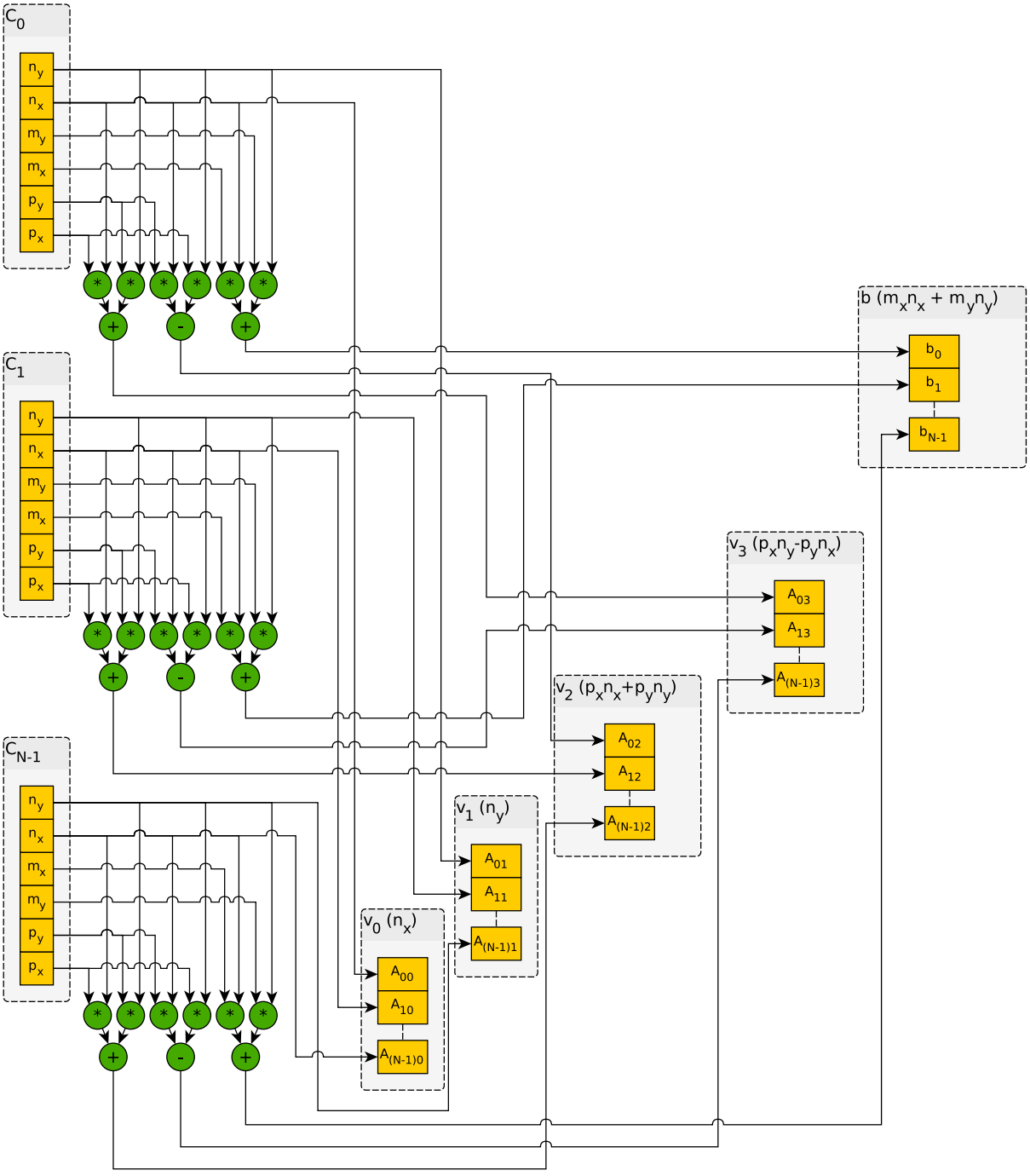


Figure 10.4: Construction of the **A** matrix and $\vec{b}$ vector using correspondences $C_0, C_1, ..., C_{N-1}$

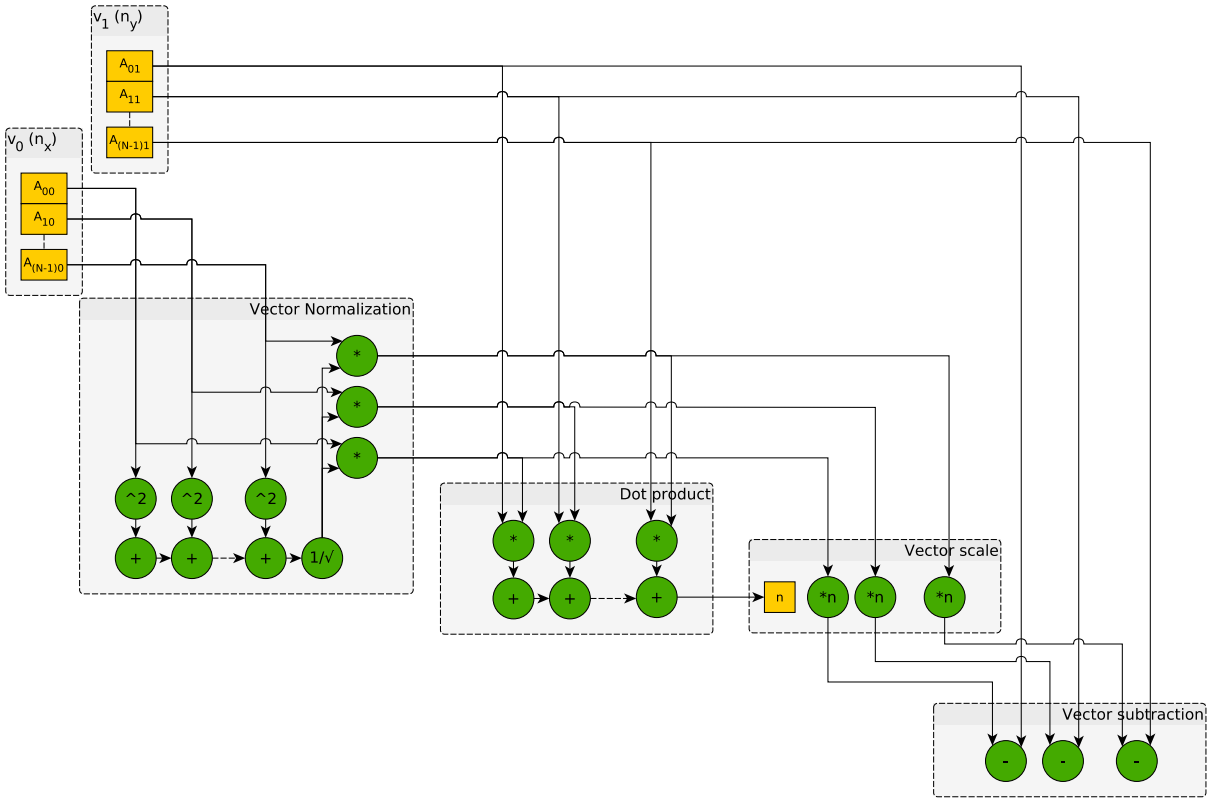


Figure 10.5: One iteration of the Gram-Schmidt, $\vec{y_1} = \vec{v_1} - (\vec{v_1} \bullet \vec{u_0})\vec{u_0}$

Constructing the (Echelon form) matrix R and $\vec{t}$ Figure 10.6 shows the full parallel architecture needed to construct the matrix **R** and vector $\vec{t}$. The optimization that some of the results of the dot products used in the Gram-Schmidt for calculating the orthonormal matrix **Q** are also needed in the matrix **R**. Note that this optimization is not shown in the figure. Using the variables calculated in the Gram-Schmidt method reduces the computation for the **R** matrix to only calculating the diagonal $(r_{00}, r_{11}, r_{22}, r_{33})$. The values above the diagonal are from the Gram-Schmidt method, and the values below the diagonal are 0.

10.3.2.2 Multiplexing DSPs

Table 10.2 shows the components used for constructing the linear system $\mathbf{A}\vec{x} = \vec{b}$, constructing the orthonormal matrix **Q**, and constructing the matrix **R** and vector $\vec{t}$. All operations require a multiple of N multipliers and adders. This regularity is also visible in the construction of selecting correspondences (Section 10.1). Apart from the 4 inverse square roots, the QR decomposition contains only the two basic operators; multiplication and addition which could be reused over time to reduce the number of components. Reusing components for operations introduces two aspects:

- Memory, for storing data over time
- Multiplexers, for selecting different input data for the multipliers

A schematic overview of the principle of reusing components with memory and multiplexers is shown in Figure 10.7 where 1 iteration of the Gram-Schmidt methods is done by reusing the multipliers. Multiplexers are used to select the input for the multipliers and the yellow blocks represent storage of data. It is important to note that the order of execution matters,

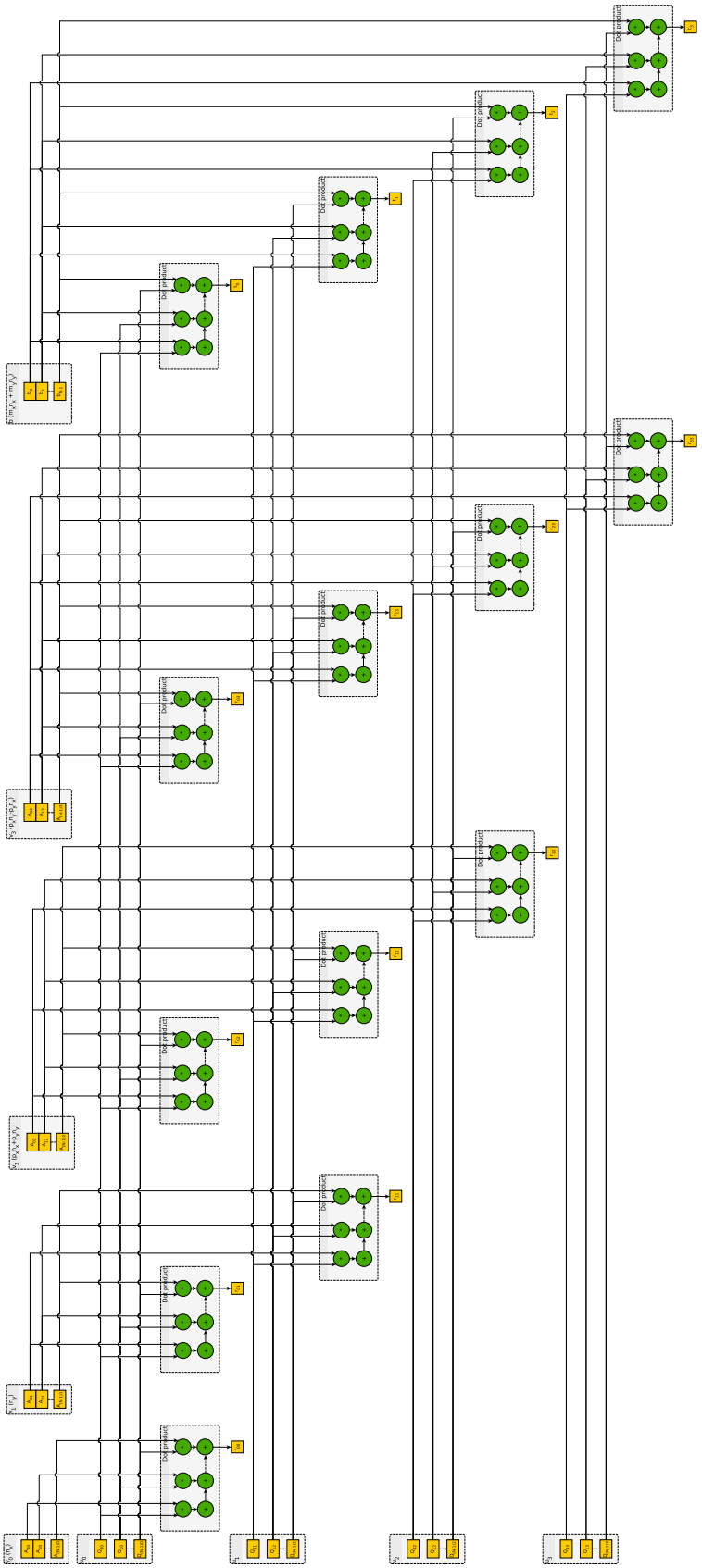


Figure 10.6: Calculate $\mathbf{Q}'\mathbf{A}$ and $\mathbf{Q}'\vec{b}$

Part		Multipliers	Adders
$\mathbf{A}\vec{x} = \vec{b}$	$\vec{v_0}$	0	0
	$\vec{v_1}$	0	0
	$\vec{v_2}$	$2N$	N
	$\vec{v_3}$	$2N$	N
	$\vec{b}$	$2N$	N
$\mathbf{Q}$	$\vec{u_0}$	$2N$	N
	$\vec{u_1}$	$4N$	$3N$
	$\vec{u_2}$	$6N$	$5N$
	$\vec{u_3}$	$8N$	$7N$
$\mathbf{Q}^T\mathbf{A}$	$\mathbf{R}$	$4N$	$4N$
$\mathbf{Q}^T\vec{b}$	$\vec{t}$	$4N$	$4N$
Total components		$34N$	$27N$

Table 10.2: Components required for a full parallel realization of the QR decomposition with N = number of correspondences

for example $\vec{u_0}$ must be known before able to calculate $\vec{u_1}$. This puts a constraint on the order of execution if calculations are done over time. The idea of splitting data and reusing operators can be drawn even further by also splitting up vectors. Figure 10.8 shows a structure in which a vector is split up in 4 parts of length M and then multiplexed through the multipliers.

10.3.2.3 Vector ALU

Almost every step of the algorithm, for both selecting and minimizing, multiplications and additions are the only operations needed. A solution for reusing those operations for all steps is an ALU structure as depicted in Figure 10.9. To realize such an ALU structure several choices must be made regarding the memory and vector operations. The ICP algorithm does not only contain vector operations but also single value operations like solving the linear system $\mathbf{R}\vec{x} = \vec{t}$, for these single operations different hardware is generated which is described in Section 10.3.6.

10.3.2.4 Conclusion: Vector ALU

Hardware limitations such as the number of multipliers, and logical units on the FPGA described in Appendix A are the reason the choice was made to design the Vector ALU structure. Further choices regarding the structure of the memory, vector operations, and linear solver are described in Section 10.3.3, 10.3.4, and 10.3.6 respectively.

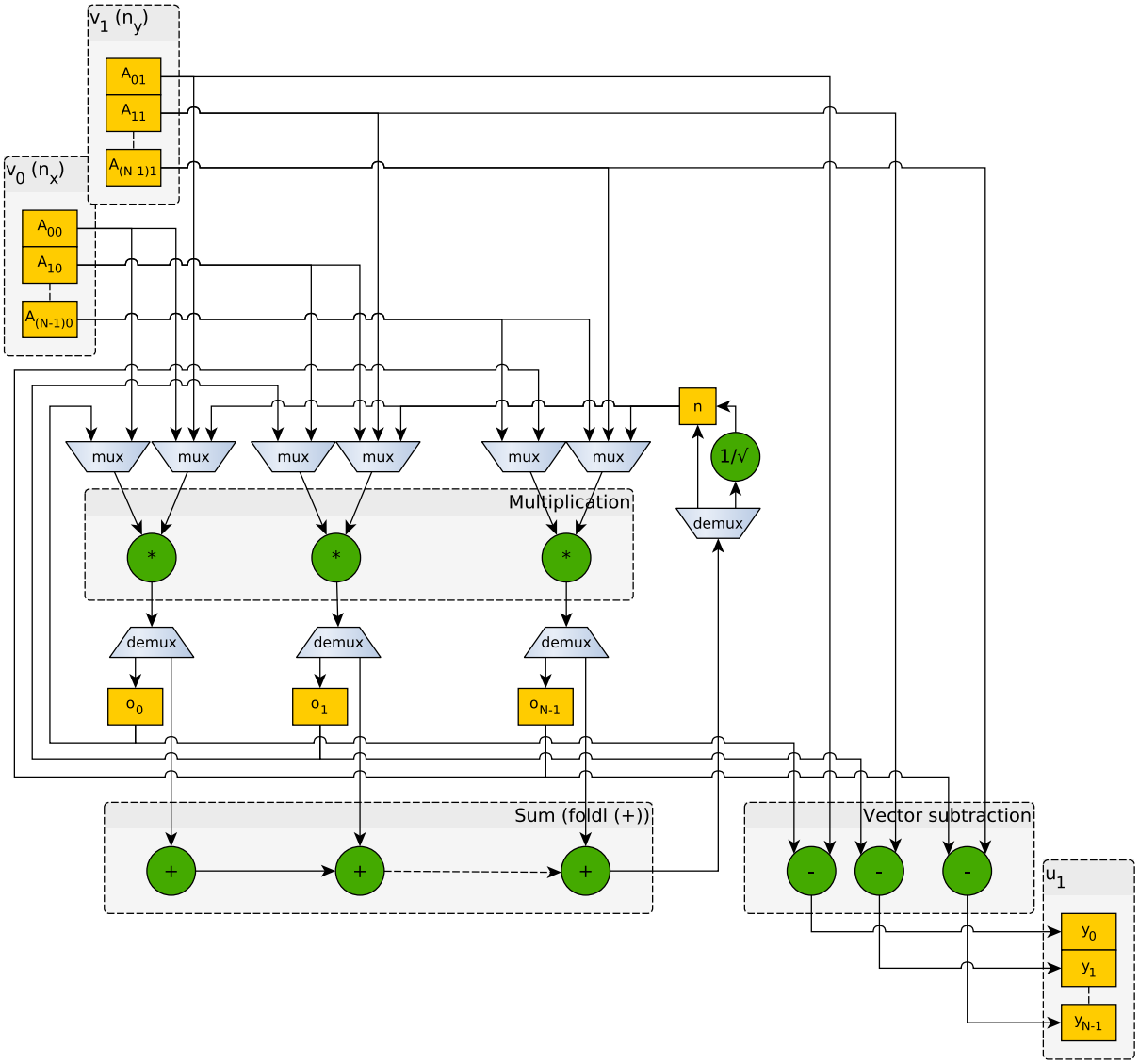


Figure 10.7: Structure of 1 iteration of the Gram-Schmidt, $\vec{y_2} = \vec{v_2} - (\vec{u_1} \bullet \vec{v_2})\vec{u_1}$

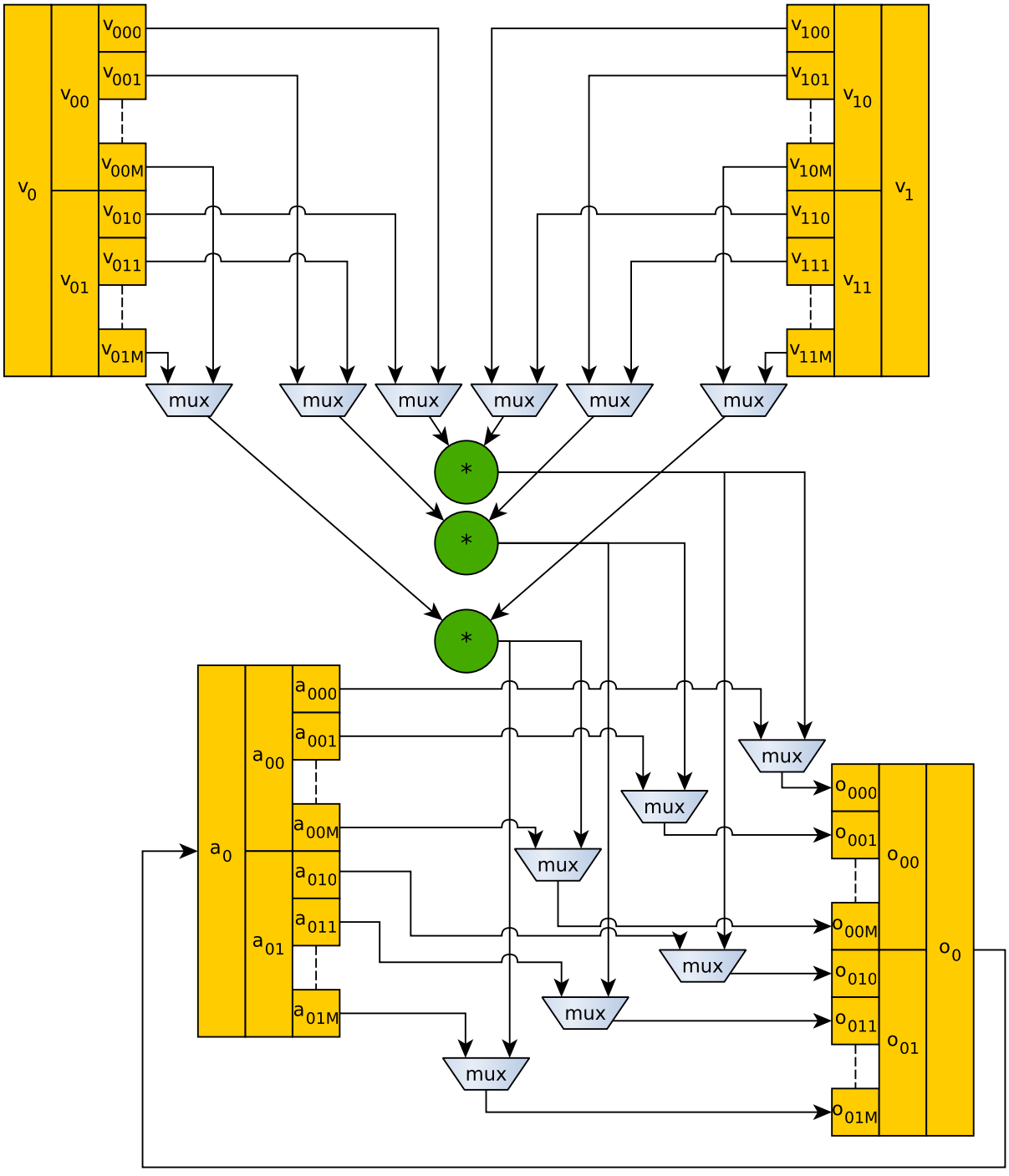


Figure 10.8: Vector multiplication split up into parts to calculate over time

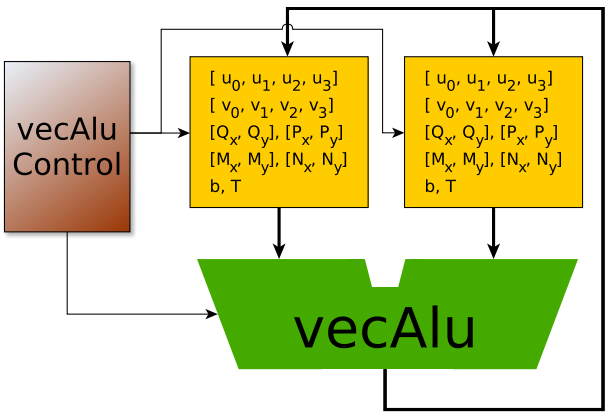
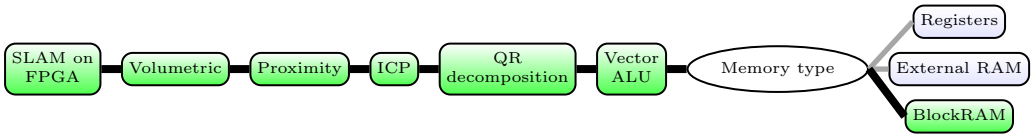


Figure 10.9: Memory with vector ALU structure for reusability of hardware operators

10.3.3 Memory layout



The structure of memory determines for a large part the structure of the entire solution. For example, if the memory component can provide only one value of an entire vector at the time then this will affect the structure of the computation. Storage on an FPGA can be done in three ways:

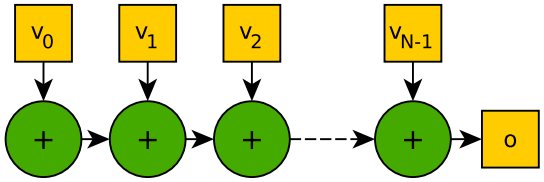
- Registers
- On-board memory called blockRAM
- External RAM

Constructing the **Q** matrix as described above, reusing the multiplications, with $N = 180$ multipliers with registers requires, according to the synthesis tooling $\pm$17000 multiplexers. This does not take the vector splitting part into account. Storing all the values in registers and multiplexing them in order to select the right values consumes a lot of hardware. BlockRAMs are specialized to do storing and fetching of data. However, on FPGA the number and depth of blockRAMs is limited. When data becomes too large external RAM can be used but this requires special drivers in order to be able to use them. The FPGA, described in Appendix A, contains 5140 kbits of blockRAM. Depending on whether to use a fixed-point data size of 1-8 or 27-bits the realized ICP solution requires 7 % or 11 % utilization. Since there is no need for external RAM the choice was made to use blockRAMs.

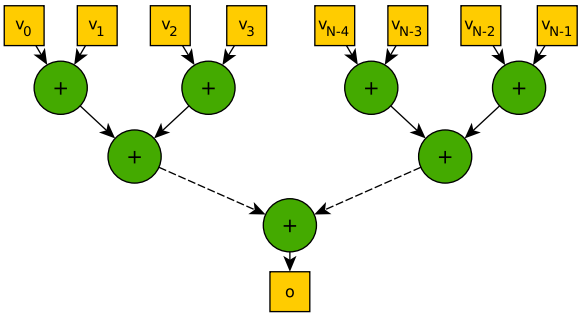
BlockRAMs provide data one clock cycle after the request for it. Creating a single blockRAM containing all the vectors will create an extra delay for loading two different vectors. One can choose to create a pipeline structure to reduce these idle cycles. This pipe-lining will create a more complex data and control path. Since the entire data for the algorithm uses only 7 % or 11 % of all the blockRAMs available the choice was made to create a dual blockRAM structure with duplicated data. This dual structure keeps the control and data path less complex and will allow one to simultaneously read two vectors and do a computation.

When the vector ALU calculates a summation (for dot product) it produces a single result which is stored in a blockRAM called *number blockRAM* . The number blockRAM contains single values that can be used for either vector scaling or for solving a linear system $\mathbf{R}\vec{x} = \vec{t}$. For single number operations another ALU is created called numberAlu which is described in Section 10.3.6. The numberAlu, like the vectorAlu, requires two inputs and for the same reason as with the vectorAlu the choice was made to duplicate the number blockRAMs as the vector blockRAMs are duplicated. A general overview is shown in Figure 10.13.

To reduce control complexity and increase speed the choice was made to use duplicate blockRAM structure for both vectors and numbers.

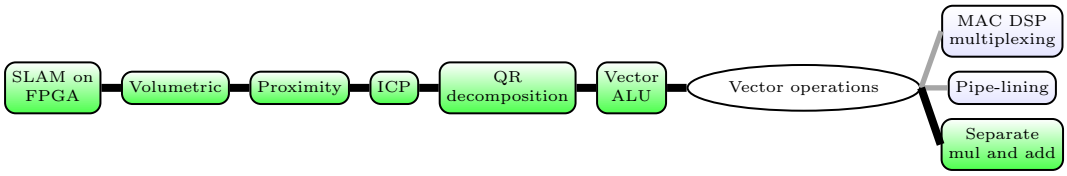


(a) Summation using a *foldl* construction



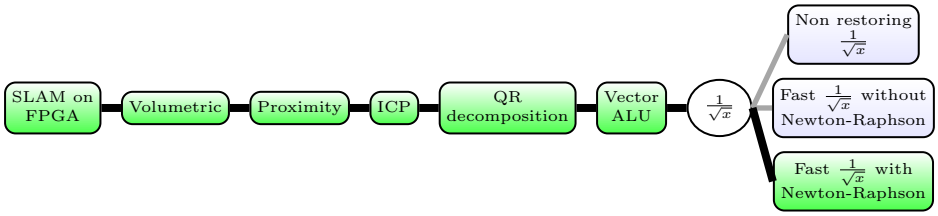
(b) Summation using a tree structure

10.3.4 Vector operations



There are two types of operations, namely parallel and sequential (with data dependency). Operations like multiplication, scaling, addition, and subtraction on vectors do not have any data dependency and can be done in parallel. Summation is an operation with data dependency and doing it with a fold left construction (Figure 10.10a) will result in a long combinational path from v_0 to o. This could be solved with pipeline stages but that introduces timing differences in the different vector operations which would make the control much more complex. A similar problem can be found in the Selection process (Section 10.1), and a similar solution can be applied here; a tree structure (Figure 10.10b). The disadvantage of a tree structure is that MAC (Multiply Accumulate) units cannot be applied. The DSP onboard an FPGA are usually capable to do a multiply accumulate, but this requires a foldl structure as depicted in Figure 10.10a. But a single DSP has a longer propagation delay than a synthesised adder, therefore, putting N DSPs in sequential order will result in a large propagation delay. The propagation delay could also be reduced by pipe-lining, but in order to reduce the propagation delay and still keep the control straight forward, without different timing paths, the choice was made to have separate hardware for parallel operations and sequential operations. Sequential operations are realized using a tree structure.

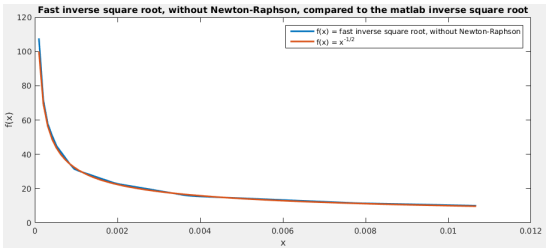
10.3.5 Inverse square root



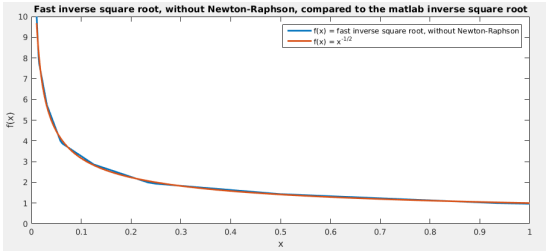
Inverse square roots are needed in two places of the algorithm:

- Constructing correspondences: creating the normal vector
- Error minimization: normalizing vectors for the orthonormal vector space

Many square root algorithms are available but in this report two are considered: Non-restoring square root and Fast inverse square root (with or without Newton-Raphson steps)



(a) Inverse square roots from $x = 0.0001$ to $x = 0.0107$



(b) Inverse square roots from $x = 0.0107$ to $x = 1$

Figure 10.11: Fast inverse square root, without Newton-Raphson, compared to the MATLAB inverse square root

10.3.5.1 Non-restoring square root

A method for solving binary a square root problem is the non-restoring square root [46], which uses the same principles as the non-restoring division (shifting register with partial and remainder). The non-restoring square root is an iterative algorithm and each iteration computes 1-bit of the square root. Iterating the algorithm as many times as the number of bits of the input provides equal precision and accuracy as the input.

10.3.5.2 Fast inverse square root

A fast and efficient way of computing the inverse square root of a number that is used in hardware design is known as the Fast Inverse Square root [53]. The design of this algorithm has been used in game engines in the early 90's. At that time they used the method to avoid using computational heavy floating point operations. The algorithm is described in Algorithm 4. On the FPGA, as described in Appendix A, fixed point variables are used in for all DSPs.

Algorithm 4 Fast inverse square root $s = n^{-\frac{1}{2}}$

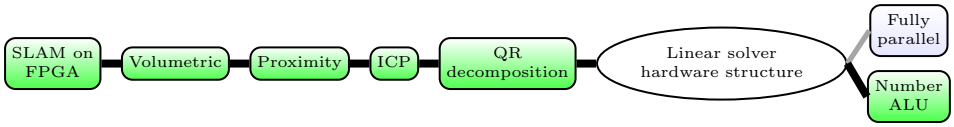
1: $x = n$ ▷ interpret the 32-bit floating point representation n, as an 32-bit Signed integer x
2: $y = x >> 1$ ▷ shift x to the right
3: $z = (0x5f3759df - y)$ ▷ subtract y from a 'magic number'
4: $k = z$ ▷ interpret the 32-bit integer representation z, as a 32-bit floating point number k
5: $//s = k * (\frac{3}{2} - (\frac{n}{2}k^2)$ ▷ (optional) one iteration of the Newton's method

Therefore, the algorithm require two extra steps; the conversion from fixed point to a 32-bit floating point representation and back. Beside the conversion the algorithm requires an adder, a shifter, a couple of multiplexers, and 3 multipliers for the Newton-Raphson method. It is however possible to do the inverse square root without Newton's method, Figure 10.11 plot the difference between the fast inverse square root method and the inverse square root from MATLAB. Figure 10.11a shows the part from 0.0001 to 0.0107 and Figure 10.11b shows the part from from 0.0107 to 1. These figures show that the fast inverse square root is relatively accurate and without the Newton-Raphson method it does not require 3 multipliers. However, simulation results, as described in Section 12.3, show that without Newton-Raphson the algorithm fails to converge.

10.3.5.3 Conclusion: Fast inverse square root

The choice was made to use the fast inverse square root method with 1 iteration of Newton-Raphson because it does not require iterations as the non-restoring square root and should be accurate enough for finding transformations.

10.3.6 Linear solver hardware structure

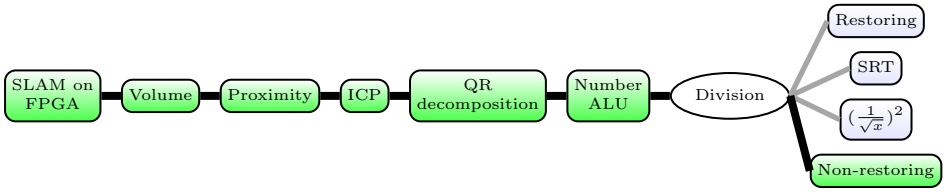


After the QR decomposition the system $\mathbf{R}\vec{x} = \vec{t}$ must be solved in order to compute the transformation between the observation $\vec{p}$ and $\vec{q}$. The row reduction process using backward substitution is shown in Equation 10.6.

$$\begin{aligned} x_3 &= \frac{t_3}{r_{33}} \\ x_2 &= \frac{t_2 - r_{23}x_3}{r_{22}} \\ x_1 &= \frac{t_1 - r_{12}x_2 - r_{13}x_3}{r_{11}} \\ x_0 &= \frac{t_0 - r_{01}x_1 - r_{02}x_2 - r_{03}x_3}{r_{00}} \end{aligned} \tag{10.6}$$

A parallel structure can be seen in Figure 10.12. Here one can see that also the division component is assumed to be a 1 clock cylce operation, in practice this is not the case. Many hardware dividers have been designed, Section 10.3.7 briefly describes some and motivates the chosen one. Linear solving occurs only once per ICP iteration, this means that for every operator used in the linear solver the utilization is very low. For this reason the choice was made to do all the linear solving operations in sequential time only using one DSP for the multiplication and addition. Looking at the QR decomposition algorithm one can see that the matrix $\mathbf{R}$ and vector $\vec{t}$ are constructed by the vector ALU using dot products. These dot products produce a single values and those are stored in a blockram containing numbers. Since the number blockrams are also duplicated, the same principle can be applied on the number ALU as is done for the vector ALU; using different read addresses both operands can be selected and writing the output to both blockrams in order to keep the data the same.

10.3.7 Division



Over the years many research has been done in the area of division [43]. Many of these algorithms, like SRT, Newton-Raphson, series expansion, have their own advantages and implementation costs. One could spend an entire master thesis finding out what is the most optimal solution. Section 10.3.5 explains a method for finding a relative accurate inverse square root in just one clock cycle. The inverse square root can also be used in a division:

$$\frac{x}{y} = x \cdot \frac{1}{y} = x \cdot \frac{1}{\sqrt{y}^2} = x \cdot \frac{1}{\sqrt{y}} \cdot \frac{1}{\sqrt{y}} \tag{10.7}$$

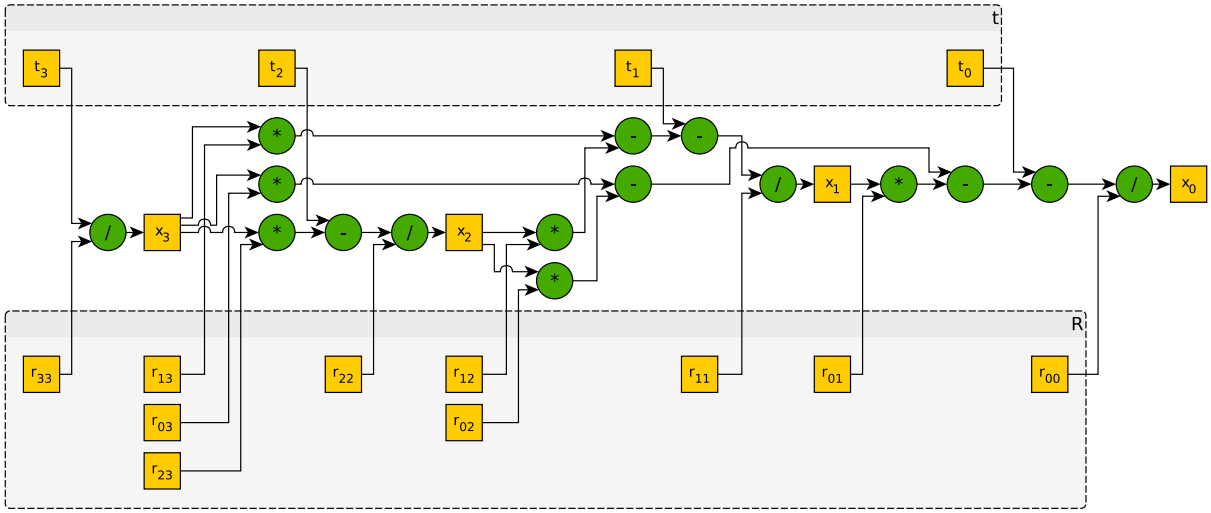


Figure 10.12: Parallel row reduction to solve $\mathbf{R}\vec{x} = \vec{t}$

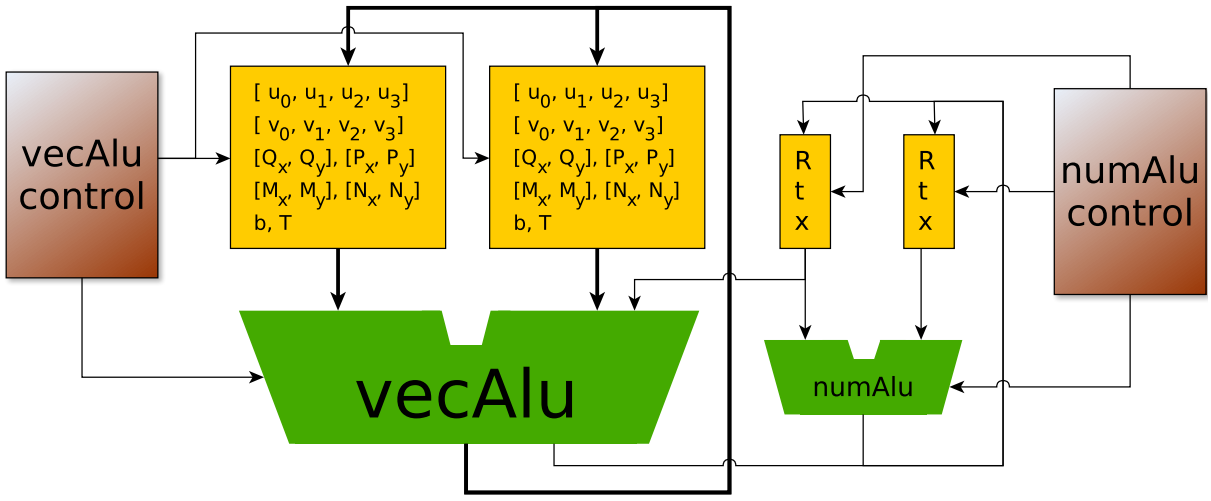


Figure 10.13: General overview of structure with vector ALU and number ALU with dual blockrams

In calculating the normal vectors of correspondences the accuracy of the inverse square root does not have a large impact on the results. But in this division the inaccuracy has a larger impact, also the inaccuracy is squared because of the multiplication. Because division is such a small part of the entire algorithm the choice was made to implement one of the most straightforward division for signed fixed point variables: non restoring division [46].

Part III

Realisation and results

11 — Graph-based SLAM

11.1 Realisation

11.1.1 Application specific multi use ALU

The ALU structure is used to execute vector operations such as pairwise operations and sum operations. Supporting operations on numbers can be achieved by adding control to the ALU that makes the ALU interpret the operands differently. This interpretation can be controlled by the operation-control signal which will be called "opcode". When the ALU is fixed as for example a single number multiplier, the first item of a vectors is taken into account and the rest of the vector is set to zero to save energy, since logic that is not transitioning will not use energy. In the ALU division is used which is not a vector operation and therefore only one divider is available.

11.1.2 Fixed size sparse vectors

The choice for a fixed size sparse vector as columns of the information matrix has been made to reduce computational complexity of the algorithm. The structure of one column has been shown in Figure 11.1. One columns contains five elements from which three elements are $d-1$, d, and $d+1$ which respectively are the element directly above the diagonal, the diagonal and the element directly below the diagonal. The other two elements are the incoming lc_o and lc_i, which are the outgoing and incoming loop closing ports respectively. By using this structure as column vector in the information matrix, the quality of the algorithm will be maintained and the amount of computations and memory use will be reduced.

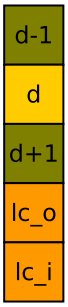


Figure 11.1: Fixed size sparse vector as column of the sparse information matrix

size of the state vector	storage method	amount of items	amount of bits per item	amount of bits
336	sparse	1.7*10^3	27	45 kb
336	dense	5.6*10^5	18	2 Mb
2000	sparse	1.0*10^4	29	290 kb
2000	dense	4.0*10^6	18	72 Mb

Table 11.1: Memory usage for different system sizes and storage methods

11.1.3 Multiple blockRAM's for efficient and fast access

BlockRAMs are a type of memory that can be used within an FPGA implementation. The width of the data is determined by the user which means the amount of data fetched simultaneously is defined by the user. However, it is only possible to read one block of the data with a defined width at one moment in time. Almost all instructions take two different operands fetched from these blockRAMs, which means if the data that is read is in different blocks, reading multiple variables will be done in multiple clock cycles. Section 10.3.3 describes an in depth motivation for using multiple dual versions of block rams to be able to read two different vectors at the same time for one operation.

The implementation uses sparse vectors and non-sparse vectors side-by-side, both vectors can be processed by one ALU. Sparse and non-sparse vectors have different data formats, additional logic is added to format the vectors in such a way that they can be used in the same operation. Sparse vectors and non-sparse vectors are stored in different instances of blockRAM memories, because they have different data lay-outs. The read addresses of the blockRAMs that store sparse and non-sparse matrices are connected to the same control signal so both data is fetched from the two memories simultaneously. Which data is multiplexed into the ALU is determined by another control signal.

11.1.4 Proposed ALU structure

The properties discussed during the design space exploration have been used to make an implementation of the graph-based SLAM algorithm on an FPGA. The resulting structure of the implemented hardware structure is shown in Figure 11.2. The architecture consists of a datapath that consists the logic used for computations. The ALU and other combinational paths are shown by green blocks. The yellow blocks are storage elements. The nine large yellow blocks are blockRAMs and the small ones are registers. The large red block is the control block which configures the other blocks to read from the correct inputs and write to the correct outputs. The control lines are denoted by lines that enter and leave blocks horizontally, the data is denoted by vertical connections.

In table 11.1 the memory use of a matrix is analysed. The difference in growth of memory between a ordinary and a sparse matrix is shown by comparing the amount of memory it would require to store the matrices. The size of a matrix when being dense matrix is defined by the number of items times the size per item. For sparse matrices, the total size is also defined by the number of items times the size of the items, but the size of the item is larger, because the index requires memory for storage as well. Storage of sparse items is performed by storing both the value and the index of an item. The amount of bits of the index can be calculated using the maximum number of items that the index can represent. The function that depicts the number of bits is a logarithm of the number of items rounded upwards.

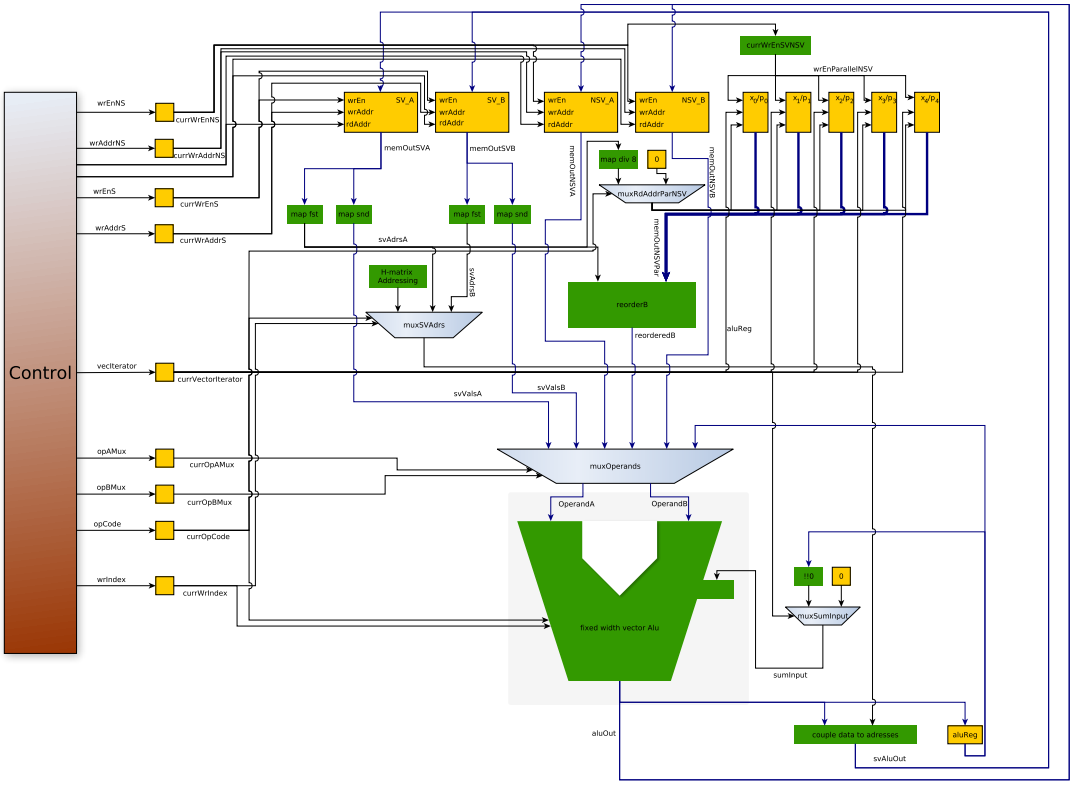


Figure 11.2: Sparse vector ALU

$$n\ of\ bits: \lceil \log_2 (n_{items}) \rceil$$

The SLAM problem that has been analysed in this thesis comes from the SLAM dataset from the Intel research building. The first complete loop of this dataset contains 336 poses. With the help of this logarithm the total amount of bits for a system with 336 poses becomes 18 for the data and $\lceil \log_2(336) \rceil = 9$ for the index. The graph in Figure 11.3 shows that there is a huge difference between the sparse and non-sparse memory usage. The larger the state vector becomes, the more rapidly the memory usage will increase.

The implemented architecture contains one transition that connects a data output to a control input. This is the vertical (data) connection between the indices of the sparse vectors read from the memories, which is connected to the horizontal (control) signal read address input of the shadow copies of the non sparse vector from which certain indices need to be fetched. The connection is marked in red in Figure 11.4.

11.1.5 Controlling the ALU

The operation that requires both parallel multiplication and summation is the dot product. The data path can be controlled to perform the dot product between a sparse vector and a dense vector or between two sparse vectors.

The control path performing a dot product of two 16-wide dense vectors with the given architecture can be done with an 8 wide vector ALU. The steps to perform this are described in the enumeration below. Each number represents a clock cycle.

1. • Load first 8-wide part of a vector

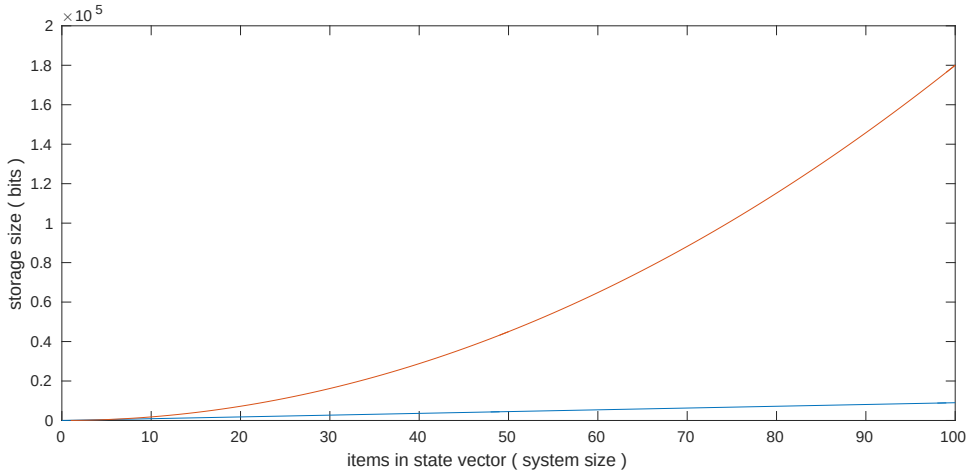


Figure 11.3: Storage sizes of sparse and non-sparse storage methods compared

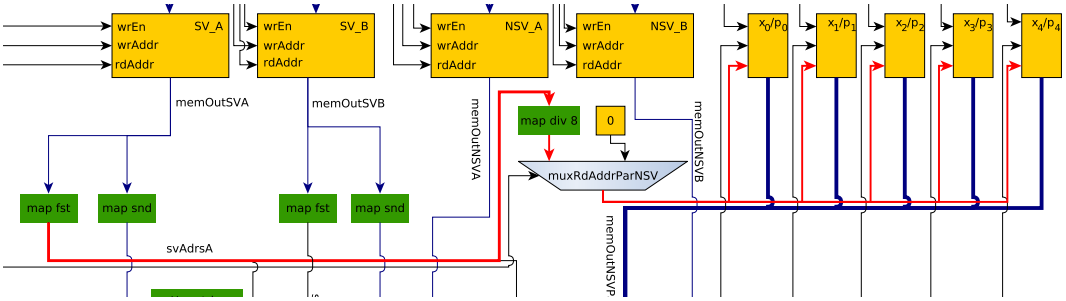


Figure 11.4: A vertical output connected to a horizontal input, which means data is used for control

2.
   - Load second vector part of the vector
   - Calculate the multiplication of the dot product on the first vector part
   - Store the multiplied first part of the vector
3.
   - Load first multiplied part of the vector
   - Calculate the multiplication of the dot product of the second part of the vector
   - Store the multiplied second part of the vector
4.
   - Load second multiplied part of the vector
   - Calculate the first summation of the dot product
   - Store the sum of the first part of the vector
5.
   - Calculate second summation and start addition from the previous found summation
   - Store the found sum in the correct memory

The enumerations shown above shows the steps in which the ALU can be used to perform a dot product. Each instruction contains can contain a load, a computational and a write instruction. The calculation is performed one clock cycle later because only then the data from the memory is present. Because of this delay, everything else will happen in the next clock cycle. This behaviour can be found in the structure of the ALU by the registers behind the control signals, except behind the memory that perform reading.

Table 11.2: Amount of clock cycles necessary to perform conjugate gradient

Type	Instances	clk cycles	parts	Total
dotSVNSV	336	3	1	1008
dotNSV	3	2	42	252
scaleNSV	3	1	42	126
subNSV	3	1	42	126
addNSV	1	1	42	42
div	2	1	1	2
total				1556

The complete implementation consists of similar instructions, at least their structure is always the same. To create a minimum amount of delay, the control has been implemented in such a way that the ALU is idle as little time as possible. The implementation creates a continuously flowing data stream to the ALU which results in a high throughput for such a small system at such a low frequency.

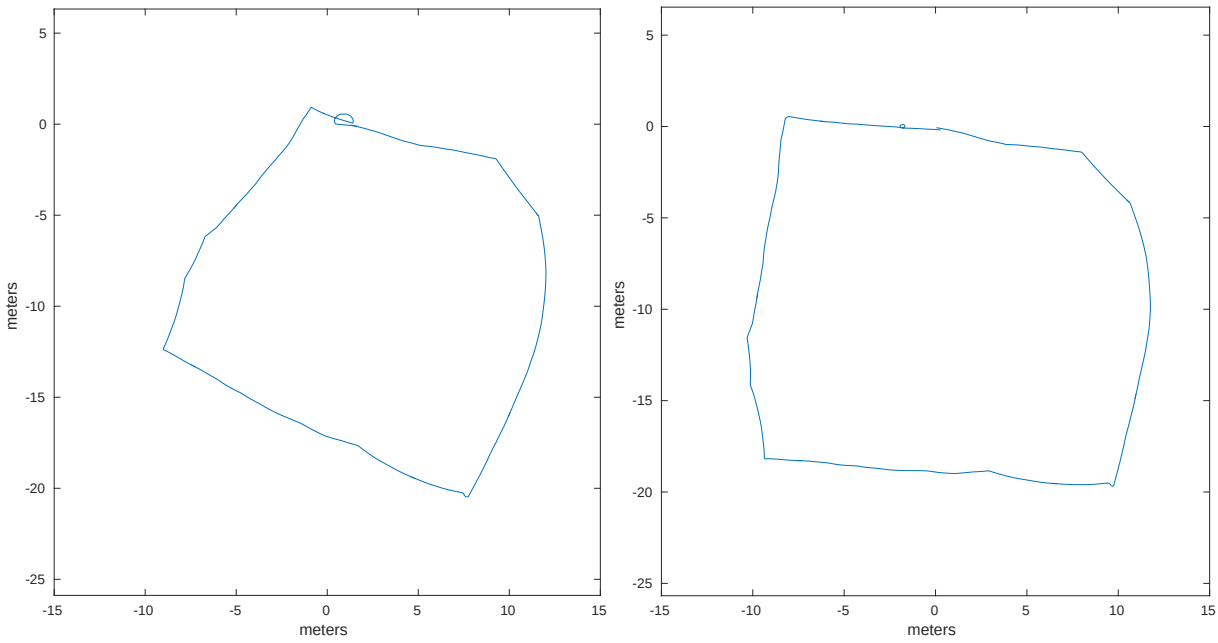
11.2 Results

The current SLAM implementation consists of a system which has a size of 336 poses. The 336 poses contain one loop of the Intel dataset that has been discussed before. The vector width of the implementation of the ALU has been set to 8, which means 8 computations will be done in parallel. Since the load of the SLAM algorithm mainly consists of correction of the graph by minimizing the error, this part will be analysed and is realized in hardware. The processor completes vector operations in multiple clock cycles, each vector that has a length of 336 items will split into $\frac{336}{8} = 42$ parts. The width of the vector ALU can be changed easily. Operations between sparse and non-sparse vectors require additional actions that consist of using indices of sparse vectors to lookup values of a non-sparse vectors with the help of copies of the vector in multiple blockRAMs. This method was used to perform a dot product between a sparse vector and a dense vector.

Given the amount of vector parts in one non-sparse vector, the amount of clock cycles for one iteration of the algorithm can be determined. Given the resources necessary for the algorithm which has been shown in table 9.1, the amount of clock cycles the algorithm demands are become as shown in table 11.2.

Figure 11.5 shows the results of a graph that has converged by solving the linear systems with the conjugate gradient hardware implementation. Both many iterations of graphslam with few iterations of conjugate gradient and vice versa give converged results. The right figure shows a better estimate of the path than the left figure. The amount of iterations that is necessary to get to a sufficient estimation of the exact vector solution is dependent on the size of the system to solve. The accuracy of the resulting vector and the convergence rate is also dependent on the error between the exact vector and the initial guess that the conjugate gradient algorithm is started with $\vec{x_0}$. An accurate proposal of the initial guess will be the input vector for the conjugate gradient algorithm. Using a method to create an initial vector that is any good, will decrease the amount of iterations drastically.

It is possible to stop iterations in two ways, the first is fixing the amount of iterations to a pessimistic amount so the result will be correct. The other way of stopping the iterations is by termination whenever the error drops below a threshold combined with an fixed maximum amount of iterations to prevent the algorithm from executing an infinite amount of iterations.



(a) 10 iterations of graph SLAM and 5000 conjugate gradient iterations per cycle

(b) 5 Iterations of graph SLAM and 10000 conjugate gradient iterations per cycle

Figure 11.5: Resulting paths with different amount of graph SLAM error convergence iterations and conjugate gradient iterations

From these examples, the right-hand figure gives the best result by using more conjugate gradient and less iterations of graph-SLAM.

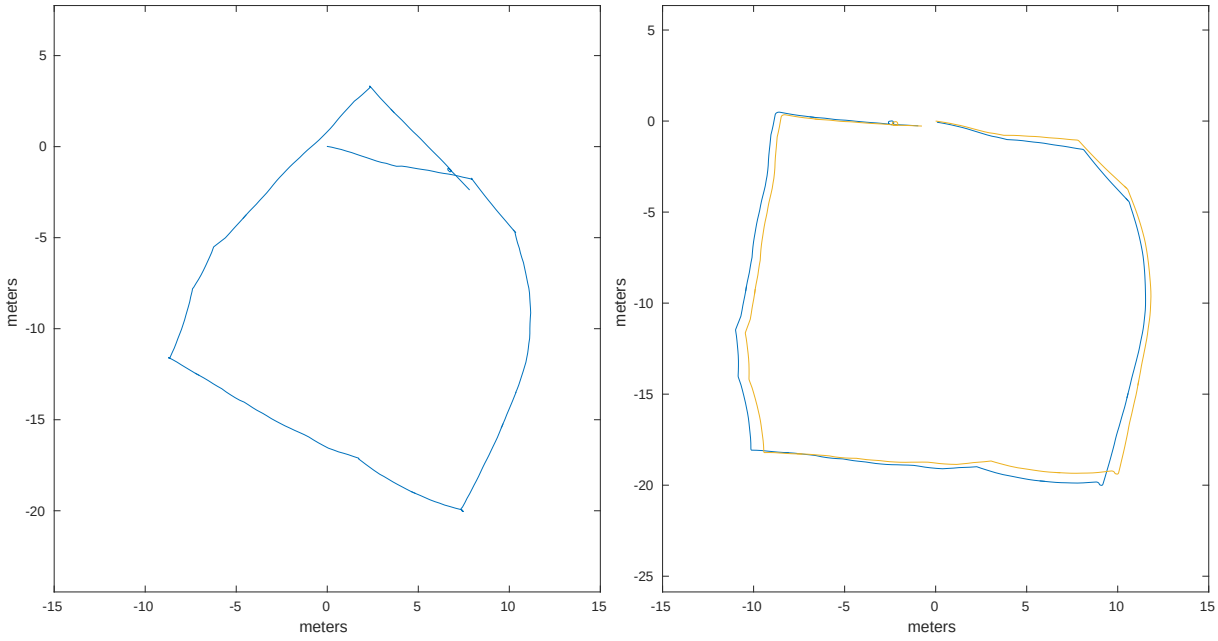
The right figure of Figure 11.6 shows the difference between a graph converged in MATLAB (the yellow path) and the results created by the conjugate gradient algorithm in hardware. The results of the hardware implementation are very comparable despite the fact there is a small difference in the angle between the graphs. In normal scenarios both these maps are of sufficient quality to use for navigation.

11.2.1 MATLAB timing results

To make a comparison to hardware, the graph-based SLAM algorithm has been implemented in MATLAB. The linear solver that MATLAB uses does not have to restrict memory use like the hardware conjugate gradient solver. In MATLAB dynamic allocation and freeing of memory can be used. An implementation that uses sparse matrices of the conjugate gradient algorithm in MATLAB shows convergence of a single dimension of the SLAM algorithm in an average of 740 ms. The linear solver that MATLAB uses is about an order of magnitude faster than the conjugate gradient implementation on MATLAB. Using the MATLAB linear solver requires significantly more memory than the hardware implementation.

11.2.2 Hardware results

The graph-based SLAM implementation has been implemented using CλaSH and the amount of clock cycles per iteration are found. The transformation from CλaSH to VHDL was not successful which means timing results are difficult to give. Due to time, hardware synthesis was not prioritized. However, because the implementation is performed while knowing the limitations of hardware synthesis, the vector ALU structure should not contain extensively long



(a) Graph before convergence

(b) Converged graph by hardware conjugate gradient (blue) and MATLAB's linear solver (yellow)

Figure 11.6: Results comparison of hardware conjugate gradient against MATLAB

combinational paths. The largest combinatorial path that would then occur will be the sum operator. The sum operator sums the vector blocks which is used in the dot product. With the vector size currently being 8, the sum operator should not result in a system that can not keep up with the FPGA's frequency of 50MHz. The assumption is therefore made that the complete implementation ALU structure will is able to run at this maximum frequency. The ALU as a combinational path has been synthesized and it shows that the amount of DSPs is equal to the width of the vector which in this case is eight. The amount of parallelism in the non-sparse vectors can be increased to any power of two and the number of DSPs will be the same as that number.

The amount of clock cycles that is needed to perform one iteration of conjugate gradient is 1556. Most of these clock cycles are used to perform the sparse matrix non-sparse vector multiplications. The loop closing error showed in the above example is corrected in 10000 cycles. The total amount of clock cycles needed to converge the error becomes $16 * 10^6$ cycles. Convergence with the assumed maximum frequency of the FPGA results in a convergence that takes 320ms per dimension. The total time to complete the algorithm and converge three dimensions with large error will add up to 960ms.

12 — ICP

Section 5.1 describes the introduction and background of the ICP algorithm. Section 10 explains the choices made for algorithms and architectures. Section 10.3.1 explains the choice and shortcomings regarding the current used error minimization methods in ICP. The choice was made to use a linear least square problem with QR decomposition as solving method, the mathematical derivation and details are described in Section 12.1. In the DSE several choices are made and the realisation of chosen architectures can be found in Section 12.2. Simulations, synthesis, and timing results can be found in Section 12.3.

12.1 QR decomposition

Low [37] explain an approach for solving the minimization using a least square problem. They use a linear approximation that the rotation angle between matching point sets $\theta \approx 0$. Using idea's from [38],[16],Horn et al. [31] and the least squares approach found for graph-based SLAM a method for finding the best rotation and translation to minimize the error is given below. The rotation (θ) and translation (t_x, t_y) can be found by minimizing the following equation, using homogeneous coordinates:

$$\underset{\mathbf{H}}{\operatorname{argmin}} \sum_i ((\mathbf{H}p_i - m_i)n_i) \tag{12.1}$$

with

$$\mathbf{H} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}, p_i = \begin{bmatrix} p_{ix} \\ p_{iy} \\ 1 \end{bmatrix}, m_i = \begin{bmatrix} m_{ix} \\ m_{iy} \\ 1 \end{bmatrix}, n_i = \begin{bmatrix} n_{ix} \\ n_{iy} \\ 1 \end{bmatrix} \tag{12.2}$$

The least squares in the form of $\mathbf{A}\vec{x} = \vec{b}$ is derived from (12.1) as follows:

$$\mathbf{H}p_i = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{ix} \\ p_{iy} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta p_{ix} - \sin\theta p_{iy} + t_x \\ \sin\theta p_{ix} + \cos\theta p_{iy} + t_y \\ 1 \end{bmatrix} \tag{12.3}$$

$$\mathbf{H}p_i - m_i = \begin{bmatrix} \cos\theta p_{ix} - \sin\theta p_{iy} + t_x \\ \sin\theta p_{ix} + \cos\theta p_{iy} + t_y \\ 1 \end{bmatrix} - \begin{bmatrix} m_{ix} \\ m_{iy} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta p_{ix} - \sin\theta p_{iy} + t_x - m_{ix} \\ \sin\theta p_{ix} + \cos\theta p_{iy} + t_y - m_{iy} \\ 0 \end{bmatrix} \tag{12.4}$$

$$
\begin{aligned}
(\mathbf{H}p_i - mi)n_i &= \begin{bmatrix} \cos\theta p_{ix} - \sin\theta p_{iy} + t_x - m_{ix} \\ \sin\theta p_{ix} + \cos\theta p_{iy} + t_y - m_{iy} \\ 0 \end{bmatrix} \begin{bmatrix} n_{ix} \\ n_{iy} \\ 1 \end{bmatrix} = 0 \\
&= \cos\theta p_{ix} n_{ix} - \sin\theta p_{iy} n_{ix} + t_x n_{ix} - m_{ix} n_{ix} + \sin\theta p_{ix} n_{iy} + \cos\theta p_{iy} n_{iy} + t_y n_{iy} - m_{iy} n_{iy} = 0 \\
&= n_{ix} t_x + n_{iy} t_y + (p_{ix} n_{ix} + p_{iy} n_{iy})\cos\theta + (p_{ix} n_{iy} - p_{iy} n_{ix})\sin\theta = (m_{ix} n_{ix} + m_{iy} n_{iy}) \\
&= \underbrace{\begin{bmatrix} n_{ix} & n_{iy} & (p_{ix} n_{ix} + p_{iy} n_{iy}) & (p_{ix} n_{iy} - p_{iy} n_{ix}) \end{bmatrix}}_{\mathbf{A}_i} \underbrace{\begin{bmatrix} t_x \\ t_y \\ \cos\theta \\ \sin\theta \end{bmatrix}}_{\vec{x}} = \underbrace{\begin{bmatrix} m_{ix} n_{ix} + m_{iy} n_{iy} \end{bmatrix}}_{\mathbf{b_i}}
\end{aligned}
\tag{12.5}
$$

For N correspondences the following system can be constructed, where $K = N - 1$:

$$
\underbrace{\begin{bmatrix} n_{0x} & n_{0y} & (p_{0x} n_{0x} + p_{0y} n_{0y}) & (p_{0x} n_{0y} - p_{0y} n_{0x}) \\ n_{2x} & n_{1y} & (p_{1x} n_{1x} + p_{1y} n_{1y}) & (p_{1x} n_{1y} - p_{1y} n_{1x}) \\ \vdots & \vdots & \vdots & \vdots \\ n_{Kx} & n_{Ky} & (p_{Kx} n_{Kx} + p_{Ky} n_{Ky}) & (p_{Kx} n_{Ky} - p_{Ky} n_{Kx}) \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} t_x \\ t_y \\ \cos\theta \\ \sin\theta \end{bmatrix}}_{\vec{x}} = \underbrace{\begin{bmatrix} m_{0x} n_{0x} + m_{0y} n_{0y} \\ m_{1x} n_{1x} + m_{1y} n_{1y} \\ \vdots \\ m_{Kx} n_{Kx} + m_{Ky} n_{Ky} \end{bmatrix}}_{\vec{b}}
\tag{12.6}
$$

The system $\mathbf{A}\vec{x} = \vec{b}$ has the size of $\mathbf{A} = N \times 4$ and the length of $\vec{b} = N$. In case of a laser range scanner N is most of the time $180, 270$ or 360, so $\mathbf{A}$ is a rectangular matrix. This system can be solved using QR decomposition where $\mathbf{A} = \mathbf{QR}$. If $\mathbf{Q}$ is an orthonormal matrix it has the following property:

$$
\mathbf{Q}^T\mathbf{Q} = \mathbf{I} \tag{12.7}
$$

Using the property from (12.7) and QR decomposition the following can be derived

$$
\begin{aligned}
\mathbf{QR} &= \mathbf{A} \\
\mathbf{Q}^T\mathbf{QR} &= \mathbf{Q}^T\mathbf{A} \\
\mathbf{IR} &= \mathbf{Q}^T\mathbf{A} \\
\mathbf{R} &= \mathbf{Q}^T\mathbf{A}
\end{aligned}
\tag{12.8}
$$

Using the property from (12.7) and QR decomposition the following can be derived for the system $\mathbf{A}\vec{x} = \vec{b}$:

$$
\begin{aligned}
\mathbf{A}\vec{x} &= \vec{b} \\
\mathbf{QR}\vec{x} &= \vec{b} \\
\mathbf{Q}^T\mathbf{QR}\vec{x} &= \mathbf{Q}^T\vec{b} \\
\mathbf{IR}\vec{x} &= \mathbf{Q}^T\vec{b} \\
\mathbf{R}\vec{x} &= \underbrace{\mathbf{Q}^T\vec{b}}_{\vec{t}}
\end{aligned}
\tag{12.9}
$$

$\mathbf{R}$ is a rectangular matrix containing an upper triangular matrix and zeros beneath it (see Equation 12.11). In normal QR decomposition the $\mathbf{Q}$ matrix is produced in a unitary form, so it would be a $N \times N$ matrix in case of this $\mathbf{A}$ matrix. But to solve the system $\mathbf{A}\vec{x} = \vec{b}$ the unitary $\mathbf{Q}$ matrix is not needed, this is due to the fact that the $\vec{x}$ vector only has a length of 4. Let the $\mathbf{Q}$ matrix be denoted as $[\mathbf{Q}_0, \mathbf{Q}_1]$, with the size of $\mathbf{Q}_0 = N \times 4$, and the size of $\mathbf{Q}_1 = N \times (N - 4)$. Then, given that the $\mathbf{R}$ matrix also has a 4×4 upper triangular matrix,

only $\mathbf{Q}_0$ is needed to solve the linear least square problem. From now on $\mathbf{Q}_0$ is referred to as $\mathbf{Q}$ To create the orthonormal space $\mathbf{Q}$ the Gram-Schmidt method is used.

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} \vec{v_0} & \vec{v_1} & \vec{v_2} & \vec{v_3} \end{bmatrix} \\ \mathbf{Q} &= \begin{bmatrix} \vec{u_0} & \vec{u_1} & \vec{u_2} & \vec{u_3} \end{bmatrix} \\ \mathbf{R} &= \mathbf{Q}^T\mathbf{A} \\ \vec{t} &= \mathbf{Q}^T\vec{b} \end{aligned} \tag{12.10}$$

$$\begin{aligned} \vec{t} &= [t_0, t_1, t_2, t_3] \\ \mathbf{R} &= \begin{bmatrix} r_{00} & r_{01} & r_{02} & r_{03} \\ 0 & r_{11} & r_{12} & r_{13} \\ 0 & 0 & r_{22} & r_{23} \\ 0 & 0 & 0 & r_{33} \end{bmatrix} \end{aligned} \tag{12.11}$$

$$\begin{aligned} \vec{u_0} &= \frac{\vec{v_0}}{||\vec{v_0}||} \\ \vec{u_1} &= \frac{\vec{v_1} - (\vec{v_1} \bullet \vec{u_0})\vec{u_0}}{||\vec{v_1} - (\vec{v_1} \bullet \vec{u_0})\vec{u_0}||} \\ \vec{u_2} &= \frac{\vec{v_2} - (\vec{v_2} \bullet \vec{u_0})\vec{u_0} - (\vec{v_2} \bullet \vec{u_1})\vec{u_1}}{||\vec{v_2} - (\vec{v_2} \bullet \vec{u_0})\vec{u_0} - (\vec{v_2} \bullet \vec{u_1})\vec{u_1}||} \\ \vec{u_3} &= \frac{\vec{v_3} - (\vec{v_3} \bullet \vec{u_0})\vec{u_0} - (\vec{v_3} \bullet \vec{u_1})\vec{u_1} - (\vec{v_3} \bullet \vec{u_2})\vec{u_2}}{||\vec{v_3} - (\vec{v_3} \bullet \vec{u_0})\vec{u_0} - (\vec{v_3} \bullet \vec{u_1})\vec{u_1} - (\vec{v_3} \bullet \vec{u_2})\vec{u_2}||} \end{aligned} \tag{12.12}$$

After the computation of $\mathbf{R}$ and $\mathbf{Q}$ the system $\mathbf{R}\vec{x} = \vec{t}$ can be calculated, where $\mathbf{R}$ is a 4×4 matrix and $\vec{t}$ is a vector with length 4. Extracting x_0, x_1, x_2, and x_3 from the system $\mathbf{R}\vec{x} = \vec{t}$ is a matter of row reduction.

12.2 Realisation

The Chapter 10 explains the different choices made for both the algorithm and the structure of the hardware. This chapter will explain the realised hardware structure for solving the ICP problem. Figure 12.1 shows the flowchart of the ICP algorithm, given there are two observations $\vec{p}$ and $\vec{m}$ (stored in blockram), the algorithm, as explained in Section 12.1, has the following segments:

1. Construction of correspondences between $\vec{p}$ and $\vec{m}$
2. QR decomposition
   - Construction of the system $\mathbf{A}\vec{x} = \vec{b}$
   - Gram-Schmidt method for creating an orthonormal space
   - Construction of the system $\mathbf{R}\vec{x} = \vec{t}$
3. Linearly solving $\mathbf{R}\vec{x} = \vec{t}$ using backwards substitution
4. Applying transformation on current observation and previous transformation
5. Go to step 1

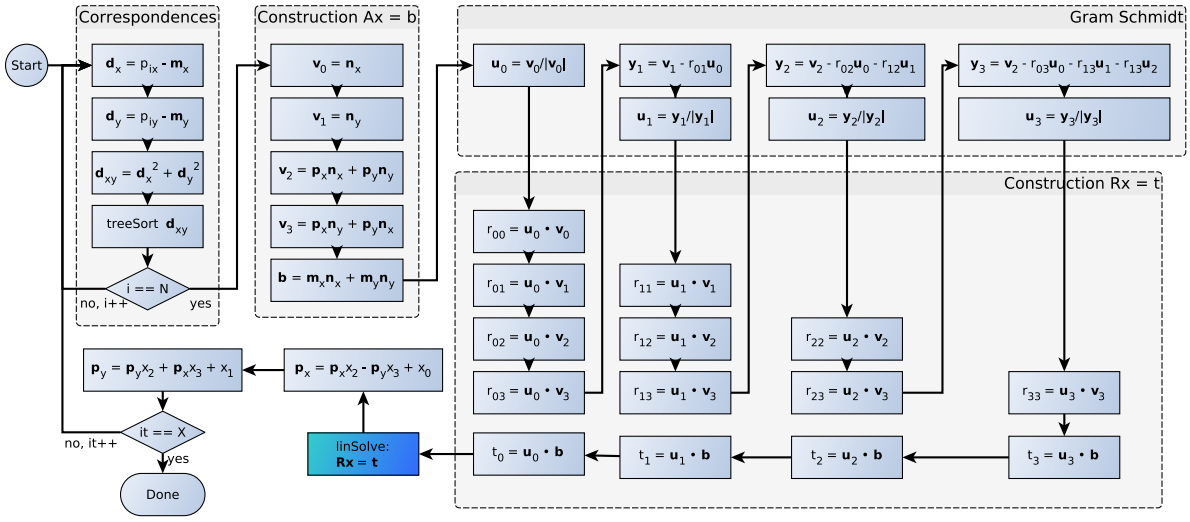


Figure 12.1: Flowchart of the ICP algorithm where the darker blue node represents the time when the main controller leaves control of the number blockRAM to the number controller

This entire process repeats until X iterations are done. This section explains the realized architecture into more detail, the synthesis, timing and simulation results can be found in Section 12.3.

An abstract view of the architecture is shown in Figure 12.2, the main architecture is shown in Figure 12.3, this architecture is shown without any control signals in order to keep it clean. The full architecture with all control signals is shown in Figure 12.4 and the names used in this figure correspond to the names used in the CλaSH code. The yellow blocks in all figures represent a type of memory, the bigger blocks are blockRAMs, while the smaller ones are registers. The red lines represent control signals and one can see that every red line ends up in a memory component, this is due to the fact that blockRAM has 1 clock cycle delay before it delivers the requested data, by delaying all control signals the control becomes easier because one can specify the wanted data (blockRAM read address) and the wanted operations (multiplexers and *vecAlu* opcode) in the same clock period. The blue lines represent the delayed control signals which go to every component except the memory components. The thick black lines are data lines that are used for vector transport with a vector length M, which also is the width of all the vector components like the vector blockRAMs, *vecAlu*, *snatcher*, *singleValueSelector*, and the *treeSorter*. The thin black lines represent data lines for transporting single number data. the green blocks are the mathematical components doing operations on the given data. Each part of the hardware architecture will be briefly explained below.

12.2.0.1 blockRAM

There are two types of blockRAM in the system, Vector blockRAM, containing vectors and matrices, and number blockRAM, containing single values like the cells of 4×1 vectors and the 4×4 matrix that are used in single value computations.

Vector blockRAMs must be able to provide two different vectors with length M. M is the length which the vector ALU can handle in one clock cycle. Every vector in the QR decomposition has length N which means that M must be chosen in such a way that N is a multiple of M. blockRAMs addresses are created in such a way that one can easily select a part

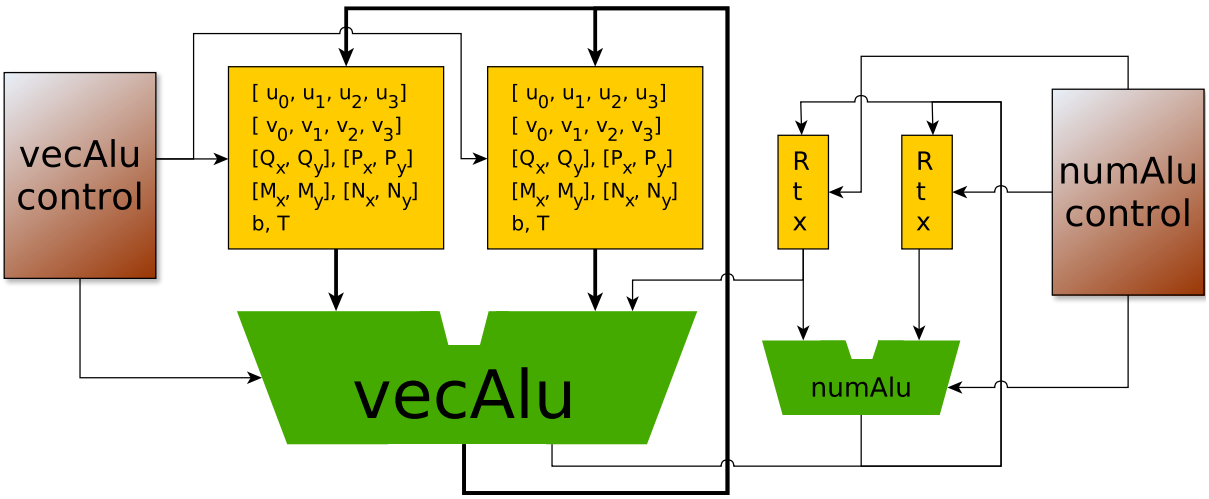


Figure 12.2: General overview of structure with vector ALU and number ALU with dual block-RAMs

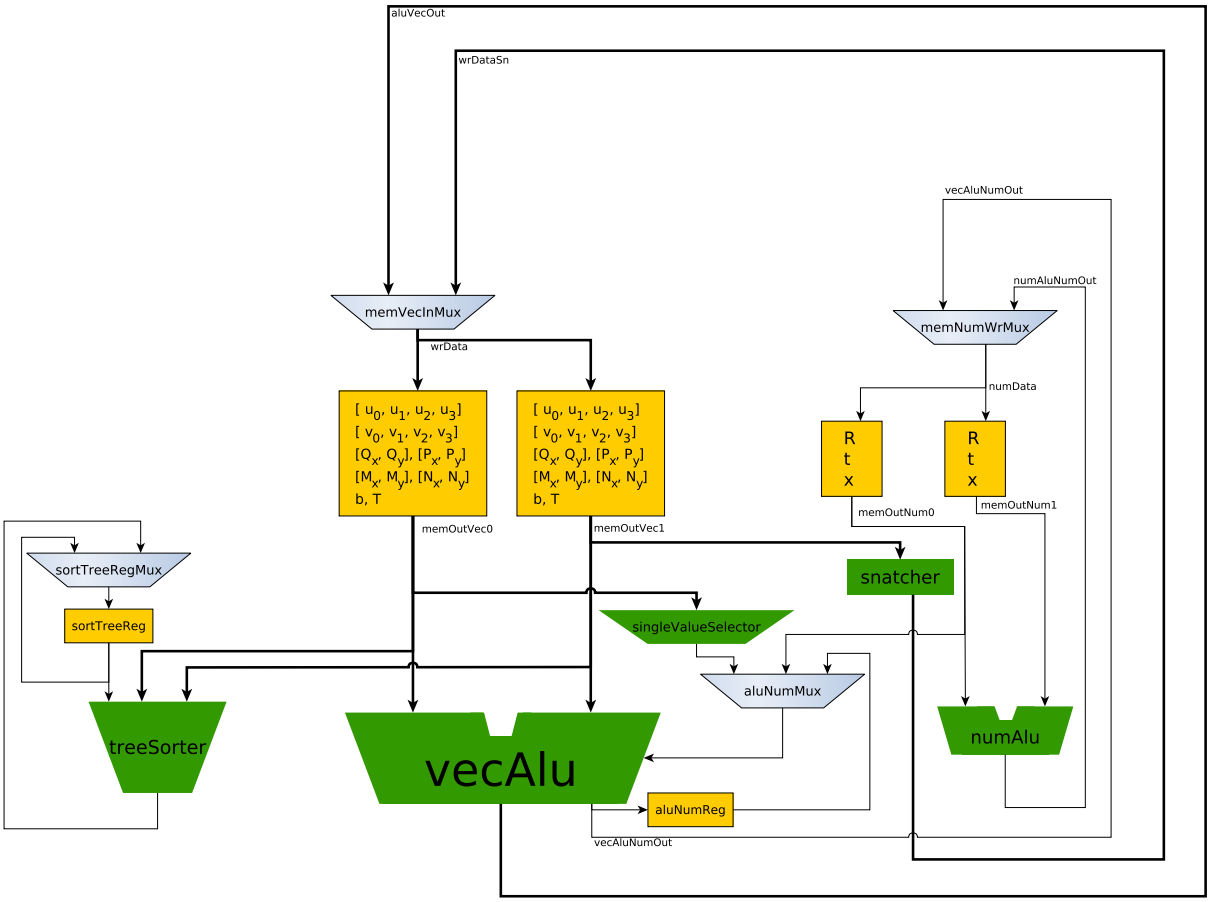


Figure 12.3: Hardware architecture without control

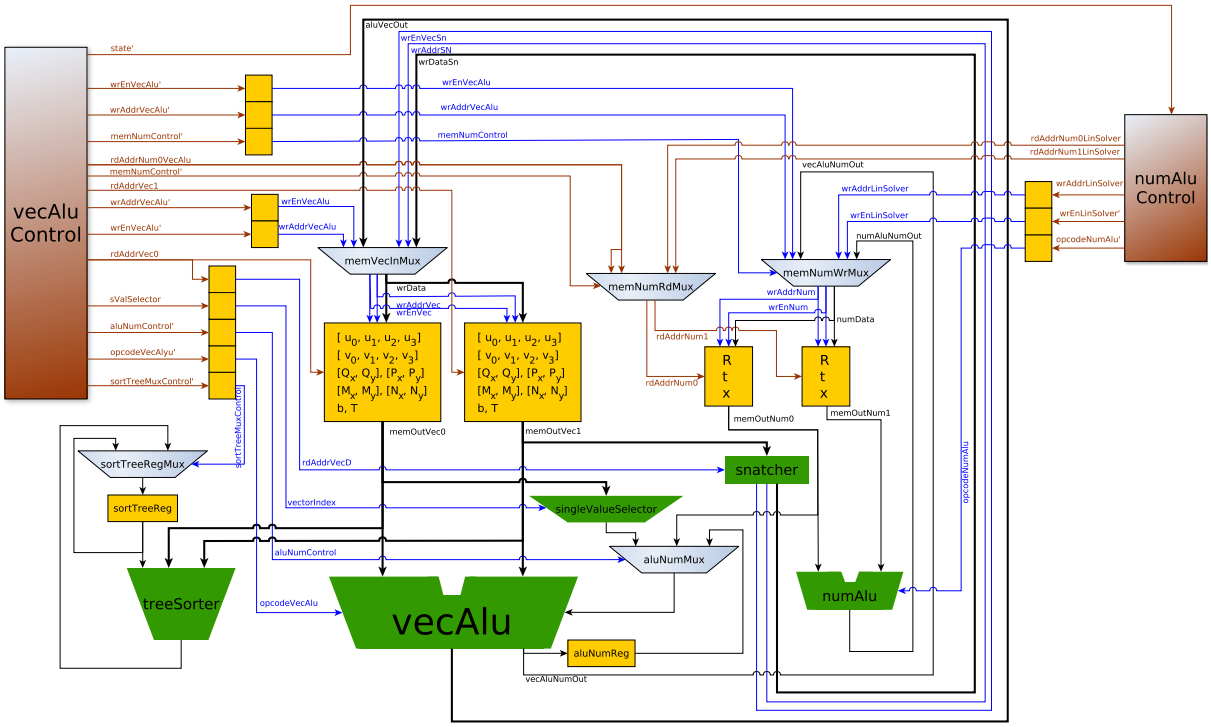


Figure 12.4: Hardware architecture

of a vector/column of a matrix. A blockRAM address looks as follows:

$$(a_0, a_1, a_2) \tag{12.13}$$

where:

a_0 is the matrix specifier, for example Q or A

a_1 is the matrix column/vector selector

a_2 is the column/vector part selector

Write addresses have the same format as the read addresses. To write a single value the *writeMask* can be used. Each element of the vector blockRAM has a *writeEnable* which enables writing a new value. Therefore, if a single value of a vector must be replaced, create a corresponding write mask and create a write vector containing the new value in every place.

Number blockRAMs are smaller blockRAMs that contain single numbers like the content of the upper triangular of the $\mathbf{R}$ matrix, values of the $\vec{x}$ vector, and values of the $\vec{t}$ vector, and these numbers are used in single value calculations like the backwards substitution. Outgoing and incoming connections to the number blockRAMs are from two different components, the *vecAlu* as a result from a dot product and used in vector scaling, and the *numberAlu* for single number calculations.

12.2.0.2 vecAlu

The vector ALU does all the vector operations and has the following four inputs:

- vector with length M as operand

- vector with length M as operand
- single input value
- opcode for control

For the vector addition, subtraction, multiplication, and scaling it uses M parallel DSPs. For the summation the ALU uses a tree structure but it does not use the onboard DSPs, instead it uses synthesised adders. The single input is used for vector scaling and to carry over the partial sum to the next clock cycle. It is also possible to do an inverse square root on the single value input. The ALU has the two following outputs:

- vector with length M, which is the result of all vector operations which produce a single vector
- single value, which is the result of for example dot products or (partial) results of vector summations.

The single value can be stored into a register (partial result summation) or written to a number blockRAM (result of final dot product).

12.2.0.3 vecAluControl

The *vecAluControl*, visible in Figure 12.2 and Figure 12.4, is the main controller of the system and is implemented as a mealy machine, this principle is explained in Chapter 6. The main controller is responsible for setting the right datapath using multiplexers and controlling the blockRAMs in order to compute the entire algorithm. Most of the time the *vecAluControl* is in control of the system, it is only during the linear solving of $\mathbf{R}\vec{x} = \vec{t}$ that the *numAluControl* controls the number blockRAMs, see the darker blue colored node in the flow chart in Figure 12.1. During this linear solving most of the system is idle and no parallel computations are done and only the number ALU is busy doing backwards substitution for which the *numberAluControl* uses the number blockRAMs and *numberAlu*.

12.2.0.4 treeSorter

The tree sorter has the same input as the vector ALU, which are two vectors with length M from blockRAM. As the name implies, the tree sorter has a tree structure (described in Section 10.1) that selects the closest and second closest point from a list with distances. The two input vectors must be two vector parts containing distances from p_i to every point m_i from $\vec{m}$. The tree sorter uses also a register to carry over partial results to the next clock cycle. The outputs are two indices which can be used to pull the closest and second closest point from blockRAM.

12.2.0.5 singleValueSelector

The singleValueSelector is a large multiplexer that can pull a single value from the vector with length M from the blockRAM. This single value can be used in vector operations like scaling or subtraction. Subtraction is for example used in the calculation of all distances to a point p_i. So it takes a single value p_i from $\vec{p}$ and it does the operations $p_i - \vec{m}$.

12.2.0.6 snatcher

The snatcher is responsible for creating the correspondences. The tree sorter provides two indices which can be used to pull the closest and second closest point from blockRAM. The snatcher is build to save clock cycles and does that because no extra cycles are needed to calculate the normal vector. To create the normal vector three points are needed: the point (p_i) from the new observation $(\vec{p})$, the closest (m_i) and second closet (m_j) from the previous observation $\vec{m}$. Equation (12.14) states the logic of creating the normal vector n.

$$\begin{aligned} l &= \begin{cases} m_i, & \text{if } m_{ix} > m_{jx} \\ m_j, & \text{otherwise} \end{cases} \\ r &= \begin{cases} m_j, & \text{if } m_{ix} > m_{jx} \\ m_i, & \text{otherwise} \end{cases} \\ c &= (r_x - l_x) \cdot (p_{iy} - l_y) - (r_y - l_y) \cdot (p_{ix} - l_x) \\ \Delta x &= l_x - r_x \\ \Delta y &= l_y - r_y \\ [\hat{n}_x, \hat{n}_y] &= \begin{cases} [-\Delta y, \Delta x], & \text{where } c > 0 \\ [\Delta x, -\Delta y], & \text{where } c < 0 \\ [0, 0], & \text{where } c == 0 \end{cases} \end{aligned} \tag{12.14}$$

Where r and l are the rightmost and leftmost points from m_i and m_j, and $\hat{n}$ is the non normalized normal vector. To understand the working of the snatcher one must understand the flow of the correspondence creation part. The correspondences creation part in the flowchart of Figure 12.1 shows the calculation of all the distances squared d_{xy} between the point p_i and the vector $\vec{m}$. Knowing the structure of the blockRAM, it can be shown that in each clock cycle of this correspondence creation part the blockRAM provides data (2 vectors with length M). The tree sorter provides two indices were the closest and second closest point are located in blockRAM. One could introduce extra clock cylces to load these points and create the normal vector. However these points will also pass by the next correspondence search. Suppose the correspondence search for point p_0. First, all the squared distances from p_0 to $\vec{m}$ have to be calculated. Then these distances go through the tree sorter which provides two indices k and z for the closest and second closest point in $\vec{m}$. Starting the next iterations all the distances from p_1 to $\vec{m}$ have to be calculated. During this calculation the closest and second closest point for p_0 pass from blockRAM to the vecAlu for distance calculations. This is the time that the snatcher snatches these values and calculates the normal vector. This way no extra clock cycles need to be introduced, except in the end to calculate the normal vector for correspondence C_{N-1}. The number of clock cycles reduced by the snatcher is $4N$ (N is the number of laser scan beams in one laser scan). If $N = 180$ the snatcher saves 720 clock cycles, with a vector ALU width $M = 45$ the reduction is approximately 15%.

12.2.0.7 numAlu

The number ALU solves the system $\mathbf{R}\vec{x} = \vec{t}$ using backward substitution (row reduction from echelon to reduced echelon form). It has non restoring divider hardware which can divide a fixed-point number with an i-bit integer part and f-bit fractional part in $i + 2f$ clock cycles.

The goal of the ICP algorithm is to determine the transformation between two observations,

after 1 iteration the transformation H_{01} is known where H has the following format:

$$\mathbf{H} = \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{12.15}$$

The next iterations the algorithm computes H_{12}, and H_{23} after that. In order to determine the total transformation, from 0 to 3, one can apply the transformation on the previous transformation, for example $H_{03} = H_{23}H_{12}H_{01}$. These transformation on transformation matrices are also done in the *numAlu*.

12.2.0.8 numAluControl

The *numAluControl* controls the *numberAlu* to solve the system $\mathbf{R}\vec{x} = \vec{t}$ and to transform the transformation matrices over iterations. The controller is only active during the linSolve state (See Figure 12.1).

12.3 Simulation, synthesis, and timing results

12.3.1 Numerical precision

The FPGA, Appendix A, contains DSPs which can be set in different modus; 9-,18-, and 27-bit. For the ICP algorithm both small and large values can occur. Large values can occur because the result of dot products of vectors with length 180 can lead to large numbers. A laser range scanner, considered for the Intel dataset, can measure up distances up to 50 meters, a dot product of a vector could lead to a value of $180 \cdot 50 = 9000$. In the correspondence creation very small vectors can occur (it is the point of the whole ICP algorithm to make minimize distance between correspondences). Normalizing small vectors with the quadratic formula means that small values are squared which makes them smaller. If a distance between two correspondences is 1mm = 0.001m then $0.001^2 = 0.000001$. The values with a the range between 9000 and 0.000001 do not fit in a fixed-point variable of 27-bit. The problem of numerical imprecision could be solved by locally, when needed, use shift operations and guard digits to up- or downscale variables on order to make a more exact computation. For example when normalizing vectors one could shift the value in such a way that the fractional part becomes larger, and when calculating a dot product one could shift the value to increase the integer part. But there is no time in this project left to implement this and the choice was made to use a 27- bit signed fixed-point variable an integer part of 12 and a fractional part of 15. With an integer part of 12 it is not possible to store values larger than 2048 (two's complement signed), but looking at the structure of the laser scans in the Intel data set it is highly unlikely that all the distances will be 50 meters, which means that the maximum value in the system is much lower than 9000. The choice was made to use an integer part of 12 bits, including sign bit, which leaves 15 bits for the fractional part. The smallest value that can be expressed in 15-bit fractional part is $1/2^{15}) = 0.00003051757$ which means that only very small squared values are neglected. The different algorithms described in Section 12.3.2 are all done in 27 bit and show no numerical instabilities due to the number of bits.

12.3.2 Simulations results using different square root algorithms

During the CλaSH simulation, fixed-point variables are used with an integer part size of 15 and a fractional part size of 12. As a reference simulation a MATLAB implementation is used and simulated for result comparison. Depending on the inverse square root, simulations in CλaSH

show promising results as can be seen in Figure 12.5, Figure 12.6, Figure 12.7, and Figure 12.8. The figures all show four different results after a number of iterations. The first (most left) plots the initial situation, these scans are given to the ICP algorithm. The second plot shows the result after one iteration. The third plot shows the result after 8 iterations, and the fourth (most right) plot shows the result after 15 iterations. All figures are discussed below. The four (inverse square root) algorithm types are as follows:

- MATLAB (a): The results of the MATLAB implementation of the ICP algorithm with double precision floating point numbers
- Non-restoring (b): The results of CλaSH implementation with a reciprocal and Non-restoring square root for an inverse square root algorithm
- Fast inverse square root with Newton-Raphson (c): The results of CλaSH implementation with a Fast inverse square root plus one iteration of Newton-Raphson
- Fast inverse square root without Newton-Raphson (d): The results of CλaSH implementation with a Fast inverse square root without Newton-Raphson

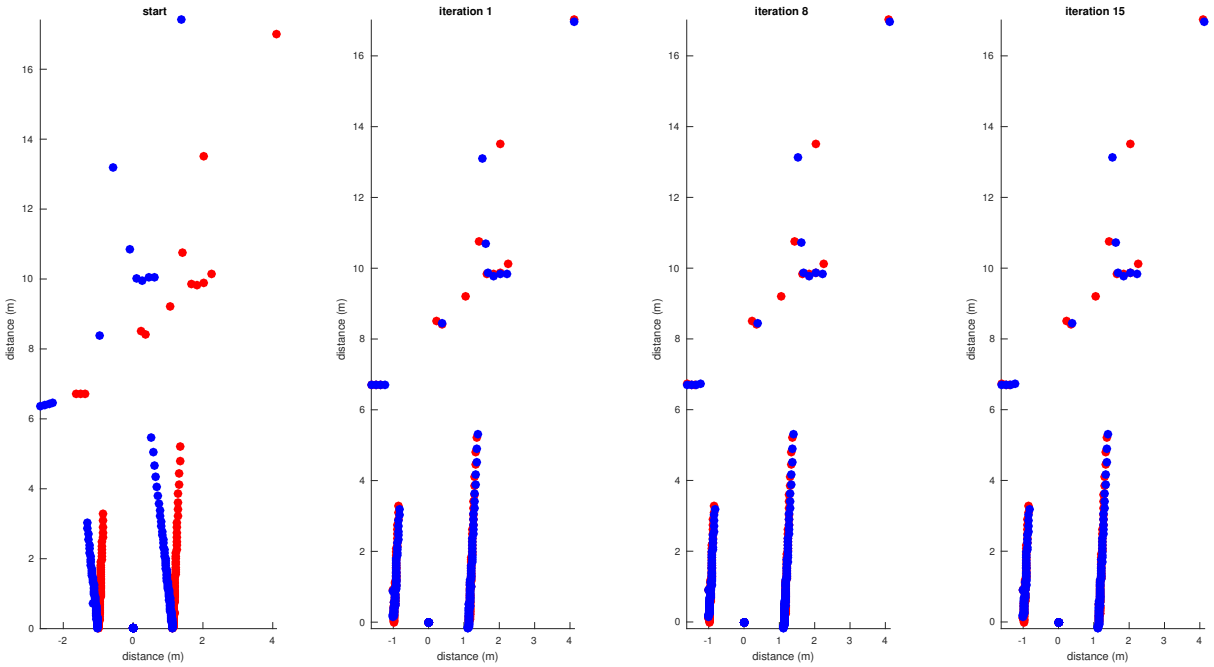
The scans considered in Figure 12.5 have only a rotation difference with no translation and one can see that all methods are able to determine the rotation. Al the solutions are almost converged after iteration 1. However, in translation there are some differences visible and since there is no stopping conditions all algorithms perform 15 iterations. The difference between MATLAB and the Non-restoring square root are small and the results from the Non-restoring method are still very accurate. Looking at the Fast inverse square root methods, translation differences are more visible. Where the Fast inverse method with Newton-Raphson show only a slight translation error, the method without Newton-Raphson show a significant translation error.

The scans shown in Figure 12.6 have a low rotation difference but a large translation difference. The implementation in MATLAB and the non-restoring method perform almost identical in terms of quality. Both methods using the Fast inverse square root lack in terms of finding the correct translation and like the case of Figure 12.5 the method without Newton-Raphson performs bad.

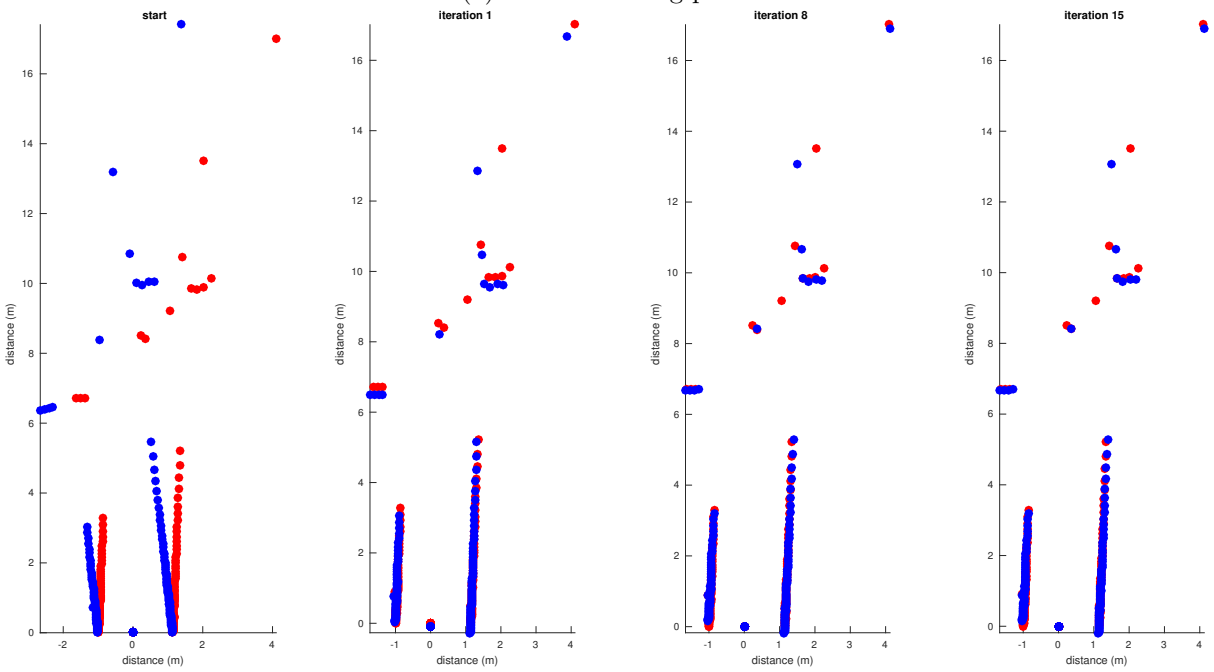
Figure 12.7 shows a case with laser scans taken in a wider area instead of a small corridor. The MATLAB results differ little compared to the Non-restoring method and the results from the Fast inverse square root with Newton-Raphson. Here it can be seen again that there is a small translation error where the method is used without Newton-Raphson.

Figure 12.8 is a case with two laser scans with a rotation difference but under a larger angle, the robot makes a 90 degree angle to go from one hallway to a orthogonal hallway and it does that in multiple steps. These two consecutive laser scans are taken from somewhere in the middle of that rotation. All methods perform accurate enough and only the Fast inverse square root without Newton-Raphson shows a slight error in translation.

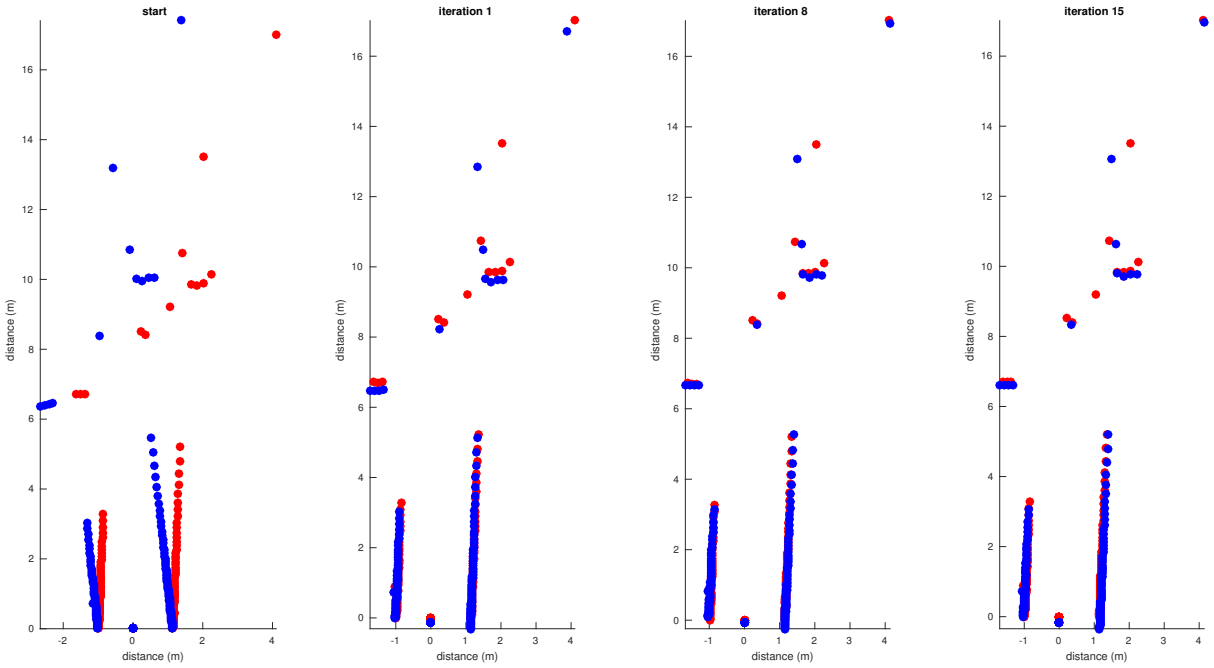
Looking at the overall performance of the algorithm one can conclude that the inverse square root has a huge impact because it is used in both correspondences creation and in error minimization.



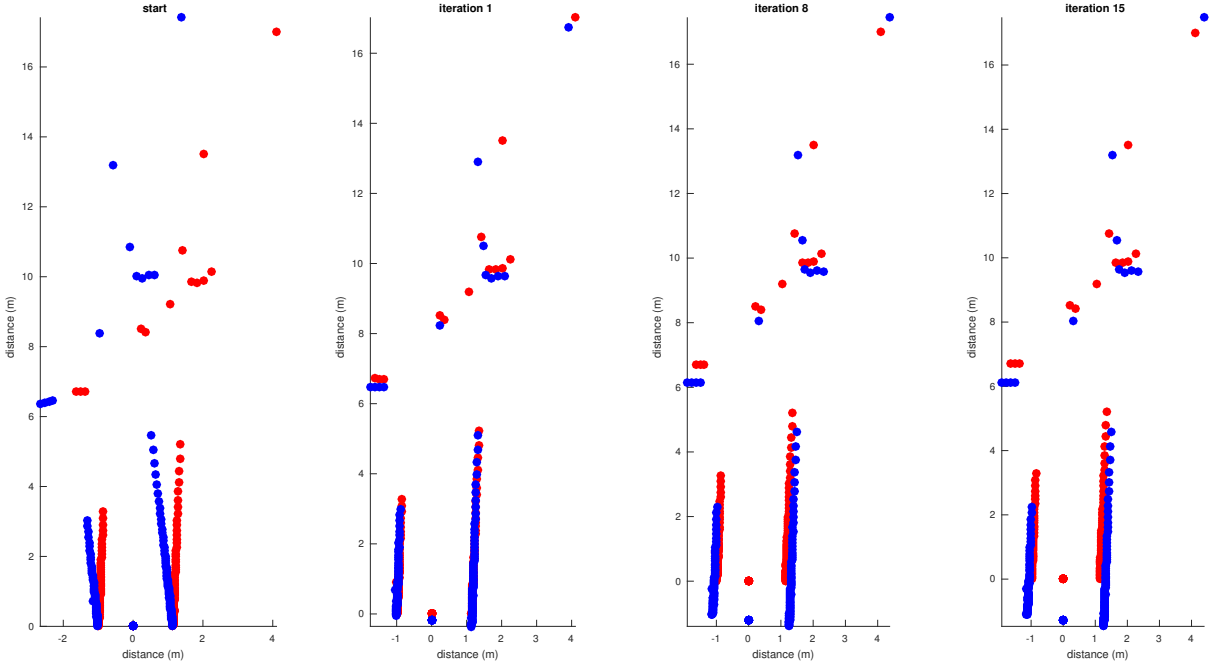
(a) Matlab floating point



(b) Non-restoring inverse square root

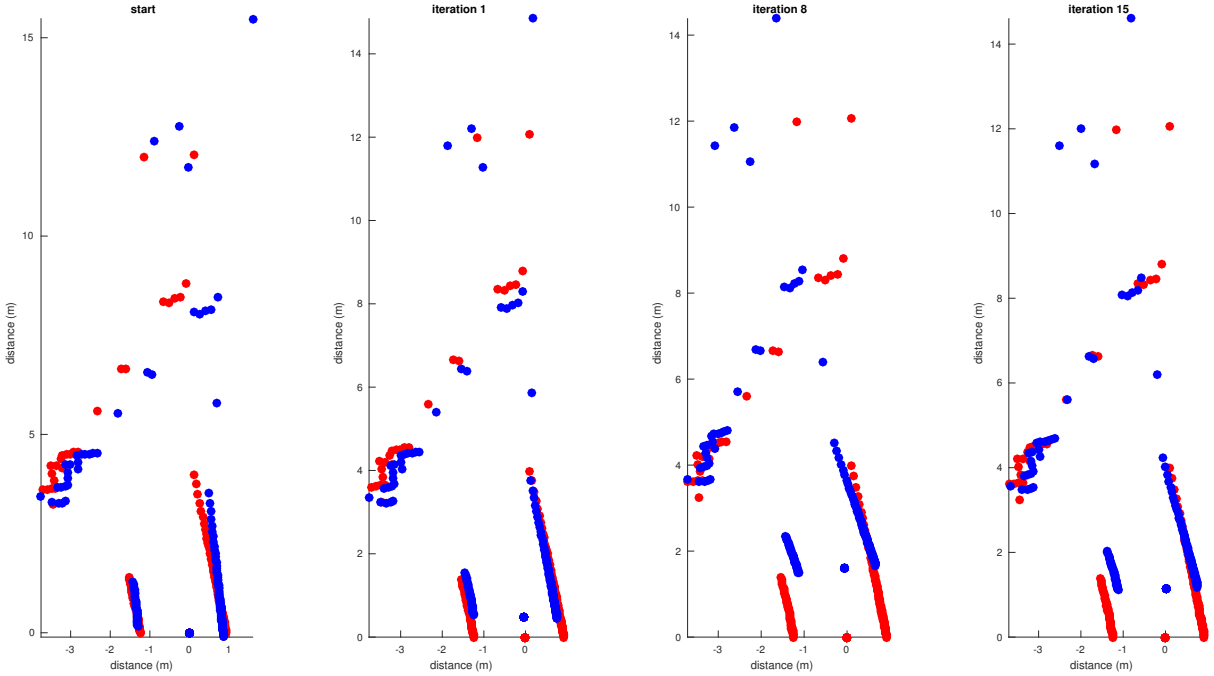


(c) Fast inverse square root with 1 iteration Newton-Raphson

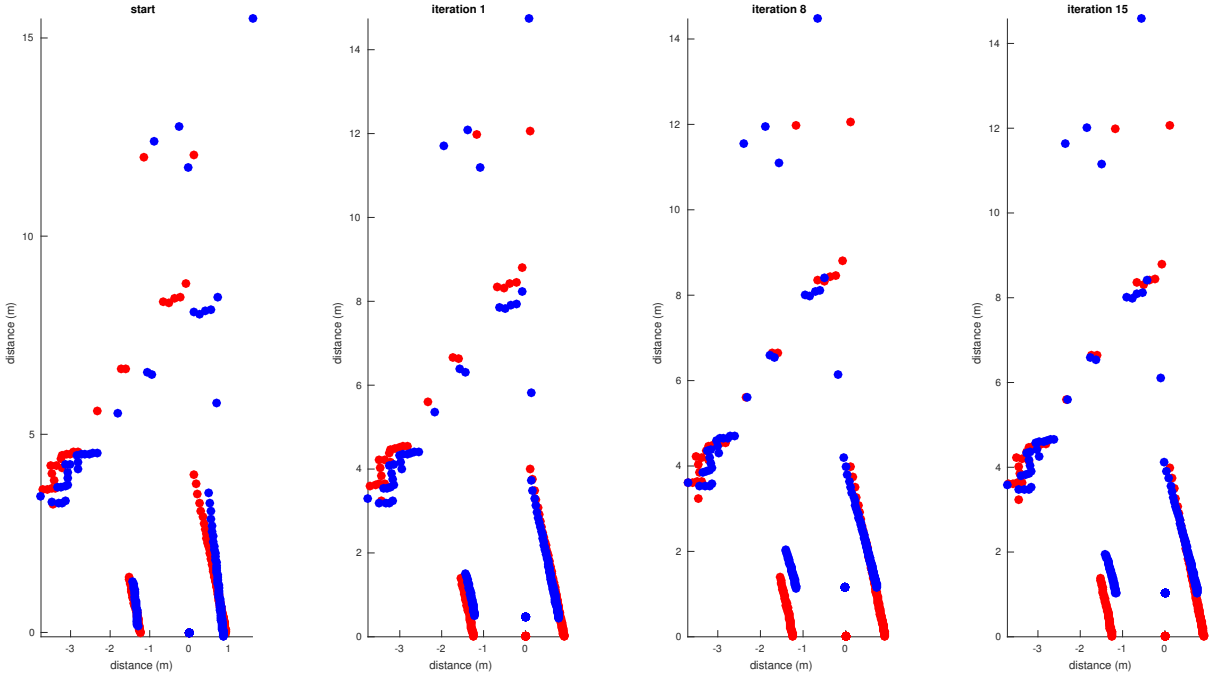


(d) Fast inverse square root with 0 iterations Newton-Raphson

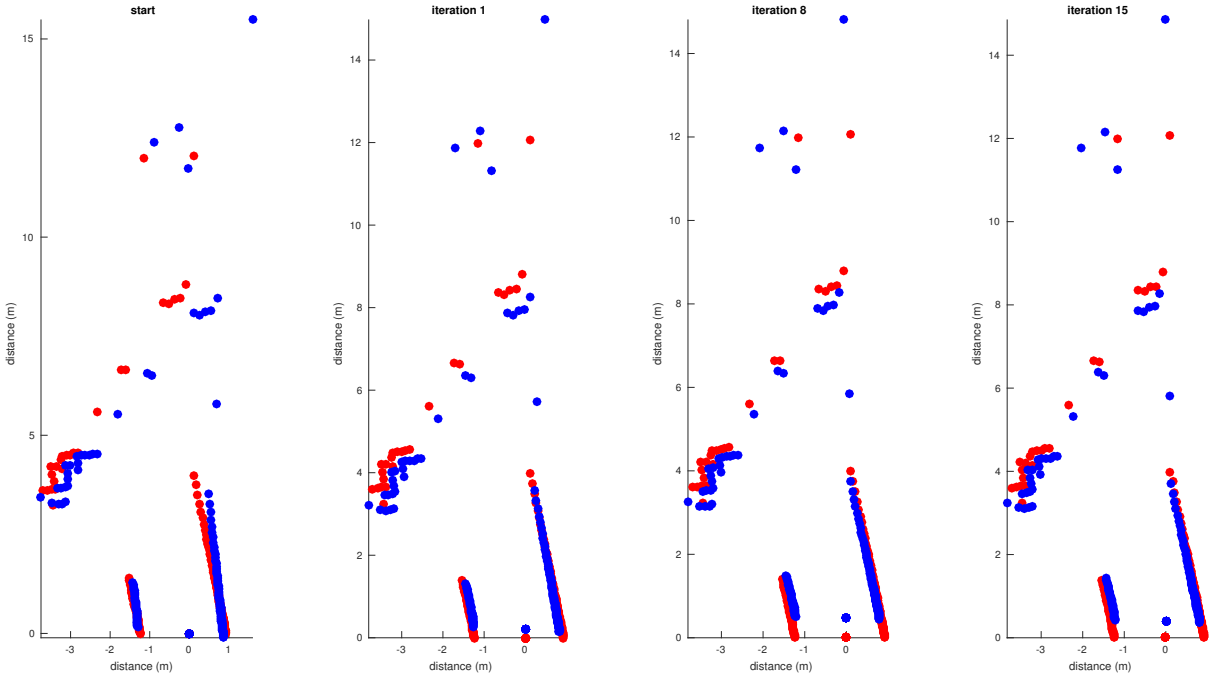
Figure 12.5: ICP algorithm performed on a laser scan with different algorithm implementations



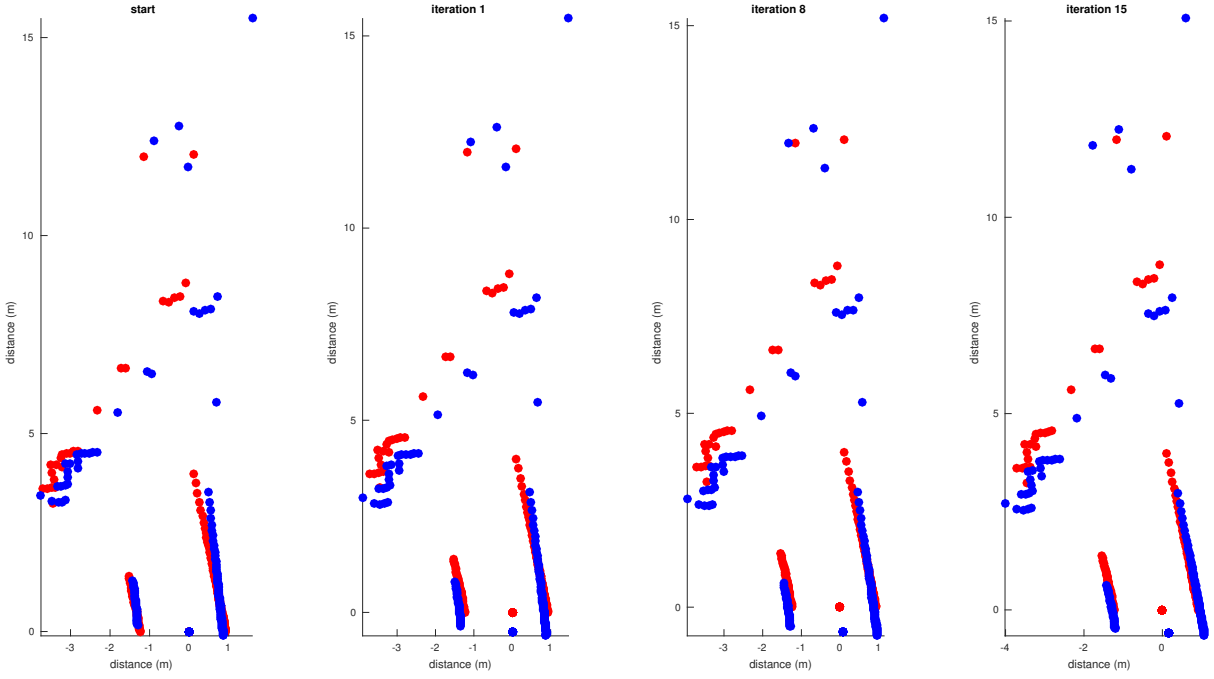
(a) Matlab floating point



(b) Non-restoring inverse square root

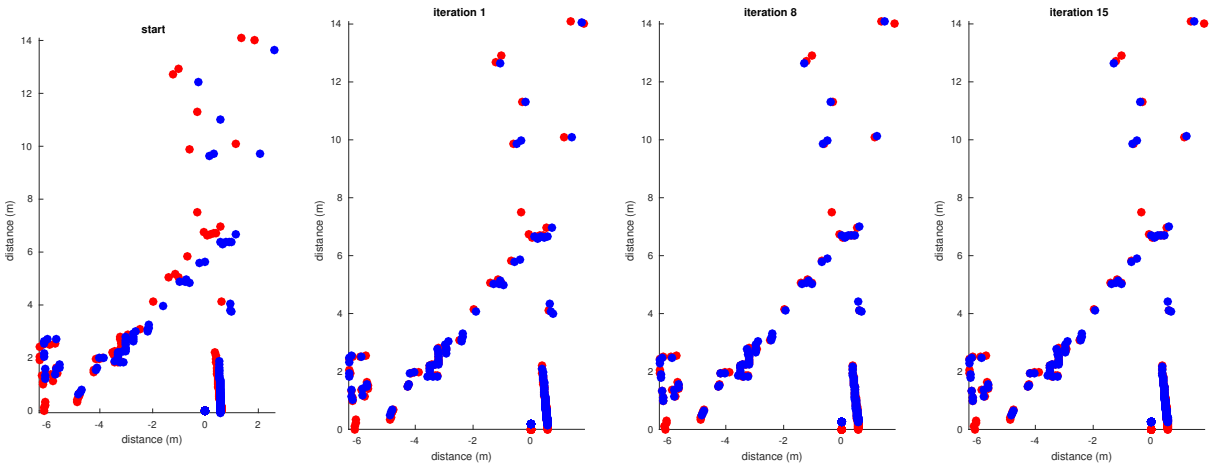


(c) Fast inverse square root with 1 iteration Newton-Raphson

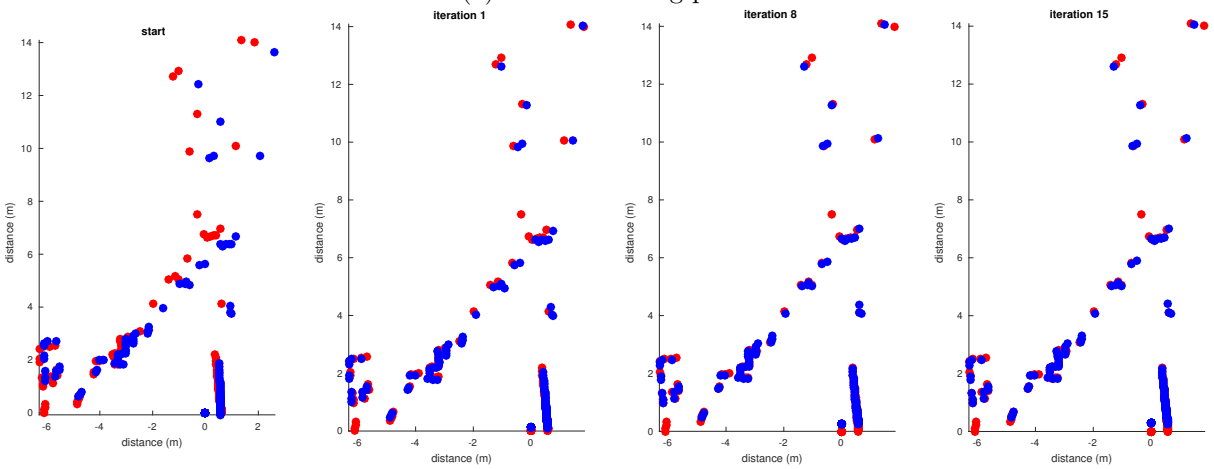


(d) Fast inverse square root with 0 iterations Newton-Raphson

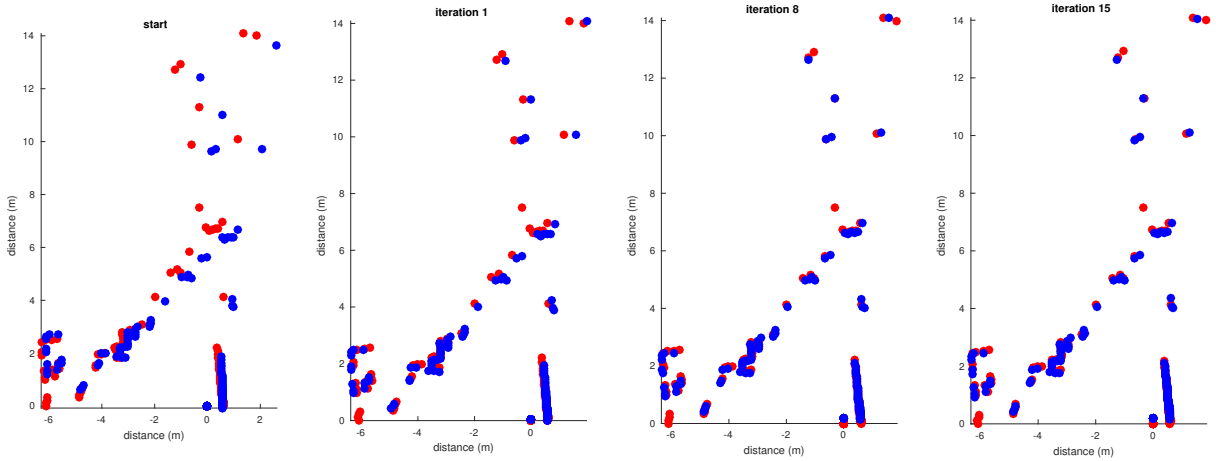
Figure 12.6: ICP algorithm performed on a laser scan with different algorithm implementations



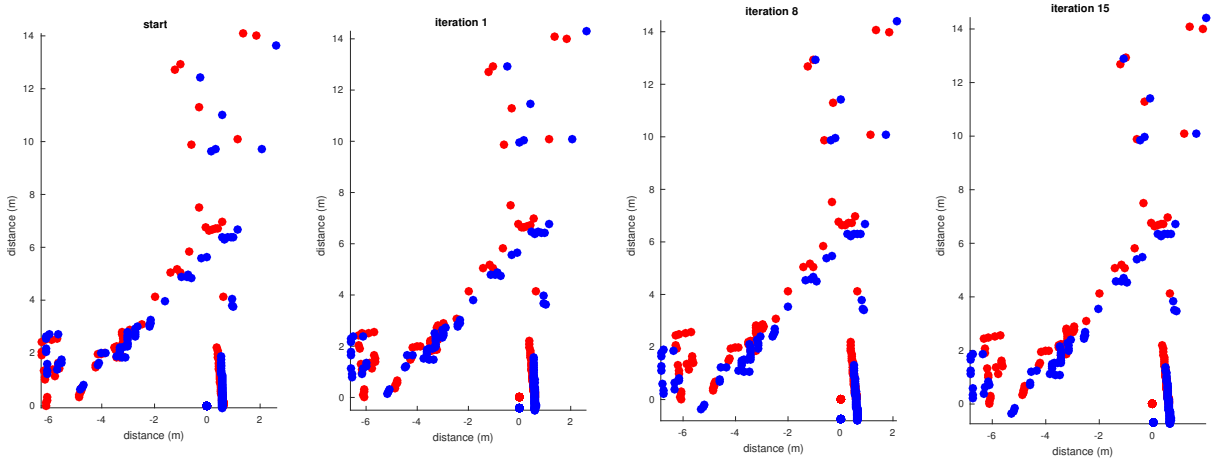
(a) Matlab floating point



(b) Non-restoring inverse square root

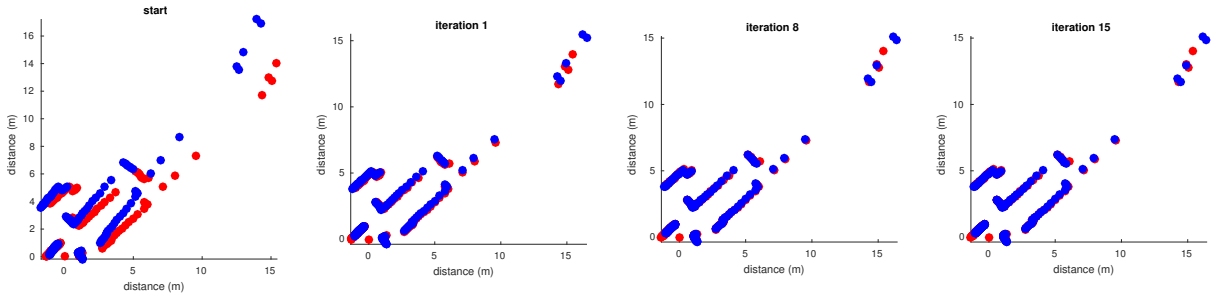


(c) Fast inverse square root with 1 iteration Newton-Raphson

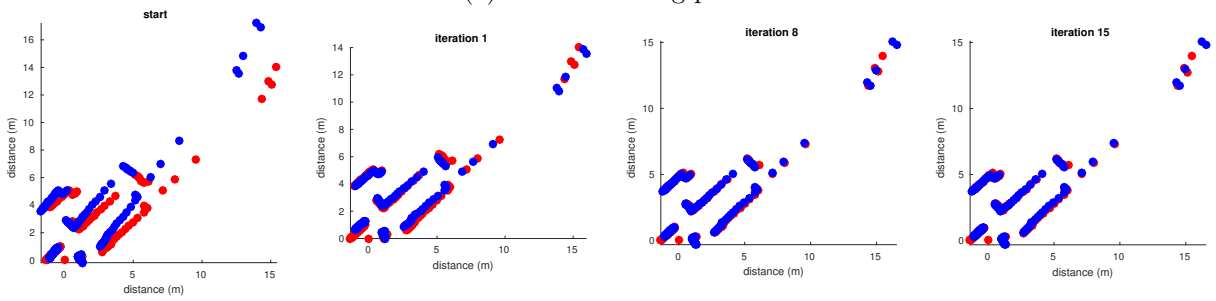


(d) Fast inverse square root with 0 iterations Newton-Raphson

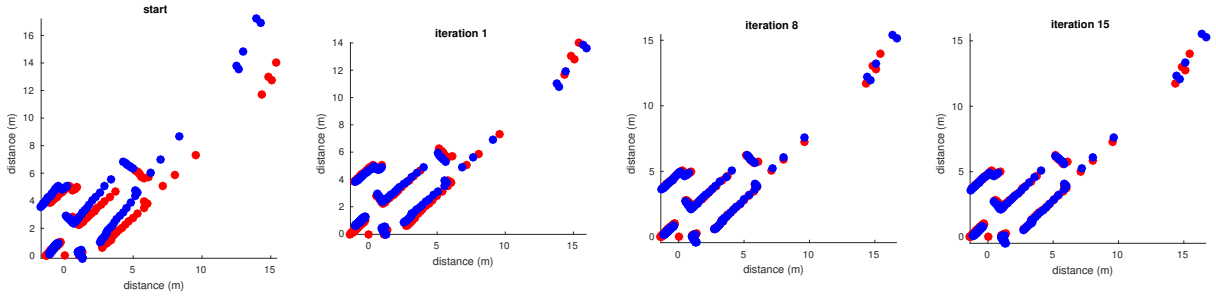
Figure 12.7: ICP algorithm performed on a laser scan with different algorithm implementations



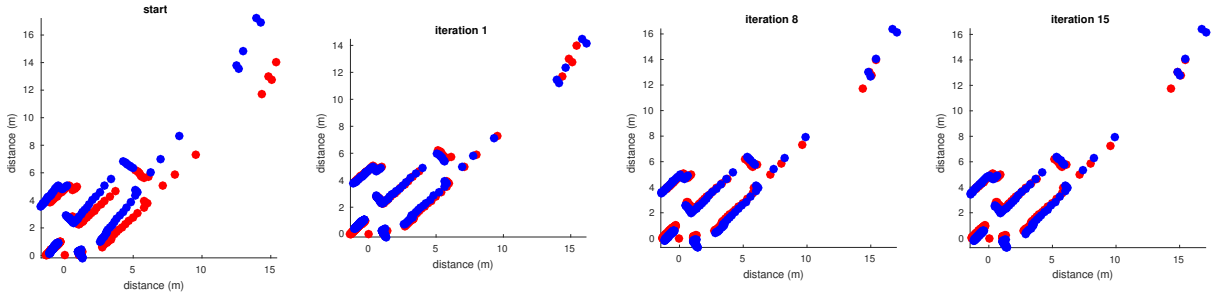
(a) Matlab floating point



(b) Non-restoring inverse square root



(c) Fast inverse square root with 1 iteration Newton-Raphson



(d) Fast inverse square root with 0 iterations Newton-Raphson

Figure 12.8: ICP algorithm performed on a laser scan with different algorithm implementations

PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Thu Nov 17 14:56:56 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	icp_alu
Top-level Entity Name	ICP_ALU_topEntity_0
Family	Cyclone V
Device	5CSXFC6D6F31C8ES
Power Models	Final
Total Thermal Power Dissipation	2609.25 mW
Core Dynamic Thermal Power Dissipation	2119.94 mW
Core Static Thermal Power Dissipation	427.24 mW
I/O Thermal Power Dissipation	62.07 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 12.9: Power analyser results from Quartus

12.3.3 MATLAB timing results vs hardware architecture timing

12.3.3.1 CλaSH/Hardware

The number of clock cycles used for one complete iteration of the ICP algorithm is depended on several variables

$$(5.5\frac{N}{M}+4)(N+1)+71\frac{N}{M}+4\frac{(i+2f)}{d}+35 \tag{12.16}$$

where

N = number of correspondences (points in a laser scan)

M = width of the vector ALU operations (parallel computations)

i = integer part of signed fixed-point

f = fractional part of signed fixed-point

d = number of non-restoring division steps taken in a single clock cycle

A laser scan from the Intel data set contains 180 points. With a vector ALU width of 45, signed fixed-point data of 27-bits, and 6 iterations of the non-restoring division in one clock cycle, the number of clock cycle used is 5053. Section 12.3.4 describes the synthesis and timing results of the hardware architecture and according to the Quartus tooling the clock frequency that can be reached is $40MHz$. With a clock frequency of $40MHz$ and 5053 clock cycles needed for a single iteration of the ICP algorithm means that a single iterations takes $0.13ms$. Section 12.3.4 describes an optimization regarding the multipliers used in the Newton-Raphson steps of the fast inverse square root. With these optimization the total number of clock cycles needed is 5057 and the clock frequency that can be reached is $50MHz$ which means that a single iteration of the ICP algorithm takes $0.10ms$. Figure 12.9 shows the results of the Quartus power analyser. The power analysis is performed on a design with a $50MHz$ clock frequency and shows that the total thermal power dissipation is 2.6 W.

12.3.3.2 MATLAB

Timing results in MATLAB cannot be determined as accurate compared to hardware simulations because external factors like a non-real-time operating system. The test performed in MATLAB for the timing results are generated on a system with an Intel(R) Core(TM) i5-3210M @ 2.50GHz processor and are meant to give an indication. According to the MATLAB profiler the computation of one iteration of the ICP algorithm takes approximately $18ms$. According to specification the thermal design power (TPD) of the target CPU is $35W$ which is usually a small overestimation with a 100% load. Since the MATLAB algorithm only runs on one virtual core of the processor it uses a 25% load. 25% of $35W$ equals $9W$. These numbers are not accurate and should be treated as indications.

	MATLAB on PC	FPGA architecture
Iteration duration	18 ms	0.10 ms
Iterations per second	55	10000
Energy consumption	9 W	2.6 W
Energy consumption per iteration	0.16 J	0.00026 J

Table 12.1: Energy consumption table

12.3.3.3 Comparison

Table 12.1 shows the energy consumption table. Despite the fact that the numbers of the MATLAB on PC are a bit inaccuracy one can clearly see that dedicated hardware can have an advantage in both energy and speed. Although the numbers shown in the figure might not be 100% accurate, they still show the order magnitude of the difference. For this example case the number of parallel operations, width of the system, is 45, increasing that 90 would result in almost half of the clock cycles needed, which would increase the speed almost twice.

12.3.4 Quartus synthesis and timing results

Synthesis results of ICP architecture, with a system width $N = 180$ and vector ALU width of $M = 45$, from the Quartus tooling is shown in Figure 12.10. The target FPGA for synthesis is 5CSXFC6D6F31C8ES. Figure 12.10a shows the summary of the synthesis of on architecture using an 18-bit data representation and a fast inverse square root without Newton-Raphson iterations. The 18-bit ICP architecture only has a logic utilization of 27% with only 15% of the blockram used. The number of DSP Blocks used is 50 wich are used in the following parts:

- 45 for vector operations
- 1 in the number ALU
- 4 for normal vector computation

Figure 12.10c show that the timing constraints are met, the target clock speed is $50Mhz$. According to the tooling the longest combinational path goes from blockram, through the compare tree, to some registers (that contain the indices of the closest and second closest points). Width a system width of $N = 180$ the compare tree, discussed in Section 10.1, has a depth of 7 nodes and forms the longest combinational path between two memory components.

Increasing the number of bits for the data representation and adding the Newton-Raphson iteration at the end of the fast inverse square root has an impact on the size and timing of the synthesized result, as can be seen in Figure 12.10b. The system with a 27-bit data representation has 40% logic utilization and uses 56 DSP blocks which are used for the following parts:

- 45 for parallel vector operations
- 1 in the number ALU
- 4 for normal vector computation
- 4 for the two fast inverse square roots (3 for each Newton-Raphson iteration)

Flow Status	Successful - Fri Sep 30 14:22:08 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	icp_alu_18
Top-level Entity Name	ICP_ALU_topEntity_0
Family	Cyclone V
Device	5CSXFC6D6F31C8ES
Timing Models	Preliminary
Logic utilization (in ALMs)	11,184 / 41,910 (27 %)
Total registers	428
Total pins	33 / 499 (7 %)
Total virtual pins	0
Total block memory bits	830,592 / 5,662,720 (15 %)
Total DSP Blocks	50 / 112 (45 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

(a) Quartus synthesis summary with 18 bits, 0 iterations of Newton-Raphson, 50 MHz

Flow Status	Successful - Fri Oct 21 11:03:24 2016
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Full Version
Revision Name	icp_alu
Top-level Entity Name	ICP_ALU_topEntity_0
Family	Cyclone V
Device	5CSXFC6D6F31C8ES
Timing Models	Preliminary
Logic utilization (in ALMs)	16,567 / 41,910 (40 %)
Total registers	599
Total pins	42 / 499 (8 %)
Total virtual pins	0
Total block memory bits	1,245,888 / 5,662,720 (22 %)
Total DSP Blocks	56 / 112 (50 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

(b) Quartus synthesis summary with 27 bits, 1 iteration of Newton-Raphson, 40 MHz

Fast 1100mV 85C Model Minimum Pulse Width Summary

	Clock	Slack	End Point TNS
1	system1000	8.787	0.000

(c) Quartus timing: 18 bits, 0 iterations of Newton-Raphson, 50 MHz

Fast 1100mV 0C Model Setup Summary

	Clock	Slack	End Point TNS
1	system1000	0.375	0.000

(d) Quartus timing: 27 bits, 1 iteration of Newton-Raphson, 40 MHz

Figure 12.10: Quartus results

Figure 12.10d shows the timing results of the circuit with a clock frequency of 40 MHz. The longest combinational path can be found in the inverse square root with the 3 sequential multipliers for the Newton-Raphson iteration. This can be made shorter by calculating the Newton-Raphson multiplications over time. This would have a minor influence on the number of clock cycles needed for computing the entire ICP algorithm. The inverse square root is used in two parts of the system, the normal vector creation (done by the snatcher), and 4 times for creating the orthonormal space $\mathbf{Q}$ (done by the vector ALU). As explained in Section 12.2, the snatcher creates normal vectors during the correspondence creation while the rest of the system is busy calculating distances. Therefore calculating the inverse square root in multiple clock cycles will not increase the total number of clock cycles for the correspondence creation part. However, the inverse square root is also used 4 times in the vector normalization for the orthonormal space $\mathbf{Q}$ and for each clock cycle added to the inverse square root calculation the total number of clock cycles needed for the entire ICP iteration will increase with 4.

Part IV

Conclusions and future work

13 — Conclusions

13.1 General conclusions

Section 1.2 summarizes the problem definition as follows:

- How can a SLAM solution be realized into a feasible hardware architecture
- Does a hardware architecture have the potential to be more efficient in terms of performance per joule compared to the commonly used computational systems

Both the Graph-based SLAM and ICP solutions have been designed using the same approach and use an vector type ALU as architecture. The realized architectures with their simulations show that both solutions are more efficient in terms of energy usage and increase speed compared to the commonly used computer systems by exploiting the parallel nature of an FPGA. Graph-SLAM specific conclusions can be found in Section 13.2, conclusions regarding the ICP algorithm can be found in Section 13.3. The algorithms and implementations have similarities and differences which are briefly discussed in this section.

Solving a linear system to find a transformation

Despite the fact that both solutions will calculate a transformation of points, the graph-based SLAM algorithm will result in a large vector describing each transformation of each point, while ICP will result in a transformation that describes the translation and rotation of the total set of points without the possibility for the points to move independent of each other. Because there is such a large difference between the final results of the algorithms, completely different architectures were found and implemented during this project.

Finding correspondences and removing outliers

Both algorithms contain steps of finding correspondences and possibly removing outliers. These steps are also different in both algorithms because format of the data is different. Correspondences in ICP are created from each point to every other point, while in graph-based SLAM only a correspondence is created from the current point to every other point. Outlier detection in graph-based SLAM will remove outlying poses from the graph, while in ICP it will remove outlying correspondences.

Vector ALU with a controlling mealy machine

The final implementations both consist of an ALU structure that does the (vector) computations and a mealy machine to control the data that is processed by the ALU. The data is stored in block RAMs from which the problem is loaded from into the registers and written back to after the computation.

Iterative convergence to solution

Both algorithms use an iterative process to converge to a solution. The processor structure of both architectures provide a control mechanism for keeping track of these iterations and reusing hardware.

13.2 Graph-based SLAM

SLAM on hardware

A hardware implementation of the graph-based SLAM algorithm has been created to prove feasibility. During the design phase, choices have been made to converge to an efficient and feasible design. The systems consists of a processor-like architecture with an ALU that has the ability to perform vector operations as main component. Parallelism is an important feature on a platform that is limited to a certain frequency and still needs to perform according to a specification. Nevertheless, parallelism is not the only property that accelerates the algorithm. Because each memory operation handles a vector, the amount of memory operations is less, and because the control is application specific, the utilization factor of the ALU is much higher than on a normal computer. Parallelism, less memory delay and dedicated control add up to a speed advantage of much more than only paralellism.

Timing and quality of the algorithm

The current implementation converges a large graph that contains a large error in three dimensions in 960ms. The resulting quality of the graph is mainly dependent on the amount of iterations of the conjugate gradient algorithm within the linear solver. The size of the resulting error is mainly determined by the amount of times the linear solver is applied in SLAM iterations. By changing the amount of iterations, the quality of the map and computation time can be affected. Correct parameters are specific for each environment and the final task of the robot. The amount of parallelism can be changed and for larger FPGAs it can be extended so the algorithm will become faster. The amount of clock cycles will roughly scale linearly with the amount of parallelism. The algorithm has been implemented in such a way that it is easy to achieve more parallelism for larger FPGAs by only changing the vector block sizes which is equivalent to the width of the ALU.

Deterministic computation times

The graph-based SLAM algorithm has been implemented in hardware and because a static amount of memory is used, the amount of clock cycles needed to execute the SLAM algorithm is also static. However, it would be a large advantage to add a stop condition which terminates the algorithm as the error is converged. By adding a stop condition, the worst case computations times can still be determined, but the algorithm will in practice always be faster.

13.3 ICP

The following conclusions can be drawn from researching and implementing an ICP algorithm on an FPGA and are discussed below:

Performance per joule increase

Section 12.3.3 shows the comparison in speed and energy usage between the implementation on an FPGA versus the implementation in MATLAB on a PC. Note that the energy and timing numbers for the MATLAB implementation are not very accurate but they indicate the order of magnitude of the difference in performance per joule between the MATLAB solution and the FPGA. The energy used for one iteration of the ICP algorithm differs with an order of magnitude of 3 between a MATLAB implementation and a hardware architecture, concluding that large power and speed savings can be achieved by using dedicated hardware for ICP.

Quality of the result

Section 12.3.2 show different results using different inverse square root methods and even the one with the Newton-Raphson iterations are slightly less accurate compared to the reference created in MATLAB. These inaccuracy are mainly caused by the inaccuracy of the inverse square root used for normal vector creation.

ICP suitable for hardware design due to regular structure, even for higher dimensional problems

Laser scan data has a fixed size and format and the algorithms used in this project to solve the ICP problem have a regular structure. The regular structure of data and algorithm is very suited for a hardware implementation and since ICP is not bounded for the 2-D case the hardware architecture does not have to be changed fundamentally to make it work with higher dimensional problem.

Regular structure of the designed hardware architecture allows for easily changing the width of the system

As described in Section 12.3.3.1 the number of clock cycles needed to compute 1 iteration of the ICP algorithm is mostly depended on two variables; the width of the system (M), and the width of the incoming data (N). In the CλaSH code these two variables can be easily changed and the entire system will change. The FPGA device chosen as target for this project has 112 DPSs on board but if one wants to choose another FPGA with more DPSs then the code can be easily adapted for a suitable hardware architecture. The flexibility to change the width of the system has also the advantage that one can easily adapt for different incoming data. The laser scan data used in this project contains 180 distances but there are also laser scanners with a data size of 270, or 360. Changing the size of the incoming data, or changing the size of the FPGA will only change the width of the system, and thus influences the number of clock cycles needed.

Problems regarding numerical stability can be fixed by using different numerical representations

Section 12.3.1 explains the problem of the representation of numbers regarding the size and precision. For the currently used data the chosen size and precision do not form a direct problem but when changing the size or dimension of the incoming laser scans one must keep in mind that numerical problems can arise.

14 — Future work

14.1 General future work

14.1.1 Coupling Graph-SLAM and ICP

Currently both solutions have a separate hardware architecture and there is no coupling between them. In order to make a fully functional SLAM solution the two parts must be linked together via communication channels where the Graph-SLAM solution can send positions between which the ICP algorithm must try to find a transformation.

14.1.2 Research towards automated parallelism

The complete SLAM solution consists of both ICP and the graph-based error convergence. The separation has been made because both algorithm are mathematically complex and have very different properties. However, the algorithms both solve a linear system and calculate and apply transformations. The principles that make the algorithms unique are already implemented in the fully parallel implementation. This features are things like QR decomposition and row reduction for ICP and sparse vector usage for graph-based SLAM. The conversion between the fully parallel implementation and an implementation with an ALU structure is only a matter of defining the amount of parallelization and timing parameters. The conversion into a ALU structure is something that could be done by some kind of a compiler.

14.2 Graph-based SLAM

14.2.1 Additions to the current SLAM implementation

During the implementation of the graph-based SLAM algorithm, the focus has been on solving the large linear systems that are needed for graph convergence. The solution works and can minimize errors by solving the linear systems iteratively. However, the data that has been presented to the linear solver is the data that already is a state vector in Cartesian coordinates, which is not the standard format SLAM datasets are in. Another important feature is the loop closing proposal function. Within the SLAM algorithm this functions is important because it will decide whether two poses should have correlation with each other. This feature has been discussed during design space exploration but has not yet been implemented in the current hardware implementation. These changes have already been implemented in MATLAB and the algorithm work as they should.

14.2.1.1 Creation of the state vector

The state vector is created by trigonometric functions which will convert a dataset of polar coordinates into a pose graph. The trigonometric functions can be evaluated by a cordic unit,

which has already been implemented in the fully parallel implementation of the SLAM algorithm. Cordic is an iterative method to find the sine and cosine of angles by using only shifts, adders, and a lookup table. Each iteration of the cordic algorithm can be divided into instructions that the ALU can execute, but since it does only contain a small amount of adders, one iteration can easily be done in parallel. The cordic unit would then become a separate mealy machine that evaluates the angles to construct the state vector.

Correction of the state vector in the theta direction is for a large part done in the same way as state vector creation, which is by using trigonometric functions. However, also a square root operation is required to find the lengths of line segments. The square root algorithm has also been implemented in the parallel implementation and similar to the cordic algorithm, it does consist of adders and shifters. The same method of implementation can be used as with the cordic unit, which means one iteration of the square root can be a combinational path and iteration can be done in multiple clock cycles.

14.2.1.2 Automated loop closing

The detection of loop closing is a combination of finding potential pairs of poses that could possibly close the loop and confirming the loop closing pairs by running the ICP algorithm between the two poses. Proposing loop closing pairs can consists of calculating distances between the poses and if a distance is low, the pair is offered to ICP for confirmation. This however requires the coupling between ICP and the SLAM algorithm which can also be considered future work. The distance calculations however just consist of two multiplications and one addition and thus can be done by the existing ALU implementation. The square root is not necessary because the smallest distances are looked for anyway.

14.2.2 Extended additions of the algorithm and implementation

These improvements will increase performance and capacity of the SLAM algorithm. They do however require an additional amount of research and much implementation time.

14.2.2.1 Improvements of the algorithm to reduce the system size dynamically

The research part discusses optimizations in the graph to reduce the amount of poses in the graph, which means less memory will be used and less computations are needed. If the algorithm is able to abstract the graph when the memory is fully used, the state vector of the robot could in theory be infinite. The quality of the map would suffer from such changes.

14.2.2.2 More parallelization by parallel sparse vectors

Whenever a ALU is available that has a large amount of parallelism, which in this case will probably be an ALU with a width of a higher power of two. For example 32, 64 still respecting the amount of available DSPs that are available on the FPGA. Operations on vectors will require less clock cycles which will make the complete algorithm run faster than it would with a smaller sized ALU.

This improvements however, will only affect the operations on non-sparse vectors, since they are large and can easily be split in a scalable way. Sparse vectors however do only contain five indices with their corresponding values. Assuming that the size of the sparse matrices will remain the same if the fixed amount of loop closing does not change.

14.3 ICP

14.3.0.1 FIFO for incoming laser scans

The current systems only works with blockrams. Both observations on which the ICP algorithm is performed on are preloaded into blockram but to couple the ICP system to other systems it required to have some sort of communication to allow loading for new observations. One could implement a FIFO structure where the observations are put in and the ICP system can fetch two observations, complete the ICP algorithm, and produce a transformation plus score (Section 14.3.0.3). Creating a communications structure from and to the ICP system results in a more separated block which can be used as an IP (intellectual property) by other systems. SLAM systems can then use one or more instances of the ICP system for its scan-matching part.

14.3.0.2 Stopping condition

The algorithm now has no stopping condition, which means that it will iterate a fixed amount of iterations but experience and results from Section 12.3.2 show that in many cases the solution has already converged and further iterations are not necessary. To implement a stopping condition one must implement a scoring mechanism (Section 14.3.0.3). If the score of the current found transformation is high enough then the algorithm can stop iterating. A disadvantage could be that the number of iterations is non deterministic but with a maxbound on the number of iteration the stopping condition will only provides a faster scan-matching process.

14.3.0.3 Scoring results

Not all transformations are 100% accurate, and scoring the correctness of a scan match is desirable for a couple of reasons:

- SLAM algorithms will be able to use the score to determine the probability of the position
- Loopclosing can be done by matching scans and determining if loopclosing occurs based on score
- Stopping conditions can be implemented based on score

In case of Graph-based SLAM each position in the state matrix has a certain value which represents the probability of a position relative to another. If a scanmating is able to provide a accurate score then it will fit nicely into the Graph-based solution. Scan matches with a low score will be corrected more due to the higher uncertainty. For loop closing it is also important to know the goodness of a scan. If the robot wants to find out if it has been on the current position before it will match the current observation with a couple of previous poses and if the score of the scan-matching is high enough the loop can be successfully closed.

14.3.0.4 Outlier rejection

In Section 10.2 the choice was made to skip the outlier rejection part due to project timing constraints. Outlier rejection could slightly improve the quality of the algorithm but a more important reason to implement outlier rejections is that it increases the convergence speed of the ICP algorithm. Increasing convergence speed decreases the number of iterations needed and with a stopping condition, (Section 14.3.0.2) one could save both energy and speed.

14.3.0.5 Transforming the transformation during the correspondence creation part

Section 12.2.0.7 describes the proccess of transforming the transformation which is needed to determine the total transformation over multiple iterations of the ICP algorithm. In the current system the process of transforming the transformation is done after linear solving the system $\mathbf{R}\vec{x} = \vec{t}$ by the number ALU and takes 14 clock cycles. The rest of the entire system is doing nothing those 14 clock cycles and an improvement can be made that the rest of the system can continue performing the next iteration of ICP while the number ALU continues transforming the transformation.

14.3.0.6 Newton-Raphson in multiple clock cycles

Section 12.3 mentions that the longest combinational path contains the 3 sequential multiplications done for the Newton-Raphson iterations for the inverse square root. The long combinational path causes the clock frequency to drop from $50MHz$ to $40MHz$ and Section 12.3 describes that a possible way to decrease the combinational path is by doing the multiplications in multiple clock cycles. If implemented like suggested in Section 12.3, the loss would be a total of 4 clock cycles, but an increase in clock frequency from $40MHz$ to $50MHz$.

14.3.0.7 Shift fixed point variables for large or small values

Section 12.3.1 describes the numerical problems with storing and calculating large and small numbers on an FPGA. It proposes a solution to up- or down-scale variables in different places where small or large values are known to occur. With this solution no extra bits are needed to increase the width of every variable, only a couple of guard digits to maintain precision.

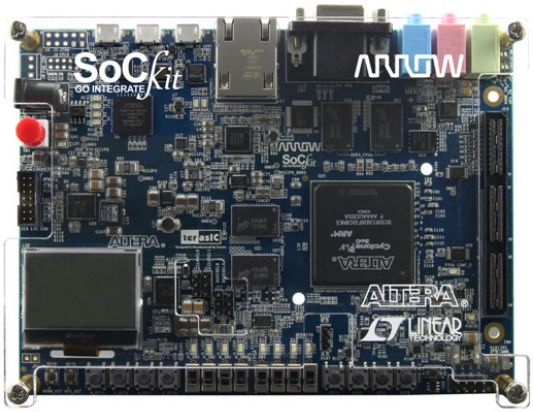
14.3.0.8 More accurate inverse square root

The fast inverse square root method is used to create a normal vector in every correspondence in the algorithm. The inverse square root is accurate but because it is used so frequently some inaccuracy is visible in the result. Developing a non-restoring inverse square root in hardware (or another type of hardware inverse square root) and simulating it remains future work.

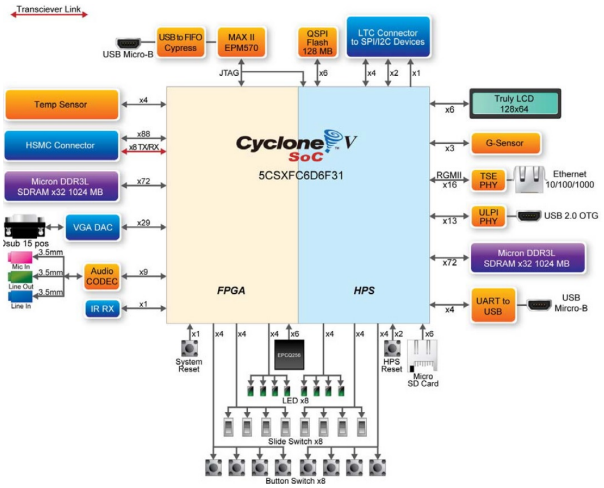
Appendices

A — FPGA

The SoCkit board shown in figure A.1a is an available platform which has both an ARM dual core processor and an Altera Cyclone FPGA on a single chip. The ARM processor is capable of running linux distributions and can easily communicate with sensors and networks. The FPGA can be used as a dedicated processor to execute the algorithm with the data coming from the ARM. Because the ARM and the FPGA are included in one package, the data bus that connects them will be faster and more robust than external connections. The complexity and size of the



(a) SoCkit PCB with SoC with onboard ARM Cortex A9 processor and Altera Cyclone FPGA



(b) Block diagram describing the peripherals connected to the FPGA and Hard Processor System (HPS)

Figure A.1: Development board SoCkit Arrow

algorithms that need to be implemented is quite high, therefore it is not possible to implement the SLAM algorithm fully parallel on the FPGA. The amount of resources is simply too low to allow for fully parallel executions. Specifications can be found in the SoC datasheet cyc [1]. The most important specifications are listed below:

FPGA:
- 110K Logic elements
- 112× Variable precision DSP blocks (27×27) / 224× 18x18 multipliers
- 4 LED's and 8 switches for debug purposes

HPS:
- ARM Cortex-A9 Dual Core, 800MHz
- USB and LAN Network Interface
- 4 LED's and 8 switches for debug purposes

B — Motion models for SLAM

The position of a robot in a 2D environment is expressed in the state $s_t = [x, y, \theta]$. The motion of the robot between two poses based on a control command is described in a motion model. The motion model can be expressed by the following distribution:

$$p(s_t|s_{t-1}, u_t) \tag{B.1}$$

where:

- s_t = new pose
- s_{t-1} = previous pose
- u_t = control command

The above equation describes how the current position is a probability distribution of the the previous pose and the current control command. The new position can then be used for map construction and path correction. The motion model can be divided into two models which are based on different sensors and user other control commands. These models are the odometry motion model and the velocity motion model.

Odometry Motion Model

Odometry data is obtained by encoders on the wheels of the robot. The odometry model represents the new robot position s_t relative to the previous pose s_{t-1}. This information can be inaccurate because of drift and slip of the robot and noise of the sensors. A robot can could in real life drive in fluent curved motions which are hard to express correctly in a model. Because the positions of the robot are discrete steps, it is possible to abstract from the fluent curves and use a simpler model, being the motion model. There are multiple ways of expressing this odometry based motion model. One way is to use a vector containing 2 rotations and 1 translation. Odometry is then represented as a change from position $(\bar{x}, \bar{y}, \bar{\theta})$ to $(\bar{x}', \bar{y}', \bar{\theta}')$ using $u = (\delta_{rot1}, \delta_{trans}, \delta_{rot2})$. See Figure B.1 where:

$$\begin{aligned}
\delta_{trans} &= \sqrt{(\bar{x}' - \bar{x})^2 + (\bar{y}' - \bar{y})^2} \\
\delta_{rot1} &= atan2(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta} \\
\delta_{rot2} &= \bar{\theta}' - \bar{\theta} - \delta_{rot1}
\end{aligned}$$

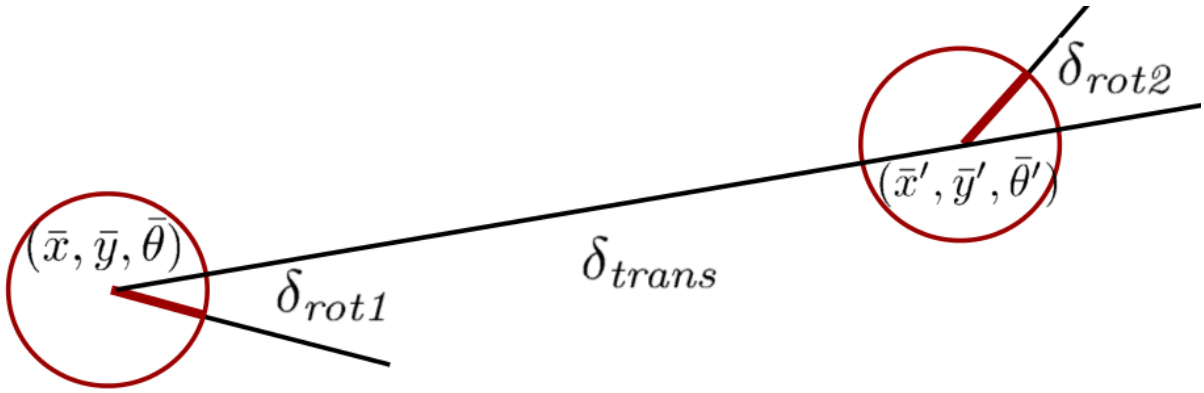


Figure B.1: Odometry motion model[55]

Velocity Motion Model

The velocity based motion model is usually applied on robots without wheels. Time (t) is used to calculate the rotational angle (using ω_t) and translation (using v_t) from the previous pose the current pose:

$$u_t = \begin{bmatrix} v_t \\ \omega_t \end{bmatrix} \tag{B.2}$$

When a robot moves from (x, y, θ) to (x', y', θ') it can be derived as follows:

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} -\frac{v}{\omega} sin(\theta) + \frac{v}{\omega} sin(\theta + \omega \Delta t) \\ \frac{v}{\omega} cos(\theta) - \frac{v}{\omega} cos(\theta + \omega \Delta t) \\ \omega \Delta t \end{bmatrix} \tag{B.3}$$

A more detailed explanation and mathematical derivation can be found in Section 5.3 and 5.4 of Thrun et al. [58].

Observation Model

In order to interpret observations within the algorithms observation models can be used that represent the sensor data of the physical world in a mathematical description. Again, due to sensor noise and inaccuracy the model is represented in a probability distribution:

$$p(z_t | s_t) \tag{B.4}$$

where:

- z_t = measured sensor data
- s_t = robot pose

Mobile robots can use different types types of sensors and multiple of them on one platform:

- *Contact sensors:* Bumpers
- *Internal sensors:*
  - Accelerometers
  - Gyroscopes

  - Compasses

- *Proximity sensors:*
  - Sonar
  - Radar
  - Laser range scanner
  - Infra-red
- *Visual sensors:* Cameras
- *Satellite-based sensors:* GPS

All of these sensors can be considered observational sensors, and will be represented by different models. Internal and satellite-based sensors can be used for the localization part of the SLAM algorithm. Contact, visual and proximity sensors can be used for both the localization and mapping part. Finding transformations between different sensor observations can be used for localization and once a position is known, the same data can be used for mapping.

Proximity sensors can be modeled using the beam-based sensor model which model represents each observation (scan) as a list consisting of K measurements.

$$z_t = z_t^1, z_t^2, ..., z_t^K$$

The measurements are independent of each other and in the case of laser scanner each z_t represents the total scan and each z^k represent the measurement under a certain angle. Because of sensor inaccuracy and noise the beam measurements are also modeled using a probabilistic approach:

$$p(z_t|s_t, m) = \prod_{k=1}^{K} p(z_t^k|s_t, m) \tag{B.5}$$

In a volumetric representation one could discretize the world into a grid cell representation where each grid cell contains a probability of whether it is a free space or occupied, this principle is known as an occupancy grid and is explained further in Section 3.1.1. Proximity sensors can be used to determine the probability of a cell to be occupied or free. In a laser range scanner each beam (z_t^k) represents a distance between the current robot position (s_t) and a cell (c), see Figure B.2 uses the following formulation, where $0 \leq p_{free} < p_{prior} < p_{occ} \leq 1$, to express the probability of a cell being occupied, which means the probability can never become larger than the probability that represents a free grid cell and never higher than the probability that represents an occupied grid cell.

$$p(c|z_t^k, s_t) = \begin{cases} p_{prior}, & z_t^k \text{ is a maximum range reading} \\ p_{prior}, & c \text{ is not covered by } z_t^k \\ p_{occ}, & |z_t^k - dist(s_t, c)| < r \\ p_{free}, & z_t^k \geq dist(s_t, c) \end{cases} \tag{B.6}$$

where:

- p_{prior} = previous probability of a cell
- p_{occ} = constant representing probability of cell being occupied

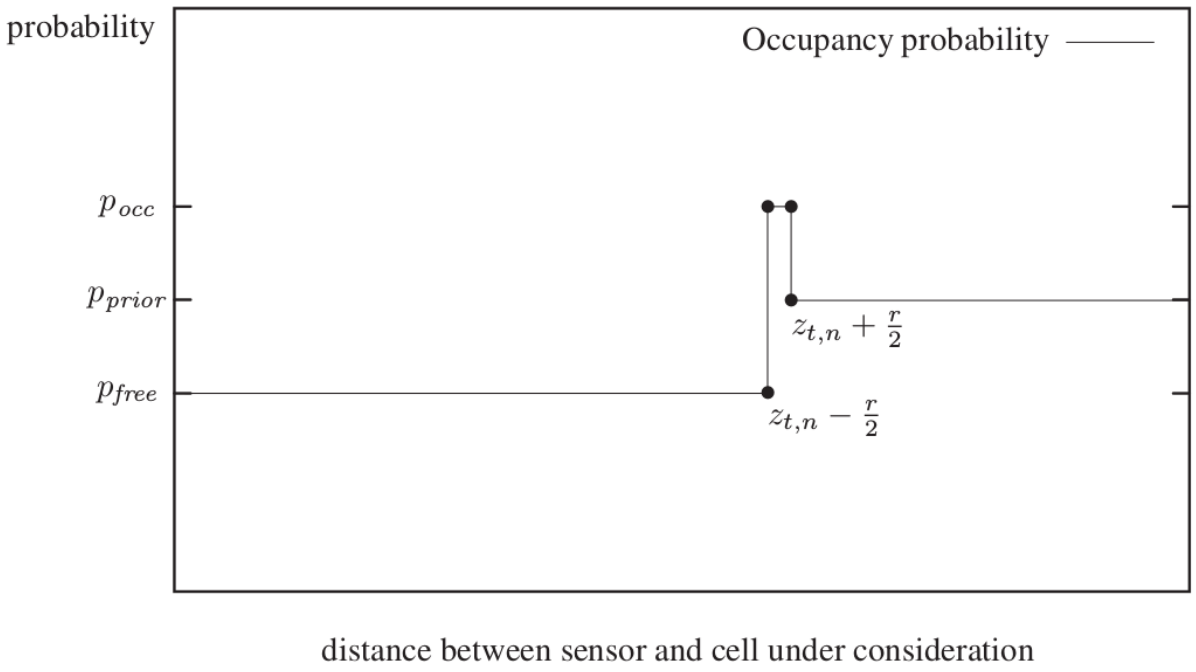


Figure B.2: Sensor model for a laser range scanner. It depicts the probability that a cell is occupied depending on the distance of that cell from the laser sensor[54].

- p_{free} = constant representing probability of cell being free
- c = grid cell
- r = resolution of grid map
- s_t = robot pose
- $dist(s_t, c)$ = distance between robot pose and cell

As described in Section 3.1.1 Thrun et al. [58] suggest a log-odds notation to avoid numerical instabilities for probabilities close to zero or one, where the **inverse_sensor_model** is used to merge the probabilities of a single observation into the grid probabilities. The function **inverse_sensor_model** implements the inverse measurement model $p(m_i|z_t, s_t)$ in its log-odds form:

$$\textbf{inverse\_sensor\_model}(m_i, s_t, z_t) = p(m_i|z_t, s_t) \tag{B.7}$$

Using Equation (B.6) and the log-odd notation of Section 3.1.1 an algorithm is suggested which creates the **inverse_sensor_model** and is shown in Algorithm 5. α Represents the width of the beam, β is the opening angle of the beam. In line 1 to 4 the range of the scan, (r) and angle (ϕ) is calculated using the robot postion $(s_t = [x, y, \theta])$. Line 5 checks if the measurement lies outside the cell, or if it lies more than half a beam width behind the detected range z_t^k. If so, then return the log-odd (l_0) of the previous occupancy (p_{prior}). However, if the cell is within the range and within the beam (line 7) then the algorithm returns occupied. For all the values that are between the object and the robot "free" is returned, line 9 and 10.

Algorithm 5 Inverse sensor model for laser range scanner

let x_i, y_i be the center-of-mass of m_i
$r = \sqrt{(x_i - x)^2 + (y_i - y)^2}$
$\phi = \text{atan2}(y_i - y, x_i - x) - \theta$
$k = \text{argmin}_j |\phi - \theta_{j,sens}|$
if $r > \min z_{max}, z_t^k + \alpha/2)$ or $|\phi - \theta_{k,sens}| > \beta/2$ **then**
    return l_0
else if $z_t^k < z_{max}$ and $|r - z_{max}| < \alpha/2$ **then**
    return l_{occ}
else if $r \leq z_t^k$ **then**
    return l_{free}
end if

C — Alternative scan-matching solutions

ICL: Iterative Closest Line

The *iterative closest line* (*ICL*) uses the same principle as the ICP algorithm, but instead of a set of points, it contains a set of shapes to match [44]. In case of a laser range scanner these shapes are often lines or corners. These shape features must be extracted first, but matching a set of lines consisting of points is less computational expensive than matching all points. Cox [18] uses this method for navigation inside structured buildings, like offices or factories.

PSM: Polar Scan Matching

Most proximity sensors return data in a format containing a bearing with a certain range. In many scan-matching algorithms this data format is transformed to a standard 2D map with x and y. *Polar scan-matching* (*PSM*) is a point to point matching algorithm that does not transform the proximity data to the x and y coordinate system [20].

NDT: Normal Distributions Transform

An occupancy grid represents the probability of a cell being occupied, see Section 3.1. The *Normal distribution transform* [14] (*NDT*) represents the probability of measuring a sample for each position within a cell. The space around the robot is divided regularly into cells with a constant size. The laser scan is transformed into this 2D space. Then, for each cell that contains at least three scan points the following is done:

1. Collect all 2D-Points $x_{i=1..n}$ contained in this box
2. Calculate the mean $\mu = \frac{1}{n}\sum_i x_i$
3. Calculate the covariance $\Sigma = \frac{1}{n}\sum_i (x_i - \mu)(x_i - \mu)^T$

Each cell then contains the following probability distribution for being occupied:

$$p(x) \sim e^{(\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu))}$$

Biber and Straßer [14] propose a method where odometry is used as an initial estimation for matching the scans. It is also mentioned that the score optimization step is done using Newton's algorithm. The structure of matching two scans is stated below

1. Create NDT of the first observation

2. Use odometry as first robot pose estimation, (or zero if no odometry is available)
3. For each scan point in the current observation: Map the reconstructed 2D points onto the NDT of the first observation according to the translation of the pose estimate of step 2.
4. Calculate al the normal distributions of the two observations combined
5. Score the transformation by summing all the distrubions of each grid point
6. Calculate the new robot position by optimizing the score using one step of the Newton's Algorithm.
7. If convergence criterion is not met, go to step 3

D — Alternative filters for SLAM

Gaussian filters

Due to errors and noise in measurements every position in SLAM has of a probability. This probability can be expressed by a function around the point. If a point is seen very often in the same position, the likelihood of the position will grow and the probability function around the position will change. In order to use Gaussian filters which are described in this section, this believe function or functions in a N-dimensional world, should be Gaussian. A Gaussian can be described by:

$$\frac{1}{\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{D.1}$$

Where μ is the center of the Gaussian function and σ is the width of the Gaussian function. The higher the sigma, the larger the uncertainty around μ. All Gaussian approaches described in this section will have the focus on feature-based SLAM with landmarks as observations.

Bayes Filter

The estimation of a robot's current position (pose) can be extracted from sensor and control data using filtering[55]. One of these filter techniques is called Bayesian filtering. The filtering uses a believe function whose definition is stated in equation D.2. The believe function uses the probability function to estimate the pose s_t when considering the given observations z_t and the control command u_t.

$$bel(s_t) = p(s_t|z_{1:t}, u_{1:t}) \tag{D.2}$$

Using this definition for the believe function, this definition can be rewritten to a form where the believe function is a product of on one hand the probability of the landmark observations given the current pose, all the previous found landmark observations and all the previous and current control inputs and on the other hand the probability of the current pose given the previous landmark observations and the previous and current control inputs. Because of the Bayes' rule D.3 this believe function can be rewritten to Equation D.4. Note that in this equation the division by B, which turns out to be a normalization term has been replaced by the normalization symbol.

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)} \tag{D.3}$$

$$p(s_t|z_{1:t}, u_{1:t}) = \eta\ p(z_t|s_t, z_{1:t-1}, u_{1:t}) p(s_t|z_{1:t-1}, u_{1:t}) \tag{D.4}$$

The $z_{1:t}$ and $u_{1:t}$ are not necessary when the pose s_t is known. Therefore the z and u terms disappear D.5 This is an application of the markov assumption.

$$p(s_t|z_{1:t}, u_{1:t}) = \eta\ p(z_t|s_t) p(s_t|z_{1:t-1}, u_{1:t}) \tag{D.5}$$

The main idea behind using the Bayes filter is to describe the recursive relation between the different steps in time. Therefore the previous state of the robot needs to be considered which state is called s_{t-1}. This variable can be used since it is a parameter that was calculated in a previous timestep. This parameter can be introduced by the law of total probability D.6.

$$p(A) = \eta \int p(A|B)p(B)dB \tag{D.6}$$

Using this law the following statement holds:

$$p(s_t|z_{1:t}, u_{1:t}) = \eta\ p(z_t|s_t) \int p(s_t|s_{t-1}, z_{1:t-1}, u_{1:t})p(s_{t-1}|z_{1:t-1}, u_{1:t})ds_{t-1} \tag{D.7}$$

Where the parameter u_t is not used for estimating s_{t-1} the second term becomes the believe function of s_{t-1}. Using the markov assumption that s_t can be estimated from only the previous pose and the current control output the complete function becomes:

$$bel(s_t) = \eta\ p(z_t|s_t) \int p(s_t|s_{t-1}, u_t)p(s_{t-1}|z_{1:t-1}, u_{1:t-1})ds_{t-1} \tag{D.8}$$

This equation is equal to:

$$bel(s_t) = \eta\ p(z_t|s_t) \int p(s_t|s_{t-1}, u_t)bel(s_{t-1})ds_{t-1} \tag{D.9}$$

Which confirms the recursive relation of the Bayes filter. This believe function can be split up into two parts. The part right of the integral sign is the is the estimation and the part before the integral is the state estimation correction by means of observations. The prediction steps consists of the so called motion model which is is the belief of the previous state (s_{t-1}), multiplied with the probability of the new state (s_t). This equation is equal to:

$$\overline{bel(s_t)} = \int p(s_t|s_{t-1}, u_t)bel(s_{t-1})ds_{t-1} \tag{D.10}$$

This barred believe function is the prediction step which gives a new probability that newly calculated position is correct. The motion model only describe the influence of odometry on the previous position and the relative probability that the new position is correct, this is multiplied by the previous probability to obtain the new probability to get the total probability.

Kalman Filter

According to Section 3.2 of Thrun et al. [58], the Kalman filter (KF) represents probability distribution at time t by the mean μ_t and covariance Σ_t.

$$p(x) = det(2\pi\Sigma)^{-\frac{1}{2}} exp\{-(x-\mu)^T\Sigma^{-1}(x-\mu)\} \tag{D.11}$$

The KF can be applied to robot localization, using feature-based landmarks. The landmarks and the current robot position is stored in the state vector μ. The first step is to predict the next robot position. This prediction is done using the motion model of the robot. Then, using the observation model, the prediction is corrected in the correction step. This correction step uses the current sensor data (observation model) to correct its prediction. Equation D.11 shows that the current position probability distribution $p(x)$ is given using a Gaussian with a probability (σ) and a mean (μ). The covariance matrix Σ holds the probabilities for the current position. The number of elements in Σ is quadratically the number of elements in the state vector μ,

which contains $3 + 2n$ elements where n is the number of landmarks, 3 because of the size of the state $s_t = [x, y, \theta]$. The entire belief is represented by the following:

$$\underbrace{\begin{pmatrix} x \\ y \\ \theta \\ m_{1,x} \\ m_{1,y} \\ \vdots \\ m_{n,x} \\ m_{n,x} \end{pmatrix}}_{\mu} \underbrace{\begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \cdots & \sigma_{xm_{n,x}} & \sigma_{xm_{n_y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \cdots & \sigma_{ym_{n,x}} & \sigma_{ym_{n_y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{thetam_{1,y}} & \cdots & \sigma_{\theta m_{n,x}} & \sigma_{\theta m_{n_y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{m_{1,x}\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \cdots & \sigma_{m_{1,x}m_{n,x}} & \sigma_{m_{1,x}m_{n_y}} \\ \sigma_{m_{1,y}x} & \sigma_{m_{1,y}y} & \sigma_{m_{1,y}\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \cdots & \sigma_{m_{1,y}m_{n,x}} & \sigma_{m_{1,y}m_{n_y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{n,x}x} & \sigma_{m_{n,x}y} & \sigma_{m_{n,x}\theta} & \sigma_{m_{n,x}m_{1,x}} & \sigma_{m_{n,x}m_{1,y}} & \cdots & \sigma_{m_{n,x}m_{n,x}} & \sigma_{m_{n,x}m_{n_y}} \\ \sigma_{m_{n,y}x} & \sigma_{m_{n,y}y} & \sigma_{m_{n,y}\theta} & \sigma_{m_{n,y}m_{1,x}} & \sigma_{m_{n,y}m_{1,y}} & \cdots & \sigma_{m_{n,y}m_{n,x}} & \sigma_{m_{n,y}m_{n_y}} \end{pmatrix}}_{\Sigma}$$

Posteriors are Gaussian if the following three properties hold, in addition to the Markov assumptions of the Bayes filter.

1. The next state probability, Equation (B.1), must be a linear function in its arguments including Gaussian noise. The motion model (Section B) is expressed using the following equation:

$$s_t = A_t s_{t-1} + B_t u_t + \epsilon_t \tag{D.12}$$

   where:

   - t = time
   - s_t = state
   - u_t = control vector
   - A_t = Matrix $(n \times n)$ for how the state evolves from $t-1$ to t without control
   - B_t = Matrix $(n \times l)$ for how the control u_t changes the state from $t-1$ to t
   - ϵ_t = Gaussian motion noise, where its mean is 0 and its covariance will be denoted R_t

   The next state probability, Equation (B.1), can be obtained by using Equation (D.12) and Equation (D.11). The mean μ_t the becomes $A_t s_{t-1} + B_t u_t$ and the covariance is R_t, so:

$$p(s_t|u_t, s_{t-1}) = det(2\pi R_t)^{-\frac{1}{2}} exp\{-\frac{1}{2}(s_t - A_t s_{t-1} - B_t u_t)^T R_t^{-1}(s_t - A_t s_{t-1} - B_t u_t)\} \tag{D.13}$$

2. The measurement probability (Equation (B.4) must also have linear arguments with Gaussian noise:

$$z_t = C_t s_t + \delta_t \tag{D.14}$$

   where:

   - C_t = Matrix $(k \times n)$ for how to map s_t to the observation z_t
   - δ_t = Gaussian measurement noise, mean is 0.0 and its covariance will be denoted Q_t

   With a mean 0.0 and a covariance Q_t the measurement probability becomes:

$$p(z_t|s_t) = det(2\pi Q_t)^{-\frac{1}{2}} exp(-\frac{1}{2}(z_t - C_t s_t)^T Q_t^{-1}(z_t - C_t s_t)) \tag{D.15}$$

3. The initial belief, $p(s_0)$ is denoted using μ_0 and covariance Σ_0 in:

$$p(s_0) = det(2\pi\Sigma_0)^{-\frac{1}{2}} exp(-\frac{1}{2}(s_0 - \mu_0)^T \Sigma_0^{-1}(s_0 - \mu_0)) \tag{D.16}$$

These three assumptions are sufficient to ensure that the posterior $bel(s_t)$ is always a Gaussian, for any point in time t. A more detailed explanation and a mathematical proof can be found in Section 3.2 of Thrun et al. [58]. The Kalman filter algorithm is shown in Algorithm 6. Step 1 and 2 calculate the state $\bar{\mu}_t$ and probability $\bar{\Sigma}_t$. The state prediction is calculated by

Algorithm 6 Kalman filter

1: $\bar{\mu}_t = A_t\mu_{t-1} + B_t u_t$
2: $\bar{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t$
3: $K_t = \bar{\Sigma}_t C_t^T (C_t\bar{\Sigma}_t C_t^T + Q_t)^{-1}$
4: $\mu_t = \bar{\mu}_t + K_t(z_t - C_t\bar{\mu}_t)$
5: $\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$

just using the motion model (control vector u_t), the previous state μ_{t-1} and the estimated motion noise (R_t). In step 3, the Kalman gain (K_t) is calculated. This calculation involves the sensor model (C_t) and sensor noise (Q_t). The Kalman gain specifies the degree to which the observation model, measurement z_t, corrects the motion model (prediction $\bar{\mu}_t$). In step 4 and 5 this prediction step is corrected by the Kalman gain (K_t) from step 3. Step 4 calculates the new mean by using the Kalman gain, the observation model and the prediction $\bar{\mu}_t$. The covariance is also adjusted based on this Kalman gain in step 5. In each iteration of the Kalman filter the covariance matrix must be inverted, with efficient algorithms a $n \times n$ matrix can be inverted with $\mathcal{O}(n^{2.373})$ complexity.

Extended Kalman Filter

The assumption that a state transition and measurements are linear is not case for most practical situations. A robot can move in directions which do not have to be linear, so the KF will not work correctly. The extended kalman filter (EKF) overcomes the assumption by means of linearization. The linear Equation D.12 and D.14 contain A_t, B_t, and C_t matrices which, in a non linear situation will be replaced by functions g and h and thus make the probability distribution no longer a Gaussian:

$$s_t = g(u_t, s_{t-1}) + \epsilon_t \tag{D.17}$$

$$z_t = h(s_t) + \delta_t \tag{D.18}$$

The idea of the EKF is to linearize the non linear function in order to have a posterior Gaussian distribution. This linearization is done via the first order Taylor expansion. Taylor expansion constructs a linear approximation of a functions value and slope. The slope is given by a partial derivative:

$$g'(u_t, s_{t-1}) = \frac{\partial g(u_t, s_{t-1})}{\partial s_{t-1}} \tag{D.19}$$

A choice for the arguments for g is the previous mean μ_{t-1} and the control vector u_t because it is likely that those arguments would be the approximation. So the linear approximation for g becomes:

$$g(u_t, s_{t-1}) \approx g(u_t, \mu_{t-1}) + G_t(s_t - \mu_{t-1}) \tag{D.20}$$

G_t is a Jacobian matrix of size $n \times n$ where n is the size of the state vector. The Gaussian next state probability becomes:

$$p(s_t|u_t, s_{t-1}) \approx det(2\pi R_t)^{-\frac{1}{2}} exp(-\frac{1}{2}[s_t - g(u_t, \mu_{t-1}) - G_t(s_{t-1} - \mu_{t-1})]^T R_t^{-1}[s_t - g(u_t, \mu_{t-1}) - G_t(s_{t-1} - \mu_{t-1})]) \tag{D.21}$$

For the observation model, used in the corrector, the EKF applies the same principle but for the closes approximation it takes the predictor outcome $\bar{\mu_t}$

$$h(s_t) \approx h(\bar{\mu_t}) + H_t(s_t - \bar{\mu_t}) \tag{D.22}$$

The Gaussian next state corrector then becomes:

$$p(z_t|s_t) \approx det(2\pi Q_t)^{-\frac{1}{2}} exp(-\frac{1}{2}[z_t - h(\bar{\mu_t}) - H_t(s_t - \mu_t)]^T Q_t^{-1}[z_t - h(\bar{\mu_t}) - H_t(s_t - \mu_t)]) \tag{D.23}$$

With these linearizations the EKF algorithm is shown in Algorithm 7:

Algorithm 7 Extended Kalman filter

1: $\bar{\mu_t} = g(u_t, \mu_{t-1})$
2: $\bar{\Sigma_t} = G_t \Sigma_{t-1} G_t^T + R_t$
3: $K_t = \bar{\Sigma_t} H_t^T (H_t \bar{\Sigma_t} H_t^T + Q_t)^{-1}$
4: $\mu_t = \bar{\mu_t} + K_t(z_t - h_t \bar{\mu_t})$
5: $\Sigma_t = (I - K_t H_t)\bar{\Sigma_t}$

The main difference between the KF and the EKF algorithm is that the linear system matrices A_t and B_t are replaced by the Jacobian G_t and the linear system C_t is replaced by the Jacobian H_t. A more detailed explanation and mathematical proofs of the EKF can be found in Section 3.2 of [58]. The cost per step is dominated by the number of landmarks n; $\mathcal{O}(n^2)$. The total cost, including building a map with n landmarks is $\mathcal{O}(n^3)$. For memory usage: $\mathcal{O}(n^2)$. Fast SLAM with Rao-Blackwellized uses EKF, see Section D for more information.

Unscented Kalman Filter

The Extended kalman filter uses the Taylor expansion for its linearization. The unscented kalman filter (UKF) uses the same state distribution but it uses a set of chosen sample points, called sigma points. Each sigma point, $\chi^{[i]}$, has a weight $w^{[i]}$ and is transformed through a non-linear function f [61][55]. The function f is the transition of the robot from pose x_{t-1} to x_t . Then the Gaussian is computed again from the transformed and weighted sigma points. The points and weights must be chosen in such a way that:

$$\begin{aligned} \sum_i w^{[i]} &= 1 \\ \mu &= \sum_i w^{[i]} \chi^{[i]} \\ \Sigma &= \sum_i w^{[i]} (\chi^{[i]} - \mu)(\chi^{[i]} - \mu)^T \end{aligned} \tag{D.24}$$

There is no unique solution for $\chi^{[i]}$ and $w^{[i]}$. Wan et al. [61] suggest a method for choosing the sigma points using the following equations

$$\begin{aligned} \chi^{[0]} &= \mu \\ \chi^{[i]} &= \mu + (\sqrt{(n+\lambda)\Sigma})_i \quad for\ i = 1, ..., n \\ \chi^{[i]} &= \mu - (\sqrt{(n+\lambda)\Sigma})_{i-n} \quad for\ i = n+1, ..., 2n \end{aligned}$$

Where n is the dimension, λ is the scaling parameter. The weight of the sigma points is calculated as follows:

$$w_m^{[0]} = \frac{\lambda}{n+\lambda}$$
$$w_c^{[0]} = w_m^{[0]} + (1 - \alpha^2 + \beta)$$
$$w_m^{[i]} = w_c^{[i]} = \frac{1}{2(n+\lambda} \quad for\ i = 1, ..., 2n$$

Once these sigma points are selected, calculated and transformed through the non-linear function it is necessary to recover the Gaussian again. This is done by substituting the non-linear function $g(x)$ in equations D.24:

$$\mu' = \sum_{i=0}^{2n} w_m^{[i]} g(\chi^{[i]})$$
$$\Sigma' = \sum_{i=0}^{2n} w_m^{[i]} (g(\chi^{[i]}) - \mu')(g(\chi^{[i]}) - \mu')T$$

Mathematical proof and further derivation of the covariance matrix calculations can be found in [61][55].

Information Filter

Like the Kalman filter the information filter(IF) can be used to solve the SLAM problem. The IF uses a so called canonical representation which consists of the information matrix, Ω (inverse covariance matrix from KF), and the information vector, ξ. This representation is shown in Equation (D.25) and (D.26).

$$\Omega = \Sigma^{-1} \tag{D.25}$$

$$\xi = \Sigma^{-1}\mu \tag{D.26}$$

Converting the information matrix and vector into the covariance matrix and mean-vector, the information matrix should be inverted again:

$$\Sigma = \Omega^{-1} \tag{D.27}$$

$$\mu = \Omega^{-1}\xi \tag{D.28}$$

Using an inverted covariance matrix will change the prediction and correction step of the Kalman filter, Equation (D.29) and (D.29) show the prediction and correction step of the Kalman filter (Section D):

$$\bar{\mu_t} = A_t\mu_{t-1} + B_t u_t$$
$$\bar{\Sigma_t} = A_t\Sigma_{t-1}A_t^T + R_t$$

Correction step:

$$\mu_t = \bar{\mu_t} + K_t(z_t - C_t\bar{\mu_t})$$
$$\Sigma_t = (I - K_t C_t)\bar{\Sigma_t}$$

Using the inverted covariance matrix, the prediction and correction steps of the Kalman filter can be rewritten into Equation (D.29) and (D.29) (for a more detailed step by step derivation see Stachniss [55]).

$$\bar{\xi_t} = \bar{\Omega_t}(A_t\Omega_{t-1}^{-1}\xi_{t-1} + B_t u_t)$$
$$\bar{\Omega_t} = (A_t\Omega_{t-1}^{-1}A_t^T + R_t)^{-1}$$

Correction step:

$$\xi = C_t^T Q_t^{-1} z_t + \bar{\xi}_t$$
$$\Omega = C_t^T Q_t^{-1} C_t + \bar{\Omega}_t$$

In comparison to the KF the IF has a more complex prediction, but the correction step is less complex.

Extended Information Filter

similar to the Kalman filter, the information filter uses a linear model. For the extended information filter (EIF) the linearization is the same as for the EKF. The Taylor expansion is used as a linear estimation of the non-linear function. See Equations D.17 and D.18, both g and h need the current state as input. In the canonical representation the state can be computed by Equation D.18. However, in the algorithm it would be more efficient to just recover the state estimation from the previous state, (see step 1 of the algorithm). The EIF also uses the Jacobians G_t and H_t (see Section D). The entire EIF algorithm is stated in Algorithm 8.

Algorithm 8 Extended information filter

1: $\mu_{t-1} = \Omega_{t-1}^{-1} \xi_{t-1}$
2: $\bar{\Omega}_t = (G_t \Omega_{t-1}^{-1} G_t^T + R_t)^{-1}$
3: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
4: $\bar{\xi}_t = \bar{\Omega}_t \bar{\mu}_t$
5: $\Omega_t = \bar{\Omega}_t + H_t^T Q_t^{-1} H_t$
6: $\xi_t = \bar{\xi}_t + H_t^T Q_t^{-1} (z_t - h(\bar{\mu}_t) + H_t \bar{\mu}_t)$

Takeover

They came in peace, they said..

Sparse Extended Information Filter

The computational complexity of the matrix manipulations in the algorithms is quite high. This is caused by the constant calculations that involve the links between the robot and the landmarks and between the landmarks themselves. The way data is represented in the information matrix the matrix is suitable for optimalization. As a robot moves around in an environment the nearest landmarks are the strongest links with the robot and the robot is most sure about its position relative to these landmarks and the positions and distance between visible landmarks. This type of algorithm is used to increase the computational efficiency. [55][57]

As always with feature-based SLAM there exists a set of landmarks. In this set of landmarks there is a subset of landmarks that are active. This subset consists of at least the current observed landmarks but can also contain other landmarks. The landmarks that are used to obtain the position of the robot are called the active landmarks. All others landmarks are called passive landmarks.

The process of separating active landmarks from passive landmarks is called sparsification. The total set of landmarks m can can be seen as:

$$m = m^+ + m^0 + m^-$$

The information matrix can be seen as a sum of three matrices which correspond to these landmark sets. The sparsification of the direct links is done by:

$$\tilde{p} = (x_t, m|z_{1,t}, u_{1:t}) \simeq \frac{p(x_t, m^+|m^- = 0, z_{1:t}, u_{1:t})}{p(m^+|m^- = 0, z_{1:t}, u_{1:t})} p(m^0, m^+, m^-|m^- = 0, z_{1:t}, u_{1:t}) \quad \text{(D.29)}$$

From this approximation the sparsified information matrix becomes a sum of matrices:

$$\tilde{\Omega}_t = \Omega_t^1 - \Omega_t^2 + \Omega_t^3$$

The matrices have been sparsified by manipulators F_{m_x}.

Particle filters

General idea

Particle filtering is a Bayesian filtering technique to determine the state variables of a system based on noisy measurements [8]. Unlike the Gaussian solutions particle filter can be used for both feature-based and volumetric-based SLAM. The particle filter approximates the posterior by a finite number of samples. The more samples are used, the more accurate the prediction becomes. Each sample consists a state hypothesis $s^{[j]}$ and importance weight $w^{[j]}$.

$$\chi = \langle s^{[j]}, w^{[j]} \rangle_{j=1,\ldots,J} \quad \text{(D.30)}$$

The posterior is represented as follows

$$p(s) = \sum_{j=1}^{J} w^{[j]} \delta_{s^{[j]}}(s) \quad \text{(D.31)}$$

A commonly used type of particle filtering is the sequential importance re-sampling filter (SIRF) which consists of four steps: prediction, update normalization and re-sampling[62]. Monte Carlo Localization (MCL) [29] is a particle filter of the SIRF type which uses particles as pose hypothesis, the four steps for MCL are explained in the following section.

Prediction

In the prediction step the next state is derived from the current state using an proposal distribution. The MCL uses the robots motion model, described in Appendix B, as a proposal function for each particle (pose hypothesis) [19]. The proposal distribution with the motion model is given in the following equation

$$x_k^{(i)} \sim p(s_t|z_{1:t}, u_{1:t}) \quad \text{(D.32)}$$

Update

In the update step, weights $w_k^{(j)}$ are assigned to all particles. The current observation model z_t, described in Appendix B, can be used to determine the weights.

$$w_t^{[j]} = p(z_t|x_t^{[m]}) \quad \text{(D.33)}$$

Normalization

In the normalization step all the weights of the particles are normalized to get an integral of 1. The normalization step can be formulated as follows:

$$\bar{w_t^{[j]}} = \frac{w_t^{[j]}}{w_{tot}} \quad \text{for} \quad j = 1..N$$
$$\text{where } w_{tot} = \sum_{j=1}^{J} w^{[j]} \tag{D.34}$$

Resampling

The re-sampling step prevents degeneracy of weights [15]. Degeneracy occurs when particles with a high weight are given an even higher weight over several iterations. This makes the particles with a low weight insignificant [62]. Particles are replicated 0, 1 or more times proportional to their normalized weight $w^{[j]}$. The Monte Carlo particle filter algorithm is shown below:

Algorithm 9 Particle Filter

1: $\bar{\chi_t} = \chi_t = \emptyset$
2: **for** j = 1 **do** J
3: *sample* $x^{[j]} \sim p(s_t|u_t, x_{t-1}^{[j]})$
4: $w^{[j]} = p(z_t|x^{[j]})$
5: $\bar{\chi_t} = \bar{\chi_t} + \langle x_t^{[j]}, w_t^{[j]} \rangle$
6: **end for**
7: **for** j=1 **do** J
8: *draw* $i \in 1, ..., J$ with probability $\propto w_t^{[j]}$
9: *add* $x_t^{[i]}$ *to* χ_t
10: **end for**
11: **return** χ_t

Rao-Blackwellized particle filter

The key idea of the Rao-Blackwellized particle filter (RBPF) is to split up the general SLAM problem of Equation (2.2) which is restated below [26].

$$p(s_{0:t}, m_{1:M}|z_{1:t}, u_{1:t}) \tag{D.35}$$

Since mapping with known poses is considered to be more trivial the equation is split up in a mapping part and localization part:

$$p(s_{1:t}|z_{1:t}, u_{1:t})p(m_{1:M}|s_{0:t}, z_{1:t}) \tag{D.36}$$

It can be shown that once all the poses are known, the landmarks are independent of each other. Which means that the mapping part of Equation (D.36) can be rewritten as follows:

$$p(s_{1:t}|z_{1:t}, u_{1:t}) \prod_{i=1}^{M} p(m_i|s_{0:t}, z_{1:t}) \tag{D.37}$$

If, for example, EKF is used as landmark estimation then instead of a large matrix containing all the landmarks the distribution can be realised by M 2×2 matrices which reduces complexity.

The poses are known because of the localization using the particle filter. Each particle represents a pose hypothesis and thus contains the current pose of the robot and the entire map. It is not necessary to keep track of all the previous parts poses because the particles do not use any methods to correct previous poses. If a particle is a bad hypothesis then it will receive a low weight and with the resampling step it will be removed. This idea of partial particle filtering is used in i.e. FastSLAM and FastSLAM2.

FastSLAM

FastSLAM [42] uses the Rao-Blackwellized particle filter for its localization. So each particle in the particle filter represents a pose hypothesis and contains a map of the environment. For the mapping part, FastSLAM uses EKF's for each landmark. However [42] implements a tree-based data structure which reduces the execution time. In FastSLAM 1.0 the four steps of the SIRF particle filter are used. In the prediction step the path posterior is extended by sampling a new pose for each sample. So each particle is extended by estimating a new pose based on the motion model:

$$s_t^{[k]} \sim p(s_t | s_{t-1}^{[k]}, u_t) \tag{D.38}$$

In the update step the particle weight is computed using the observation model, so it matches the current observation with the expected observation and uses the measurement covariance to determine how those two relate:

$$w^{[k]} = |2\pi Q|^{-\frac{1}{2}} exp(-\frac{1}{2}(z_t - \hat{z}^{[k]})^T Q^{-1}(z_t - \hat{z}^{[k]}) \tag{D.39}$$

where:

1. z_t = current observation
2. $\hat{z}^{[k]}$ = expected observation
3. Q = measurement covariance

After the weight calculation all the landmark probabilities are updated using the EKF update, see Appendix D and Algorithm 7. The update step also contains the normalization. Finally the particles are resampled. The mathematical proof and algorithm can be found in Montemerlo et al. [42]. The complexity of this basic implementation is $\mathcal{O}(NM)$ where N is the number of particles and M is the number of map features. However, [42] proposes a tree data structure which reduces the complexity to $\mathcal{O}(N \log M)$.
FastSLAM 1.0 only uses the motion model as the proposal distribution for the prediction step. FastSLAM 2.0 also considers the measurement during the sampling as a proposal distribution:

$$s_t^{[k]} \sim p(s_t | s_{t-1}^{[k]}, u_{1:t}, z_{1:t}) \tag{D.40}$$

In practice this means that FastSLAM 2.0 tries to align the current observation with the previous one to get a better pose estimate. The FastSLAM algorithm cannot only be applied to feature-based data representation but also volumetric-based data representation, using grid maps (see Section 3.1). Each particle represents a possible robot path. This means that each particle should also contain its own map. The number of particles determines the accuracy, because more particles increases the chance of a correct particle. But storing multiple maps eventually uses a lot of memory. The only way to limit memory use and still get a good estimate is by resampling. Resampling in SLAM context consists of getting rid of bad particles and use the memory it frees to store copies of good particles from which the particle filter can estimate again.

This resampling step is necessary for convergence. Equation (D.41) shows the calculation of n_{eff}.

$$n_{eff} = \frac{1}{\Sigma_i (w_t^{[i]})^2} \tag{D.41}$$

What equation D.41 basically does is integrating all particle weights to find the overall quality of the samples at that point in time. If the total weight (n_{eff}) of the particles drops between a certain value, resampling will be carried out.

Loop closing

In a particle filter based solution to the SLAM problem the loop closing is done by matching the newly found observation data to old scans based on the pose distribution of the particle. If the particle crosses a certain point where it believes to have been before, on a positive scan match, its believe will increase drastically and resampling can be carried out.

E — Graph-based SLAM algorithm

Algorithm 10 Basic Graph SLAM algorithm

initialize:
$H_{0,0} = 1$
repeat every timestep:
if (new_pose_added) then:
    $\theta_{t+1} = \theta_t \ + \ \Delta\theta$
    $\mathbf{x}_{t+1} = \mathbf{x}_t \ + \ \Delta d_t \ \times \ cos \ \theta_{t+1}$
    $\mathbf{y}_{t+1} = \mathbf{y}_t \ + \ \Delta d_t \ \times \ sin \ \theta_{t+1}$
    $H^* = H \ + \ H_{i,i+1}$
    if (loop_closing_found) then:
    repeat k times or until convergence :
        $H = H \ + \ H_{i,j}$
        $\mathbf{e}_x = \mathbf{z}_{i,j} - (\mathbf{x}_j - \mathbf{x}_i) \forall (i,j)$ *loop closing pairs*
        $\mathbf{e}_y = \mathbf{z}_{i,j} - (\mathbf{x}_j - \mathbf{x}_i) \forall (i,j)$ *loop closing pairs*
        $\mathbf{e}_\theta = \mathbf{z}_{i,j} - (\mathbf{x}_j - \mathbf{x}_i) \forall (i,j)$ *loop closing pairs*
        $\mathbf{b}_{x(i)} = \mathbf{b}_{y(i)} = \mathbf{b}_{\theta(i)} = \mathbf{0} \ \forall i$
        $\mathbf{b}_x = \mathbf{b}_x \ + \ \mathbf{b}_{x_{i,j}} \forall (i,j)$ *loop closing pairs*
        $\mathbf{b}_y = \mathbf{b}_y \ + \ \mathbf{b}_{y_{i,j}} \forall (i,j)$ *loop closing pairs*
        $\mathbf{b}_\theta = \mathbf{b}_\theta \ + \ \mathbf{b}_{\theta_{i,j}} \forall (i,j)$ *loop closing pairs*
        solve $H \ \Delta\mathbf{x} \ = \ -\mathbf{b}_x$
        solve $H \ \Delta\mathbf{y} \ = \ -\mathbf{b}_y$
        solve $H \ \Delta\theta \ = \ -\mathbf{b}\theta$
        $\mathbf{x} \ = \ \mathbf{x} \ + \ \Delta\mathbf{x}$
        $\mathbf{y} \ = \ \mathbf{y} \ + \ \Delta\mathbf{y}$
        $\theta \ = \ \theta \ + \ \Delta\theta$
        $\mathbf{length}_i \ = \ \sqrt{(\mathbf{x}_{i+1} - \mathbf{x}_i)^2 + (\mathbf{y}_{i+1} - \mathbf{y}_i)^2} \ \forall i$
        $\mathbf{x}_{i+1} = \mathbf{x}_i \ + \ \mathbf{length}_i \times \cos \ \theta_i \ \forall i$
        $\mathbf{y}_{i+1} = \mathbf{y}_i \ + \ \mathbf{length}_i \times \sin \ \theta_i \ \forall i$

F — Feature Extraction

Scale Invariant Feature Transform

An algorithm that has been proven to be very robust in feature detection but also computationally heavy is the scale-invariant feature transform (SIFT) which is an algorithm that is robust in a way that the algorithm finds the same keypoints in different images with the same objects[38]. SIFT makes use of the Laplacians of Gaussians to determine a scale space. To find keypoints, the extrema is searched within an image by comparing the pixel values. If the pixel is larger or smaller than the pixels that lie around it and also then the 9 pixels in the same area in the next and previous scale in the scale space, the pixel is a candidate keypoint. The slope of the keypoints will also be checked to see if it is indeed a well recognizable keypoint. To estimate orientation a histogram is created with the derivatives of the local image after which the image is rotated around the found angle θ. The descriptors are histograms of the gradients of the neighbouring blocks of pixels. These descriptors can be matched to compare the position of the keypoints in different frames.

Speeded up robust features

A feature extraction technique similar to SIFT is the speeded up robust features algorithm (SURF)[12]. The filtering in SURF is done by integrating the gray-scale values around a point as a faster alternative to Gaussian. This is given by:

$$S(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j)$$

SURF uses the matrix of second order derivatives (Hessian matrix) for blob detection to detect potential keypoints and the determinant of the Hessian to detect local changes around the point of interest. Second order derivatives will provide the position of corners explicitly. The Hessian matrix is give by:

$$H(p, \sigma) = \begin{pmatrix} L_{xx}(p, \sigma) & L_{xy}(p, \sigma) \\ L_{yx}(p, \sigma) & L_{yy}(p, \sigma) \end{pmatrix}$$

The scaling of the images is determined by calculating an image pyramid for the image. The image pyramid contains copies of the image in different scales. This is done by calculating the changes around a point with different filter sizes for different sigma values and a square filter. With this method it is possible to match the distance between the features and determine the difference in scale between the two compared images.

Laser range scanner data

The SIFT and SURF feature extraction methods are known to be robust and precise. Feature extraction on image data seems more straightforward than feature extraction on data obtained

from a laser range scanner because of the very different data representation. By means of segmentation and shape recognition it is possible to extract lines, circles, and ellipses from data from a laser range scanner[49].

Segmentation needs to be done to distinguish whether each point found in a laser scan belongs to an earlier found feature. A possible method to do this is by looking at the euclidean distance between points that are at the beginning or end of lines. Segmentation is an important aspect of data association.

Index

Bibliography

[1] Altera Cyclone V device overview. URL https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf. (Cited on page 145.)

[2] Advanced Robotics Tutorial. URL https://msdn.microsoft.com/en-us/library/bb905452.aspx. (Cited on page 9.)

[3] OpenSLAM. URL https://www.openslam.org. (Cited on page 51.)

[4] Quartic Formula. URL http://planetmath.org/QuarticFormula. (Cited on page 86.)

[5] SLAM Benchmarking Datasets, 2009. URL http://kaspar.informatik.uni-freiburg.de/~slamEvaluation/datasets.php. (Cited on page 9.)

[6] Haskell, An advanced, purely functional programming language, 2014-2016. URL https://www.haskell.org/. (Cited on page 41.)

[7] S Anderson, F Klassner, P Lawhead, and M McNally. LMICSE: Lego Mindstorms in Computer Science Education, 2001. URL http://www.mcs.alma.edu/LMICSE/LabMaterials/AlgoComp/Lab3/occgrid.gif. [Online; accessed January 19, 2016]. (Cited on page 14.)

[8] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174–188, 2002. (Cited on page 160.)

[9] K Somani Arun, Thomas S Huang, and Steven D Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on pattern analysis and machine intelligence*, (5):698–700, 1987. (Cited on page 37.)

[10] C Baaij. CλaSH, From Haskell to Hardware. URL http://www.clash-lang.org/. (Cited on page 41.)

[11] Kirk Baker. Singular value decomposition tutorial. 2005. (Cited on pages 38 and 39.)

[12] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-Up Robust Features (SURF). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008. ISSN 1077-3142. doi: 10.1016/j.cviu.2007.09.014. URL http://dx.doi.org/10.1016/j.cviu.2007.09.014. (Cited on page 165.)

[13] Paul J Besl and Neil D McKay. Method for registration of 3-D shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992. (Cited on page 29.)

[14] Peter Biber and Wolfgang Straßer. The normal distributions transform: A new approach to laser scan matching. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2743–2748. IEEE, 2003. (Cited on page 151.)

[15] Olivier Cappé, Simon J Godsill, and Eric Moulines. An overview of existing methods and recent advances in sequential Monte Carlo. *Proceedings of the IEEE*, 95(5):899–924, 2007. (Cited on page 161.)

[16] Andrea Censi. An ICP variant using a point-to-line metric. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 19–25. IEEE, 2008. (Cited on pages 30, 34, 37, 86, and 113.)

[17] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992. (Cited on page 30.)

[18] Ingemar J Cox. Blanche-an experiment in guidance and navigation of an autonomous robot vehicle. *Robotics and Automation, IEEE Transactions on*, 7(2):193–204, 1991. (Cited on page 151.)

[19] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1322–1328. IEEE, 1999. (Cited on page 160.)

[20] Albert Diosi and Lindsay Kleeman. Laser scan matching in polar coordinates with application to SLAM. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 3317–3322. IEEE, 2005. (Cited on page 151.)

[21] Chitra Dorai, Gang Wang, Anil K Jain, and Carolyn Mercer. Registration and integration of multiple object views for 3D model construction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(1):83–89, 1998. (Cited on page 34.)

[22] Sébastien Druon, Marie-José Aldon, and André Crosnier. Color constrained icp for registration of large unstructured 3d color data sets. In *Information Acquisition, 2006 IEEE International Conference on*, pages 249–255. IEEE, 2006. (Cited on page 34.)

[23] David W Eggert, Adele Lorusso, and Robert B Fisher. Estimating 3-D rigid body transformations: a comparison of four major algorithms. *Machine Vision and Applications*, 9 (5-6):272–290, 1997. (Cited on page 37.)

[24] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420, 1970. (Cited on page 39.)

[25] G. Grisetti, R. Kuemmerle, C. Stachniss, and W. Burgard. A Tutorial on Graph-Based SLAM. *Intelligent Transportation Systems Magazine, IEEE*, 2(4):31–43, 2010. doi: 10.1109/MITS.2010.939925. (Cited on page 17.)

[26] Giorgio Grisetti, Gian Diego Tipaldi, Cyrill Stachniss, Wolfram Burgard, and Daniele Nardi. Fast and accurate SLAM with Rao–Blackwellized particle filters. *Robotics and Autonomous Systems*, 55(1):30–38, 2007. (Cited on page 161.)

[27] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, Udo Frese, and Christoph Hertzberg. *Hierarchical optimization on manifolds for online 2D and 3D mapping*, pages 273–278. 2010. ISBN 9781424450381. doi: 10.1109/ROBOT.2010.5509407. (Cited on page 26.)

[28] Rui Guo, Fengchi Sun, and Jing Yuan. ICP based on Polar Point Matching with application to Graph-SLAM. In *Mechatronics and Automation, 2009. ICMA 2009. International Conference on*, pages 1122–1127. IEEE, 2009. (Cited on pages 39 and 40.)

[29] JE Handschin. Monte Carlo techniques for prediction and filtering of non-linear stochastic processes. *Automatica*, 6(4):555–563, 1970. (Cited on page 160.)

[30] Berthold KP Horn. Closed-form solution of absolute orientation using unit quaternions. *JOSA A*, 4(4):629–642, 1987. (Cited on page 37.)

[31] Berthold KP Horn, Hugh M Hilden, and Shahriar Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *JOSA A*, 5(7):1127–1135, 1988. (Cited on pages 37 and 113.)

[32] A Hornung, K.M Wurm, M Bennewitz, C Stachniss, and W Burgard. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees, 2013. URL `https://octomap.github.io/freiburg_outdoor_big.png`. [Online; accessed February 11, 2016]. (Cited on page 13.)

[33] D Hähnel. Dataset: Intel Research Lab (Seattle). URL `http://kaspar.informatik.uni-freiburg.de/~slamEvaluation/datasets/images/intel_relations.png`. [Online; accessed February 11, 2016]. (Cited on page 13.)

[34] Margaret E Jefferies and Wai K Yeap. Robot and Cognitive Approaches to Spatial Mapping. In *Robotics and Cognitive Approaches to Spatial Mapping*, pages 1–5. Springer, 2007. (Cited on pages 9 and 51.)

[35] Henrik Kretzschmar, Cyrill Stachniss, and Giorgio Grisetti. Efficient information-theoretic graph pruning for graph-based SLAM with laser range finders. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 865–871. IEEE, 2011. (Cited on pages 26 and 55.)

[36] Dmitriy Leykekhman. Linear Least Squares using SVD Decomposition, Fall 2008. (Cited on page 39.)

[37] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. 2004. (Cited on pages 37, 38, 39, 86, and 113.)

[38] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL `http://dx.doi.org/10.1023/B:VISI.0000029664.99615.94`. (Cited on pages 113 and 165.)

[39] Feng Lu and Evangelos Milios. Robot pose estimation in unknown environments by matching 2D range scans. *Journal of Intelligent and Robotic Systems*, 18(3):249–275, 1997. (Cited on page 30.)

[40] Takeshi Masuda, Katsuhiko Sakaue, and Naokazu Yokoya. Registration and integration of multiple range images for 3-D model construction. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 1, pages 879–883. IEEE, 1996. (Cited on page 34.)

[41] Javier Minguez, Luis Montesano, and Florent Lamiraux. Metric-based iterative closest point scan matching for sensor displacement estimation. *Robotics, IEEE Transactions on*, 22(5):1047–1054, 2006. (Cited on page 28.)

[42] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada, 2002. AAAI. (Cited on page 162.)

[43] Stuart F Obermann and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on computers*, 46(8):833–854, 1997. (Cited on page 99.)

[44] Edwin B. Olson. *Robust and Efficient Robotic Mapping*. PhD thesis, Cambridge, MA, USA, 2008. AAI0821013. (Cited on page 151.)

[45] Edwin B Olson. Real-time correlative scan matching. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 4387–4393. IEEE, 2009. (Cited on page 28.)

[46] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Oxford, UK, 2000. ISBN 0-19-512583-5. (Cited on pages 98 and 101.)

[47] P Piniés. CI-Graph SLAM execution on Victoria Park data set, 2007. URL `http://webdiis.unizar.es/~ppinies/research_archivos/map_victoria_google.jpg`. [Online; accessed January 27, 2016]. (Cited on page 16.)

[48] François Pomerleau, Francis Colas, François Ferland, and François Michaud. Relative motion threshold for rejection in ICP registration. In *Field and Service Robotics*, pages 229–238. Springer, 2010. (Cited on page 34.)

[49] Cristiano Premebida and Urbano Nunes. Segmentation and geometric primitives extraction from 2d laser range data for mobile robot applications. *Robotica*, 2005:17–25, 2005. (Cited on page 166.)

[50] Kari Pulli. Multiview registration for large data sets. In *3-D Digital Imaging and Modeling, 1999. Proceedings. Second International Conference on*, pages 160–168. IEEE, 1999. (Cited on page 34.)

[51] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*, pages 145–152. IEEE, 2001. (Cited on pages 28, 29, 30, 34, and 37.)

[52] Aleksandr Segal, Dirk Haehnel, and Sebastian Thrun. Generalized-ICP. In *Robotics: Science and Systems*, 2009. (Cited on page 28.)

[53] Rys Sommerfeldt. Origin of Quake 3's Fast invSqrt(), 2006. URL `https://www.beyond3d.com/content/articles/8/`. (Cited on page 98.)

[54] C. Stachniss. *Exploration and Mapping with Mobile Robots*. PhD thesis, University of Freiburg, Department of Computer Science, April 2006. (Cited on page 149.)

[55] Cyrill Stachniss. Robot Mapping, SLAM course. 2013. (Cited on pages 9, 147, 153, 157, 158, and 159.)

[56] Monika Thakur. Samsung's robotic vacuum cleaner comes with laser point technology, 2014. URL `http://www.homecrux.com/wp-content/uploads/2014/08/Samsung-Robotic-Vacuum-Cleaner-VR9000H_1.jpg`. [Online; accessed November 9, 2016]. (Cited on page 7.)

[57] Sebastian Thrun, Yufeng Liu, Daphne Koller, Andrew Y Ng, Zoubin Ghahramani, and Hugh Durrant-Whyte. Simultaneous localization and mapping with sparse extended information filters. *The International Journal of Robotics Research*, 23(7-8):693–716, 2004. (Cited on page 159.)

[58] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005. (Cited on pages 9, 14, 147, 149, 154, 156, and 157.)

[59] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318. ACM, 1994. (Cited on page 34.)

[60] NASA/JPL/Cornell University. An artist's concept portrays a NASA Mars Exploration Rover on the surface of Mars. Two rovers were launched in 2003 and arrived at sites on Mars in January 2004. Each rover was built to have the mobility and toolkit for functioning as a robotic geologist., 2003. URL `http://photojournal.jpl.nasa.gov/jpeg/PIA04413.jpg`. [Online; accessed November 9, 2016]. (Cited on page 7.)

[61] Eric Wan, Ronell Van Der Merwe, et al. The unscented Kalman filter for nonlinear estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158. IEEE, 2000. (Cited on pages 157 and 158.)

[62] Rinse Wester. *A transformation-based approach to hardware design using higher-order functions*. PhD thesis, UTwente, 2015. (Cited on pages 160 and 161.)

[63] Lei ZhangO. Weighted Point-to-Line ICP for 2D Laser Scan Based SLAM Application. (Cited on page 30.)