CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Systematic Evaluation of Autonomous Parking Modules Using Constrained Hardware

Master's thesis in Software engineering and technology

Nils Gangby
David Johansson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Systematic Evaluation of Autonomous Parking Modules Using Constrained Hardware

Nils Gangby
David Johansson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Systematic Evaluation of Autonomous Parking Modules Using Constrained Hardware

Nils Gangby
David Johansson

Supervisor: Federico Giaimo, Department of Computer Science and Engineering
Advisor: Olle Norelius, Infotiv AB
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Systematic Evaluation of Autonomous Parking Modules Using Constrained Hardware

Nils Gangby
David Johansson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In modern vehicles, autonomous features are highly sought after. However, the complexity of the problem domain and the security aspects make the software expensive to develop. Therefore, autonomous features are available only in the most exclusive vehicles. If proper software engineering procedures can be applied in the domain of autonomous driving, the cost of development can be reduced, increasing the number of autonomous features on the market.

This thesis aims to research how systematic evaluation can be performed on a series of software modules. The modules were developed as a proof of concept that it is possible to achieve specific AD features using fewer sensors and cheaper hardware than what currently is used in the automotive industry.

The development and testing are done working in an agile manner, following the design science framework. All the separate software modules are systematically evaluated individually to confirm their viability and compared to similar algorithms to decide which ones would be the best fit for the project. This is done mainly through experimental studies in a controlled environment.

The results of the systematic evaluation show that the implemented methods for parking space detection, pathfinding, path following, and parking algorithm are viable and can accurately detect and locate a parking space. The pathfinding algorithm can then generate a valid path to a desired starting position which the path follower can use to relocate the platform to that position with adequate accuracy. However, the choice of using BreezySLAM due to the constrained hardware proved to be a problem because of a large amount of spatial drift in the coordinates.

Even though the specific testing methods used in this thesis are tailored to test the specific types of algorithms used in the project, the general methodology of how the tests were designed and the practice of utilising systematic evaluation throughout the development of an artifact could potentially be generalised and utilised in any, or most software engineering project.

Keywords: Systematic Evaluation, Design Science, Requirements Engineering, Image processing, SLAM, Autonomous Platform, Pathfinding, Path Following

Acknowledgements

We would like to thank first our supervisor at Chalmers, Federico Giaimo for his invaluable help and guidance throughout the project. Second we would like to thank all the people working at Infotiv for letting us doing this project at their company and for all their help. We would especially like to thank Olle Norelius as our technical supervisor from Infotiv for all his help. Lastly, we would like to thank our examiner Christian Berger for trying to guide us through the jungle of SE academic guidelines.

Nils Gangby and David Johansson

Contents

Nomenclature

A*	A-star Algorithm
AD	Autonomous Driving
ADAS	Advanced Driver Assistance System
APP	Adaptive Pure Pursuit
BF	Brute Force
BFS	Breadth-First-Search
BMS	Battery Management System
BRISK	Binary Robust Invariant Scalable Keypoints
CEM	Central Electronic Module
CI	Continuous Integration
ECUs	Electronic Control Units
IPAS	Intelligent Parking Assist System
LiDAR	Light Detection and Ranging
OpenCV	Open Computer Vision
SAE	Society of Automotive Engineers
SE	Software Engineering
SIFT	Scale Invariant Feature Transform
SLAM	Simultaneous Localization And Mapping
SURF	Speeded up robust features
SWEBOK	Software Engineering Body of Knowledge
TEM	Telematics Engineering Module
VCU	Vehicle Control Unit

List of Figures

List of Tables

1
Introduction

Some *software engineering* (SE) processes have been around since the 1960s and have throughout the years radically changed the way software is developed [3]. These processes were in 2004 compiled into an IEEE-standard called *Software Engineering Body of Knowledge* (SWEBOK) [4]. SWEBOK contains 15 distinct areas of knowledge for an SE engineer to consider whilst in a project. These areas and processes exist to counteract the fact that developing good and robust software is hard. Before the development phase begins, the goals of the development must be decided. This can be achieved by eliciting needs from stakeholders and to create a rigorous requirements specification. When software is in the development phase, many challenges erupt, such as managing quality characteristics and reducing technical debt. Traditionally, the software is to be tested after development, however recently a more agile approach has become popular. Instead of testing when the software is deemed to be done, the testing can be done throughout the development process with the help of methodologies such as agile and continuous integration. Using these methods, technical debt can be reduced, and code quality standards can be improved [5]. Furthermore, systematically evaluating the decisions and code sections throughout the process can improve the overall quality, code quality, and help motivate the decisions made. However, every domain in which the development process takes place is unique, and the software engineering procedures may impact the end results differently between different domains.

Many of SWEBOK's processes can be applied to the area of *Autonomous Driving* (AD). AD is an area that is evolving and is estimated to evolve even more in the coming years [6]. AD is generally conceived as vehicles driving without external manipulation from the driver, however, it also entails the vehicle performing specific functions autonomously. One such example is parking by itself. Parking is an issue for a significant amount of people since they feel afraid of the parking manoeuvres [7].

Intelligent Parking Assist System (IPAS) was introduced in 1999 by Toyota and was the first parking assist system on the market [8]. Since then, the feature has evolved and it is now in every modern car. IPAS helps the driver park by using cameras and sonar sensors to inform the driver when the vehicle is about to hit an object. The system warns the user but does not control the vehicle in any way, nor does it perform the parking itself. To achieve full autonomy, the vehicle will have to perform the parking entirely on its own. Autonomous parking is also proposed as part of a solution in order to achieve level four AD as proposed by *Society of Automotive Engineers* (SAE) which is a global standards developing organization for the automotive industry [9].

1.1 Motivation

AD is, as stated before, an area that has been researched throughout the years with many different methods and research questions. *Suppé et al.* [10] and *Tripathi et al.* [11] tried to test a semi-autonomous valet service using an AD vehicle. Instead of focusing on the implementation and systematic evaluation of the system, they chose to focus on the human computer interaction and how people would respond to such a system. *Han et al.* [12] focuses solely on optimizing the calculations needed to process a 3D-LiDAR output for localization. *Paromtchik et al.* [13], *Jiang et al.* [14] and *Dhivya et al.* [15] are all also trying to create an autonomous parking algorithm by utilizing arrays of ultrasonic sensors to map their surroundings and then manually drive the vehicle to an optimal starting position next to the parking spot before initializing their parking algorithms.

It is estimated that technical debt costs, on average, 1 million dollars for each business application [16]. The benefit of embracing SWEBOK's areas of knowledge to this unique domain is that the cost of developing and evaluating autonomous functionality in cars hopefully can be made cheaper to further the acceleration of the technology's growth on the market. As can be seen in the papers above, there is research on the subject of AD in general, but none of the papers has focused on the utilization of established software engineering procedures and what benefits that can produce. Neither have they aimed to create and evaluate a fully autonomous parking system by conducting both parking space detection and the execution of the parking itself on any type of constrained hardware. Instead they use a variety of sensors, most commonly ultrasonic or radar sensors, and utilize sensor fusion to achieve their results. This same pattern can be observed in other papers researched as well.

The concept of systematic evaluation has been used in multiple research papers. A study made by *B.M Williamson et al.* used systematic evaluation when testing different configurations of arrays for an application based on Simultaneous Localisation and Mapping (SLAM) [17]. The study utilized a camera-based SLAM with multiple cameras. The systematic evaluation was used in a different context than what this thesis intends to do, however their systematic evaluation still proved useful. Through their systematic evaluation, they discovered several new interesting features, such as specific singular cameras performed worse than others and that there was little to no difference between a three-camera configuration and a four-camera configuration. Another study made by *Fan Zhang et al.* utilized systematic evaluation to evaluate their side-channel attack strategy [18]. They tested their entire framework through a series of comparative experiments where they used four metrics to measure the different configurations. They used systematic evaluation to guarantee completeness. Once again, their context differs from that of this thesis, however, the systematic evaluation helped them guarantee the completeness of their project.

If systematic evaluation can be used to validate an implemented design for a SLAM-based AD platform which only utilizes a 2D-LiDAR device and computationally constrained hardware for performing autonomous parking, this software engineering

process can be applied to other AD features as well. By applying known software engineering processes like these and systematically evaluating the features, the number of *Electronic Control Units* (ECUs) required would likely be lower or at least, they could be made cheaper because of the lower computational power required. This would likely result in a significantly cheaper product regarding hardware costs as well. When manufacturers aim to construct high volumes of cars for the general population cost is of most importance, which is why the premise of this thesis is relevant. If the cost of sensors and ECUs can be lowered, the systems can be implemented in cheaper cars and likely speed up the deployment of autonomous vehicles in traffic which can improve both traffic safety and the environmental impact of the transport sector [19].

1.1.1 Research Goal and Questions

The goal of this thesis is to research how systematic evaluation can be performed on a series of software modules. The modules were developed as a proof of concept that it is possible to achieve specific AD features, such as an autonomous parking algorithm, using fewer sensors and cheaper hardware than what currently is used in the automotive industry.
This is divided into two research questions:

1. *How can a systematic evaluation be performed on pure software modules such as BreezySLAM-based mapping and pathfinding algorithms on a constrained processing unit?*
2. *How can a systematic evaluation be performed on software modules such as path following and autonomous parking, which are designed and implemented on a small-scale vehicular platform?*

1.2 Scope

In this project, only one hardware configuration will be developed and tested on. This is done due to the work being conducted at Infotiv AB and with their current AD development platform, see Section 2.1.1 for details. Furthermore, only one SLAM algorithm, *BreezySLAM*, will be tested, this is done because of two reasons. Firstly, this is the algorithm that Infotiv aimed to examine and expected to be the best fit for the scenario. Second, the major rework needed to test other SLAM algorithms due to their differences in how they operate and their data outputs. Ideally, multiple algorithms and hardware configurations would be tested and evaluated against each other, but it is considered unfeasible in this project. By extension, only algorithms that can reasonably be run on the project-specific hardware will be considered.

When performing systematic evaluation, it is desirable to perform as exhaustive tests as possible. A high evaluation coverage can be difficult to achieve as there are usually multiple possibilities and factors to test. Due to the complexity and time needed to test this thesis will not test, for example, a high number of different maps when evaluating path finding or test an extremely high number of frames when detecting parking spaces. Furthermore, the systematic evaluations are not designed to be statistically significant in terms of sample sizes or give a definitive answer on e.g.,

what the best pathfinding algorithm in existence is, but instead give an indication if they would be a good fit for this specific project. This is because the limited scope of the project does not allow to conduct tests with big sample sizes. It is also not the goal of the thesis to determine which algorithms are best in every case, but instead systematically evaluate them to determine their viability within the project scope.

This thesis will not try to accomplish a method to implement a self-parking system that would function well in the real world and no such dynamics that are present in a real traffic scenario will be accounted for. The goal is, as previously stated, to research how systematic evaluation can be performed on a series of software modules that were developed as a proof of concept that it is possible to achieve specific AD features, such as an autonomous parking algorithm, using fewer sensors and cheaper hardware than what currently is used in the automotive industry. This could then be applied in settings where the safety aspects are not as crucial or where the environment is static, e.g., an autonomous robot in a robotized warehouse. However, the systematic evaluation does not demand the software modules to function in a real-life setting.

2
Background

In this chapter we will first describe the technical background to the project followed by a series of sections describing all the underlying technologies, libraries and algorithms that are utilized in the different software modules.

SWEBOK's areas of knowledge is divided into multiple phases of a project development timeline. The phases that will be mainly focused on in this thesis are the software requirements, design, construction, testing and quality. These practices are then used to facilitate an effective project with rigorous and systematic testing and evaluation of the different software components.

In this thesis, these practices are applied to a subject of study within the AD area. There are several different methods to create an autonomous vehicle and, in most cases, they involve some sort of sensor fusion technology which uses a combination of e.g., cameras, ultrasonic sensors etc. The approach taken in this thesis contains no sensor fusion of any kind and only utilizes one type of sensor. The general idea is to create a series of images of the environment using a 2D *Light detection and ranging* (LiDAR) sensor [20], see Section 2.1.2, and process it using *Open Computer Vision* (OpenCV) [21]. OpenCV is an open source image manipulation library which will be used to help clean up the images, analyse them and identify parking spaces using a feature detection algorithm called *Speeded up robust features* (SURF) in combination with a *brute force* (BF) feature matching algorithm, see Section 2.1.4. The identified parking space's coordinates are then fed together with the vehicle's current coordinates to a *pathfinding* algorithm, see Section 2.1.6, that finds the shortest path between the points without hitting any obstacles in the map. Using a path following algorithm called *Adaptive Pure Pursuit* (APP) [22], see Section 2.1.7, the vehicle follows the generated path and relocates itself to a good starting position for the parking manoeuvres to take place. The parking manoeuvres are then executed using a custom parking algorithm.

2.1 Subject of study

The goal is to research how systematic evaluation can be performed on a series of software modules. In the case of this thesis, the modules were not provided beforehand, but were developed for this specific project. The artifact that was provided from Infotiv was the Infotiv's AD development platform to use as a basis for further development. The modules to be systematically evaluated as the subject of study were parking space detection, pathfinding, path following and parking algorithm.

2.1.1 Infotiv Embedded Platform

The Infotiv Embedded Platform, see Picture, was the platform used throughout the project for implementation and testing of the software. The platform has three CAN-buses, five ECUs and several sensors, including sonar, radar, and camera, although these sensors was not used for this project but instead only a 2D RPLiDAR A2 was used. The LiDAR system is detailed in Section 2.1.2. The ECUs are:

- *Battery Management System* (BMS) on an STM32 F1, which is a 32-bit ARM Microcontroller developed by STMicroelectronics.
- *Vehicle Control Unit* (VCU) on a STM32 F1.
- *Central Electronic Module* (CEM) on a STM32 F1.
- *Telematics Engineering Module* (TEM) on a Raspberry pi 3b+.
- *Advanced Driver Assistance System* (ADAS) on a Raspberry Pi 4.

The platform is a four-wheels vehicle with four electric motors, one for each wheel, and the steering is done through skid steer by individually controlling each wheel's rotation.

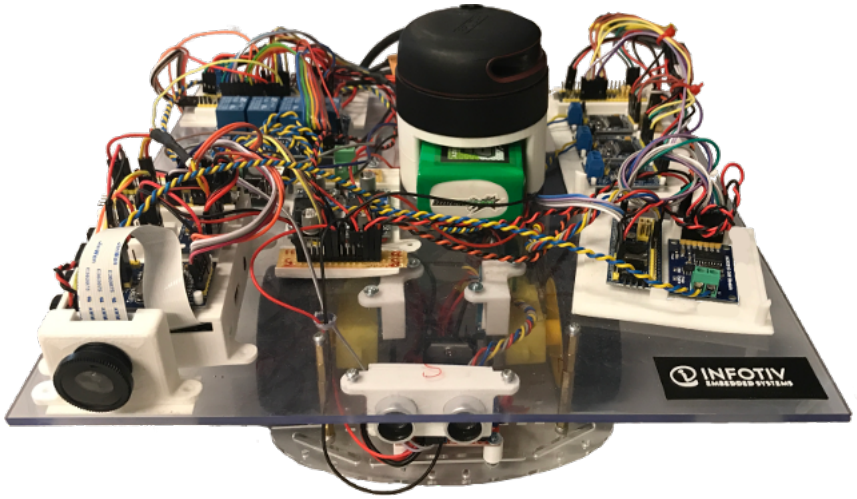


Figure 2.1: The Infotiv Embedded Platform.

2.1.2 LiDAR

LiDAR is an optical measuring instrument that utilizes a light pulse, commonly an IR laser. By measuring the time it takes for the pulse to return to the sensor, it determines the distance that pulse has travelled [23].

LiDAR's are used for many different applications in areas such as the construction industry and topological mapping. The version of LiDAR that is commonly used in the automotive industry is a 360-degree LiDAR that rotates at a constant rate while pulsing the light to create a 360-degree image of the environment. There are both 3D and 2D LiDAR's currently available on the market where the 3D LiDAR is what is commonly used for AD today. A 3D and 2D LiDAR both work by the same principles, but a 3D LiDAR can generate a 3D image of its surroundings while a 2D LiDAR is limited to only generating a 2D image. While the 3D LiDAR does provide a clear performance advantage over the 2D ones, the increase in cost and computational power required for calculations are vast. Therefore, this project

utilizes a 2D LiDAR model called RPLiDAR A2. The RPLiDAR A2 uses a 10Hz rotational speed and provides a maximum of 18m measurement radius with a listed measurement resolution of <1% of the distance between the LiDAR and the target surface [24].

2.1.3 BreezySLAM

BreezySLAM is the SLAM algorithm of choice. It was advocated by the company as they had an implementation integrated on the platform. BreezySLAM is an API partly developed by Simon D. Levy and it was specifically designed for fast, efficient, and easy handling of LiDAR data [25]. It utilizes Python C extensions for efficiency and as a result, the algorithm can run with similar speed in Python as in C. Python is generally seen as a slow language [26]. However, the speed of making a C call using Python is not going to make a significant difference in this case. On 64-bit platforms such as the Raspberry Pi used in this thesis, it also utilizes the NEON extensions for ARMv7, which allows the processor to receive the data for a given instruction from multiple registers in parallel, this further improves performance for video and image handling [27].

This API is used to process the LiDAR data into a continuously updating 2D-map of the platform's surroundings. These maps can then be further processed and analysed by other software, which are discussed in detail below.

2.1.4 OpenCV

Open Computer Vision (OpenCV) is an open source programming library designed for computer vision [21]. It was developed by Intel and provides a cross platform support for a huge variety of useful functions for computer vision such as feature detection, motion tracking and various machine learning based image manipulation functions. This library can be used to manipulate and detect features in the output images from the BreezySLAM algorithm to detect parking spots, obstacles, and other features.

Parking space detection can be performed in multiple ways. In this thesis, the SLAM-map is modified to make it easier to work with and then key features are detected and referenced to a pre-set image of a parking space. If enough features are matched, the coordinates in the SLAM-map corresponding to that of the reference image are classified as a parking space.

2.1.4.1 SLAM-map noise reduction

The original SLAM-map, as seen in Figure 2.2 is cluttered with noise from inaccuracies from the SLAM-algorithm as well as smaller objects that are not necessary for the detection of the parking space. This noise makes the map hard to effectively analyse so the original map is modified using several OpenCV functions to facilitate the feature detection and feature matching.

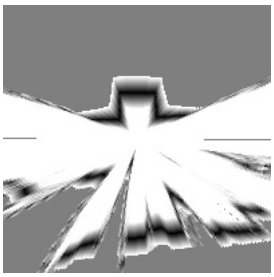


Figure 2.2: A caption of the unmodified SLAM-map.

The first step is to apply a threshold to the image. Thresholding in OpenCV is done by converting the image to grayscale before analysing every pixel value. If the value of the pixel is less than the threshold, the value is set to 0, resulting in a white colour. Otherwise, the value is set to the maximum value of 255 resulting in a black colour. The result of the threshold operation can be seen in Figure 2.3.

Figure 2.3: The map after threshold has been applied.

The thresholded image is less noisy than the original image, however there are still some irrelevant data in it. In the thresholded image, the parking space is clearly visible, and a human can easily distinguish the parking space from the noise. However, when working with raw data, it is important to remove as much unimportant information as possible without modifying key features. As can be seen in the picture, there are still several smaller objects that, for a human being, clearly is not a parking space, but a computer will have trouble distinguishing between them and the actual parking space. To remove these small objects, physically represented by chair and table legs, the OpenCV function dilation is used. The kernel is configured to dilute the image a certain distance. The result of the dilation can be seen in Figure 2.4.

Figure 2.4: The map after threshold and dilation has been applied.

The dilation facilitates the feature recognition process by further simplifying the image. However, the dilation process can sometimes remove too much and distort key features. For example, the left corner of the parking space is but a pixel wide in the dilated image, which could pose issues when performing the feature detection. In certain scenarios, the dilation may even make the parking space unrecognizable. Because of this, erosion is used to restore the potential damage that the dilation process has caused. Erosion works the same way as dilation, but in reverse. A kernel is convoluted with the image and based on the kernels settings the image's black pixels are eroded. The result of the erosion process can be seen in Figure 2.5.

Figure 2.5: The map after threshold, dilation and erosion has been applied.

Comparing the map with just thresholding applied and the resulting map after dilation and erosion it is a clear difference and the parking space can be easily distinguished from the rest of the image. However, even if the features are immediately apparent to humans, there is no guarantee that computers can make the same distinction. For example, if humans were to count the number of corners in the eroded image, they might come up with an answer around eight given that a corner is defined as a place where two edges meet. A computer on the other hand may come up with an entirely different answer, as technically, all separated pixels contain four corners. To make it easier for the computer to identify key features that are not pixeled corners, the image is blurred before the feature detection is performed. The blurred image can be seen in Figure 2.6.

Figure 2.6: The map after threshold, dilation, erosion, and blur has been applied.

2.1.5 Feature Detection and Feature Recognition

The parking space detection algorithm is built on BreezySLAM. The SLAM algorithm uses the LiDAR readings to create a map, where coordinates and direction of the vehicle can be retrieved. More computationally intensive implementations of SLAM can accurately remember the previous map and can deduce, based on its movement, where and how the object should look. BreezySLAM is a very lightweight implementation of SLAM which makes it great for running on constrained hardware. It does however have some drawbacks in that it is not as accurate as more computationally intensive versions of SLAM [28]. It works well enough to keep track of coordinates, but it often forgets how objects looked in previous frames. When the algorithm forgets how objects looks in previous frames, the merge of the previous frame with the current one becomes skewed and objects can look distorted. A downside of this is that the objects often look distorted when the LiDAR views them from slightly off angles.

Optimally, the parking space detection algorithm would only be run once; when the user would want to identify a parking space based on the SLAM map previously generated. This would allow for a better, more computationally heavy feature detection algorithm to be used. However, due to the limitations with the SLAM algorithm, this cannot be done. If the image of the parking spot gets distorted whenever the LiDAR is not perfectly aligned with the parking spot, the final image will have a distorted image of the parking spot. This is because in the case of the platform passing the parking spot, the LiDAR will continue moving away from the parking spot after achieving the perfect frame. This issue is solved by having the parking space detection algorithm run in real time. If a parking space is ever detected, the coordinates of the parking space are saved. Whenever the user wants to find the parking space, the coordinates for the parking space are already saved and can be displayed immediately.

To detect a parking space in a larger map, a separate image of a standardized parking space will be referenced to the map. The feature detection will be done using either OpenCV's *Speeded Up Robust Features* (SURF) function, *Scale-Invariant Feature Transform* (SIFT) or *Binary robust invariant scalable keypoints* (BRISK). For space detection, the algorithm is applied to both the reference image of a parking space and the actual map. The algorithm will detect key features for both the images separately. The key features are then passed to OpenCV's *Brute Force Matcher* (BF) which matches the key points from the two different images to each other. The result from the BF are destinations (pixel values) where the keypoints from the two images align as well as the transformation matrix that was used to achieve the results. If enough correct key points are matched to each other, it can be concluded that there is a parking space visible in the image.

2.1.6 Pathfinding / Path generation

To reach the research goal, the autonomous vehicle needs to be able to design a valid path through a map filled with obstacles. To achieve this, a pathfinding algorithm is used. Pathfinding is a well explored field of science with several famous algorithms such as *Dijkstra's algorithm for shortest path* (Dijkstra's). Although this algorithm works well and is guaranteed to provide the shortest path, it can be computationally intensive to do an exhaustive search in something a big as the LiDAR map images. It is possible to downscale the images to a smaller size but then there is a chance to lose crucial details in the image. However, its advantages of always returning the shortest path does make it an interesting candidate to test.

Nowadays, many more algorithms have been developed with different strengths and weaknesses. One of the most popular ones in use today is the *A-star* (A*) Algorithm [29]. The A* Algorithm is an informed search algorithm, meaning that the algorithm has knowledge of the maze and where its final goal is. The main difference between it and Dijkstra's algorithm is that A* uses a heuristic function that tries to calculate which nodes are the most likely to be in the shortest path to the target. This heuristic is often some sort of general distance measurement to the target, e.g., the straight line or the rectilinear distance to the target. This gives the algorithm a way of knowing if it is moving in the right general direction and therefore hopefully, does not have to search as many nodes as unguided algorithms, such as Dijkstra's.

Another commonly used algorithm for pathfinding is *Breadth-First-Search* (BFS). This algorithm takes a simpler approach to pathfinding. Instead of having a heuristic function that guides it, it simply looks at all nodes in one layer before moving down to the next layer. It then repeats the process until the goal is found and traces back via the shortest path from end to start.

There are, as mentioned, numerous algorithms to choose from that aren't described in this thesis. The choice to focus on Dijkstra's, A* and BFS was done after researching many different alternatives and comparing their strengths and weaknesses to try to find the best subset of algorithms that would most likely provide good results in the scope of this project. In this thesis, all three algorithms mentioned above were implemented in software and A* was implemented using two different Heuristic functions for a total of four algorithms. These are then evaluated for performance to determine which one is the best candidate for the autonomous parking platform. Evaluation methods are described in Section 4.3.

The way that a pathfinding algorithm works within the context of an image is that every pixel is considered a node in the graph. Each node has a distance of one to its nearest neighbour and the pathfinder simply traverses the image pixel by pixel. After the pathfinders have searched the image, the generated path contains every single pixel on the path. Since a pixel on the map corresponds to a very small distance in the actual environment, it is unnecessary to have such a high resolution for the path. Therefore, the path is first deconstructed into a set of coordinates consisting of only the start and end points and all the points where the angle of the path changes. Because the path has no real curvatures and only consist of straight lines and 45- or 90-degree corners, no information or resolution is lost by doing this. This subset of

the path is then injected with evenly spaced points on the map and any sharp angles are rounded out to generate the final path that the vehicle should try to follow.

2.1.7 Path following

When the path is found and generated by the pathfinding algorithms, the vehicle needs to be able to follow the given path. To achieve this, an algorithm called *Adaptive Pure Pursuit* (APP) was used. APP is a commonly used algorithm for path following within the fields of robotics [22]. The specific version of the algorithm used in this thesis was developed by *Team Dawgma* which is a team competing in *FIRST Robotics Competition* [30]. There are several other methods than the one that was chosen, such as Bakker et al. [31] which utilizes a robot with four wheels that can all be turned separately, or Kapitanyuk et al.[32] which uses the path to create a guiding vector field covering the entire map to control the robot. Although not explicitly stated in the paper, it is most likely not a viable option to run on a Raspberry Pi 4 due to its need to calculate a very high number of guiding vectors to cover the map. The APP algorithm was chosen after researching a variety of different path following algorithms due to its relative simplicity, good results and how well it matched the situation at hand. It is not very computationally intensive compared to many of the others and Team Dawgma's implementation [22] already uses the same type of skid steer drive the Infotiv Embedded Platform uses as well.

The APP controller is at its core simply directing the robot to travel from its current location to a *lookahead point* located on the path in a predetermined distance called the *lookahead distance* in an arc.

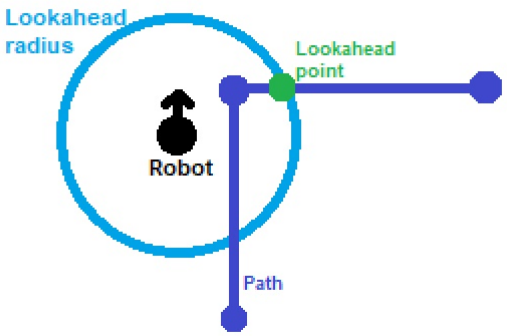


Figure 2.7: Locating the Lookahead point.

When the robot moves to a new location, the lookahead point will also move along the path and the robot is in a constant pursuit of that lookahead point, hence its name. The algorithm needs to know its current location, angle, and the path waypoints, all of which can be provided by the RPLiDAR fitted to the Infotiv Autonomous Platform. For each iteration of the algorithm it performs the following steps:

- Find the closest point on the path to the robot.

- Find the lookahead point on the path.
- Calculate an arc that intersects both its own position and the lookahead point.
- Calculate target left and right wheel speed.
- Utilize a proportional control loop to achieve target wheel speeds.

All these steps and the underlying math are described in detail in the technical paper by Team Dawgma [22]. The APP algorithm was first implemented on a laptop and simulated in software in conjunction with the pathfinding algorithm to be able to test their combined functionality and performance before implementing them on the platform.

2.2 Continuous Integration

Continuous integration (CI) is a practice that was introduced in 1991 by Grady Booch [33]. Lately the practice has received much attention and praise within the software community. CI requires that every time somebody commits any change, the entire application is built, and a comprehensive set of automated tests is run against it [34]. CI would assist the research by allowing for high levels of automation when it comes to tests and deployment. This automation can lead to saved time that can be better utilized elsewhere. Furthermore, CI can assist with code quality checks to set a base code standard, which in the end allows for less implementation specific deviations when it comes to systematically evaluate the implementations.

Although not strictly necessary to answer the research questions or reach the goal in this thesis, it is a powerful tool and strategy that can be helpful throughout the project. It is also very relevant for the field since many other companies and projects within the field of SE have already implemented CI because of it is benefits. To implement and follow CI, there are three prerequisites: version control, some form of automated build and the agreement of the team. For version control, GitHub will be used as that is what the company advocated. For CI there are loads of tools to aid the process. Most known are Jenkins, Travis CI, and Bamboo. Bamboo is made by Atlassian, the same company that is behind Bitbucket. Therefore, many of the functions that are provided by Bamboo works best in collaboration with Bitbucket [35]. Since the project intends to use GitHub, Bamboo might not be the best suitable software. Furthermore, Bamboo is not a free service. Because of these reasons, Bamboo was not the software of choice for this project. Both Travis and Jenkins however would be valid choices. Travis has easy integration with GitHub and is free for open source projects [36]. Jenkins is arguably the most popular choice for CI software on the market today [35]. It is free and completely open source with support for a wide variety of languages and version control services.

When comparing the features of Travis CI and Jenkins, many similarities of features occur. Travis CI supports more languages than Jenkins, however that is not relevant for this project as all the code will be written in Python, which is a programming language that Jenkins do support. Jenkins has more plugins than Travis, while Travis supports cloud-based hosting which Jenkins does not. Travis also has an arguably easier setup and interface than Jenkins. Weighing all the pros and cons

of both these tools it is decided that Jenkins will be used in this project. The pros of Travis CI such as support for more languages and cloud-based hosting does not provide any value in this project. Because of this, and the fact that Jenkins is a more used tool overall, which arguably implies more documentation and support online, Jenkins was the platform that was utilized in this project.

The Jenkins server was setup on a local Ubuntu machine and was installed using the Docker image provided by Jenkins. Jenkins allows for many plugins to be installed and automatically run together with the build. The ones that were integrated in this project GitHub branch source plugin, Violations plugin, and Cobertura. GitHub branch source plugin allows to make a new project based on the repository structure. Violations supports integration with Pep8 and allows to create reports of code standard followage with each build. All reports generated is shown in a graph with each build to provide an overview of the improvement with each build. Cobertura is used to create a code coverage report with each build, and in a similar way to Violations, provide statistics of the changes between the builds. Jenkins throughout the project helped with keeping all commits clean, following a set code standard and providing reports for each build. Even though CI did not necessarily provide any help with the systematic evaluation, it did help with developing the software in a standardized way, removing one factor of deviation from the evaluation.

2.3 Code Quality Assurance

When systematically evaluating software artifacts, it is important to know that the way the software was written does not influence the metrics collected in the other tests such as e.g. path length or performance in any significant way. Great software quality can help alleviate the strain of poorly written software artifacts. According to SWEBOK there are two main notions included in the term software quality [37]. One being how well the software complies with the functional requirements of the software. The other being how well the software complies with the non-functional requirements such as maintainability and robustness.

To facilitate the process of evaluating software quality there are several software tools available. Some alternatives include SonarQube, ReSharper and Black Duck. SonarQube is a software that allows developers to visualize their code and analyse their code quality by measures such as dependencies, code duplicates etc. ReSharper gives a very similar set of tools compared to SonarQube but it requires a paid license at $300/year, which is out of budget in this project. Black Duck is free to use and has many good features, but it lacks features such as automatic visualization, which provides a good overview of the code quality. For the purposes of analysing the quality, SonarQube was the best fit and ultimately the one that was used.

Since there were multiple developers working on the project simultaneously, there was an impending risk that the different code snippets will be written to different standards. To counteract this, there are multiple Python code standards available that tells the developer how to write the code. The most used one is called Pep8 and there is software that checks the code for violations to Pep8. One way of facilitating the process is to integrate a program called Pycodestyle into Jenkins and have

it check for Pep8 violations on every push to the version control software. Using Pycodestyle can help increase the consistency of the software and make sure that it is up to official coding standards.

The standard was followed by integrating Pycodestyle into Jenkins, causing it to check that the code is up to standards every time the version control gets a new commit. The first time that Pycodestyle was applied to the software more than 800 violations were detected. Some of them came from libraries and previously existing code, but many of the violations were caused by the developers. In the end of the project, the amount of violations had been significantly decreased and the code quality was higher.

Although these methods for code quality assurance is not providing answers to the research questions, they are an important foundation for the project in the same way that the requirement engineering and CI are. All these foundational methods are used to have a stable and efficient project which provides good quality code, measurable and quantifiable results in an efficient manner.

3
Related Work

A literature study was performed where several databases, including ACM, IEEE Xplore, Scopus, and Google Scholar, were searched for related works. For all of them, the search query ""SLAM" AND "parking"" was used to find relevant work. The query provides a lot of hits, but everything is not relevant for the purpose of this thesis. Some of them were similar to each other, in which case one was chosen. When considering papers as related works, only papers using some type of SLAM and/or LiDAR was deemed relevant. This, aside from inlcuding LiDAR-SLAM and 2D-LiDARs included papers using 3D-Lidars, camera based SLAM and different types of sensor fusion. Even though these specific technique combinations are not directly applicable or usable for the purpose of this thesis, they were relevant in order to understand and consider the limitations of the intended implementation. In cases where the techniques and sensors used in the papers did not match the techniques and sensors of this thesis, studies with environments that were deemed interesting was chosen. Furthermore, where similar implementations were made, the hardware used was considered in order to understand the restrictions of the intended hardware.

Several papers of interest were found. Some were using SLAM as a technique for localization where GPS signals are absent, such as parking houses and garages [38]. Others using SLAM in conjunction with other techniques to both identify available parking spots and autonomously park vehicles. The paper that was most similar to what this thesis is going to encompass is a research paper done by *Jie Song et al.* [39] released in the summer of 2019. The paper studied an automatic parking system (APS) that used short range LiDAR data and simultaneous localization to identify available parking slots. They compared the laser-based solution to ultrasonic based solutions and found that the laser-based solution achieved slightly better performance than the ultrasonic based solution, however it still suffered from the same main drawback in that it cannot detect parking spots unless other vehicles or obstacles are adjacent since it cannot see the parking slot markings in the ground. They developed an algorithm for parallel parking and tested their solution experimentally with good results. They used a MicroAutoBox as their vehicle control unit (VCU). The MicroAutoBox has a total of six processors, two NVIDIA Denver 2 Cores and four ARM Cortex-A57. [40]

Another interesting study made by *Gyubeom Im et al.* uses LiDAR together with around view monitor (AVM) cameras to improve loop closing and achieve accurate parking space detection [41]. They successfully created a SLAM implementation that, after testing, proved better than other, previously existing, SLAM algorithms. They identified the parking lines using the AVM camera in conjunction with the LiDAR and achieved an accuracy of up to 98%.

The first step of autonomously parking a vehicle is to identify the parking spot. The course of action taken in this thesis is to use a series of OpenCV functions to modify the map and detect potential parking spots. A common practice when identifying objects in OpenCV is to use a combination of feature detection and feature matching algorithms. The feature detection algorithms detect characteristics in two images, and the feature matching algorithm tries to match the features of the different images to each other. There are many different feature detection and feature recognition algorithms available through OpenCV and what combination is optimal can be hard to deduce. A study made by *F.K.Noble* [1] compared the algorithms individually and in combination with each other. The paper identified that many different algorithms and combinations existed and set out to answer the question "Which ones should be used". All algorithms were tested with the same benchmark image set of five images of a face taken from different angles. The main code in the tests was written using C++, which is a slight deviation from intentions of this thesis. However, since OpenCV mostly uses C calls internally, the difference between using Python and C++ should be minimal if the main computation comes from OpenCV functions. The paper concluded that BRISK and BF was the optimal combination. However, due to this thesis's specific setting, the test results and the graphs, as seen in figures 3.1, 3.2, 3.3 and 3.4 generated from the paper is of more importance. The graphs show a comparison between the different feature detection algorithms in terms of features detected and the computation time. Even though the paper concluded BRISK and BF to be the best combination, it can be seen in the graphs that other algorithm combinations, more suitable for this project, achieves similar results.

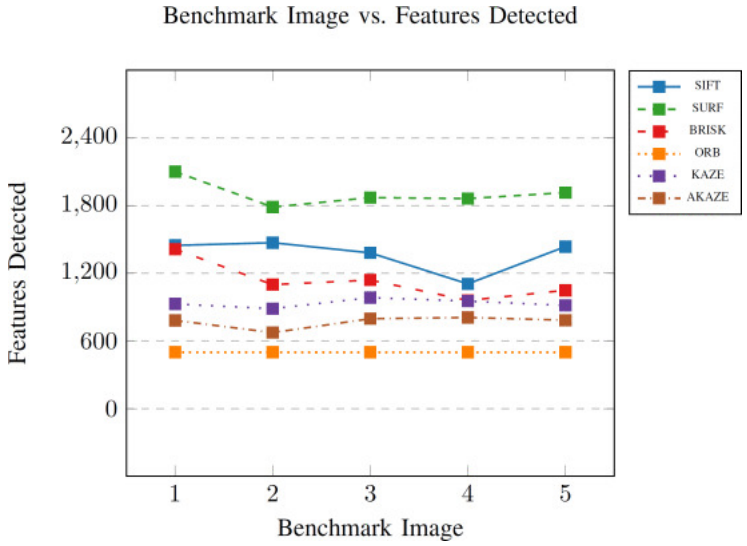


Figure 3.1: Benchmark image vs. features detected. Graph from *F.K.Noble [1]*

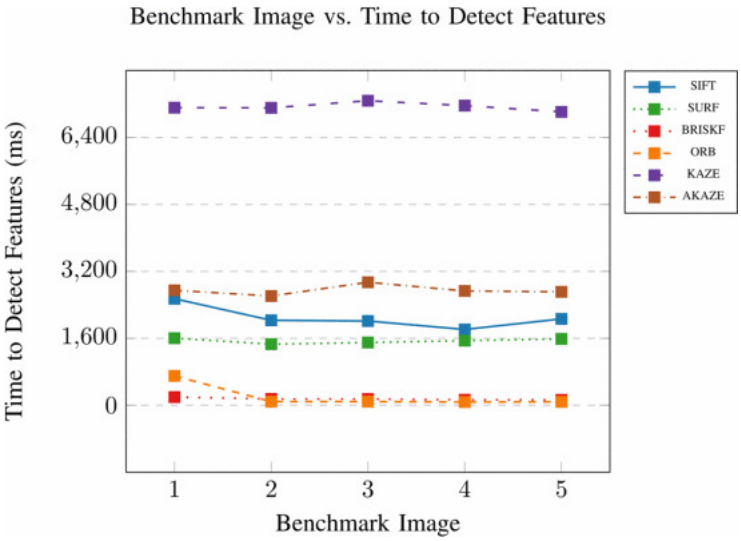


Figure 3.2: Benchmark image vs. time to detect features. Graph from *F.K.Noble [1]*

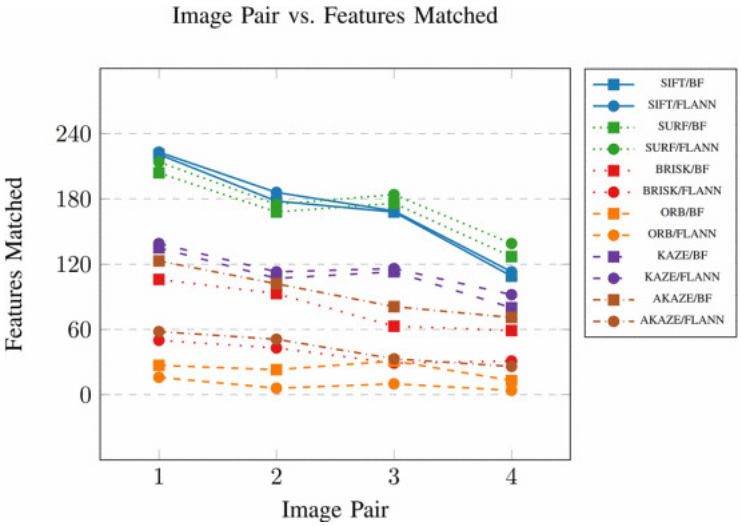


Figure 3.3: Image pair vs. features matched. Graph from *F.K.Noble [1]*

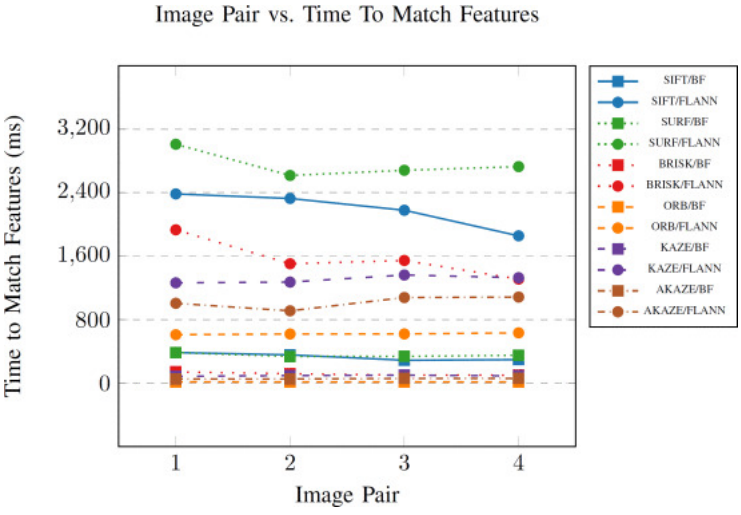


Figure 3.4: Image pair vs. time to match features. Graph from *F.K.Noble [1]*

4
Methodology

The methodology that was utilized throughout the project was divided into the different work areas and software modules that were to be evaluated. In this chapter, the general development methodologies are described followed by a separate section, Section 4.3, that describes all the methodology utilized for testing and evaluation of the developed modules.

4.1 Development methodology

The approach that was chosen for this project is the Design Science method [42]. This approach is inherently a problem-solving process and is a commonly used to create innovations and define technical capabilities of a product. The methodology can, according to *Hevner et al.*, be divided into seven distinct guidelines which are described in the table below [2].

Table 4.1: Table of guidelines [2].

#	Guideline	Description
1	Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
2	Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
3	Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
4	Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
5	Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
6	Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
7	Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Although Hevner et al. advises against mandatory or rote use of these guidelines, the goal is to follow them as closely as possible.
Wieringa further extends on the Design Science framework and divides it into two parts called “Design” and “Investigation”. The design part consists of designing the actual artifact to solve or improve on a problem within a given context. From this, valuable knowledge about the context and artifact itself will be gained and transferred to the investigation. The investigation part is where the team aims to answer the questions that might have risen during the development. From the investigation one might get new knowledge or design problem which in turn are brought over to the design part again. These two distinct parts of the framework are utilized in parallel during the development and affect each other continuously.

The aim was to be able to systematically evaluate the software and the artifact and the goal of the artifact was to be able to autonomously park itself using the limited hardware that was available on the platform. To answer the research questions of the thesis, several evaluation methodologies were needed. The first step was to come up with different metrics that can accurately show if the prototype works as intended. These metrics were decided upon through rigorous discussion with key stakeholders and include variables such as consistency, accuracy, performance etc. These same metrics were generally used on several of the software modules the project was divided into.

After the metrics were decided, different tests were designed for each of the different software modules and the final artifact to get a rigorous methodology on how to gather these metrics in a reliable and efficient manner. These tests were designed internally and in collaboration with key stakeholders at both Chalmers and Infotiv and by looking at how other papers have tested similar algorithms or features.

4.2 Requirements specification

The goal of this thesis was to research how systematic evaluation can be performed on a series of software modules. The modules were developed as a proof of concept that it is possible to achieve specific AD features, such as an autonomous parking algorithm, using fewer sensors and cheaper hardware than what currently is used in the automotive industry. For this to be achievable, an artifact in the form of a working AD-platform needs to be developed and the artifact needs to be able to meet certain needs and perform in the scope of the thesis. Because of this, clear requirements were needed as a baseline to be evaluated against. The process of requirements engineering is a systematic approach to reduce the likelihood to develop an incorrect solution to a problem [43]. This is where the foundation for the artifact is laid out and it must be detailed enough so that everyone involved with the project has a clear image of the system and how it is supposed to function.
The first thing to consider when compiling a set of requirements is to identify the stakeholders of the project. The stakeholders are any person that has a stake in the project. These stakeholders could be for example the company that funds the project, the school, the people conducting the project etc. Identification of stakeholders is important to specify from the start so that e.g., the elicitation then allows as many stakeholders as possible to get their say. For this project, the stakehold-

ers were identified by discussion within the team and plotting them in a quadrant divided graph with Influence on the Y-axis and Interest on the X-axis, see Figure 4.1.

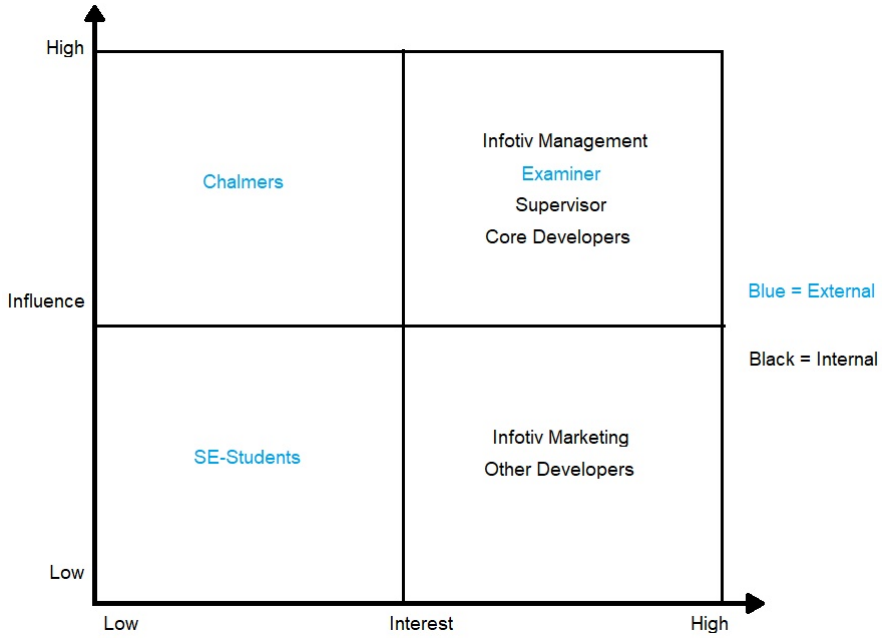


Figure 4.1: Identified stakeholders in the project.

- **Chalmers:** Chalmers as an academic institution. Regarded as having a high grade of influence over the project but low interest in this individual thesis.
- **SE-Students:** SE-students as a whole that might benefit from the work done in this project. Regarded as having a low interest and low influence.
- **Infotiv Management:** The management at the company this project is conducted at. Regarded as having a high influence and interest surrounding the project.
- **Examiner:** The academic examiner from Chalmers. Regarded as having high influence and interest surrounding the project.
- **Supervisor:** The academic supervisor from Chalmers. Regarded as having influence and interest surrounding the project.
- **Core Developers:** The students conducting this project. Regarded as having influence and interest surrounding the project.
- **Infotiv Marketing:** The marketing department at the company this project is conducted at. Regarded as having a low influence but high interest in the project.
- **Other Developers:** Other developers, primarily at Infotiv that can use this project as a foundation for future projects. Regarded as having a low influence but high interest in the project.

This stakeholder identification and classification was done to get a better understanding of what stakeholders' needs to prioritize in the requirements.

Requirement elicitation was conducted to gather data on what the stakeholders in the project felt was needed. This was done mainly through semi-structured interviews and brainstorming with stakeholders at Infotiv. Through this elicitation, an

understanding for the problem at hand was built and the functional requirements could then be written down. In this project, the requirement specification was made as a set of user tasks, this was chosen because of their easy-to-follow nature and their ease of mapping the requirements to things such as business goals.

Once all requirements were written, they had to be prioritized. This was done using a 100-dollar test [44] with developers and a bubble sort ranking [45] with Infotiv management. These scores were then combined and resulted in a final table of the importance of the requirements. The prioritization is done both to get an understanding if some requirements are redundant or completely unwanted from the stakeholders, but also, it helps with time management in case of deviations from the original time plan. If a problem occurs and something takes more time than expected, it is important to know what to focus the work on and if something can be cut from the original plan of requirements.

The final part of the requirement specification is to validate the requirement. Validation is the process of verifying that the requirements that are written satisfies the stakeholders' needs and requests. The validation process can be done either by inspections, formal proofs, or tests. The methodology chosen for this project was mainly inspection by perspective-based reading. This was chosen mainly because there are quite a few key stakeholders and a compromise that satisfies all parties needs to be the goal. So, by using perspective-based reading from both Chalmers' and Infotiv's point of view, these two main stakeholder perspectives could be evaluated. The other method used was to discuss the requirements and the project with both the supervisor from Chalmers and both developers and management from Infotiv. The resulting requirements document is appended to this document as Appendix A.

4.3 Testing and Evaluation

In every scientific project and especially this one with regards to our Research Goals and Questions it is important to have a planned structured methodology for testing and evaluating the ideas that are being worked on.
Evaluation of many different technologies in many different phases were designed, so each technology needed to be evaluated individually before being integrated into a complete package. The different testing and evaluation phases and their methods of testing are detailed one by one below.

4.3.1 Pathfinding

To answer Research Question 1, the different possible pathfinding algorithms needed to be tested and evaluated. These algorithms were evaluated by testing a set of 10 different mazes and real LiDAR maps. These ten images were chosen as a sample of scenarios, five of them was actual LiDAR images from different environments within the office, five of them were of mazes. The reasoning behind these numbers were mostly because of the limitation that the controlled environment, in this case Infotiv's office spaces, does not contain very many unique rooms that could be used to

create real LiDAR maps. It would also be time consuming to create a map representative of our controlled environment since the rooms would have to be emptied from any objects to reduce unwanted noise in the map. Additionally, if one were to create LiDAR maps in e.g., 20 different rooms of roughly the same shape and size, the results would not really differ since the algorithms cannot see a difference between one squared room and another. Optimally one would want to conduct experiments with a much bigger sample size or even try to create a fully automated test scenario where an almost exhaustive search is conducted through a maze generator that creates millions of mazes. Even though it would be theoretically possible to have e.g., five LiDAR maps and five million automatically generated mazes as a substitute, it was unfeasible and would not yield much valuable insight because of the reasons listed below:

- The project timeline did not allow to create an entirely automated maze generator and run it all the time.
- The pathfinding algorithm took a few seconds each to compute a path. To run an almost exhaustive set of mazes on both algorithms would take days to compute each time any change is done.
- Mazes are not necessarily the best depiction of reality. An algorithm could perform very well in a very complex maze but much worse in an actual LiDAR image. For this reason, both mazes and actual LiDAR maps were present in the test. It shows both real-world expected performance but also some worst-case environment data from the mazes.
- Given several images with roughly the same maze complexity, the algorithms would perform similarly from run to run and most likely, no valuable insights can be gathered from a few milliseconds' differences.

The set of ten pictures was then fed to four different pathfinding algorithms: BFS, A* with Euclidean distance as its heuristic function, A* with Manhattan distance as its heuristic function, and Dijkstra's.
These different algorithms were evaluated against each other and the following metrics were evaluated:

- **Accuracy:** Accuracy measured by generated path length. Dijkstra's is mathematically guaranteed to provide shortest path and was used as baseline.
- **Efficiency:** Number of seconds it takes for the algorithms from start to finish. These numbers are then compared to each other
- **Convergence:** Given a small variation in start or end positions on a given map, does the generated path length differ much or does it remain stable?
- **Robustness:** How does the algorithms handle scenarios where there is no valid path or where e.g., the only valid path goes through a 1-pixel wide passage?

Setup:

- Ten different images of mazes and LiDAR maps, five each.
- Four different pathfinding algorithms are implemented (BFS, A* Euclidean, A* Manhattan and Dijkstra's).

- All algorithms running on the same hardware with only the Operating System running in the background
- For each image, a start and endpoint are manually chosen.
- All ten images are fed in serial to the pathfinding algorithms for a total of 10 runs per algorithm.
- From these runs, the time-to-compute and path length are gathered. The data is then evaluated by comparing time-to-compute and path lengths to each other.

To test for convergence, the test setup had to be slightly modified because variation between runs on the same maps was desired. The tests were conducted with the same basic setup as the test above, but with two key differences:

- All algorithms are run with the five LiDAR images ten times each. Every run, the starting position was slightly altered by randomly generating a start position that is +/-10 pixels from a centre in both X and Y directions. A total of 50 runs per algorithm will be performed.
- All path lengths are then recorded and evaluated for standard deviation.

See the flow chart in Figure 4.2 for an overview of the exact testing procedures.

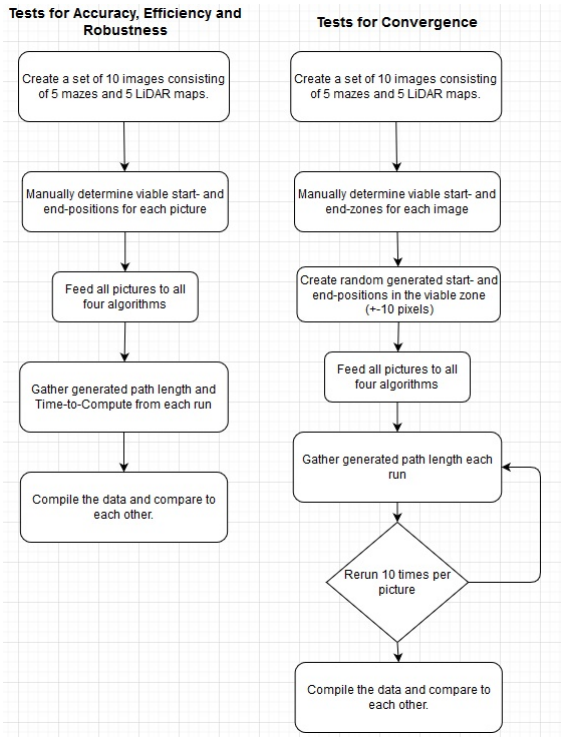


Figure 4.2: Flow-chart for the pathfinder tests

Note that in the testing for convergence, only the 5 LiDAR maps will be tested on. This is because of the nature of a maze where a slight variation in position might lead to a huge variation in path length even though the algorithms work perfectly and generate the shortest possible path. In a LiDAR map, this is usually not the case

since they are more open and any variation will be either due to the actual distance between the start and end point being longer, or the algorithm not behaving as expected.

4.3.2 Path Following

To answer Research Question 1, the path following algorithm also needed to be evaluated to verify its functionality and fit regarding the requirements of the artifact. The path following algorithm was not evaluated against any other algorithms in this thesis since no other algorithms were implemented due to time constraints. Due to the reasons mentioned in Section 2.1.7, the APP was deemed to likely be the best choice for this implementation.

To evaluate the path following, the algorithm was implemented first on a laptop and a simple visualization window, see Figure 4.3, was used to track how the robot followed the path. This was useful because otherwise it would be difficult to verify that the algorithm did not e.g., cut corners or jump between parallel paths that might occur in the path generation.

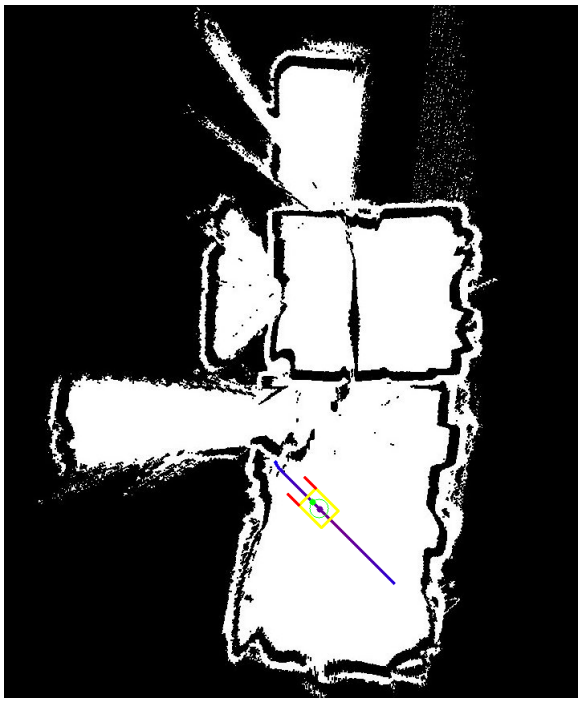


Figure 4.3: Visualization window for testing. Note that the robot size is exaggerated for visibility purposes.

The program was then fed four different paths generated by the pathfinding algorithm which in turn was given actual LiDAR maps to create a realistic setting. The algorithm then executed and the mean, minimum and maximum orthogonal distances to the path were collected and evaluated. If they were deemed as adequate from the start, a few different *lookahead distances* were tested and compared in the same manners as the algorithm itself to get the optimal performance for this specific use case. What to consider adequate was determined by the resolution of the image. If the pixel deviation corresponds to e.g., two centimetres in real life, it probably

would not be an issue. However, if the deviation corresponds to 50 centimetres, it would be an issue in many cases. Because of this, the accepted values were only decided upon once a standard map quality was set.

4.3.3 OpenCV

To answer research question 1 and 2, the different algorithms that were used in the OpenCV library had to be evaluated. All evaluation tests were run on the same hardware that was used in the final product i.e., a Raspberry Pi 4 and an RPLiDAR A2. Even though it would be theoretically possible to generate an infinite number of unique frames to test the different values on it would not be feasible. The processing power of the Raspberry Pi 4 is severely limited, so given a very large number of frames, the processing time would be too large to fit within the scope of the project. Due to this, the number of frames tested had to be limited. Seeing as the tests were performed in a controlled environment while the platform was stationary, there is only so many unique frames that could occur given that the environment is controlled to a sufficient degree. Given the limited amounts of unique frames and the time constraints linked to the one processing units, 50 frames for each configuration were tested in real time with a LiDAR map. Once the best algorithm was decided, another more extensive test was conducted where the optimal parameters for the specific arguments of that algorithm were determined. All tests were run with a thresholded map as argument and the platform placed 25 centimetres away from the delimiting vehicles.

SURF, SIFT and BRISK are three feature detection algorithms. According to F.K.Noble, BRISK in combination with BF is the optimal combination for maximizing features found and matched. However, SURF in combination with BF also achieved great results and was therefore tested as well. Furthermore, the predecessor of SURF, SIFT was also in the test as it could simulate the power of a more computationally heavy algorithm. SURF is generally accepted as a better algorithm than SIFT, but there are special cases where SIFT outperforms SURF, for example when there is no blur and there is no rotation in the map.

The factors that were tested were feature detection algorithm, blur on map and blur on reference image. The feature detection algorithm has three levels, SURF, SIFT and BRISK. When the blur value is increased, the resulting image will be more blurred. Because of this, even though the amount of levels is theoretically infinite, the resulting image after a threshold was so blurred that the images are indistinguishable from each other. Furthermore, the OpenCV function only takes odd numbers as arguments, so (1, 3, 5) is valid, whereas (2, 4, 6) is not. This further decreased the amount of levels that had to be tested.

A full factorial design was chosen since there are rather few factors with limited levels. There are three different levels of blur for both cases where blur is applied, on the map and on the reference parking space. The three different levels that were used are 1, 5 and 9, where 1 means that the image is less blurred and 9 means that it is significantly blurred. The reason not all odd numbers up until a specific

point were tested is that this would drastically increase the amount of tests that had to be made and would likely not have changed the outcome of which algorithm is deemed most suitable. When the initial test generated sufficient results to deduce the most optimal algorithm, another test was conducted to perfect the values of blur.

To mitigate differences in the tests, all tests were run in a controlled environment. The controlled environment consisted of a room approximately three by three metres wide. All objects not necessary for the experiment were removed or placed out of sight for the LiDAR. The platform was placed 2.5 metres away from the parking space, which was represented by three cardboard boxes of approximately 100 by 40cm. The boxes were tall enough to be in line of sight of the LiDAR.

A fractional factorial design could have been done if additional factors were to be added, such as threshold, dilation, and erosion. However, these factors were omitted as they were considered crucial for any matches to be discovered and their levels did not affect the outcome significantly. Furthermore, omitting these factors helped delimit the testing. To ensure as little change in the environment as possible, the room in which the test was performed was locked down to prevent unnecessary disturbance from people interfering with the LiDAR image. The platform was stationary throughout the tests and the platform was never rotated. This makes for a fair testing environment.

The results of the experiment will be evaluated using the following criteria:

- **Quality:** the number of correct features detected and matched.
- **Efficacy:** the time it takes to compute.
- **Utility:** how well the implementation works in our specific case.

4.3.4 Parking Algorithm

To answer Research Question 2 and verifying the parking algorithm in hardware, the parking algorithm needs to be evaluated by placing the vehicle in a starting position and having the platform autonomously park. The algorithm is tested for the following metrics:

- **Accuracy:** What is the difference between the platform and the desired location once the algorithm is finished?
- **Angular accuracy:** How many degrees misalignment does the platform have in relation to parallel with the parking space once the algorithm is finished?
- **Safety:** Does the algorithm finish without the platform bumping into objects that it should not?
- **Efficiency:** How much time passes from start to finish?

Example setup:

- Platform is placed in starting position, a set distance from the aligned vehicle.
- Algorithm decides optimal end coordinates and distance between end coordinates and other vehicles.
- Timer starts before algorithms is started and stops when vehicle is parked.
- Algorithm is performed and throughout the parking process the platform is visually observed to see if the platform bumps into any objects.
- Once the algorithm is performed, the deviation from the optimal coordinates and the angular misalignment is measured.
- These tests will be run four times.

4.3.5 Final Product

To answer Research Question 2, the final artifact with all software modules implemented in hardware needed to be evaluated against the requirements as well. To evaluate and validate the finally integrated product, an experimental study in a controlled environment was designed, but the tests were never completed. The issue of the tests not being conducted is discussed in detail in section 6.2.4.
The finally integrated product was to be evaluated and validated by experimental studies in a controlled environment. The controlled environment consisted of an enclosed room with no tables or chairs and a single parking spot created by placing wooden planks and/or boxes as outlines. The vehicle was then supposed to be driven around the area once to let it map its surroundings. Once it detected the parking space, it was to be placed at a fixed starting position and commanded to conduct the parking manoeuvres necessary for reaching the space and parking autonomously. The fixed starting position was to be done in order to have a baseline of performance from test to test. From this test, metrics such as convergence, time-to-park, longitudinal and latitudinal accuracy, angular misalignment and the time it took to locate

the parking space could have been gathered, evaluated and compared to other studies or real-life test cases. All the data gathered from this experiment could then be referenced to previous studies that tested similar technologies and a general opinion could have been formed as to if the results were good. The experiment should then have been rerun several times to get stronger data and to evaluate eventual outliers.

Example setup:

- Enclosed office room with approximately $15m^2$ floor area.
- All chairs, tables etc. removed.
- Vehicle allowed to drive one lap around the room to create map.
- When parking a parking space is detected, the time it took is noted and the vehicle is moved to a fixed starting position.
- The vehicle is commanded to relocate itself to the desired position for the parking manoeuvres to take place.
- The vehicle will without further input drive to the desired position and commence parking manoeuvres to relocate itself to a desired end position.
- The time taken from park command until the entire parking procedure is concluded will be noted down as time-to-park.
- The longitudinal and latitudinal accuracy will be measured manually by measuring the distance from the wall parallel to it and the "car" in front and in the back.
- The angular misalignment will be measured in relation to the wall to the side as well, where optimal alignment would be considered a completely parallel parking line.
- The same test will be conducted 5 times to get measurement for convergence by looking at the mean and standard deviation of the data points
- To get data to compare ours to, other autonomous parking studies will be analysed and the car will also be driven manually by 5 different people that has little previous knowledge of how the vehicle behaves to get data on how a human would perform in a similar situation. Every subject gets one trial run and afterwards, the test will only be done once by each person to minimize learning bias.

5
Results

This chapter contains all results gathered from the systematic evaluation of both the testing done on the software modules individually and the final product in hardware. First, the results that are relevant to research question 1 are presented and in the end, the results related to research questions 2 are presented.

5.1 Pathfinding

To answer research question 1, pathfinding was systematically evaluated. The first test-run of the pathfinding algorithms were run using ten different images and four different algorithms. The algorithms that were run for this test was A* with an Euclidean distance heuristic, A* Euclidean in the table, A* with a Manhattan distance heuristic, A* Manhattan in the table, Dijkstra's algorithm, Dijkstra in the table, and BFS. All tests were run on a laptop with a core i7 4500U CPU.

The ten different images were all a subsection or resized resolution version of the three images below, e.g., by running one test with the image in figure 5.1 with 2000 by 2000 pixels and another test by resizing or cropping it to 500 by 500 pixels and choosing different start- and end positions. The mazes were reused by choosing completely different start- and end-positions between tests to create different complexity solutions and attempt to elicit different behaviour from the algorithms.

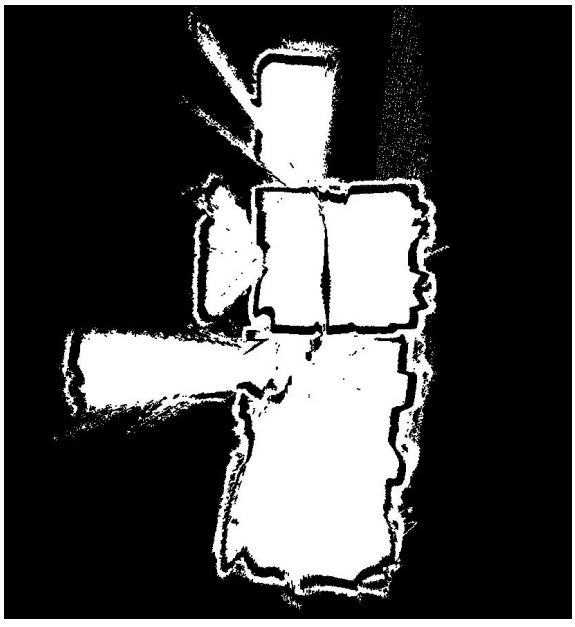


Figure 5.1: High resolution LiDAR image of Infotiv office spaces.

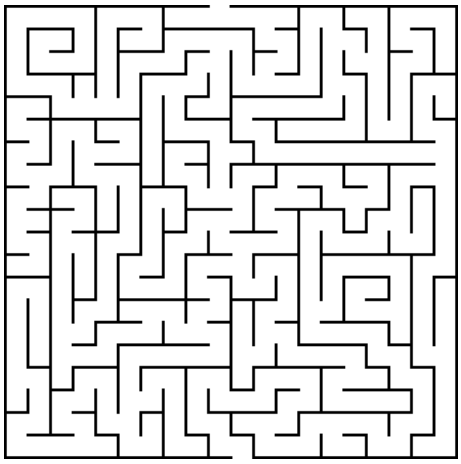


Figure 5.2: Low resolution and complexity maze

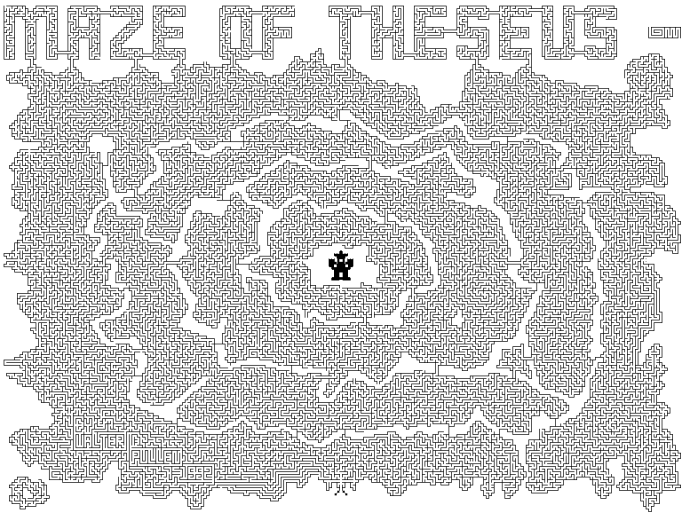


Figure 5.3: High resolution and complexity maze

The data that were gathered during the experiment was the time it took to find the path and the generated path length. Table 5.1 contains the time-to-compute metrics for each of the algorithms and Table 5.2 contains all the generated path lengths from the experiment.

Table 5.1: Time to compute a valid path in seconds (smaller is better). Column 5 describes what image was used for the test. LIDAR meaning a real LiDAR map of a room while MAZE indicates a Maze.

A* Euclidean	A* Manhattan	Dijkstra	BFS	Image
0.306	0.341	0.436	0.406	LIDAR1
0.371	0.337	0.547	0.491	LIDAR2
3.321	2.320	8.137	3.596	LIDAR3
4.848	4.267	6.421	4.194	LIDAR4
2.991	3.212	9.492	4.735	LIDAR5
1.483	1.622	4.246	2.887	MAZE1
5.851	4.061	5.387	3.379	MAZE2
18.104	27.009	19.980	10.501	MAZE3
2.023	4.081	3.140	1.826	MAZE4
1.101	1.351	1.983	1.107	MAZE5

Table 5.2: Generated Path length in pixels (smaller is better). Column 5 describes what image was used for the test. LIDAR meaning a real LiDAR map of a room while MAZE indicates a Maze.

A* Euclidean	A* Manhattan	Dijkstra	BFS	Image
79	75	75	75	LIDAR1
123	121	121	121	LIDAR2
529	529	521	521	LIDAR3
407	407	401	401	LIDAR4
344	342	331	331	LIDAR5
711	681	681	681	MAZE1
828	776	776	776	MAZE2
2219	1839	1834	1834	MAZE3
796	719	719	719	MAZE4
413	387	387	387	MAZE5

As the tables show, there is not any clear superiority that can be distinguished between the algorithms at hand. In some cases, the Euclidean A* is the quickest and in others BFS etc. The only thing that is a trend throughout all the cases is that the Euclidean A* consistently provides a longer path than the other algorithms. The BFS and Dijkstra always provide the shortest path but especially Dijkstra does this at a big performance penalty with an average of 64% longer time to compute than BFS. BFS performed well overall but the only times it was the fastest alternative was on a small LiDAR map with a short path and in a very complex maze.

Even though it is not possible to deem a clear winner for every situation from the testing, the best fit for this project, judging from the data above, would be some variation of A*, either with Euclidean or Manhattan Heuristic, this is mainly because due to the constrained hardware aspect of the project. Because of this, the time it takes to compute the path is weighed much more heavily than if the path generated is a few pixels longer. Even though a difference of e.g., 0.5 seconds would

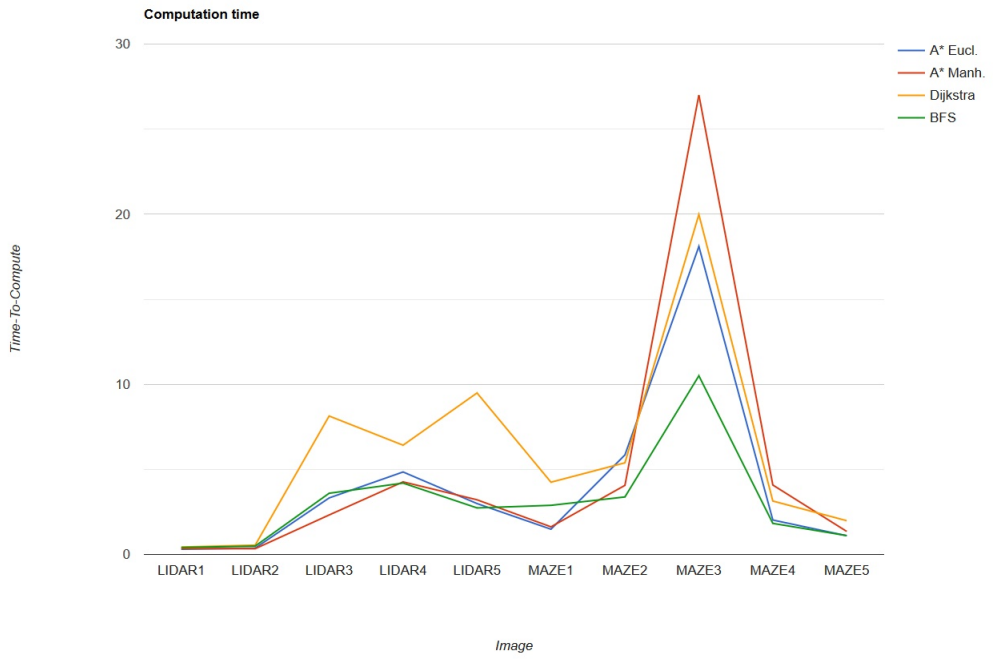


Figure 5.4: Graph of the results from Table 5.1

barely be noticeable for an end-user and would not make the project in any way unfeasible, the computer that was used for these tests is almost ten times faster than a Raspberry Pi 4 when it comes to processor performance and a five second extra delay to compute a path would most certainly be noticeable.

To get the data needed to test the convergence metric, a test was conducted where each algorithm was run ten times with slightly altered start and end positions. This was done by defining the same start and end positions as in the tests above but with a random generated variation of $\pm$ 10 pixels in X- and Y-directions for both the start and end points. Given the rather small sample size of these tests there is a high margin of error, but again, doing a more exhaustive search would be unfeasible due to the time it would take to compute. Furthermore, this test is not meant to show the exact convergence numbers for each algorithm or give a definitive answer on what the best pathfinding algorithm in existence is, but instead give an indication if they would be a good fit for this specific project. The data gathered from the test can be seen in Table 5.3

Table 5.3: Standard Deviation given a random generated start and end point within $\pm$ 10-pixel zone. Only conducted on the same five LiDAR images used in the previous test.

A* Euclidean	A* Manhattan	Dijkstra	BFS
7.86	11.87	8.16	9.41
13.35	5.73	8.24	9.08
7.21	6.41	9.74	9.62
7.34	11.44	7.33	9.88
20.91	11.04	9.62	7.5

By looking at the data for Dijkstra's algorithm, which is mathematically proven to

always provide the shortest path, it is possible to deduct that a standard deviation of around 10 pixels is to be expected due to the paths generated having a variation in length due to the randomly generated start and end positions. All numbers in the test seem to align well with that assumption and converges well. The only exception to this is the A* with a Euclidean heuristic function. Especially in the last test these path lengths fluctuated wildly and varied between 323 and 377 pixels, which is too long of a distance to be explained by the variation induced in the test. Because of this, that specific test was run several times to verify the result, but similar results emerged every time.

The robustness of the algorithms is all equal. If no valid path is possible, they return an array with the length of zero and terminate so that they can be initiated again at a later stage. All algorithms also always accept a one-pixel wide gap as a valid path due to the algorithms only looking at their closest neighbours for information . This issue could be resolved by checking more neighbours at every stage, but it would be much more computationally intensive and this thesis therefore decided to handle that issue by adding more erosion to the pictures instead to eliminate any gaps smaller than the vehicle's width.

Given that A* Euclidean and A* Manhattan both provided similar performance and the high deviation that A* Euclidean showed, A* Manhattan was chosen as the best fit for the project.

5.2 Path following

To answer research question 1, path following was systematically evaluated. The evaluation for the path follower was conducted by giving the algorithms four different paths generated in a real LiDAR image by the A* pathfinding algorithm with Manhattan distance as a heuristic function. During the run, every time the algorithm updated the loop by calculating the nearest neighbouring point on the path, the orthogonal distance to that point was calculated and recorded. For each run, the minimum, maximum and average recorded distance during the entire run was recorded and compiled in Table 5.4 below.

Table 5.4: Minimum, maximum, and average orthogonal distances to desired path measured in pixels.

Min	**Max**	**Average**
0.0	2.90	0.66
0.0	1.48	0.55
0.0	1.98	0.57
0.0	1.63	0.52

In the LiDAR image chosen for this evaluation, every pixel corresponds to a square roughly two centimetres wide. All tests were run with a lookahead distance of seven pixels or roughly 14 cm. The maximum measured deviation from the path was 5.8 cm and the average deviation was between 1-1.3 cm. The reason behind the

minimum distance being zero pixels in every test is because the test does not differentiate between distance to the left and to the right of the path, i.e., there's no positive or negative distance and any distance from the path is considered positive. The minimum distance was still considered valuable data to highlight that there's no constant offsets from the path throughout the test. This was deemed as a good result and a new test was therefore conducted to try to find the best lookahead distance.

In theory, a smaller lookahead distance leads to more aggressive behaviour of the controller and can cause oscillations around the path while a larger lookahead distance gives a smoother behaviour but at the cost of precision around corners and curves where it might cut the corner. Because of these reasons, it is important to find an optimal middle ground where the algorithm is precise but does not display overly aggressive behaviour. The different lookahead distances were tested on a single path to eliminate the path as a source of variance. The tests were first conducted with lookahead distances with an increment of five pixels, roughly corresponding to ten centimetres. The results can be seen in Table 5.5 below.

Table 5.5: Maximum and average orthogonal distances, measured in pixels, with different lookahead distances in five-pixel increments.

Lookahead Dist.	**Max**	**Average**
2	17.21	3.66
7	1.59	0.59
13	2.88	0.95
18	4.41	1.29

As can be seen in the table above, the performance gets worse at the very low end and shows the best performance around seven pixels, or 14cm, lookahead distance and the performance gets gradually worse once the distance is increased further. Another test was therefore conducted with smaller granularity where the lookahead distance was increased in increments of one pixel or two centimetres. The results can be seen in Table 5.6 below.

Table 5.6: Maximum and average orthogonal distances, measured in pixels, with different lookahead distances in one-pixel increments.

Lookahead Dist.	**Max**	**Average**
3	1.02	0.43
4	1.15	0.45
5	1.31	0.49
6	1.47	0.54
7	1.59	0.59
8	1.72	0.65
9	1.96	0.70
10	2.17	0.76
11	2.35	0.82
12	2.53	0.88

As shown in the table above, the best performance is achieved at a lookahead distance of three pixels. However, at this distance, visual inspection of the simulated robot showed some oscillations back and forth and a lookahead distance of five pixels provides a much smoother driving experience while still having a precision of less than one centimetre.

5.3 OpenCV (Computer Vision API)

To answer research questions 1 and 2, parking space detection was systematically evaluated. A full factorial design experiment was conducted to find the optimal parameters for blurring as well as deducing if SURF, SIFT or BRISK would be most optimal algorithm for using together with constrained hardware. SIFT has previously been deemed an unlikely competitor against SURF and BRISK as SIFT performs worse when rotation and blur is introduced. The results of the experiment can be seen in Table 5.7.
The test results were evaluated using three criteria:

- **Quality:** the number of correct features detected/matched
- **Efficacy:** the time it takes to compute
- **Utility:** how well does the implementation work for us in our specific case

Based on the test, it was concluded that SURF was the most suitable algorithm for this specific scenario. SIFT scored fewer matches than SURF, giving it a lower score in the metric quality. Furthermore, the data implies that SIFT has an average computation time of a factor three larger than SURF. This gives SIFT a bad score when it comes to the efficacy metric. The combination of a poor quality and efficacy score makes SIFT a inappropriate algorithm choice. Furthermore, with the efficacy being so poor, it is simply not feasible to have a computation time of almost two seconds in a real time application. The overall utility score is therefore also low as SIFT is not a viable choice for real time applications on constrained hardware.

According to F.K.Noble [1], BRISK in combination with BF was the best combination when wanting to maximize the features detected and matched. BRISK scored very poor in the quality metric as it had an average matches of around 0.25. Its average computation time is not much slower than the rest, however, it is worse than SURF with a factor of around 2.5. BRISK therefore scores an average on the efficacy metric. Overall, BRISK in its current implementation is probably not suitable for use in a real time application with constrained hardware, providing a low score on the utility metric.

SURF scored the highest number of average matches of all the algorithms independent of the values of blur, with few exceptions. This gives SURF a clear advantage in terms of the quality metric. Furthermore, SURF was the fastest of all the algorithms, scoring an average computation time of around 0.3 seconds. Having an algorithm that can perform real time computations is of great importance as this allows for more frames with potential parking spots to be analysed. Using SURF would allow for the algorithm to run at approximately 3Hz which is sufficient.

It is concluded that SURF is the most suitable algorithm for the specific setting at hand. To further determine what parameters of blur is optimal for SURF, another full factorial design was performed. The test uses the same base premise as the previous one, that is a thresholded map as argument and the platform placed 25 centimetres from the delimiting vehicle. The number of frames tested was changed from 50 to 200 to gather more distinct data. The number of levels tested was increased from (1,5,9) to (1,3,5,7,9,11). The results of the factorial design can be seen in Table 5.8

Seeing as all the configurations have similar execution time, this parameter will not be taken into consideration. Furthermore, SURF has not had a single false positive, neither in this test nor the previous one, so it is deemed robust. With these parameters excluded, the two factors that decides is average matches and max matches. There are several configurations that performs well, namely (5,1), (3,3), (5,3) and (5,5). All these configurations have an average matches over 8.3. While (5,3) showed the best result for max matches, (5,1) showed a higher average result. While max matches is good, the average amount of matches is slightly more important as the algorithm will be run in real time. Because of this, the most optimal configuration is deemed to be (5,1) as it has 9.95 average matches and a max match of 12.

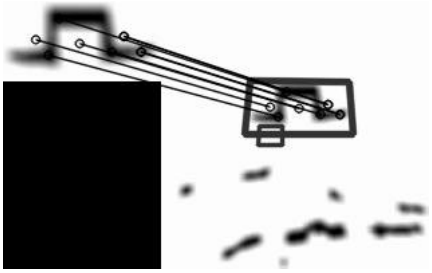


Figure 5.5: The result of the parking space detection algorithm.

The load on the processor of the Raspberry Pi 4 is around 70% with SURF and the implemented parking algorithm running where three out of four cores are used, see figure 5.6. The load was measured using the Linux command “top”, which allows to monitor resource usage [46]. Once all the processes was started, including the SLAM algorithm, SURF and the parking algorithm, the load was measured. An example of how the result of the parking space detection algorithm can look can be seen in figure 5.5

5.4 Parking Algorithm

To answer research question 2, the parking algorithm should have been systematically evaluated. The test for the parking algorithm would have been conducted by placing the platform at a predetermined starting position close to the parking spot. The parking algorithm is then started and can execute on its own. After its completion, the data for accuracy, angular accuracy, safety, and efficiency were to be recorded, the same test is run four times.

Table 5.7: Data from the full factorial design experiment.

Full Factorial Design, n = 50							
Algorithm	Blur Map	Blur Parking Spot	AVG Matches	Max Matches	Average Time(Sec)	True Positives	False Positives
SURF	1	1	6.22	9	0.3	9	0
SURF	5	1	7.7	10	0.3	10	0
SURF	9	1	3.4	7	0.29	7	0
SURF	1	5	6.72	10	0.31	10	0
SURF	5	5	7.32	12	0.31	12	0
SURF	9	5	4.3	9	0.3	9	0
SURF	1	9	5.2	8	0.31	8	0
SURF	5	9	5.92	9	0.3	9	0
SURF	9	9	3.88	7	0.3	7	0
SIFT	1	1	2.5	5	1.87	6	0
SIFT	5	1	2.5	6	1.85	6	0
SIFT	9	1	2.76	5	1.95	5	0
SIFT	1	5	3.56	7	1.87	7	0
SIFT	5	5	4.88	6	1.87	6	0
SIFT	9	5	4.12	6	1.88	6	0
SIFT	1	9	3.34	5	1.88	5	0
SIFT	5	9	4.82	7	1.95	7	0
SIFT	9	9	6.4	9	1.88	9	0
BRISK	1	1	0.28	2	0.8	1	1
BRISK	5	1	0.35	3	0.81	0	3
BRISK	9	1	0.26	2	0.81	0	2
BRISK	1	5	0.33	2	0.81	1	1
BRISK	5	5	0.38	3	0.8	1	2
BRISK	9	5	0.24	2	0.81	0	2
BRISK	1	9	0.24	2	0.81	0	2
BRISK	5	9	0.22	2	0.81	0	2
BRISK	9	9	0.25	2	0.8	0	2

Table 5.8: Data from the full factorial design experiment.

Full Factorial Design, n = 200							
Algorithm	Blur Map	Blur Parking Spot	AVG Matches	Max Matches	Average Time(Sec)	True Positives	False Positives
SURF	1	1	6.52	10	0.31	10	0
SURF	3	1	7.67	11	0.30	11	0
SURF	5	1	9.95	12	0.30	12	0
SURF	7	1	6.70	11	0.31	11	0
SURF	9	1	3.68	8	0.30	8	0
SURF	11	1	2.03	8	0.29	8	0
SURF	1	3	7.06	10	0.31	10	0
SURF	3	3	8.30	11	0.31	11	0
SURF	5	3	9.39	14	0.32	14	0
SURF	7	3	7.09	10	0.31	10	0
SURF	9	3	4.88	10	0.31	10	0
SURF	11	3	3.19	8	0.32	8	0
SURF	1	5	7.08	12	0.33	12	0
SURF	3	5	7.81	11	0.33	11	0
SURF	5	5	8.38	12	0.34	12	0
SURF	7	5	7.59	12	0.34	12	0
SURF	9	5	3.50	8	0.34	8	0
SURF	11	5	3.67	8	0.33	8	0
SURF	1	7	6.27	10	0.34	10	0
SURF	3	7	5.42	9	0.33	9	0
SURF	5	7	6.48	10	0.35	10	0
SURF	7	7	5.80	10	0.35	10	0
SURF	9	7	2.70	7	0.35	7	0
SURF	11	7	3.20	6	0.35	6	0
SURF	1	9	5.30	8	0.35	8	0
SURF	3	9	6.12	10	0.34	10	0
SURF	5	9	6.30	9	0.36	9	0
SURF	7	9	5.60	8	0.37	8	0
SURF	9	9	3.45	8	0.35	8	0
SURF	11	9	4.00	7	0.36	7	0
SURF	1	11	5.70	8	0.36	8	0
SURF	3	11	5.70	8	0.36	8	0
SURF	5	11	6.57	10	0.31	10	0
SURF	7	11	6.50	10	0.33	10	0
SURF	9	11	5.03	8	0.34	8	0
SURF	11	11	5.67	9	0.32	9	0

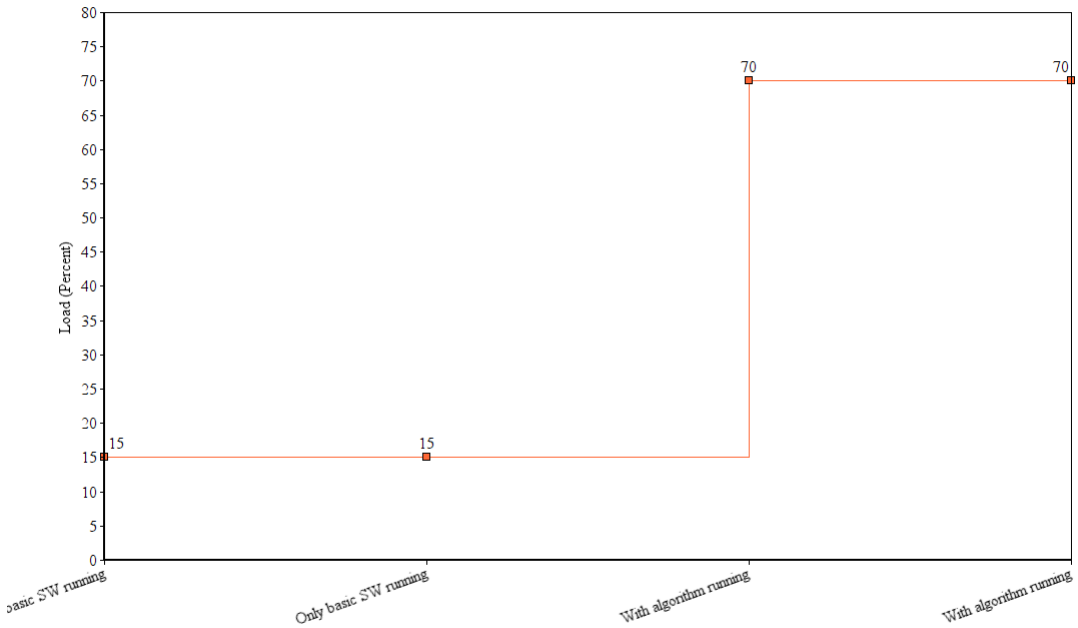


Figure 5.6: Load of the processor

These tests could not be completed due to one main reason. An issue with the BreezySLAM algorithm causing problems for the parking and the path following algorithms when running in hardware. These issues were severe enough for the few tests successfully conducted to not yield any valuable results to draw conclusions from.
This issues are discussed in detail in Section 6.2.4

5.5 The Final product

To answer research question 2, the final product should have been systematically evaluated. The test for the final product was designed to test the performance of all modules working as a unit on hardware. The test was designed as an experimental study in a controlled environment consisting of an enclosed and empty room containing only a single parking spot. The vehicle was then supposed to drive around the room once and map its surroundings and when a parking spot was detected, it was supposed to do everything from locating the parking spot to parking itself fully autonomously.

Due to the same issues discussed in Section 5.4 above, this test could not be conducted properly either. The issues that caused this are also discussed in detail in Section 6.2.4.

6
Discussion

This thesis aimed at systematically evaluating AD software modules specific to autonomous parking while following known software engineering principles. According to some software models such as the waterfall model, it is advised to test and evaluate the software after the development is done. This could result in increased technical debt and many unnecessary development iterations. Working with known software engineering processes and procedures can increase code quality and reduce development cost. If cost can be reduced in both regards to the development and the hardware required for its operation, AD features can be integrated into cheaper vehicles and ultimately make driving safer and more environmentally friendly. Throughout the thesis, the development methodology framework design science has been used while following the concepts of SWEBOK.

6.1 Development Methodology

The methodology framework that was used throughout this project was the framework of design science [2, 42]. In this section this framework's realization and its impact on the project will be discussed.

After working with the framework of design science throughout the project, we observed that it has had a positive impact in almost every regard. It was a straightforward framework to work with since nothing felt forced and most things mentioned in the framework were natural to conduct in any design- or evaluation-based project. For example, working in two distinct cyclical phases, design and investigation, is something that comes naturally when working with design and evaluation since one always starts by investigating the issue at hand, then tries possible solutions and therefore encountering new problems to bring back to investigation.

Guidelines such as communication of research was a major issue that was encountered throughout the project. It had to be managed so that both the business aspects from Infotiv's side and the research aspects Chalmers were conducted in a way that made both stakeholders satisfied. Therefore, time was spent on finding ways of producing something of value for both the company and the SE research community at large. To have this guideline in the back of the mind that perhaps the communication with different parties had to be modified to fit that specific context was helpful.
The method of Design Science was an effective and good framework to work with since it has many benefits and few, if any, guidelines and items that feel unnecessary and that did not have a purpose in the project.

6.2 Results

In this section, all the results are discussed in the same order as they were presented in Chapter 5. In each section, the methods for coming up with the systematic evaluation procedures are discussed first to help answer the research questions followed by a discussion surrounding the actual results of these procedures.

6.2.1 Pathfinding

To evaluate pathfinding algorithms, the initial plan was to find the the fastest algorithm by looking at, e.g., Big-O analyses [47], for different pathfinders. In theory this shows with certainty which the fastest algorithm is. This however proved to be an unfeasible method since different pathfinders can be better suited for some scenarios but not others. Because of this, a method was needed to evaluate different pathfinders for the specific needs of this project.

First, different metrics that were deemed important for this project were deducted. From these metrics, test cases were designed so that they specifically gave results for these metrics. The tests were also designed in a way that tried to eliminate other factors that could influence the results, e.g., by doing them on the same hardware, and the algorithms were developed within the project to make sure that code errors did not influence them. By conducting these tests, the information needed to draw conclusions for this project was successfully gathered and the testing was deemed to answer for the needs of the project.

In the pathfinding results, a clear winner is not apparent. Every pathfinding algorithm has its own set of benefits and drawbacks, this is something that was clear even before the project started by reading up on different possible solutions. The ones that were tested in this project were the algorithms that were most used for similar applications. Even though the margins were, in many cases, modest and all the algorithms could probably have been viable for the task at hand, the A* algorithm with Manhattan distance as a heuristic function prevailed with significantly lower time-to-compute and adequate path lengths in our specific scenario and was therefore chosen. But if one were to apply these pathfinding results to something different than this specific use case, e.g., a different environment or pathfinding in video games, the tests done would probably show a different result. This does not mean that the tests are not accurate or valid for this project, but pathfinding algorithms do perform differently given different scenarios and there is no perfect answer for what algorithm is best for every use case. The pathfinders worked as intended and provided valid paths with low variance in the results and did so with performance good enough to be run on our specific hardware setting.

6.2.2 Path following

For the evaluation of path following algorithms, only one algorithm was implemented and evaluated. Because of their higher complexity than, e.g., pathfinders, the scope of the project did not allow to implement multiple algorithms to test against each other. Different pathfinding algorithms were researched to find an algorithm that fit

this project, this process is described in Section 2.1.7. From that research, Adaptive Pure Pursuit was chosen as a likely best candidate to be evaluated. The metrics that the were desired to gather for the path following was decided upon by discussion together with the technical supervisor at Infotiv. The tests were yet again designed to accurately measure the metrics without other influencing factors.

The path following algorithm was tested successfully in software and performed well with an average deviation of around one centimetre. The lowest average deviation was measured with a lookahead distance of three pixels (six cm). However, with a lookahead distance this short, visual inspection showed that the algorithm was aggressive and constantly adjusted its path to match the curve. This is good if the path has many tight corners or if the path is jagged, but when it comes to more straight paths with less curves, this behaviour could cause a higher deviation. But more importantly, in a scenario that includes real hardware, this could cause small deviations in e.g., the motor response to be exaggerated and the driving experience to become unpleasant. Because of this, a lookahead distance of five pixels (ten cm) was deemed as the best fit for the project.

6.2.3 Parking space detection

For the evaluation of the parking space detection's performance, tests were conducted to determine which algorithm to use and what parameters yielded the best results. In the algorithm there are several factors with multiple levels. The most important factor is the only one that was rigorously tested where the others were deemed crucial and therefore omitted from testing. The amount of blur added to the different images impacted the number of features and ultimately the number of features matched greatly.

The experiments that were conducted aimed to achieve two goals. The first being to decide which feature detection algorithm was most suitable for being used in real time using constrained hardware. Once the best algorithm was decided more rigorous test would decide the optimal parameters for that specific algorithm. The metrics that were chosen, quality, efficacy and utility were used to determine how suitable the different algorithms and configurations were. Quality was derived as the number of correct matches are important. If the number of matches is too low, the algorithm would not be as certain that the parking space is located where it says it is. The threshold for what the algorithm classifies as an identified parking space could be lowered, but that would increase the possibility of a misclassified parking space. Efficacy was derived as the computational time of the algorithm is of great importance. In a system that must run in real time, the computation time of the algorithm is a deciding factor that could eliminate an algorithm as a viable option. Utility was derived as there could be cases where two algorithms or configurations could be tied. For example, a situation where one argument configuration had a slightly higher number of average matches and another one had a slightly higher number of maximum matches could occur. In this case a delimiter must exist. Utility would be used as the deciding factor in a situation where the data showed two configurations that were similar in performance and a human decision had to be made.

All experiments were conducted in a controlled environment with as little external disturbances as possible. The platform was placed in a concealed room that no one had access to, and all moving objects were removed. Furthermore, all objects that were unnecessary for the detection to be made and could be moved, were removed. In this sterile environment the platform was placed in a static location a set distance from the parking space and the platform was not moved throughout the course of the experiments. All frames that were fed into the algorithm were gathered in real time from the platform.

The parking space detection algorithm had to be severely limited in the earlier stages of the project. The algorithm could not run only once due to the limitations with BreezySLAM but had to be run continuously and scanning in real time. Because of this, the computational time of the parking space detection algorithm had to be in focus. Algorithms such as SIFT had little to no chance of competing with more lightweight algorithms due to this limitation. For the parking space detection algorithm, the choice of using SURF for feature detection and BF for feature matching was made. The reasoning behind what algorithms were chosen for evaluation is largely based on the study made by F.K.Noble [1]. There is a possibility that there are algorithms and combinations of algorithms more suited for this specific application. However, the limited nature of the project requires that certain delimitations be made.

F.K.Noble argued that BRISK in combination with BF yielded the best results. However, based on our experiments, BRISK was not a suitable choice. Not only did SURF outperform BRISK in terms of identified features, but it was also significantly faster. Furthermore, the features that BRISK managed to identify, and that BF later matched resulted in many false positives. This does not necessarily have to be a result of BRISKs shortcomings but could be that BF just performed poorly with the specific features that BRISK identified. BRISKs poor ability to identify features could also be due to it not being implemented correctly. The algorithm has many different arguments, and to optimize its ability to identify features a set of separate experiments would have to be conducted. The parameters with which the algorithm is initialized should however not affect the computation time which would still be too large to work in this scenario, making it a poor candidate. BRISK could also just not work well with the simple images that are used in this project. Even though BRISK might not be viable for this project, F.K.Nobles research makes it interesting to investigate further. Maybe BRISK, given the correct arguments and slightly more powerful hardware could heavily outshine the competitors. This could be something to investigate in future works.

SIFT identified more features than BRISK and had, like SURF, no false positives. The algorithms computation time is however longer than what can be considered acceptable in this project. With an average computation time ranging between 1.85 and 1.95 seconds per identification, it was quickly deemed unsuitable for use in a real time application.
Once SURF was deemed the most suitable for this specific application, a series of more in-depth experiments were conducted to establish the optimal parameters.

The experiments did not only help decide the most optimal parameters, but also revealed some interesting facts. Firstly, when using SURF, there were never even a single false positive, indicating that it works well with BF. Secondly, the average computation time did not deviate much. This was not unexpected, but it is good to know that independent of the configuration, the time will not be severely affected. Lastly, there seems to be no clear pattern in the amount of blur on both the images and the number of matches. A limit with the test performed is that it was done in a controlled environment. The vehicle was static, and the parking space looked the same for each configuration tested. In a real-life scenario, this would rarely be the case, perhaps making it harder to have the optimal configuration in every situation.

The amounts of features matched using SURF and BF is, in comparison to the example on OpenCV's website, still rather low. In those examples, they match features on a cereal box, with features from an image containing that same cereal box, positioned on a table with several other items. The amount of features they managed to extract from both the images and the matches they get is larger than what we managed. This could be because the images of the parking space and the map contains comparatively few features. The example images contain plenty colours and much information around the object detected which could be a cause for the fact that more features are detected. One way to perhaps increase the number of features detected is to introduce some white noise to the images, causing them to not be identical anymore. This could make the feature matching algorithm detect more features as the image would no longer only contain the colour values 0 and 255. Furthermore, it could perhaps make the feature matching algorithm match more features or be more secure on the features that it does match. However, there is also the risk that white noise could create features that previously did not exist, thereby causing it to make more false positive matches. Introducing white noise would once again require a great deal of configuration as to make the algorithms work optimally. Unless the feature matching algorithm is configured correctly, there is a high chance that it will either not match any features or match way too many features. Once again, another set of rigorous tests would need to be performed to make a sensible judgment.

6.2.4 Issues with hardware platform testing

When testing the systems on the hardware platform, severe issues were discovered. Firstly, after just a few preliminary tests to test basic functionality, the hardware platform started to display different technical issues. This was something that it had done intermittently throughout the project. This took a lot of time as the issues had to be fixed, and we did not have much previous experience with the platform or its issues.

Second, during the brief tests we conducted, the platform did not act as expected. This was caused by the coordinates calculated by BreezySLAM not being correct. According to the coordinates from BreezySLAM, everything was working as intended and the car's positioning was close to what the algorithms tried to achieve. However, the platform was, in reality, not located where the algorithm claimed it to be. There was in fact a big difference in what the BreezySLAM claimed and the real position-

ing. This problem did not seem to be caused by the algorithms that were developed in the project but by BreezySLAM. BreezySLAM was simply not accurate enough to calculate a good enough depiction of its surroundings to let it position itself in space while driving. This caused major spatial drift when driving the vehicle and was verified to be the cause of our issues by a couple of different methods. First, the platform was positioned at a fix position in the room marked as coordinates (0,0). The platform was then manually driven 50 cm straight forward and then reversed back to its original position. By doing this small and simple manoeuvre, the coordinates could drift as much as 50 pixels in any or both X- and Y-directions in an image that was just 500x500 pixels. We tried to mitigate the issue by trying to see if there was a pattern to the drift that could simply be offset by another function, but the data seemed random. Second, to eliminate the possibility of the LiDAR hardware causing the issue, another more advanced SLAM algorithm called *Google Cartographer*[48] was implemented using the LiDAR from the platform and a more powerful laptop as computational unit. By doing the same experiments of driving the vehicle straight forward and back again, we could conclude that no such drift was present in that algorithm.

Unfortunately, implementing Google Cartographer was not in the scope of this thesis because of it being much more computationally intensive and therefore not viable to run on the platform. Furthermore, the way that Cartographer provides its output data such as coordinates, angle in space and images are extremely different from how BreezySLAM works and a complete rebuild of the entire code base would have been required to integrate it into the project.

Because of these issues, the project could not provide accurate or meaningful results for the parking algorithm or final product tests.

6.3 Threats to Validity

In this section, threats to validity will be discussed. The different type of threats are separated in several different subsections. Conclusion validity is the reasoning surrounding the relationship between treatment and outcome, i.e., can we draw reasonable and correct conclusions for our outcome from our treatment construction. Internal validity also concerns the relation between treatment and outcome, but instead focuses on if something outside of our control could have caused the effect seen in the results. Construct validity concerns if the construction of the study actually measures what it is intended to measure. External validity concerns how the findings of the study can be generalised or if the cause-effect relationship is context-dependent.

6.3.1 Conclusion Validity

Some tests were designed to not always have a high degree of statistical power, i.e., with a $p < 0.05$, in terms of sample size. This was done primarily because of time limitations where large enough sample sizes for e.g., the pathfinding algorithm's convergence would be unfeasible to test. In this thesis, this was by design and a choice made because of the fact that this thesis goal was not to determine with absolute

certainty the best algorithms in their respective fields, but instead use the tests for development purposes and highlight how one can conduct a systematic evaluation in a project like this. Because of this, some of the findings regarding specific algorithms' performance might not be accurately generalised and should not be used at face value for a project developing a similar artefact.

6.3.2 Internal Validity

The biggest threat to internal validity is the lack of accuracy in the chosen SLAM algorithm. Because of this, the validation of the other components' performance was not possible to conduct properly. Because SLAM is such an integral part of every algorithm, the real-world performance could only be guessed from how the platform acted since it did not have correct coordinates. The algorithms that could be validated in software were however validated properly and the issue was confirmed to be caused by BreezySLAM and not anything else, so the algorithms should therefore function properly given more accurate SLAM data.

6.3.3 Construct Validity

To safeguard against construct validity threats, it is important to make sure that the metrics you are gathering data on are actually the correct metrics for the situation at hand. This was achieved mainly through a rigorous requirements engineering procedure to gather and specify the various metrics that were relevant. The procedures for how the requirements elicitation and specification were conducted was from lector Eric Knauss's course on the subject [49]. The requirements specification created could then be used to verify that the different metrics that were decided upon were actually the correct ones for the project.

Due to the issues presented in 6.2.4 the final product was never tested or evaluated. Because of this, no data was gathered for the final product and the metrics can not be evaluated. However, seeing as all the previous tests on the individual components were successful and their metrics were deemed suitable, the metrics and design of the systematic evaluation for the final product should in theory still be valid, although further experiments would be needed to validate the functionality and validity of the individual parts as a complete unit in the future.

6.3.4 External Validity

Most of the tests were run in a controlled environment with constrained hardware or purely in software, which poses threats to external validity. As the platform is unique with other software running concurrently, the exact environment of the study could be hard to replicate. Furthermore, not all factors that could have affected the outcome of some of the experiments were tested. For example, when conducting the experiments for the parking space detection algorithm, the threshold factor was deemed crucial to identify a significant number of features at all, and tests without it were therefore omitted as a factor. However this could, given a larger time span of the project, be tested more rigorously.

The outcome of the thesis may vary based on the environment where the vehicle is placed. If the parking algorithm is used in a controlled environment that is empty except for the two vehicles delimiting the parking slot, the algorithms should be able to perform adequately. However, in a real-life traffic situation with several objects of different shapes and sizes, and a constantly changing landscape, it will perhaps not work as intended. This is due to all calculations not being done in real time due to the computational limitations on the hardware and the 2D LiDAR's inability to see objects that are not located on the same height as itself. Because the vehicle will not be able to do all calculations for everything in real time, some things must be precalculated before doing its operations, for example the path generation for the path to the parking space. It will not be able to change its path while following it, e.g., if the parking space becomes occupied or if a pedestrian suddenly decides to stand in its way. Although this could be mitigated by using some other types of sensors while driving, e.g., ultrasonic sensors, to be able to interrupt its manoeuvres and recalculate a path, it was not in the scope of this study due to hardware limitations and time constraints.

6.3.5 Lessons Learned

The project in general has been successful, but some things could have been performed better. From the setbacks experienced, a few key lessons can be learned. The biggest problem throughout the project was that BreezySLAM did not perform as expected. The choice of using BreezySLAM was based on the fact that the company already had an implementation running on their platform and that it probably was lightweight enough to run on the constrained hardware. However, the implementation had never formally been evaluated nor verified. The lesson learned here is to always early in a project secure any assumptions in the time plan by verifying prerequsities, and to later on use a good test suite that can empirically verify fit and functionality to minimize the risk for delays. Issues could then have been detected earlier and mitigated to some extent if more rigorous tests were performed on SLAM in the earlier stages to prove that the assumptions that were made was correct. If the issue with the spatial drift was discovered earlier, alternative SLAM algorithms could have been researched. Even if no other SLAM algorithm that could be run on hardware that constrained could be found, the project could have been redesigned at an earlier stage.

Another key lesson learned is to have an even more detailed plan on exactly what data should be gathered from each experiment. The way that the experiments were performed was rarely changed as they were planned to a great extent. However, some of these had to be run multiple times as some key data was missing. Furthermore, automating experiments and data collection should be done to the greatest extent possible. The first time some of the experiments were run they were performed manually, and the data collected manually inserted into spreadsheets. However, after having to re-run a few experiments, they were automated. Not only did this make it easier to re-run the experiments, but the base code for these could be reused for other experiments, making it easier to setup new tests.

7
Conclusion

The goal of this thesis was to research how systematic evaluation can be performed on a series of software modules developed as a proof of concept that it is possible to achieve specific AD features, such as an autonomous parking algorithm, using fewer sensors and cheaper hardware than what currently is used in the automotive industry.

In this thesis, several methods of evaluation for testing, both in software and in hardware, has been presented. The methods presented in this thesis cannot be concluded to be the optimal tests to conduct in every scenario, but instead, it can guide future research on how to come up with testing methodologies and act as general guidelines. However, given the same type of algorithms to test the tests proved suitable and provided the data needed to make decisions on the best fitting algorithms and technologies.

The tests were mostly successful and provided the results that were needed to answer the two research questions. However while testing in hardware, one specific component, BreezySLAM, was not accurate enough to evaluate and verify the performance of the product as a whole. This was because of the spatial drift caused by BreezySLAM's lack of ability to correctly position itself within space as discussed in Section 6.2.4. The issue was verified to be caused by BreezySLAM by both testing it individually with simple experimental tests and by implementing a more computationally intensive SLAM algorithm in the form of Google Cartographer, which proved more accurate. However, its implementation and integration with the other software components was outside the scope of the project.
For future projects, the first thing to consider would be to use another SLAM algorithm, preferably e.g., Google Cartographer, that gave better results in terms of both map- and positioning accuracy during the quick testing done in this thesis. This would require the scoping to be a bit different and allow for more advanced hardware for running the SLAM algorithm at a viable speed.
This increase in compute performance would also allow for further testing surrounding BRISK, SURF and SIFT feature recognition to verify if BRISK is better than the others if calculation speed is not as crucial or hardware significantly increased.

For the implementations to work in a real-world scenario, some sort of handling of dynamics in the environment would also be needed. In the work done here, if an object is placed in front of the platform e.g., after a valid path has been calculated to the goal, the platform takes no new information into consideration while it is driving there, which would not be feasible in a real-world environment.

Bibliography

[1] F. Noble, "Comparison of opencv's feature detectors and feature matchers," 11 2016, pp. 1–6.

[2] A. Hevner, A. R, S. March, S. T, Park, J. Park, Ram, and Sudha, "Design science in information systems research," *Management Information Systems Quarterly*, vol. 28, pp. 75–, 03 2004.

[3] (2020) Software development process. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Software_development_process

[4] (2020) Software engineering body of knowledge (swebok). IEEE Computer Society. [Online]. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering

[5] (2020) Escaping the black hole of technical debt. Atlassian. [Online]. Available: https://www.atlassian.com/agile/software-development/technical-debt

[6] (2020) The evolution of autonomous vehicles. Protivity. [Online]. Available: https://www.protiviti.com/US-en/insights/evolution-autonomous-vehicles

[7] (2016) Driven to distractions. The Sun. [Online]. Available: https://thesun.co.uk/news/2480273/millions-of-drivers-are-suffering-from-%parallelophobia-a-fear-of-parallel-parking-a-new-study-finds/

[8] (2019) Intelligent parking assist system. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Intelligent_Parking_Assist_System

[9] (2019) Sae international releases updated visual chart for its "levels of driving automation" standard for self-driving vehicles. SAE. [Online]. Available: https://www.sae.org/news/press-room/2018/12/sae-international-releases-updated-visual-chart-for-its-%E2%80%9Clevels-of-driving-automation%E2%80%9D-standard-for-self-driving-vehicles

[10] A. Suppé, L. E. Navarro-Serment, and A. Steinfeld, "Semi-autonomous virtual valet parking," in *Proceedings of the 2nd International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, 2010, pp. 139–145.

[11] N. Tripathi, G. Sistu, and S. Yogamani, "Trained trajectory based automated parking system using visual slam," *arXiv preprint arXiv:2001.02161*, 2020.

[12] J. Han, J. Kim, and D. H. Shim, "Precise localization and mapping in indoor parking structures via parameterized slam," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 12, pp. 4415–4426, 2018.

[13] I. E. Paromtchik and C. Laugier, "Autonomous parallel parking of a nonholonomic vehicle," in *Proceedings of Conference on Intelligent Vehicles*. IEEE, 1996, pp. 13–18.

[14] K. Jiang and L. D. Seneviratne, "A sensor guided autonomous parking system for nonholonomic mobile robots," in *Proceedings 1999 IEEE International Con-*

ference on Robotics and Automation (Cat. No. 99CH36288C), vol. 1. IEEE, 1999, pp. 311–316.

[15] K. Dhivya, R. Nagarajan, and G. Prabhakar, "Autonomous lane parking system for nonholonomic vehicles," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. IEEE, 2016, pp. 421–425.

[16] (2020) The cost of 'technical debt' per average business application exceeds $1 million. Itbusinessedge. [Online]. Available: https://www.itbusinessedge.com/slideshows/show.aspx?c=93589

[17] B. M. Williamson, E. M. Taranta, P. Garrity, R. Sottilare, and J. J. LaViola, "A systematic evaluation of multi-sensor array configurations for slam tracking with agile movements," in *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2019, pp. 1231–1232.

[18] F. Zhang, X. Dong, B. Yang, Y. Zhou, and K. Ren, "A systematic evaluation of wavelet-based attack framework on random delay countermeasures," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1407–1422, 2020.

[19] (2020) Automated vehicles for safety. NHTSA. [Online]. Available: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety

[20] (2019) Rplidar a2. slamtec. [Online]. Available: https://www.slamtec.com/en/Lidar/A2

[21] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to opencv," in *2012 proceedings of the 35th international convention MIPRO*. IEEE, 2012, pp. 1725–1730.

[22] T. 1712, "Implementation of the Adaptive Pure Pursuit Controller," FRC Team 1712 Dawgma, Tech. Rep., 08 2018.

[23] (2020) What is lidar? NOAA. [Online]. Available: https://oceanservice.noaa.gov/facts/lidar.html

[24] *RPLidar A2 Technical Specifications*, SLAMTEC, May 2020, rev. 1.0.

[25] S. Bajracharya, "Breezyslam: A simple, efficient, cross-platform python package for simultaneous localization and mapping (thesis)," 2014.

[26] (2019) Why python is popular despite being (super) slow. Medium. [Online]. Available: https://medium.com/@trungluongquang/why-python-is-popular-despite-being-super-slow-83a8320412a9

[27] (2020) Arm architecture. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/ARM_architecture#Advanced_SIMD_(Neon)

[28] K. Krinkin, A. Filatov, A. y. Filatov, A. Huletski, and D. Kartashov, "Evaluation of modern laser based indoor slam algorithms," in *2018 22nd Conference of Open Innovations Association (FRUCT)*, 2018, pp. 101–106.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[30] "First robotics competition," https://www.firstinspires.org/robotics/frc, accessed: 2020-03-20.

[31] T. Bakker, K. Van Asselt, J. Bontsema, J. Müller, and G. van Straten, "A path following algorithm for mobile robots," *Autonomous Robots*, vol. 29, no. 1, pp. 85–97, 2010.

[32] Y. A. Kapitanyuk, A. V. Proskurnikov, and M. Cao, "A guiding vector-field algorithm for path-following control of nonholonomic mobile robots," *IEEE Transactions on Control Systems Technology*, vol. 26, no. 4, pp. 1372–1385, 2017.
[33] (2020) Continuous integration. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Continuous_integration
[34] "Dat321/dit847 software testing," p. 22.
[35] (2020) Bamboo vs jenkins vs travis - full comparison of continuous integration tools. Deployplace. [Online]. Available: https://deployplace.com/blog/bamboo-vs-jenkins-vs-travis/
[36] (2020) Built for every team. TravisCI. [Online]. Available: https://travis-ci.com/plans
[37] (2020) Software quality. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Software_quality
[38] K. Rainer, H. Dirk, D. Dmitri, T. Sebastian, and B. Wolfram, "Autonomous driving in a multi-level parking structure," *IEEE*, 2009. [Online]. Available: https://ieeexplore-ieee-org.proxy.lib.chalmers.se/document/5152365
[39] S. Jie, Z. Weiwei, W. Xuncheng, G. Qiaoming, and L. Suyun, "Laser-based slam automatic parallel parking path planning and tracking for passenger vehicle," *IET*, vol. 13, 2019. [Online]. Available: https://ieeexplore-ieee-org.proxy.lib.chalmers.se/stamp/stamp.jsp?tp=&arnumber=8851494&isnumber=8851488&tag=1%5C%5C&tag=1
[40] (2020) dSpace. [Online]. Available: https://www.dspace.com/en/pub/home/products/hw/micautob/microautobox2.cfm
[41] I. Gyubeom, K. Minsung, and P. Jaeheung, "Parking line based slam approach using avm/lidar sensor fusion for rapid and accurate loop closing and parking space detection," *MDPI*, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/21/4811/html
[42] R. J. Wieringa, *What Is Design Science?* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 3–11. [Online]. Available: https://doi.org/10.1007/978-3-662-43839-8_1
[43] "Dat231/dit276 requirements engineering," p. 73.
[44] D. Leffingwell and D. Widrig, *Managing software requirements: a unified approach*. Addison-Wesley Professional, 2000.
[45] J. Karlsson, C. Wohlin, and B. Regnell, "An evaluation of methods for prioritizing software requirements," *Information and software technology*, vol. 39, no. 14-15, pp. 939–947, 1998.
[46] (2018) Boolean World. [Online]. Available: https://www.booleanworld.com/guide-linux-top-command/
[47] I. Chivers and J. Sleightholme, "An introduction to algorithms and the big o notation," in *Introduction to Programming with Fortran*. Springer, 2015, pp. 359–364.
[48] *Google Cartographer*, Google, 5 2020. [Online]. Available: https://google-cartographer.readthedocs.io/en/latest/#
[49] "Dat231/dit276 requirements engineering," p. 18.

Appendix A

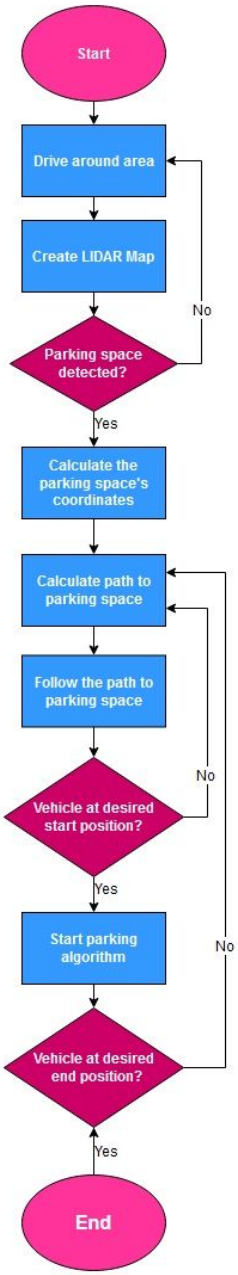
Requirement specification for Autonomous Parking Platform

A. High-level demands

This chapter explains the high-level requirements, giving a understanding about the project's goal and and overview of the functions of the system.

A1. Flows

The chart below is the desired logical flow of the parking algorithm.



Steps in execution	Related tasks and subtasks
1. Users should be able to manually drive the vehicle.	C6, E1.1
2. The vehicle autonomously detects parking spaces in real time.	C1, C2, D1.2, D1.3, E1.1
3. The vehicle calculates and follows a path to the parking space	C3, C4, C6, E1.2
4. After reaching the desired start position, the vehicle shall autonomously park itself.	C5, C6, C7, D1.4

A2. Business goals

Goals for the new system	Solution vision	Related tasks	Deadline
1. Efficient and easy to use for the user	The entire parking procedure should include a maximum of TWO manual steps for the user (e.g., accept a detected parking spot and commence parking algorithm)	C5, C7, C8, E1, E2, E3, E4	2020-06-20
2. The platform should be as easy as possible to adapt to other domains, e.g., a warehouse robot or something similar.	Make the software as modular as possible, it should be easy to change e.g., the parking space detection to detection of another object instead.	C1, C2, C3, C4, C6, D1, D4	2020-06-20
3. Low cost of implementation	The system should use only a 2D-LiDAR and constrained hardware (such as a Raspberry Pi 4)	C6, D2, D3	2020-06-20

A3. Requirements prioritization

The table below shows the order in which requirements should be prioritized. The priority score is a combined score from two tests:

Test 1: The 100-dollar test[1], from a developer viewpoint.
Test 2: The ranking test(bubble sort)[2], from an Infotiv management viewpoint.

The rank goes from high priority (1) to low priority (10).

Rank	Requirement	Priority score	Test 1 score (developer)	Test 2 score (Business Manager)
1	C2	0,83	0,77	0,95
2	C1	0,74	0,57	0,9
3	C3	0,73	1	0,45
4	C4	0,71	0,57	0,85
5	C6	0,69	0,57	0,8
6	D1	0,66	0,57	0,75
7	C5	0,64	0,57	0,7
8	E1	0,61	0,57	0,65
9	C7	0,45	0,66	0,25
10	C8	0,33	0.57	0,1

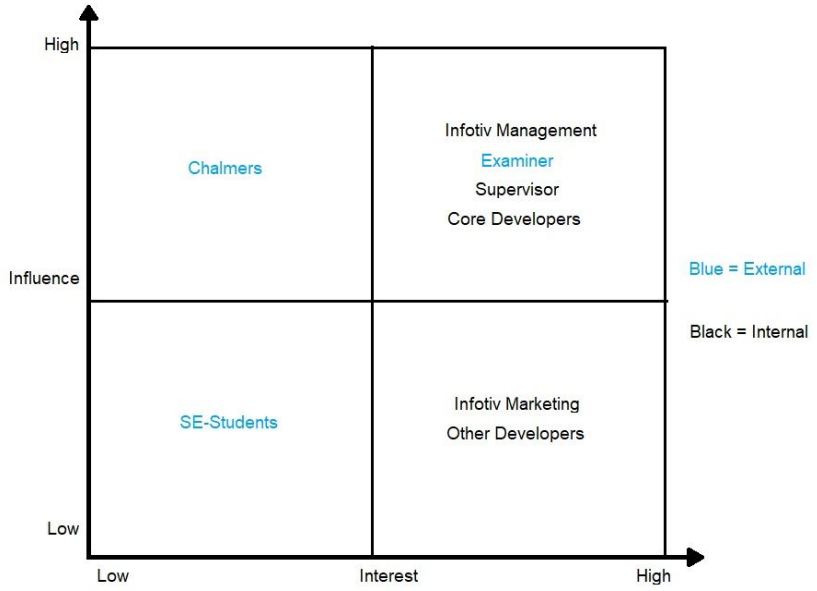
[1] "Managing software requirements; a unified approach," Scitech BookNews, vol.24, no.1, 03 2000, copyright - Copyright BookNews, Inc. Mar 2000; Last updated - 2010-06-06. [Online].

[2] Karlsson, C. Wohlin and B. Regnell, "An evaluation of methods for prioritizing software requirements," *Information and Software Technology*, vol. 39, no. 14-15, pp. 939–947, 1998.

B. Stakeholders

To get an overview of the stakeholders in this project and their relationships between Interest and Influence, see the stakeholder diagram below. For an explanation of what the different stakeholders infer, see the list below.

- **Core Developers** are the owners of the project, in this case Nils Gangby and David Johansson
- **Infotiv Management** refers to the management at Infotiv AB where the project is conducted.
- **Examiner** refers to the examiner of the project from the Software Engineering division at Chalmers
- **Supervisor** refers to our supervisor for the project which guides us with academic and technical know-how throughout the project
- **Chalmers** refers to Chalmers University of Technology as an institution.
- **SE-Students** refers to other Software Engineering students that might take an interest in the work done in this project.
- **Infotiv Marketing** refers to the marketing division at Infotiv AB which may or may not sell this as a product if deemed a success.
- **Other Developers** refers to the other developers at Infotiv AB who might gain knowledge from the project that they can apply in other projects in the future.



C. Tasks to support

The autonomous platform shall implement some of the following tasks based on the prioritization. Each task is executed from start to finish with possible subtasks. Subtasks are marked with another number after the main task number, e.g., C1.2.

The example solutions should be used as guidelines, they can be used as-is or changed if needed.

C1. Generate a SLAM map

This task allows the vehicle to generate a graphical SLAM map of its environment so that further image processing can be applied to it.

User: Vehicle
Start: The vehicle starts driving around the area.
End: When the vehicle has created an image of the generated SLAM map.

Subtasks and variants:	Example solutions:	Code:
1. Receive raw data from BreezySLAM		C1.1
2. Generate an Image from the data that can be used for image processing		C1.2

C2. Detect a parking space

This task allows the vehicle to detect and locate the coordinates for a valid parking space in real time.

User: Vehicle
Start: The vehicle has been driven around the area long enough to create an OK SLAM map of the area.
End: When the vehicle has detected and located the coordinates for a parking space

Subtasks and variants:	Example solutions:	Code:
1. Receive an image of the SLAM map generated in BreezySLAM		C2.1
2. Apply image processing algorithms if needed to clean up the map.		C2.2
3. Detect a parking space	Utilize Feature recognition algorithms to match the features in the SLAM map to a reference image of a parking space	C2.3
4. Calculate the detected parking space's coordinates in the SLAM map.	Calculate the translational matrix between the coordinate system of the reference image and the SLAM map.	C2.4
5. Calculate a desired position for the vehicle to navigate to	Use the same translational matrix as above to translate a marker on the reference image to the SLAM map.	C2.5

C3. Generate a path to reach parking space

This task allows the vehicle to generate a path to reach the parking space and its desired starting position.

User: Vehicle

Start: The vehicle has been given the coordinates to a parking space and its desired starting position
End: When the vehicle has generated a path that it can follow to reach said starting position

Subtasks and variants:	Example solutions:	Code:
1. Receive the SLAM map and desired start & end positions		C3.1
2. Generate a valid path between those two points without collisions.	Use a pathfinding algorithm	C3.2

C4. Follow the path to the parking space

This task allows the vehicle to follow a given path to reach the parking space and its desired starting position.

User: Vehicle
Start: The vehicle has been given a generated path to a parking space and its desired starting position
End: When the vehicle has relocated itself to the parking space via said path.

Subtasks and variants:	Example solutions:	Code:
1. Receive the SLAM map and a generated path		C4.1
2. Follow given path with acceptable levels of deviation	Use a path following algorithm	C4.2

C5. Parking algorithm

This task allows the vehicle to generate a path to park itself in the parking space

User: Vehicle
Start: The vehicle is located at a desired starting position to commence parking.
End: When the vehicle has relocated itself to its final parked position

Subtasks and variants:	Example solutions:	Code:
1. Receive the SLAM map and its current position		C5.1
2. Generate a path that ultimately takes it to a parked position	Use the same pathfinding algorithm as in B4 but with several waypoints	C5.2
3. Follow given path and conduct parking maneuvers	Use the same path finding algorithm as in B4	C5.3

C6. The software can control the Infotiv Embedded Platform

This task allows the Software to control the hardware on the Infotiv Embedded Platform (IEP)

User: Software
Start: The software has generated paths and/or commands that the hardware should execute
End: When the entire parking procedure is completed

Subtasks and variants:	Example solutions:	Code:
1. Receive commands from the tasks above.		C6.1
2. Apply those commands to the vehicle so it can drive.	E.g., Use necessary PID controllers.	C6.2

C7. Accept proposed parking spot

This task allows driver to accept a proposed parking spot so that the vehicle knows it should park there

User: Driver
Start: When a parking spot has been detected by the vehicle
End: When the user has accepted the parking spot

Subtasks and variants:	Example solutions:	Code:
1. Driver receives a notification that a parking spot has been detected	Provide text in terminal or have some GUI (GUI not demanded)	C7.1
2. Give the user an option to accept the parking spot	Either let the user type “yes” in a terminal or click a button depending if GUI is existant	C7.2

C8. Data feedback

This task allows driver to accept a proposed parking spot so that the vehicle knows it should park there

User: Vehicle
Start: When the program launches
End: When the program shuts down

Subtasks and variants:	Example solutions:	Code:
1. If wanted, the user should be able to extract data from the column.	Should be able to extract Current position, current speed, goal position etc.	C8.1

D. Technical IT architecture

This chapter presents the compatibility requirements in regards to the Infotiv Embedded Platform (IEP)

D1. New hardware and software

Platform requirements:	Example solutions:	Code:
1. The software should be written mainly in Python		D1.1
2. The software has to be able to run on a Raspberry Pi 4	Make every component of the software as lightweight as possible	D1.2
3. The software must only use a RPLiDAR-A2 as sensory equipment		D1.3
4. The software should be compatible with IEP's underlying communication protocols	Use ZeroMQ for handling of data	D1.4

E. Usability and design

The following requirements are for the user experience of the platform

E1. Usability

Platform requirements:	Example solutions:	Code:
1. The vehicle should require no manual input to detect a parking space	The vehicle can autonomously detect and locate a valid parking space.	E1.1
2. The driver should not need to relocate the vehicle next to the parking space manually	The vehicle should relocate itself to a desired starting position to commence parking.	E1.2
3. The driver should get a notification only when a parking space is detected and located	Notify either via Command line or a GUI.	E1.3
4. The notification requires as little manual labour as possible to accept	It can be accepted in one command or one click (If there's any GUI).	E1.4

Tracability Analysis

Tracability between Tasks and Business goals

Tasks	**A2.1**	**A2.2**	**A2.3**

C1		X	
C2		X	
C3		X	
C4		X	
C5	X		
C6		X	X
C7	X		
C8	X		
D1.1		X	
D1.2			X
D1.3			X
D1.4		X	
E1.1	X		
E1.2	X		
E1.3	X		
E1.4	X		