AALBORG UNIVERSITY

STUDENT REPORT

ED2 - P2 - H103

Mapping and navigation of unknown terrain

Students:
Antal János Monori
Emil Már Einarsson
Gustavo Smidth Buschle
Thomas Thuesen Enevoldsen

Supervisor:
Akbar Hussain
Torben Rosenørn

June 21, 2015

AALBORG UNIVERSITY
STUDENT REPORT

School of Information and Communication Technology
Niels Bohrs Vej 8
DK-6700 Esbjerg
http://sict.aau.dk

Title:
Mapping and navigation of unknown terrain

Theme:
Scientific Theme

Project Period:
Spring Semester 2015

Project Group:
H103

Participant(s):
Antal János Monori
Emil Már Einarsson
Gustavo Smidth Buschle
Thomas Thuesen Enevoldsen

Supervisor(s):
Akbar Hussain
Torben Rosenørn

Copies: 3

Page Numbers: 91

Date of Completion:
June 21, 2015

Abstract:

This project looks at the future needs for autonomous exploration of unknown environments and terrain. In the coming years humanity will push for more planetary and deep-sea exploration, and during this autonomous vehicles with capabilities of mapping unknown environments will play a major role. Maps of unknown planets or unmapped ocean bed will help prevent hazardous expeditions and will also gather data for researchers to process before visiting the unknown terrain with a larger scale expedition. Using a laser, ultrasonic sensors and a chassis, a low intelligence autonomous rover was created. It is capable of creating 2D maps using Hector SLAM on ROS with readings from the laser. The ultrasonic sensors were used to create a close proximity detection system, which helped the rover avoid obstacles it encountered in the unknown environments.

Contents

Preface

The project entitled 'Mapping and navigation of unknown terrain' was made by four students from the Electronics and Computer Engineering programme at Aalborg University Esbjerg, for the P2 project in the second semester.

From hereby on, every mention of 'we' refers to the four co-authors listed below.

Aalborg University, June 21, 2015

Antal János Monori
<amonor14@student.aau.dk>

Emil Már Einarsson
<eeinar14@student.aau.dk>

Gustavo Smidth Buschle
<gbusch14@student.aau.dk>

Thomas Thuesen Enevoldsen
<tten14@student.aau.dk>

Chapter 1

Introduction

We started out the project with a brainstorming session where we discussed different problems we would like to solve. By the end of the session, we came up with the following list of problems:

- Noise pollution
- Navigation using echolocation
- Re-usability of heat waste
- Mapping of unknown terrains without any external technologies
- Maze navigation

We could identify a common theme amongst these problems, which led us to the conclusion that the problem-domain should be within *mapping and navigation*.

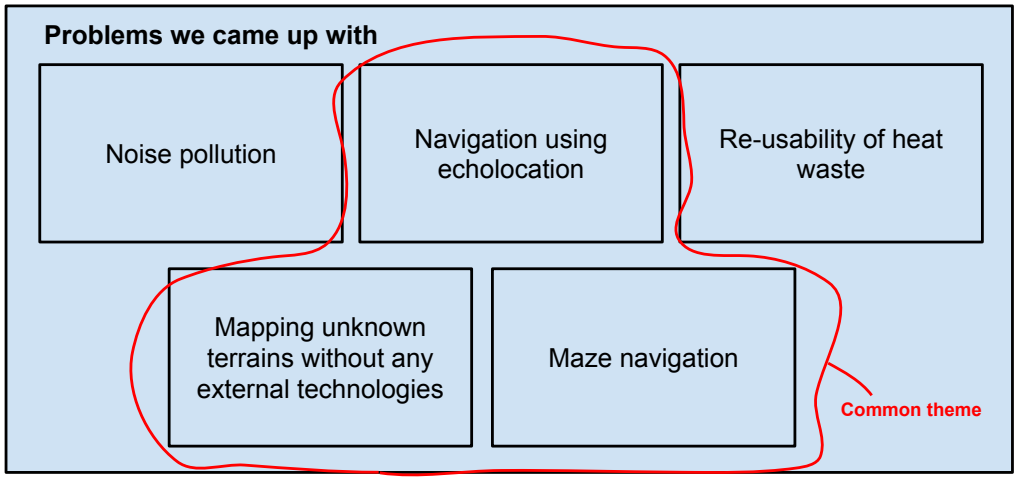


Figure 1.1: High-level block diagram expanding upon our brainstorming session

Furthermore we decided to expand upon the problem-domain we set for ourselves and to further explore it. This led to more discussions on possible outcomes, implementations and benefits of such system.

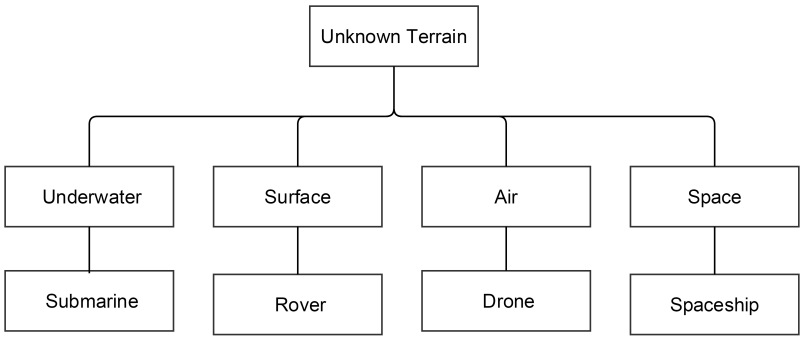


Figure 1.2: Flowchart of our first main ideas

In the flowchart above, it is possible to see the different approaches we could take to map different unknown terrains. The outcome of this flow chart came from a brainstorming session too, after some very initial research has been done within the problem-domain of *mapping and navigation*.

Thereafter, we decided to continue with the selected problem-domain and started using a 6W-diagram model to expand upon it further and to identify the parts needed to be researched. This method helped us to come up with a solution to the problem at hand.

1.1 Initiating Problem

The selected initiating problem is ***mapping and navigation of unknown terrain***.

Chapter 2

Problem Analysis

Since the beginning of time, early human settlements were eager to explore their environment and surroundings. It did not take long, until the very first humans started moving even further and eventually became explorers. They left marks in the environment and started using landmarks to remember pathways and places. This primitive representation of prehistoric places and early history maps can be traced back to 24000-25000 BC [1]. Soon after, they started using maps, which revolutionized the way we navigate and the way we travel, - and therefore the field of Cartography was born. During the 19th century, terra incognita *(orig. Latin, 'unknown land.')* disappeared from maps, since both the coastlines and the inner parts of the continents had been fully explored. Today, using technology and satellites, Earth is completely mapped [2], yet still not completely explored [3]. Around 95 percent of our oceans still remain unexplored, considering that they take up 70 percent of the planet's surface [4]. This is mostly due to extreme conditions and high depths, that humans can not be put up against. As technology advanced and matured, remotely operated underwater vehicles (ROVs) became a popular way of exploring the depths of the oceans, due to their accessibility to places where humans could not possibly go before for a long period of time. In the meantime, interplanetary exploration took off during the mid-20th century with the start of the Cold War between the United States of America and the Soviet Union. This led to many great achievements in the field of rocket science and space exploration, like the first successful interplanetary encounter, where the US Mariner 2 flew by Venus [5] or when the Soviet Venera 3 made first impact on the surface of Venus [6]. One of the latest successful programs is the Mars rover CURIOSITY [7], which landed and still is making progress in exploring and sending back scientific data of the planet Mars. Exploring planets for future-settlements and first-contact is very important in order to quench out human curiosity but also for advancing the different fields in science and technology.

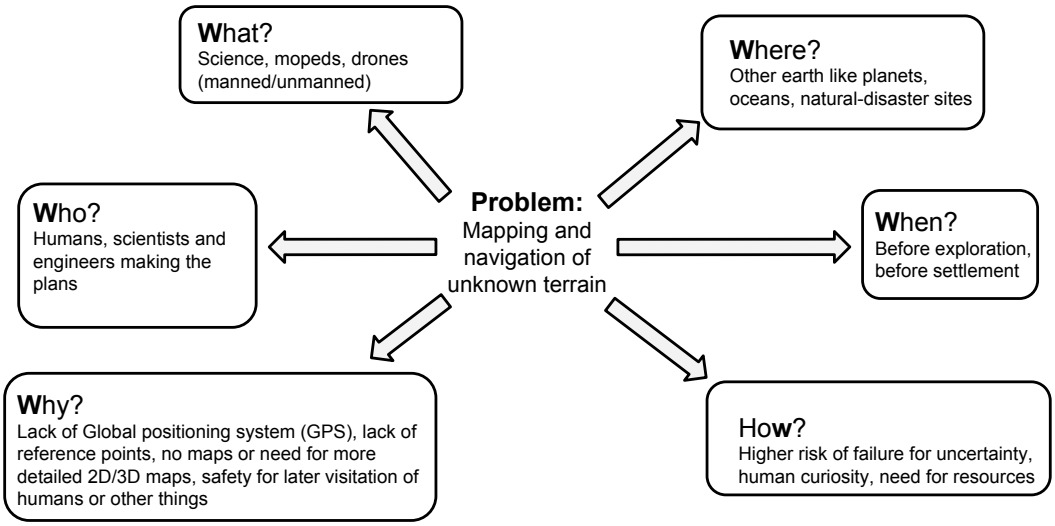


Figure 2.1: W-diagram

2.0.1 When is this actually a problem?

Mapping unknown terrains becomes a problem when new planets are discovered and mapping of the surface and its atmosphere is necessary for further exploration and investigation of the planet. Scientists use the so-called Earth Similarity Index (ESI) for mapping extrasolar planets (also called exoplanets) for potentially habitable places in the Universe [8] [9]. The ESI index implies many factors, like surface temperature and other Earth-like properties.

Another scenario is underwater exploration, where trained divers are unable to explore and map the underwater terrain due to its depth and dangerous circumstances. In these cases robots and submarines, that are either autonomously or remotely controlled, can take up the task of mapping the bottom of the ocean. It is also much faster and more reliable than sending down people to do the same task.

Besides exploration, the task of mapping unknown terrains is used in the event of a hostage situation or natural disaster as well, where people cannot enter a specific area without it being mapped upfront for various factors, like radiation leakage or armed terrorists. One example would be the Fukushima Nuclear Plant catastrophe in Fukushima, Japan on March, 2011. After the leakage in the reactor, rescue forces and scientist used autonomous drones to map the inside of the factory [10], which was put under quarantine for high levels of radiation and therefore no person was allowed inside of the building without proper precautions. Drones and autonomous robots are perfect for scenarios like this.

2.0.2 Where does this problem actually occur?

An unknown terrain can be defined as a terrain where no mapping has been done before.

The bottom of the ocean is mostly sand, mud, and water [11]. This makes it difficult to navigate with a rover, but relatively simple with a submarine. In water, sensors that use sound, need to be calibrated to work with the correct pressure of the surroundings, which varies with the depth of the water. If the water is murky, light-based sensors may also not work, depending on the wavelength of the light they use. Light also refracts in water depending on the water density. Since the bottom of the ocean is not very difficult to reach by cables or certain wireless signals, it is possible to remote-control vehicles that are there.

Other planets can be much more complicated to navigate. The consistency of their atmosphere and terrain may be not completely known. This means that there is a higher chance of a rover being stuck on loose terrain, falling down sheer faces, and so on. It may also be the case that the planet has no atmosphere, so a drone would not be able to hover. One of the biggest issues with extraplanetary exploration is input lag. Other planets are very far away, which means that signals travelling to these distant places can take several minutes at least and years at most to reach from planet Earth. Thus automation is better suited for this task. Other planets also share the issues that the bottom of the ocean has, namely that sensors need to be calibrated to the different atmospheric levels and other characteristics of the planet.

Disaster scenarios can also be classified as unknown terrain. This means that the terrain might be loose or difficult to navigate, so a drone might be best suited for it. In most cases, disaster scenarios happen in locations where a vehicle can be controlled manually.

2.0.3 How is it a problem?

To answer the question of how this is a problem, we have to look into some possible scenarios discussed above. Exploration of unknown terrain and environments has been practised for many years now. It originates from the human curiosity and the spirit to explore and gather information about our surroundings, as discussed previously.

When dealing with unknown environments, the word *'unknown'* does not strictly refer to the environment itself but the persons state of knowledge about the physics and composition of the given environment. In a known environment, outcomes from every action can more or less be calculated or estimated. Where as in an unknown environment, it is a matter of investigation and figuring out what works and what does not. In the unknown environment it is important to gain knowledge of how everything works, so that in the future it is possible for an individual to take the best possible choices and decisions in the said environment [12]. Even though most

of the land has been explored and is being used for its vast amount of resources, the time will come where planetary and ocean exploration becomes a key factor for our technological advancements and our own resources on planet Earth. Ocean exploration is important because it provides data from deep-sea areas, which in turn will reduce the amount of unknown environments left on our planet. Gathering data and intelligence from the ocean also helps with managing the resources that are available in the deep-sea areas, so that future generations can benefit from them. The ocean also provides information about future environmental conditions and can help predict earthquakes and tsunamis. Investigating the deep-sea also reveals new ecosystems and possible sources for medication, food and energy, which are all vital for scientific advancements or survival on Earth [13].

Humans have always had never-ending interest and the need to push science and technology beyond its limits, and a desire to achieve something even further than what is possible. The many challenges humans have faced so far, has led to many benefits for our society since its creation. Space exploration helps to further our understanding about the history of our universe and solar system [14].

2.0.4 To whom and what is this a problem?

Unknown terrains and environments pose a big issue for scientists and engineers who want to explore these areas. Designing vehicles and devices for the deep-sea ocean or planetary exploration is impossible without any background information on what environmental factors they will be dealing with or encountering during the set mission. Exploration is essential when, in the future, new resources are needed for scientific and technological advancements, that are currently out of our reach.

In the long run, humans in general will be affected by the lack of exploration. Alternative resources and habitable areas for expansion will be necessary in the future, when Earth's natural resource-deposits become depleted and no habitable places are left.

Scientists are heavily hindered by society, because it is becoming too focused on risks. Only 5 percent of the ocean has been explored so far, which leaves a large amount of areas untouched and unmapped. Being concerned about taking risks is what will put the future development and science in jeopardy [15].

2.0.5 Why is this a problem?

The reason why there is a need to explore other unknown terrains is to help better understand everything around us and to be able to go to other planets. If the trips are well planned and a sufficient 3D map of the terrain is generated, it is possible to more optimally plan-ahead for the next mission or to determine if the environment is worth exploring further or visiting once again.

The problems with navigating a robot in an unknown terrain is the lack of reference

points that are needed to make a 2D or 3D map and reference points for the robot itself to being able to navigate and find its way around. This creates two challenges: one, is for the robot to be able to navigate through a completely unknown terrain and secondly, to make sense of it and come up with reference points and data that can be converted into a map.

If a robot is successful at navigating unknown terrains and gathering data, it can be turned into a detailed map, where the next mission can rely on that information and have a easier way to navigate and possibly a more successful mission.

On Earth, we have the GPS (global positioning system), which is a space-based satellite navigation system placed above Earth [16]. Putting up a system like that on another planet would cost a lot of resources and is not beneficial for first-time exploration missions. GPS also does not penetrate water [17].

2.1 Preliminary research

2.1.1 GPS theory

"The Global Positioning System (GPS) is a space-based satellite navigation system that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites. The system provides critical capabilities to military, civil, and commercial users around the world. The United States government created the system, maintains it, and makes it freely accessible to anyone with a GPS receiver." [18]

The GPS satellites each carry an atomic clock and positions the user when connected to 4 or more satellites. The satellites measure the time it takes for the signal to go back and forth between the satellites and a GPS receiver [19]. There is also a space "GPS" system in development, that would use the X-ray signal from dying stars to give you the exact location in space with an error rate of about 5km [20].

Either of those systems would not be beneficial for our project in space, since we would need a high-accuracy location of each picture we take to make a map and a model for navigation purposes. Later in the future when people will *hypothetically* be moving to Mars, another GPS-like system would have to be put in place up for navigation purposes. For now, we would need to use encoders and short-to-medium range, high-accuracy signals. But these signal systems needs to be different for every planet we would go to. [19]

2.1.2 Photogrammetry theory

"Photogrammetry is the practice of determining the geometric properties of objects from photographic images. This is done by comparing and matching pixels or reference points across a series of photos." [21]

As a simple example, a map can be constructed from a series of images by overlaying them in such a way that the areas the pictures have in common, overlap. The distance between two points that lie on a plane parallel to the photographic image plane can be determined by measuring their distance on the image, if the scale (s) of the image is known. This is done by multiplying the measured distance by $1/s$ [22] [21].

"Algorithms for photogrammetry typically attempt to minimize the sum of the squares of errors over the coordinates and relative displacements of the reference points. This minimization is known as bundle adjustment and is often performed using the Levenberg–Marquardt algorithm." [21]

Photogrammetry could also be used in our project with an AUV or a rover. There is no need for outdoor reference points to make the 3D model. But this technology is much more complicated to set-up and maintain without an outside assistance, which is a major inconvenience when used for space exploration. It would also take a large

quantity of pictures in a high resolution, so in turn more space on hard drives are needed. The algorithm needs to create the 3D model from the pictures, which will need a lot of computing power and memory. So doing on board rendering would be really hard and resource consuming for smaller computing units. [22]

2.1.3 Reference points theory

To make a 2D or 3D map, a reference point is needed. It can be GPS positions of every picture taken or it can also be by comparing two pictures and overlapping them. It is also possible to combine both options, but with the lack of GPS in space and the cost of memory and power to make an overlapping 3D map, another solution is needed. For an example, Mars has a thin atmosphere, so using sound sensors could be a possibility. The sensors would need to be calibrated to work correctly on Mars, since there is a difference in the speed of sound. Other sensors such as light (X-ray, laser, etc.) can also be used. That would mean setting up and array of sensors, so a rover or an AUV could potentially connect to them and know their position. The use of encoders and the wheels of the rover is also another solution, but this requires error calculation in it, since sometimes the tires will attempt to move in a circle but there is no traction to help create the movement [23].

2.1.4 SLAM

Simultaneous Localization and Mapping (SLAM) is a particular method used to construct maps. In SLAM, the mapping device maps its environment whilst simultaneously keeping track of its location within that map, hence the name. SLAM is used mostly in autonomous vehicles that need to traverse some terrain without any prior information about it. More precisely, SLAM is a combination of algorithms, most notably *Kalman filtering*, that all relate to either mapping or navigation. The specifics of these algorithms are too complex for the purpose and timescale of this project.

2.1.5 Sensors

Pressure Based

Pressure-based sensors work by simply measuring how much pressure there is in a point in space. They can be used to measure the vertical distance between two points in a planet with an atmosphere [24] [25]. By combining height information with the data gathered by other sensors, it is possible to create a crude three-dimensional map.

Image Based

Image-based sensors are often cameras or camera-like equipment. The pictures that the sensor takes, must be mathematically processed to yield any useful data. The two most common types of image sensors are stereoscopic and projected image sensors.

A projected image sensor is where some known image is projected on the environment (a grid, for example), then a camera takes a picture. Since the projection is known, it is simple to detect the distance of an object on the image, based on what the projection looks like. This method is limited to how fine the projected image is.

Stereoscopy, also known as stereo-photogrammetry, is more complicated, but also more reliable. Two photos of the same scene, but from different vantage points, are taken. Small sections of one of the images are taken, and overlayed onto the other image. By finding the place on the other image where the small section fits, the algorithm can determine the depth of each pixel in that subsection. This procedure is repeated until the whole image has been processed.

2.1.6 Optical-flow sensor

An Optical-flow sensor is a camera mounted on a PCB board that compares changes in pixels to look out for drift and movement in a system. The concept is the same as your computer mouse follows. It will tell you the movement of your vehicle in a XY-grid.

2.1.7 Rangefinder

A rangefinder is a device that measures the distance between itself, and a point at a certain distance away from it. Rangefinders work based on the principle that the speed of an object is defined to be *distance travelled over time travelled*, which means that the *distance travelled* is the speed of an object times the amount of time it travelled. To put this in math terms: $v = d/t$ or $d = t * v$.

Most rangefinders work on either sound or light. Both sound- and light-based rangefinders work by emitting a pulse in a specific direction, and counting how long it takes for the pulse to come back. Since the pulse has travelled back and forth, the distance is calculated as $\frac{1}{2} * v * t$, where v is the speed of the pulse, and t is the time between the emission and the detection.

The pulse will travel from the rangefinder, straight towards whatever object is in front of it. Often, the pulse is not normal to the object it hits; meaning that the pulse is at an angle relative to the object, so the pulse will not reflect back to the rangefinder. However, a small portion of the pulse will scatter upon impact, meaning the rangefinder might still detect the pulse once it comes back.

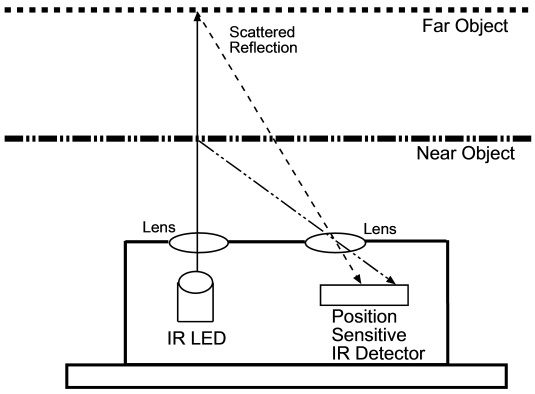


Figure 2.2: A slightly more complex laser rangefinder. [26]

In this picture we see a different setup for a rangefinder. Instead of measuring the time it takes for the pulse to get back, it measures the distance between where the pulse was emmited, and where it was absorbed. Given this distance, and the fact that the pulse is emmited at a right angle to this distance, simple trigonometry can be used to determine the distance the pulse travelled.

It is required that rangefinders know what the speed of the pulse is, which is a problem in unknown environments. For a rangefinder to be reliably used in such environments, it needs either to have the speed of its pulse pre-calculated, or it needs to measure the speed of the pulse by itself. Also, in situations where the speed of the pulse varies, a rangefinder is either not useful, or it needs a more complicated model of the environment it is in.

The main cause for a pulse to travel at a different speed than expected is in situations where the environment the sensor is in varies in density, which depends on temperature [27]. Hence, a simple rangefinder is not fit for an environment that varies heavily in temperature.

Sound moves faster through denser mediums, where light travels slower. The speed of light is less affected by density than sound is. In earth-like atmospheres, light only travels 0.03% slower in air when compared to the vacuum. [27] [28].

Another issue in environments with a high variation in density is refraction. Refraction is a phenomenon that occurs when a wave switches between mediums where it travels at different speeds. This change causes the wave to change its direction [29]. Refraction would cause rangefinders to become effectively blind, as neither their direction nor the length measured can be trusted.

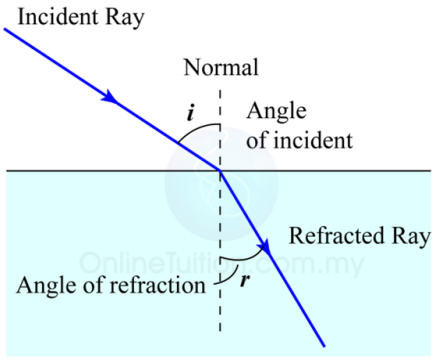


Figure 2.3: An illustration of refraction into a denser medium. [30]

We can categorize reflectivity of an object or surface into three categories: [31]

- Diffuse reflective
- Specular
- Retro-reflective

Just to expand upon these categories, it is possible to look at the reflective characteristics of an object's surface.

We find **diffuse reflective** objects to have a textured quality, which causes the reflected energy to disperse uniformly from the object. These materials tend to be read very well by a large number of laser sensors, due to the relatively predictable percentage of dispersed laser energy finding its way back to the laser receiver. Some examples of these objects would be: paper, granite and textured walls. It is important to note, that different wavelengths may exhibit unexpected results with certain receivers.

Specular objects or surfaces tend to not read very well, due to the emitted energy not being dispersed correctly. Examples of specular surfaces are glasses and mirrors.

With very reflective properties, we categorize objects and surfaces as **retro-reflective**. These usually return a very high percentage of emitted energy to the receiver, therefore are a good target to many lasers. Some examples would be: reflectors, license plates and animal eyes.

2.1.8 Autonomous Vehicles and Robots

Autonomous robots or vehicles can work for an extended period of time, gathering information from its surroundings, while being able to work without human interaction. The robot or vehicle gathers different kinds of data, depending on what the goal of the device is. Positional data can be utilized for navigation and path finding in a known and unknown environment. Intelligent autonomous robots and vehicles are able to adapt to changes happening in its surroundings. Currently there are

many robots on the market that are self-reliant, ranging from autonomous vacuum cleaners to drones and helicopters [32].

Simple autonomous robots use ultrasonic sensors or infrared LEDs to manipulate its own behaviour, if it detects obstacles or unexpected scenarios in the environment. This is useful for obstacle-avoidance and mapping of unknown areas, where the robot, through different reference points, can pin-point the obstacles it has encountered along the way and in the future, more efficiently avoid them. More advanced robots use more advanced vision to grant them the ability to see their surroundings. Algorithms then analyse the vision data and gives the robot a depth perception, which grants the ability to instantly identify objects and locate them and memorize them accordingly [33].

There are two main kinds of autonomous robots: a single computer-autonomous robot and insect robots. The single-computer autonomous robot uses its own on-board computing unit to do its computations and its decisions, whereas the insect robots are a fleet of many smaller robots, which are controlled by a single and separate computing unit. The advantage of having a single-computer autonomous robot is that the tasks it performs can be done using more dedicated computer resources. It has the possibility of utilizing the computing power to its full potential, instead of relying on a separate unit that is also making decisions and calculations for many other robots. The insect robots are usually fairly simple in terms of capabilities, but the whole robot fleet can be utilized to perform advanced and possibly more sophisticated tasks, that requires a fleet simpler robots designed to work together autonomously [34].

2.1.9 Pathfinding and Mapping

Pathfinding is done by a computer, where it uses plotting to find the shortest path between two different points. It can be viewed as an efficient way of navigating a maze. The main objective is getting from some location to a goal location. There are different pathfinding algorithms already existing that are used for software simulations, but also for mobile robot navigation. Common pathfinding algorithms are A* (pronounced A star) and its extended version named D* (Dynamic A*).

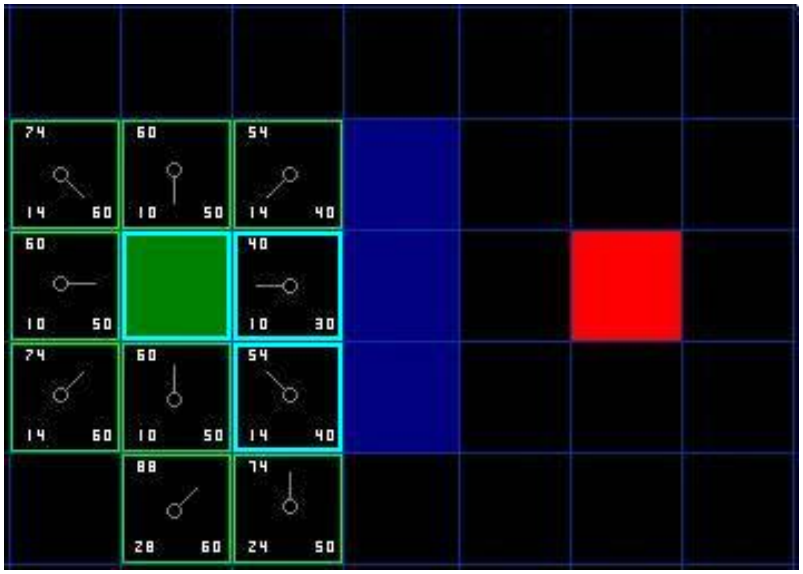
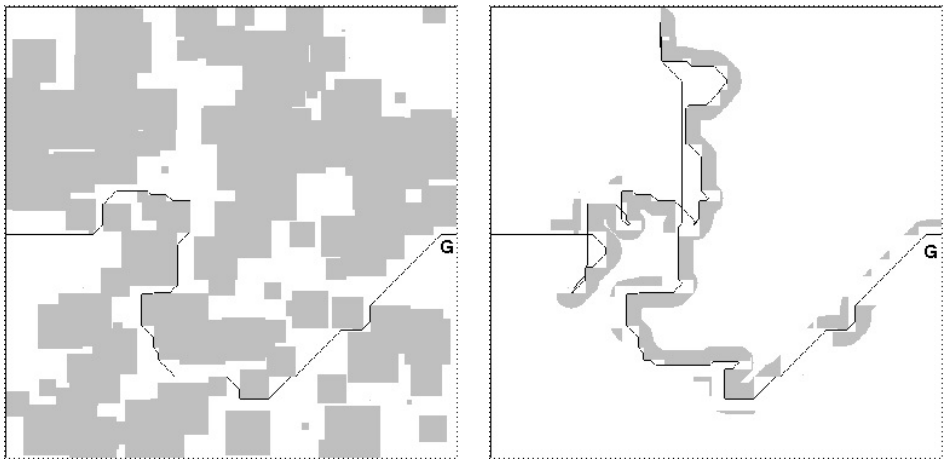


Figure 2.4: Upper-left number is F, Lower-left G and Lower-right H [35]

The A* algorithm is most commonly used in video games. The algorithm splits a known map into nodes (these can be any geometric shape, for example squares), and assigns a cost between each node. It also notes which nodes cannot be moved to, such as pits or walls. Based on this map, it determines the cost of movement from its position to some adjacent node(G), and estimates the cost of movement from that node to its final destination(H). It assigns this node with the value F which is the sum of G and H. It continues checking adjacent nodes with the same procedure, occasionally changing the calculated path based on the F values of various nodes [36].

In the early stages of robot automation, it was always assumed that the environment around the robot was known and the path was generated based on this. This was okay until the robot was introduced to the unknown, and then started to meet inconsistencies and obstacles in its path (this being discrepancies between the true state and the world state), the robot then either had to re-plan completely from scratch or alter the plan through trial-and-error. Computational wise, it would require too much power to re-plan from scratch (Even though this is the optimal choice) and most of the time trial-and-error does not yield an optimal outcome.



(a) Complete Map **(b)** Optimistic Map

Figure 2.5: Mapping simulations with D* [37]

The D* algorithm creates an initial path, based on assumed and known information. It then edits the path based on new information it receives whilst navigating the path. The final version of the edited plan is equivalent to re-planning the whole thing from scratch, but avoiding the limitations of the computation power. Whenever an on-board sensor receives information about an obstacle, the algorithm can edit the path based on what possible routes there are available near-by. D* is fast at navigating large scale unknown environments because during its exploration it is able to edit and repair its desired path and navigate towards its goal. It would require too much effort to re-plan everything from scratch every time it failed, especially for large scale environments [37] [38].

Figure 2.5a shows the map where the given robot knows its environment (the gray boxes) and can therefore easily determine the optimal path or the *Complete Map* in this case. Figure 2.5b on the other hand shows the *Optimistic Map*, since before navigating on the planned path it assumes that there are no obstacles. Whenever the robot meets an obstacle during the optimistic map, it adjust its course to continue towards its goal. On figure 2.5b it only displays the obstacles which the robot met during its journey along the path.

Simple autonomous robots navigate by the use of infrared LEDs or photo-resistors and LEDs, by following lines drawn on a surface. Robots that use photo-resistors to follow a line are continuously looking for a change in brightness of the surface. If a line-following robot is tracking a black line, then whenever the photo-resistor picks up the brightness from the surface that is not a black line, the path will change accordingly. A large number of photo-resistors can be used to increase the intelligence of an autonomous line-following robot, since the greater amount of LEDs can detect intersections between lines and different routes and be identified [39].
Some autonomous robots and vehicles use multiples of different range-sensors and other sensory equipment, to map and locate themselves in indoor and outdoor environments. The map that is generated can be used to keep track of static items in the environment such as structures and difference in terrain, but the map can

also distinguish non-static items such as humans and other moving objects, from the static items. Since the maps are created by the vehicle itself whilst exploring, this technology can be used in places where there are no reference points, such as GPS [40] [41]. Robots and vehicles have been created using this technology to explore known and unknown environments. Because of advanced algorithms and hardware these autonomous devices are capable of performing some tasks more efficiently than humans, but are also able to perform them in places that are unsafe and hard to reach.

Autonomous robots also use these tools to work together. Using a shared map, the robots can keep track of one another and either perform tasks together or separately, depending on what is required from them.

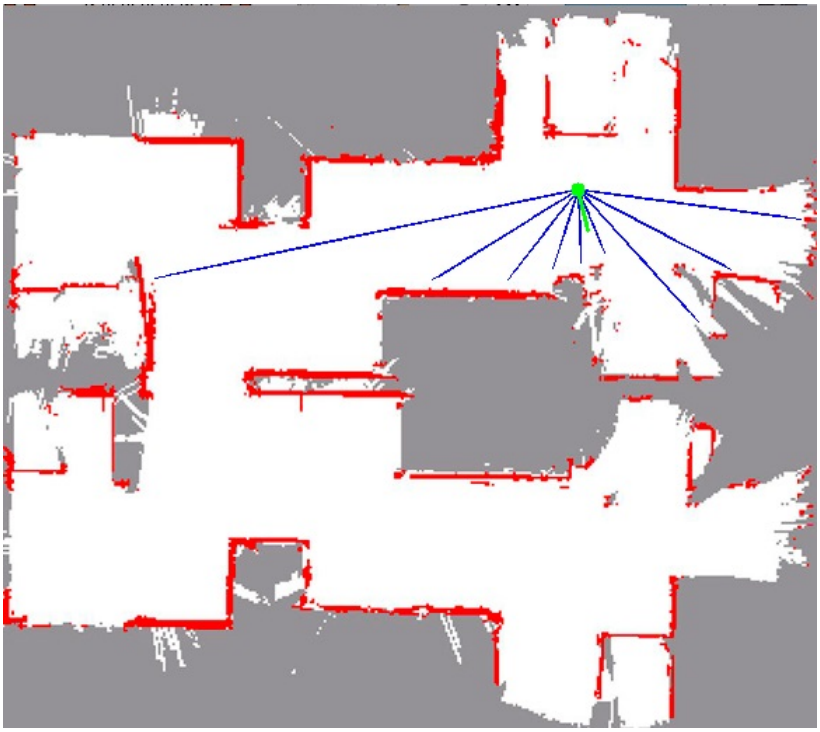


Figure 2.6: Laser Range Map [42]

Laser-range finders and sonar arrays are used to navigate and determine the shortest possible path to a given destination. The sensory equipment is used to give the autonomous robot a sense of distance towards objects in an environment, giving it vital data regarding optimal travel directions and information on how to avoid obstacles [43].

2.1.10 Resources

For the purpose of exploration and resource gathering, planets can be split into two categories. These two categories are: [44]

- **Terrestrial planets**, also known as the *Inner Planets*

- **Gas giants**, also known as the *Jovian planets* or the *Outer Planets*

There is very little reason to explore a gas giant, as they smoothly transition between gas, liquid, and solid. This means that it is extremely difficult to colonize such a planet, or build anything on it for that matter, due to the layers of condensed helium and hydrogen that make out their atmosphere [45]. On the other hand, Earth-like planets have a larger variety of resources, and it is easier to build upon them. These planets are often composed largely of metals and silicon, which is what gives them their rocky and sandy surfaces.

2.1.11 Environment

Atmosphere

There are quite a few planets, in our solar system, with a negligible atmosphere. On these planets sound based sensors cannot work, but light based ones might even work better. These planets also tend to have a temperature very close to absolute zero [46]. Planets with an atmosphere can be much harder to cope with. On these planets there can be threats such as, high-winds, high-pressure, high-temperatures, corrosive gases, and liquids. When designing equipment for such environments, the specific environment must be kept in mind.

Terrain

Earth-like planets are, by definition, rocky/sandy planets, which tend to have mountains, canyons, and craters. Earth-like planets do not have a very big variation in terrain, as most of them do not have any liquid substances that could change the terrain.

2.2 Stakeholder analysis

2.2.1 Interested parties

Interested parties are the groups of people who are being affected either negatively or positively by the problem in question. They have some influence over the development, as the trends and actions of these interested parties has to be taken into consideration by the development team and the people involved with the project. They do not necessarily participate directly in the project.

NASA

The National Aeronautics and Space Administration, or more commonly known as NASA, is a governmental agency in the United States, responsible for aerospace research and civilian space programs. Many manned- and unmanned missions to new planets require extensive research of that planet prior to the mission. It might even require sending exploratory rovers or satellites to examine the environment and to map out possible landing sites. Rovers are commonly used and equipped with similar technology that we are producing, as in all cases, the environment is unknown, external technologies, like GPS are not available and no maps are generated of these planets without previous research missions.

ESA

The European Space Agency or simply ESA, is the European counter-part of NASA, governed by 22 member states, with the aim of space exploration and research. Similar interest applies to ESA as to NASA, therefore they can be classified as an *Interested party* in our assessment.

The Royal Danish Army

The Royal Danish Army, or simply The Army, is the land warfare branch of the Danish Defence Forces, together with the Danish Home Guard. Their main tasks are to prevent conflicts and wars, through crisis management and co-operation with NATO and allied forces [47]. In case of a conflict or war, drones are commonly used to map out potential war sites before a planned attack or defence, thus making them a good candidate at potentially implementing our technology.

The National Police

Also known as *Rigspolitet* [48], The National Police is the national police force of Denmark, also having authority over regions governed by The Kingdom of Denmark, the Faroe Islands and Greenland. As the nations main police force, it is part of

their duty to avert any danger or possible terrorist attack in order to protect the population and public peace. As we identified dealing with terrorist attacks and hostage situations as a possible application for our technology, The National Police would be a potential user of such technology in these unfortunate events, assisting them with mapping a building where a kidnapping might take place or any other criminal offence, as an example.

2.2.2 Actors

We define **actors** as a subset of *Interested parties*, who have any influence on the development of the project, either by financing it, providing knowledge or help the development of the project.

Supervisors

The supervisors involved with this project are Akbar Hussain and Torben Rosenørn. Their involvement consists of giving directions regarding where to look for knowledge, having a general overview of the process and evaluating the end results of the project. They also provide valuable feedback and help through supervision to Group H103 during the duration of the project.

Department of Electronic Systems, AAU

The Department of Electronic Systems is one of the largest departments at Aalborg University with a total of more than 300 employees. The department is internationally recognized in particular for its contributions within Information and Communication Technology [49]. As the project relates to a study line which is under the Electronics department, and therefore the tools, equipment and funding necessary for the project potentially could be funded by the department.

2.2.3 Technology carriers

We define **technology carriers** as a subset of *Actors* who have influence to change the direction of the project, be involved with it or be a potential customer or partner in developing it even further.

Danish Explosive Ordnance Disposal squad

Also known as *Ammunitionsrydningstjenesten* in Danish or *EOD* for short, refers to the Danish bomb disposal squad [50]. They are the one called in for emergency situations or bomb scares, where they use robots to map the area and find the possible suspicious items to investigate. The function of this squad is very crucial to

the Danish Army, as these men and women need special training and education to work with these tools and technologies to ensure the public safety in these special cases. The interest that the Danish EOD squad would have in this project would be in regards of mapping technology and autonomous navigation in unknown terrains. Such technology could be used on rovers that bomb squads generally use for going in to building to find possible threats and to investigate it. This could mean they would have possible interest in the development, customization of the project to their needs and even financial interest. They are also considered as a potential users and carriers of this technology.

Group H103

Group H103 is the group who develops the solution for this project. The group consists of the following members:

- Antal János Monori
- Emil Már Einarsson
- Gustavo Smidth Buschle
- Thomas Thuesen Enevoldsen

2.2.4 Summary

Name of stakeholder	Type	Role
Group H103	Technology carriers	Developing the project
Danish Explosive Ordnance Disposal squad	Technology carriers	Knowledge sharing and potential partner/client
Electronics dept., AAU	Actors	Funding
Supervisors	Actors	Knowledge sharing and assistance
NASA	Interested parties	Potential partner/client
ESA	Interested parties	Potential partner/client
The Royal Danish Army	Interested parties	Potential partner/client
The National Police	Interested parties	Potential partner/client

Table 2.1: Stakeholder table

The table above contains the names of the different people involved with or interested in the project, with the type, role and impact of their involvement specified. The type of the stakeholder represents a level of involvement within the project, meanwhile the role represents a more concrete function.

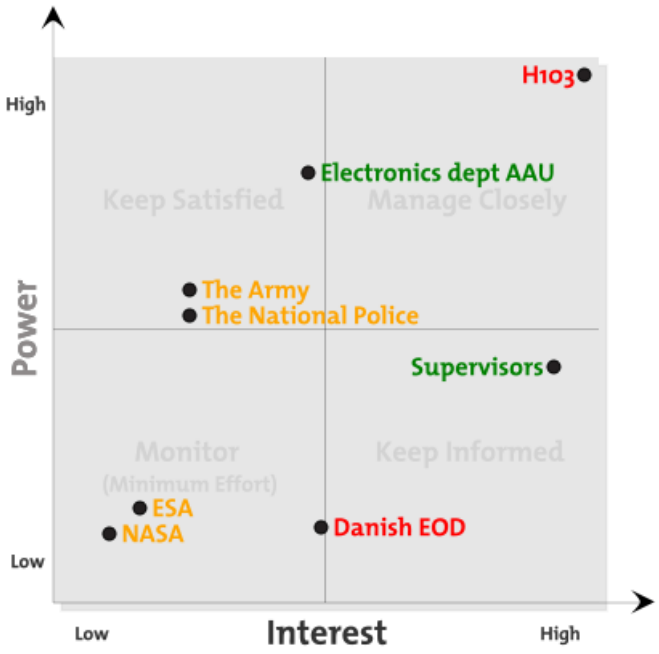


Figure 2.7: Stakeholder analysis diagram

Figure 2.7 showcases how do we categorize our stakeholders on the Power/Interest grid, used commonly for stakeholder analysis. The different colors used for distinguishing between the three types of stakeholders we identified: orange for *interested parties*, green for *actors* and red for *technology carriers*. The for sections can be defined as:

- **High power, interested people:** these are the people you must fully engage and make the greatest efforts to satisfy.
- **High power, less interested people:** put enough work in with these people to keep them satisfied, but not so much that they become bored with your message.
- **Low power, interested people:** keep these people adequately informed, and talk to them to ensure that no major issues are arising. These people can often be very helpful with the detail of your project.
- **Low power, less interested people:** again, monitor these people, but do not bore them with excessive communication.

2.3 Risk management

Possible risks that could occur before, during and after the implementation of the proposed 2D mapping system, elaborated in this report, and presented in the form of a table together with a mitigation plan for each and every identified risk.

We listed most of the risks that would influence the design process of our product. These risks will be used as criteria for the end-product.

Risks are considered on two different scales with 4 levels. As the legend under the table shows, it starts from Very High (VH) being the most severe to Low (L), which can be considered as a minor issue. The two scales are **Probability**, which represents the chance of occurrence, and **Impact**, which represents the severity of the risk in case it would actually appear.

Risk factor	**Probability**	**Impact**	**Mitigation strategy**
Going over the budget	M	VL	Carefully select the products and make sure we order the same models as specified in the excursion request form
Parts not arriving in time	L	VH	Estimate a 2 week delivery time for each package
Short-circuiting electronics during development	VL	H	Buy a new one or consider replacing it with similar hardware.
Lack of knowledge about software	M	H	Find and compile a list of contact persons with knowledge about it and time to support us
Compatibility issues or lack of support for hardware	L	VH	Research before ordering/selecting parts; otherwise replace
Too big of a scope and too little time to complete	H	H	Cut down on scope and write a solid argument in the discussion about it

Table 2.2: Risk management and mitigation

Legend: VH - Very High | H - High | M - Medium | L - Low

Chapter 3

Problem Definition

3.1 Conclusion of Problem Analysis

During the research for this project, we have decided on what technologies we will use for solving the problem stated in the W-diagram (Figure 2.1).

We will use a rover with multiple sensors to navigate around in a set environment. This rover will produce a 2D map of its surroundings, with the possibility of implementing equipment for 3D mapping. We will use both sound and light sensors for navigation and mapping. Ultrasonic-sensors will be used for close encounter manoeuvres for their high precision at a close range (up to 4m) [51] and a optical flow sensor will be used to measure the distance travelled. For mapping we will be using a laser sensor, that will either automatically turn 360° or we will implement a motor to rotate the sensor.

The rover will both be remote controlled and autonomous for navigation. For reference points we will use the optical flow sensor, located on the bottom or in front of the vehicle to help determine our current location and the distance travelled from the starting location.

The controller used in this prototype will be a Raspberry Pi running an optimized version of the Debian Linux operating system, called Raspbian. On top of the operating system, we will install the ROS (Robot Operating System) framework, which provides libraries and tools to help software developers create robot applications [52]. The reason for using the Raspberry Pi is that it can multi-task, read many sensors at the same time and build the 2D map on the go, with enough processing power and resources at hand. Whereas for other microcontrollers, like the Arduino, we would only be able to collect data and need to build the map from it later, using another computer.

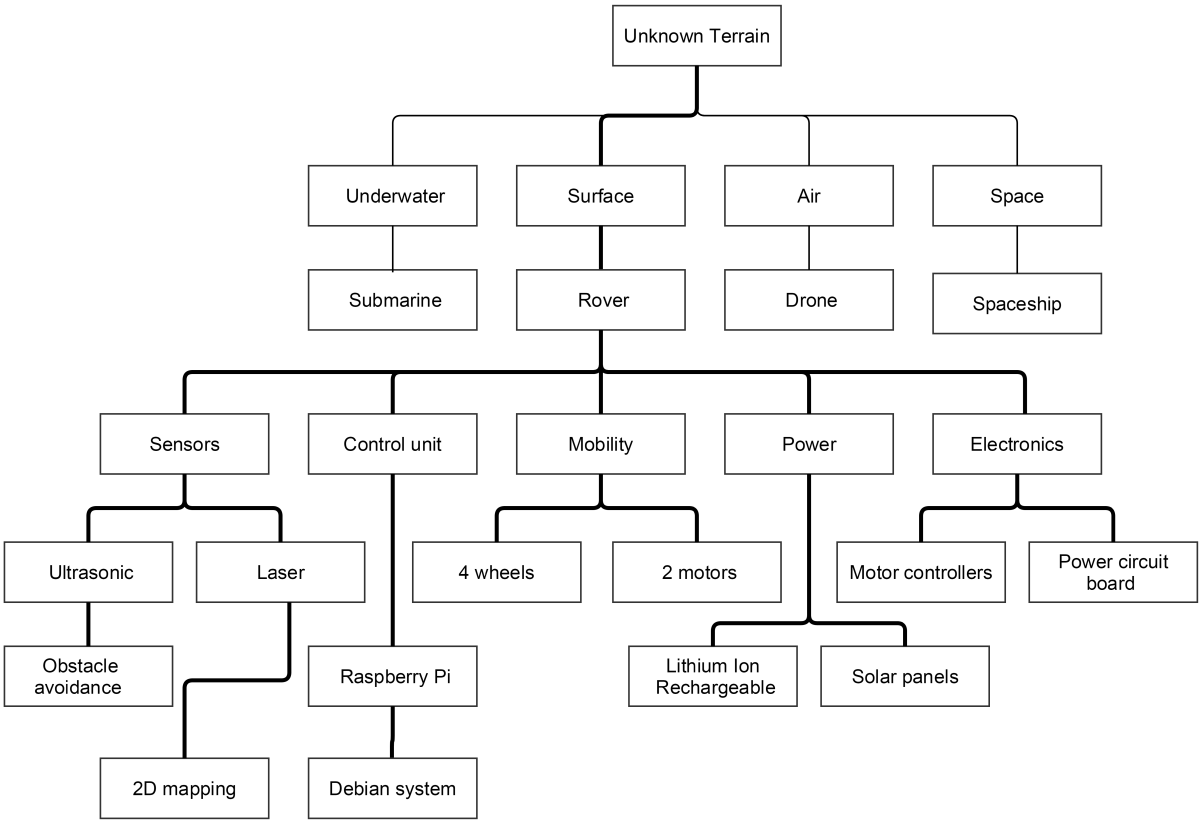


Figure 3.1: Flowchart of the prototype idea

This picture shows how the flowchart from Chapter 1 was expanded. We decided to focus on rovers, and created the prototype requirements accordingly.

3.2 Problem Delimitation

Based on previous discussions, the following equipment has been selected:

Rover:
A 4 wheel rover to move the sensor around.

Laser:
LIDAR-Lite rangefinder laser, used for distance measurement from points.

Brushless DC motor:
Used to spin the laser a full circle.

Ultra-sonic sensor:
For close proximity navigation.

Raspberry Pi:
As a control computer for navigation and processing of 2D map.

Optical-flow sensor:
Used for looking at a movement of the rover for reference points.

Motor controller:
Used to power the motors used on the rover.

Power source:
Used to power the various microcontrollers and equipment.

3.2.1 Prototype delimitation

The main limiting-factor for us is time. In the time-frame of the project, we cannot build a full-scope product. Finding all the parts, ordering, and assembling them completely would take too long to complete and over our given budget. Furthermore, we do not have the tools or place to build a full-scope product, either. Hence we will build a prototype as a proof of concept using the means we have available.

For the prototype, we decided to use a laser to gather the data needed for a 2D map as a proof of concept, but at the same time allowing the capability of implementing cameras for 3D mapping. A laser can also be used for 3D mapping, but it would be easier to make a 3D map from cameras and using cameras would also give a better picture of the surroundings.

The optical flow sensor is used to determine reference points. We will be using Simultaneous Localization and Mapping (SLAM) technique for the actual map creation. The laser will be mounted on top of the rover and spin 360° using a stepper motor. The optical flow sensor will tell the system about the movement of the rover on a XY grid, to determine the direction and distance the vehicle has travelled.

This prototype will not have all the features we wish to have for the final product, but the goal is to show the interaction between an laser sensor for 2D mapping and

a rover navigating through unknown terrain, using a Raspberry Pi.

3.3 Requirements

This section outlines the requirements for both the prototype that we are aiming to achieve through this project, and the requirements for testing and evaluating it.

3.3.1 Prototype requirements

1. The rover should be remotely controlled.
2. The rover should be autonomous.
3. The laser sensor should produce a 2D map of the surroundings.
4. Running a Raspberry Pi that controls mapping and navigation asynchronously.
5. The program should be written in Python.
6. The system should include a laser sensor.
7. The system should include ultrasonic sensors.
8. The prototype should be capable of implementing a 3D mapping sensor without too much change.

3.3.2 Testing requirements

1. The rover can be controlled using a remote device.
2. The rover should navigate autonomously through a maze.
3. The laser sensor should make a usable 2D map of its surroundings.

These criteria will be used to design, build, and test our implementation of the prototype. These requirements will be re-evaluated in the *Testing* section, Chapter 5.

3.4 Hypothesis

Due to the fact that our report deals with such a broad scope, we can not completely cover our initiating problem in such a short time. Therefore, a hypothesis is used to measure the success of our project.

The solution to the initiating problem discussed in this project, is to make a rover capable of 2D mapping of unknown terrains and finding its own path to navigate across it. Then provide a usable 2D map of the surroundings from point A to B.

The rover should also be capable of implementing equipment for 3D mapping in the future.

3.5 Problem Definition

The scope of this project is to create a *working and affordable* prototype of the above delimited problem, that would showcase the main capabilities of the end product, and would validate the idea of solving the presented problem in Chapter 2. Through testing our hypothesis and by fulfilling the requirements displayed above, we will be able to determine whether or not the project is a success.

Chapter 4

Development

4.1 Description

The figure below shows the modules we are planning to develop during the Development phase of our project. Three sensor modules will be developed to return readings to the controller running ROS, one navigation system which will be relying the ultrasonic sensory readings and a remote module for optionally controlling the rover by a remote. These modules may be prioritized accordingly to their importance and some of these modules might not be developed fully in the end.

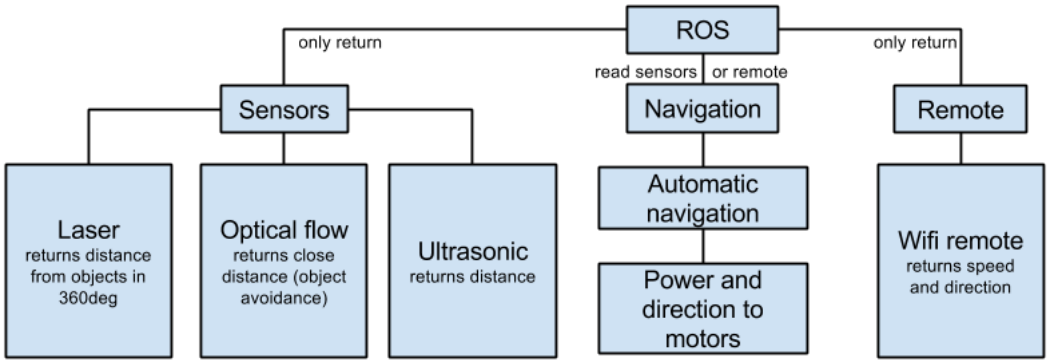


Figure 4.1: Diagram of modules we are developing

The rover will consist of two main parts: A LIDAR-Lite laser module, recording distances and creating a map using the SLAM technique and a close proximity detection and navigation system.

4.2 Microcontroller

Our microcontroller of choice, the Raspberry Pi 2 Model B, is a preferred choice for embedded robotic systems. One of the deciding factors was the GPIO capabilities on the controller and the interface that it has to offer. It also features a powerful 900MHz quad-core ARM Cortex-A7 CPU and 1 GB of RAM. Because it has an ARMv7 processor, it can run the full range of ARM GNU/Linux distributions [53].

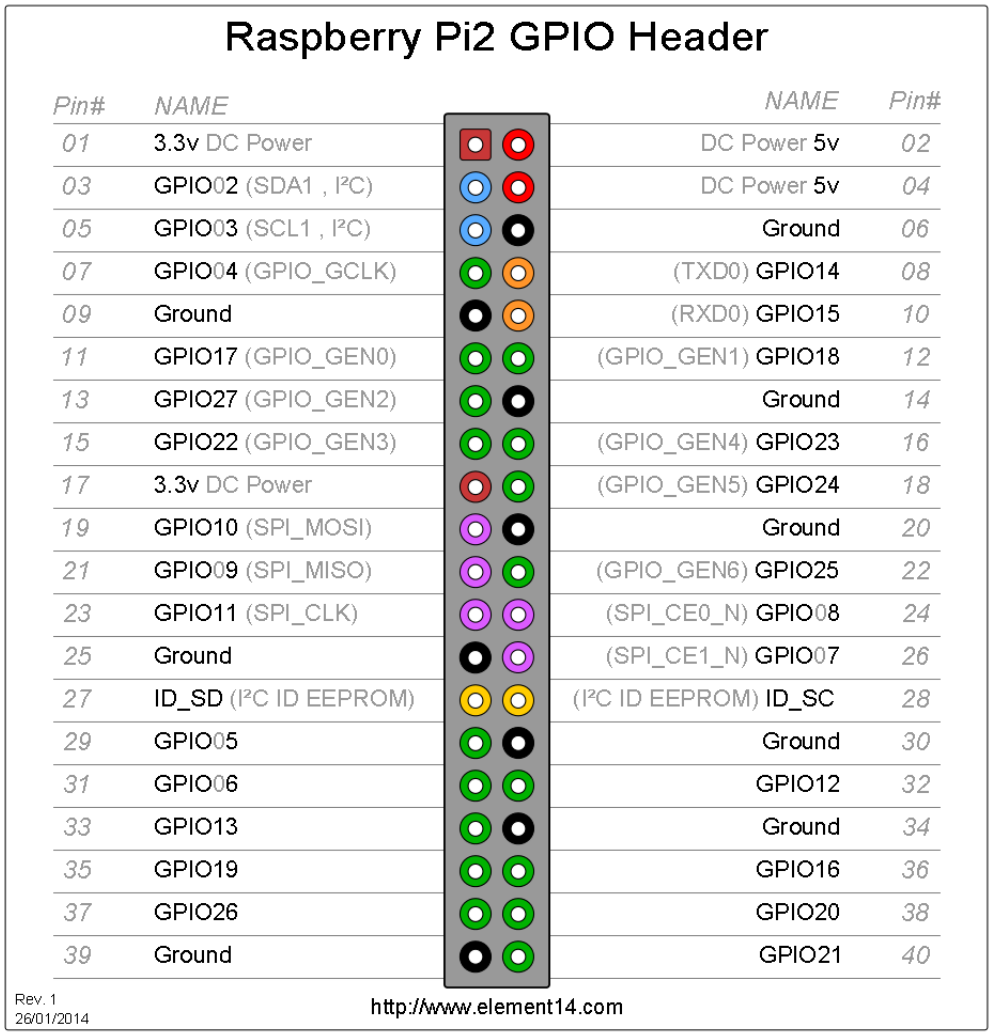


Figure 4.2: The pinout of the Raspberry Pi 2 model B, revision 2 [54]

During the initial stages of using ROS we attempted to get the installation up and running on the Raspbian operating system, which is a modified version of the Debian distribution specifically tailored for the Raspberry Pi. The installation was done using the official guide on the Raspberry Pi website [55].

After the installation of the Raspbian operating system, we installed the Indigo distribution of ROS, a Robot Operating System, which is really just a framework

for wiring robot software, using their step-by-step guide for the Raspbian operating system [56]. A source installation had to be done, due to poor support for the Raspbian, and this resulted in an 6-to-8 hour long setup process, because of the need for compiling, building and testing of the whole system. During the gathering of the different packages needed for ROS, some unforeseen issues were encountered, like the GMapping package causing issues during installation, since it required Python 2.7.1 and upwards. Through troubleshooting it was then discovered that GMapping package was looking for *Python-2.7.1-0ubuntu2*, instead of the *Python-2.7.3-6+deb7u2* that was installed together with the Raspbian operating system.

Instead of continuing to deal with the ROS issues on the Raspbian operating system, Ubuntu ARM was instead installed on the Raspberry Pi [57]. The installation process using Ubuntu ARM was much faster and more fluent compared to installing ROS on the Raspbian operating system, due to the much broader support for the Ubuntu distribution. The installation only took a couple of hours, by using the built-in package manager, because no installation was required from source [58].

4.3 Optical flow sensor

An optical flow sensor uses a camera to detect motion on a the x and y plane. It looks for movement by finding patterns on the ground to be able to determine the direction it is travelling in and how far the sensor has moved. This is vital information for SLAM mapping, since it is necessary to know where the mapping device is placed compared to previous measurements, this means that the optical flow sensor is used for finding the reference points. The sensor should be mounted quite high above ground on the rover to get a bigger view of the ground area, and is faced downwards to the ground [59].

The optical flow sensor works in the same manner as a computer mouse and returns information to the rover. What is important is the surface that the optical flow is looking at. If it is reflective or has many differences that the camera can use for reference, the readings will be incorrect. It is therefore good to have a backup system to ensure that the rover receives a correct reference point, like encoders on the wheels so the program knows that the rover is moving.

We used the CJMCU-110 Optical flow V1.0 from 3Drobotics. It uses the ADNS-3080 optical chip. It can take both 3V and 5V. It uses the Serial Peripheral Interface (SPI) bus to communicate with the master [60].

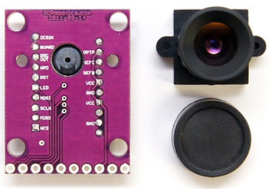


Figure 4.3: The optical flow sensor from 3D Robotics. [61]

During the development phase, we found out that no optical flow sensor was required for the SLAM package of our choice for ROS.

4.4 Laser

Our laser of choice, the LIDAR-Lite SEN-13167, is a SoC (System-on-a-Chip) solution for optical distance measurement applications. It has a 4.7 - 5.5 V nominal and 6 V potentially maximum DC operating range. The sensor has a theoretical limit of a 40m range and 100Hz rep rate. The laser emitters' accuracy is estimated to be +/- 2.5cm with an acquisition time of less then 20ms. These specifications allows us to get an accurate image of the environments specified in our test cases [62].

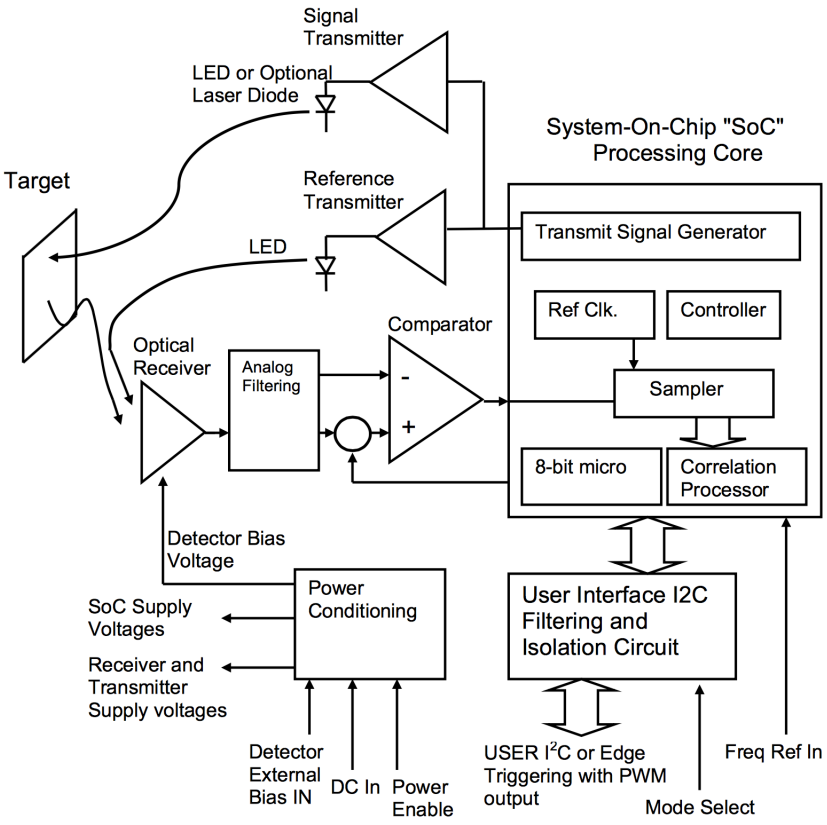


Figure 4.4: An overview of the laser's internals [63]

As previously discussed how the laser rangefinders work (Figure 2.2), our laser has two 14mm optic lenses, one as a transmitter emitting a 905nm infrared (IR) pulse and a receiver, which is measuring the time it takes for the pulse to travel back and the angle at which it comes in.

At distances less than a meter, the pulse is about the size of the receiver's optic lens, and at distances longer than that, you can estimate it using the following equation: $\frac{d}{100} = psize$, where d = distance and psize = the pulse size at that specific distance, in the units they are measured in. The actual spread is ~ 8 Mil or $\sim \frac{1}{2}^{\circ}$ [64].

It is possible to receive multiple valid return signals from a single measurement if the pulse illuminates more than one surface along the beam path. The sensor has

the capability to process two distinct reflections as long as they are separated by more than 3.5m and the reflection at the shorter distance does not saturate the correlation record masking the more distant object.

Figure 4.5 shows an example of two reflections in the signal correlation record separated by approximately 3.5m.

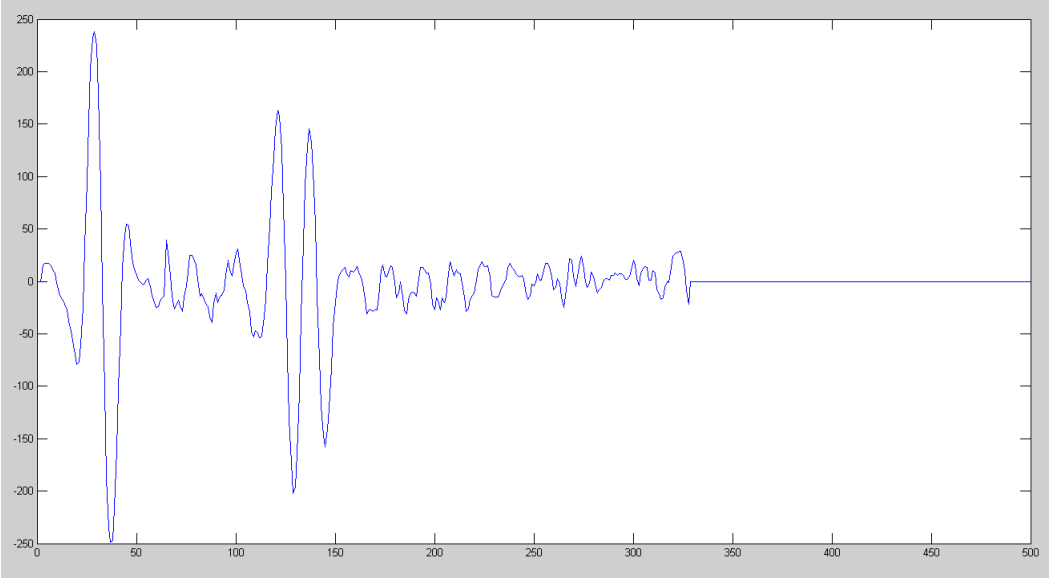


Figure 4.5: Measuring two valid return signals from one measurement [65]

The sensor detection criteria may be selected to pick the nearer signal, the more distant signal or the strongest signal strength. Currently, we do not handle multiple reflection in our code.

A reference signal is sent from the transmitter to the receiver, prior to the first distance measurement with a received pulse reflected from the target. The time delay between these two stored signals is estimated through a signal processing approach known as *correlation*, which effectively provides a signature match between these two closely related signals. This accurately calculates the time delay, and is translated into distance based on the known speed-of-light, which is $\sim 3.3ns\backslash m$ [65].

By default, the sensor runs at about 50Hz, but could vary given the signal strength and distance. To optimize the rep rate, it is possible to decrease the max-acquisition count (on register 0x02). By default, register 0x02 is 0x80 (value 128 in decimal), if you decrease this, the maximum number of acquisitions the sensor can take and use to get a reading, will decrease, too [66].

The assembling of the laser happened as shown on figure 4.6.

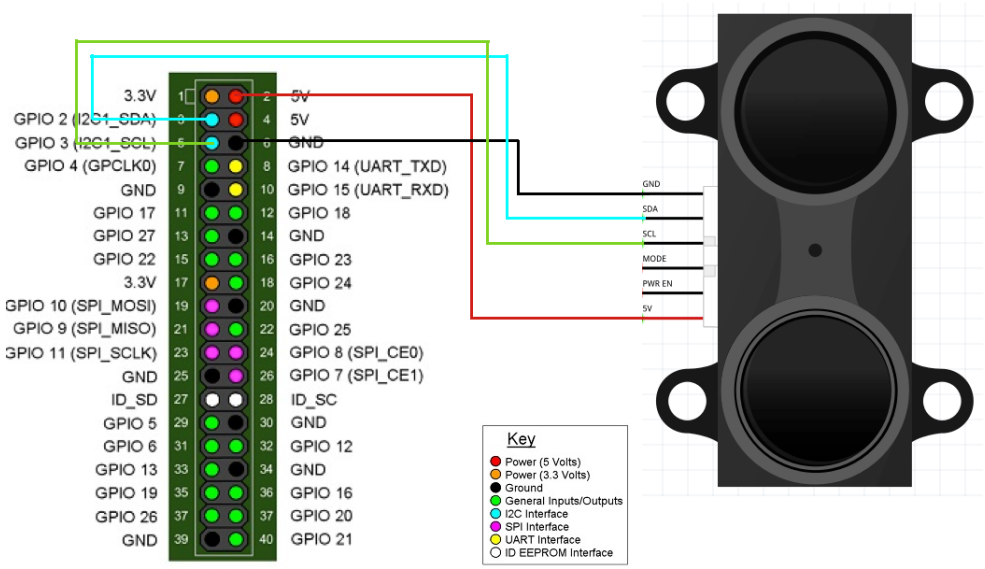


Figure 4.6: The pinout and wiring of the LIDAR-Lite laser and Raspberry Pi

Raspberry pin	Description	LIDAR pin
2	5V	Red
5	GND	Black
4	SCL	Green
3	SDA	Blue

Table 4.1: The pins of the microcontroller mapped to the laser pins

The laser can be interfaced through I^2C (Inter-Integrated Circuit) or PWM (Pulsed Width Modulation) [62]. The default I^2C address for the laser is 0x62, which we could validate by running the following command:

```
>> $ sudo i2cdetect -y 1
>> $
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- 62 -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

As the output shows, the address is available and the I^2C connection is usable. The -y 1 relates to the revision B of the Raspberry Pi board.

We started using a library called lidarLite, which provides a simple C interface to the LIDAR-Lite laser on the Raspberry Pi. It is using the WiringPi access library written in C for the BCM2835 SoC, used on the Raspberry Pi. It is usable with C and C++ or many other languages with suitable wrappers. It was designed very familiarly to Arduino's wiring library [67].

With the lidarLite library [68], we managed to read data using the I^2C protocol. To do this, we had to enable the kernel support for I^2C on the Raspberry Pi [69].

A small program was been written in order to test the output of the laser sensor, to read out the distance value of the next closest point in centimeters.

```
LidarLite V0.1

write res=0
Lo=13
Hi=0  13 cm

write res=0
Lo=19
Hi=0  19 cm

write res=0
Lo=24
Hi=0  24 cm

write res=0
Lo=29
Hi=0  29 cm
```

sample output from our laser test code

This output is generated by the function called *lidar_read(int fd)* from the lidarLite library. This code simply calculates by low and high value from the sensor returns the distance measurement in centimeters, in form of an integer value.

```
int lidar_read(int fd) {
   int hiVal, loVal, i=0;

   // send "measure" command
   hiVal = wiringPiI2CWriteReg8(fd, MEASURE_REG, MEASURE_VAL);
   if (_dbg) printf("write res=%d\n", hiVal);
   delay(20); // DO NOT CHANGE! Might get inconsistent readings from the
    sensor

   // Read second byte and append with first
   loVal = _read_byteNZ(fd, DISTANCE_REG_LO) ;
   if (_dbg) printf(" Lo=%d\n", loVal);

   // read first byte
   hiVal = _read_byte(fd, DISTANCE_REG_HI) ;
   if (_dbg) printf ("Hi=%d ", hiVal);

   return ( (hiVal << 8) + loVal);
}
```

lidarLite.c

We tried to incorporate this measurement code with a stepper-motor, so that it would return the distances around a circle. The first version had the stepper-motor spinning the laser around at a constant speed, which was driven by an Arduino board (Code reference 4.10), while the laser made measurements that was driven by the Pi.

In order to determine how long it takes for the lidarLite *lidar_read(int fd)* function to execute, some testing was made.

```
#include <lidarLite.h>

int main(){
    int nread = 1000;
    for(int i = 0; i<nread; i++){
        lidar_read();
    }
}
```

laserTiming.cpp

This code was run with the settings of 1000 and 2000, and each version was ran 3 times. The time was measured using the UNIX *time* utility.

Measurements of how long it takes to run *lidar_read(int fd)*:

Number of Calls	Measurements			Average
1000	24.372s	24.476s	24.490s	24.446s
2000	48.400s	48.111s	48.172s	48.228s

Table 4.2: Data from testing

According to this data, we can isolate the time it takes to execute the *lidar_read(int fd)* function from everything else:

$t_{total} = n * t_{lidar} + t_{rest}$

$24.446s = 1000 * t_{lidar} + t_{rest}$
$48.228s = 2000 * t_{lidar} + t_{rest}$

Subtract the two equations:

$23.782s = 1000 * t_{lidar}$
$t_{lidar} = 0.023782s \approx 0.024s$

This proved ineffective, since the rotation and measurements were independent from each other. If the time for one measurement is not precise enough the measurements to be skewed over time.

We then attempted to control the stepper-motor with the Raspberry Pi, added to the same code as we were running the laser with. However, the wiringPi library required root privileges, and ROS was not designed to be run under root privileges. The solution was to use a system call library to call an external file, which then would drive the stepper-motor. This makes the stepping and measuring synchronous, but making system call is very expensive in terms of computational resources, meaning that the code is significantly slower than the top speed that the laser is capable of.

The full code written for the laser sensor can be found in Appendix 9.2. The sequence diagram shown in figure 4.7 gives us an overview of the calls happening in the code.

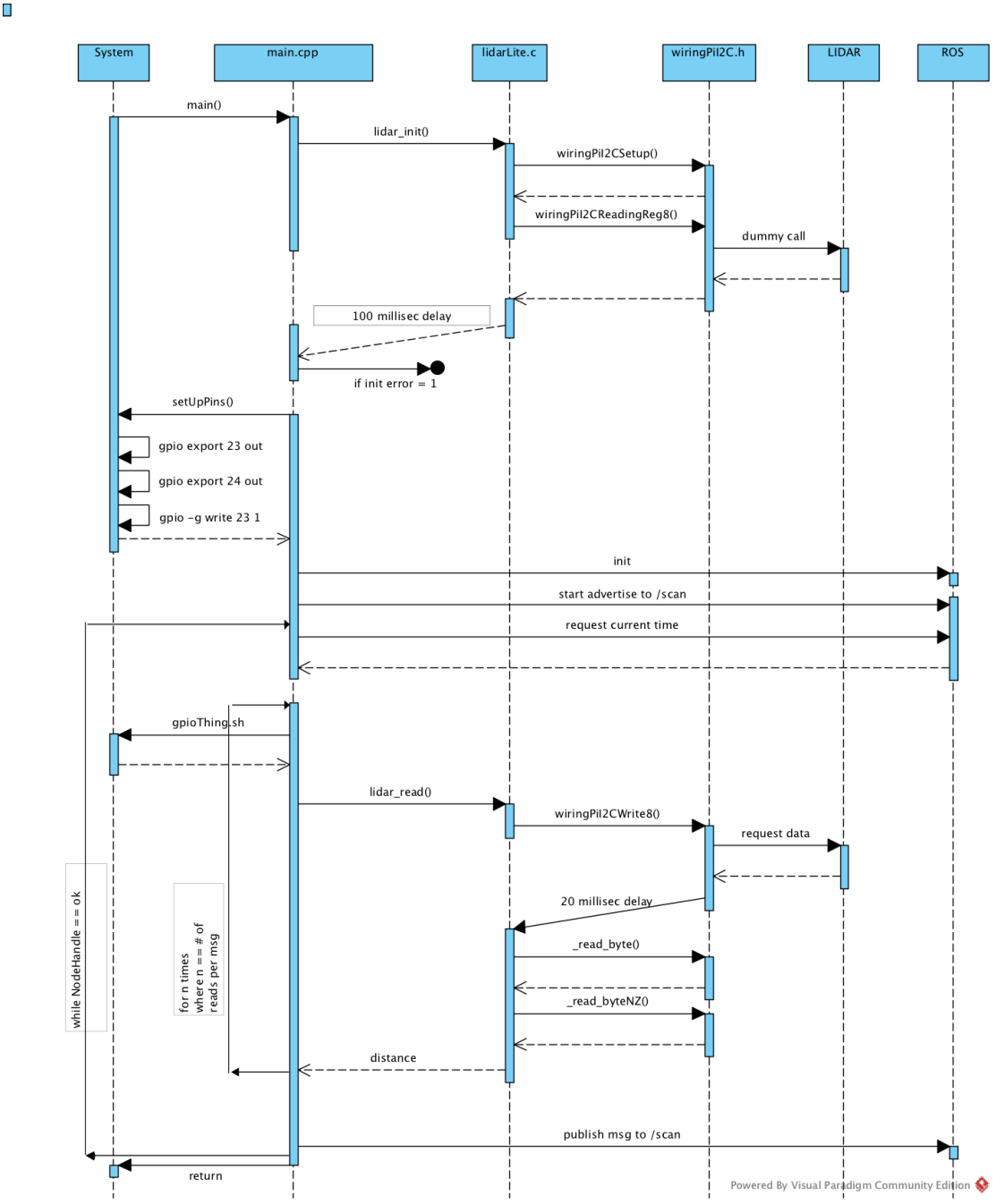


Figure 4.7: Sequence diagram of the main function call in main.cpp

The first step is to initialize the the LIDAR-Lite laser sensor, by giving a dummy "wake-up" call to it via the wiringPi library, using the I^2C protocol. There is a 100 millisec delay, which is necessary for the device to wake up and for the reading of registry to be completed. If the initialization fails, the program terminates by printing out a message to the terminal.

Thereafter, we call the *setUpPins()* function which makes the system call 3 separate calls using the GPIO utility, setting up the GPIO on the Raspberry Pi to be able to communicate with the stepper motor.

The initialization of ROS happens afterwards, by calling the *init* function with arguments and the name of our module. The module name is used as the node name in ROS. In the next line, ROS also starts to advertise messages to the */scan* topic, where all the scan results are being published to.

The rest of the code consists of two loops, one *while* loop and one *for* loop. The *while* loop continues until the node returns anything other than an *OK* status code. In here, a new LaserScan variable is being created where we ask ROS for the current time, set some hardcoded values and populate an array with the readings. The *for* loop is then used to get 200 readings from the LIDAR-Lite laser using the *lidar_read(fs)* function and also rotating the stepper motor before each of the calculation using the *step(num_step)* function. The *step(num_step)* function is calling a shell script that is doing calls to the GPIO interface, making the stepper motor to take 8 micro-steps, or in other terms, one full step. Before the end of the *while* loop, the message is being published to */scan*.

```
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
gpio -g write 24 0
gpio -g write 24 1
```

gpioStuff.sh

The only purpose of the code above, is to be called from the main file. The gpio utility is capable of interfacing with the GPIO pins on the Pi without root privileges. The change from low to high causes a micro-step, and 8 micro-steps makes one full step.

4.4.1 Laser Safety Analysis

In the following section we analyse and review the classification of the laser product of choice, the LIDAR-Lite rangefinder laser.

This product emits laser radiation and it is classified as a *Class 1 laser product* during all procedures of operation. This means that the laser is safe to look at with the unaided eye, however, it is very advisable to avoid looking into the beam and power the module off when not in use [70].

The operation of LIDAR-Lite, without proper housing and optics, or modification of the housing or optics that exposes the laser source may result in direct exposure to laser radiation and the risk of permanent eye damage. Removal or modification of the diffuser in front of the laser optic may result in the risk of permanent eye damage and the declassification of the laser as a *Class 1 laser product*. The product is also RoHS compliant [71].

4.5 The building process

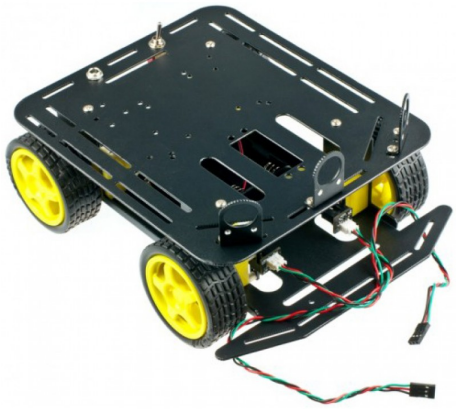


Figure 4.8: The chassis used for our prototype

To attach the different sensors to the rover, different enclosures and brackets were needed. For the ultrasonic sensors, two different kind of brackets have been designed, since the main chassis required them to be mounted differently based on their placement on the chassis of the rover. Using Autodesk Inventor the two enclosures for the ultrasonic sensors have been modelled and 3D printed.

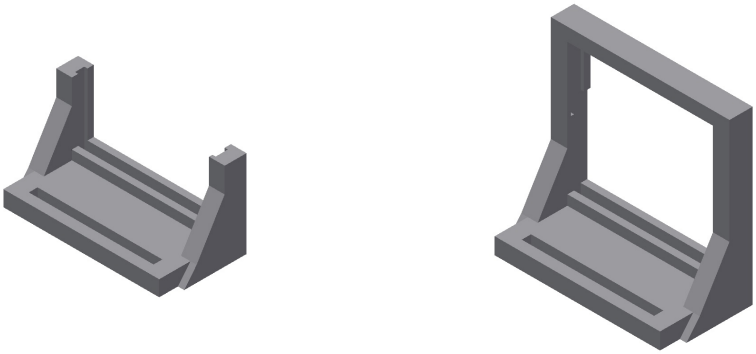


Figure 4.9: The enclosures used to mount the ultrasonic sensors on the chassis

The enclosures for the ultrasonic sensors have be designed to place all of the sensors at the same height on each side of the chassis, to ensure that the measurements are similar. The left-most holder is the one which is primarily used.

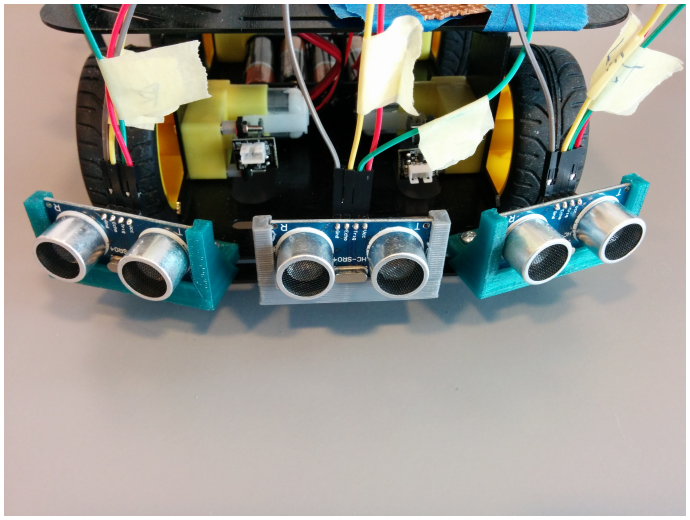


Figure 4.10: Front view of the mounted ultrasonic sensors

The design for the LIDAR-Lite laser sensor enclosure was found on the Internet, modelled by someone who is working on a project using the same laser sensor as us. The enclosure enables the sensor to rotate 360° using 3D printed gears that are driven by a stepper motor [72].

The enclosure design required a high-end 3D printer due to its size and print time. The prints were completed within a day by the university, which would have taken much longer with our personal 3D printers.

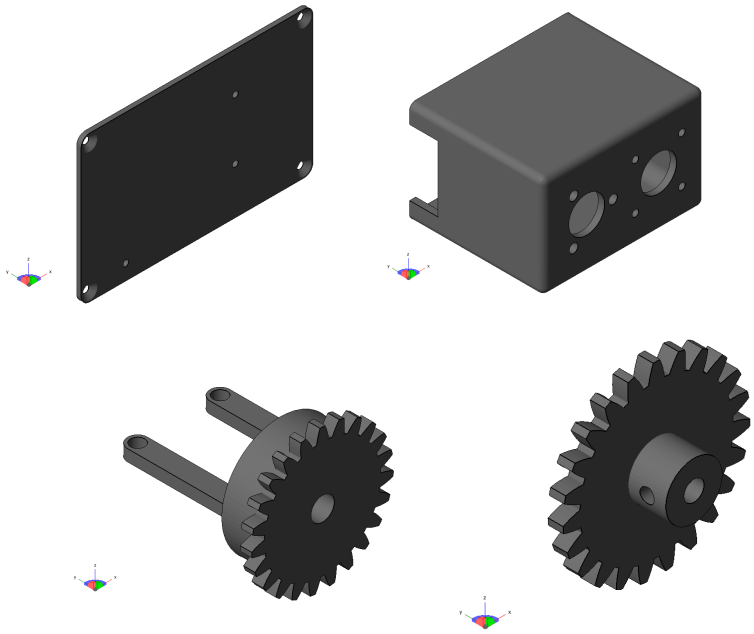


Figure 4.11: Parts of the enclosure used for rotating the LIDAR laser sensor

The idea of using a separate enclosure driven by a stepper motor to rotate the laser sensor is to constantly get a 360° map of the environment by rotating the sensor

separately from the chassis, rather than rotating the chassis to do so. This way, we can also continue to update the map while the rover is on the move. The strength behind using a stepper motor is that it is possible to control each step with great precision, which makes it more reliable when rotating it.

The project ended up being assembled in two separate parts: The rover with the ultrasonic sensors and the LIDAR-Lite laser sensor module by itself.

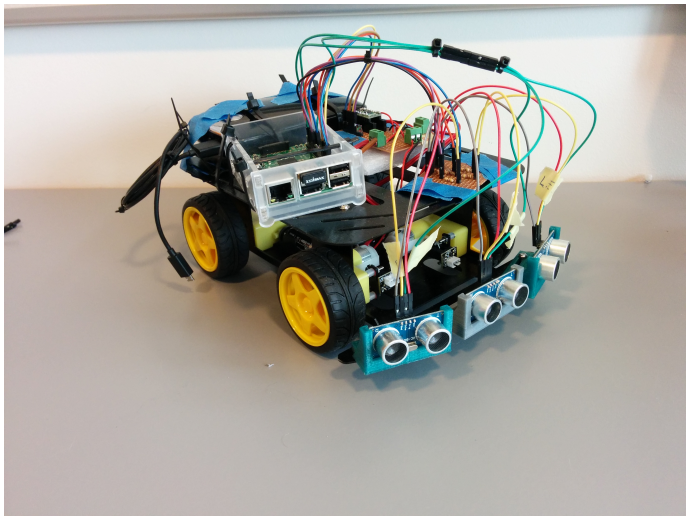


Figure 4.12: The final prototype for the rover

Figure 4.13: The final prototype for the LIDAR-Lite laser sensor

4.6 Slip ring

To allow the laser sensor to continuously rotate and take measurements, a slip ring was needed to avoid issues with the cables breaking or getting tangled, which would result in damage to the equipment. The slip ring used for this build is a SNM022A-06 [73], which is a 6-wire slip ring that supports up to 210DC/240AC.

Figure 4.14: The SNM022A-06 [74]

The slip ring allows the transfer of electrical signal from a stationary object to a rotating object. The center of the slip ring remains stationary as the surrounding housing rotates, in our case 6, conducting brushes along the center core in separate lanes, which then transfers the electrical signal. The specific slip ring used has 6 lanes on the core, since the slip ring is compatible up to 6 wires. Between each of the lanes a high resistance insulation is put into place to avoid short circuits and signal interference between the wires [75].
Slip rings are often used in electrical generators, in cable reels and wind turbines.

4.7 Ultrasonic sensor

For the autonomous navigation, the rover will be using 3 ultrasonic sensors for close-proximity detection. These sensors will be mounted on the front arc of the rover, at the same height as the wheels to help detect objects within a close distance of the rover.

Figure 4.15: The specific ultrasonic sensors are 3 **HC-SR04**. [51]

To avoid damaging the Raspberry pi when using these ultrasonic sensors, a voltage divider was needed in order to protect its input pins. The ultrasonic sensor operates at 5V, resulting in the output from the sensor being 5V. The GPIO ports on the Raspberry Pi operate at 3.3V, so the voltage divider is used to limit the voltage going into the input port from the ultrasonic sensor, while still providing a correct *input-HIGH* signal to the Raspberry Pi. When the sensors are initially powered on they require a few milliseconds to initialize, before they can be used to take measurements. This is recommended in the datasheet [51].

The HC-SR04 operates using 4 pins: Vcc, Trigger, Echo, Gnd. The trigger pin on the sensor is connected to the Raspberry Pi as an GPIO-output and the echo pin is connected as a GPIO-input. The echo pin outputs a 5V signal, so to protect the Raspberry Pi we have created the following setup:

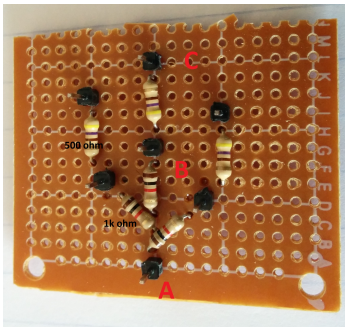


Figure 4.16: Picture of the voltage dividers

The circuit in figure 4.16 has three sets of voltage dividers. Besides protecting the Raspberry Pi from the 5V, there are also other advantages of the board. Instead of using 12 pins for the three ultrasonic sensors the board reduces it to only end up using 8 pins, since the board above combines their 5V Vcc's and the ground pins into two common pins. The three pins at the top of the figure labeled C are the output pins from the ultrasonic sensors, which at that time has a voltage of 5V. The

pins labeled as B is where the Raspberry Pi is connected, the voltage divider has at this point reduced the voltage to the suitable 3.3V for the Raspberry Pi.
The final pin labeled C is the common ground for the three voltage dividers.

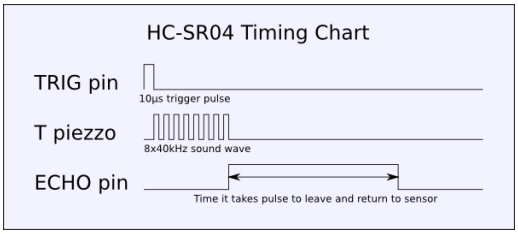


Figure 4.17: The HC-SR04 timing chart [76]

To retrieve a distance measurement, a 10μs pulse is sent from the sensor using the trigger pin, the echo pin then receives a HIGH-pulse equivalent to how long it took to hear the echo. This means that length of this high−pulse is proportional to how far away the object is that distance is being measured between [77].

```
def getdist(trigger, echo):
    #Sending the 10microsecond pulse nedded
    GPIO.output(trigger, True)
    time.sleep(0.00001)
    GPIO.output(trigger, False)
    #When echo goes high the time starts
    while GPIO.input(echo)==0:
        pass
    start = time.time()
    #When echo goes low the time stops
    while GPIO.input(echo)==1:
        pass
    stop = time.time()

    #The time difference * speedofsound
    #Divided by two since the since the distance meassured is back and
    forth
    distance = ((stop-start) * 34000)/2

    return distance
```

ultrasonic.py

The snippet of code above is the function that retrieves the distance measurement between the ultrasonic sensor module and the current object in front of it. On line 3, the trigger output is set to High for 10μs to create the ultrasonic burst. When the burst returns the input pin ECHO goes high for the duration of the pulse, which is measured by the subtracting the stop time from the start time. The distance is returned in centimetres, since the speed of sound ($340m/s$) has been converted to centimetres per second.

4.8 Motorcontroller

The chassis used for the rover has 4 pre-installed DC motors, which are all four used for movement. To control the direction of the rover a Pololu DRV8833 motor controller has been connected to the motors and the Raspberry Pi. The motor controller is in charge of operating the steering-, forward- and reversing motion of the rover.

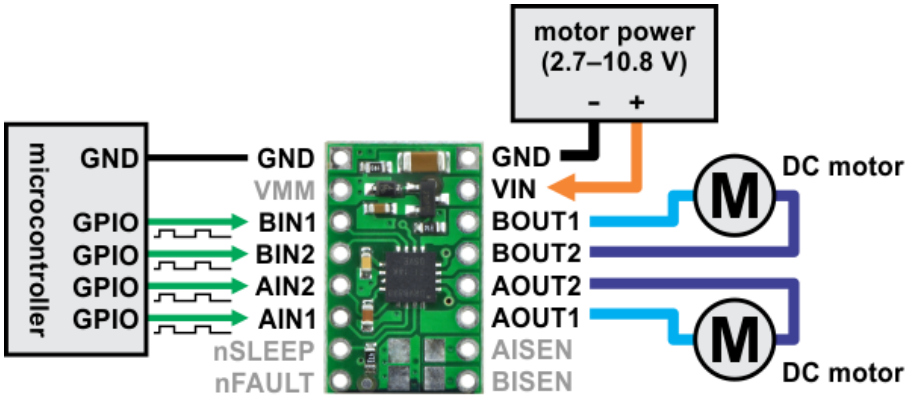


Figure 4.18: Pololu DRV8833 [78]

The driver uses a 2.7-10.8V supply voltage to operate, which is supplied to the motor controller by an external battery pack. This battery-pack is used to power the motor controller and the four DC motors on the rover. The DRV8833 has two separate DC motor outputs labeled as *BOUT* and *AOUT*, that can be independently controlled by their matching inputs from the microcontroller, since this rover uses 4 DC motors, each of the outputs from the motor controller will be connected to 2 DC motors.

The advantage of this particular motor controller is the fact that it supports analog and digital inputs. Since the Raspberry Pi has no analog outputs the controller will be given digital inputs.

xIN1	xIN2	xOUT1	xOUT2
0	0	Z	Z
0	1	H	L
1	0	L	H
1	1	L	L

Table 4.3: The truth table for the outputs

Table 4.3 shows the possible combinations of pins to achieve different desired directions/states, where the pins must be set to high and low according to what is the direction of choice. To turn the rover, the A and B inputs will be flipped, so that when *AOUT* goes forward, the *BOUT* reverses, which results in the rover turning

in its place. The input pins on the motor controller also allow for control with PWM (Pulse Width Modulation), which will change the length of the duty cycle and influence the speed of which the rover is moving at [79].

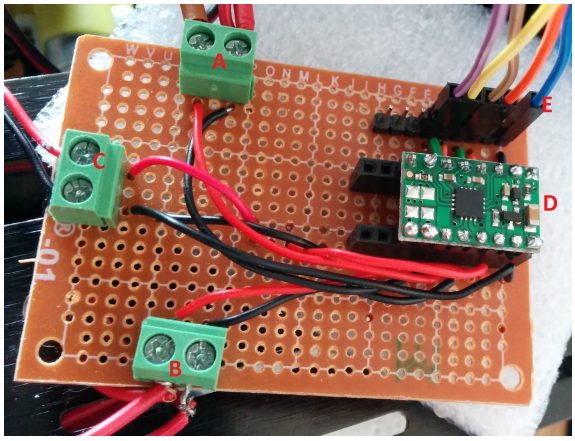


Figure 4.19: Custom board for the motor controller

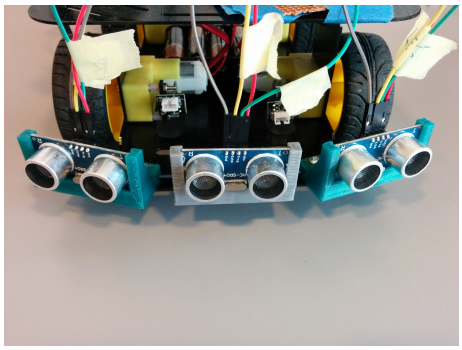
To enable easier access to the pins on the motor controller, a custom board has been created for ease of use. The green terminals, A and B, are connection terminals for the *AOUT* and *BOUT* respectively. The terminal C is for the battery pack that supplies the voltage to the motor controller and the DC motors, which in this case is 5 AA batteries. The label D is the Pololu DRV8833 and the final label E are the connections to the Raspberry Pi.

Operating the motor controller requires 5 pins from the Raspberry Pi, 4 to control the input pins on the DRV8833 and a ground pin. Since we are controlling the motor controller using HIGH and LOW, the controller can be connected to any of the regular digital GPIO pins.

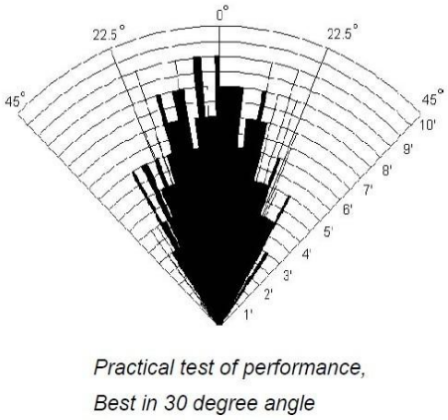
4.9 Close proximity detection and navigation

The close proximity system consists of two main components, one part that is reading from the ultrasonic sensors and another one for controlling the rover using the motor controller.

The ultrasonic sensors return distance measurements to the algorithm that then determines the optimal direction according to some threshold distances and different parameters.



(a) The mounted sensors



(b) The measuring angle [51]

The current vision of the rover, which relies on the distance measurements from the sensors above, is limited to three different measurements. This unfortunately results in some blind spots between the ultrasonic sensors. The ultrasonic sensor performs best at a 30° reading angle. This angle helps determine the area of which the ultrasonic sensor can read upon. The larger the angle, the less of a blind spot there will be between two ultrasonic sensors, depending on how they are mounted on the rover [51]. Besides the 30° models there also exists the same ultrasonic sensors with a measuring angle of 15°.

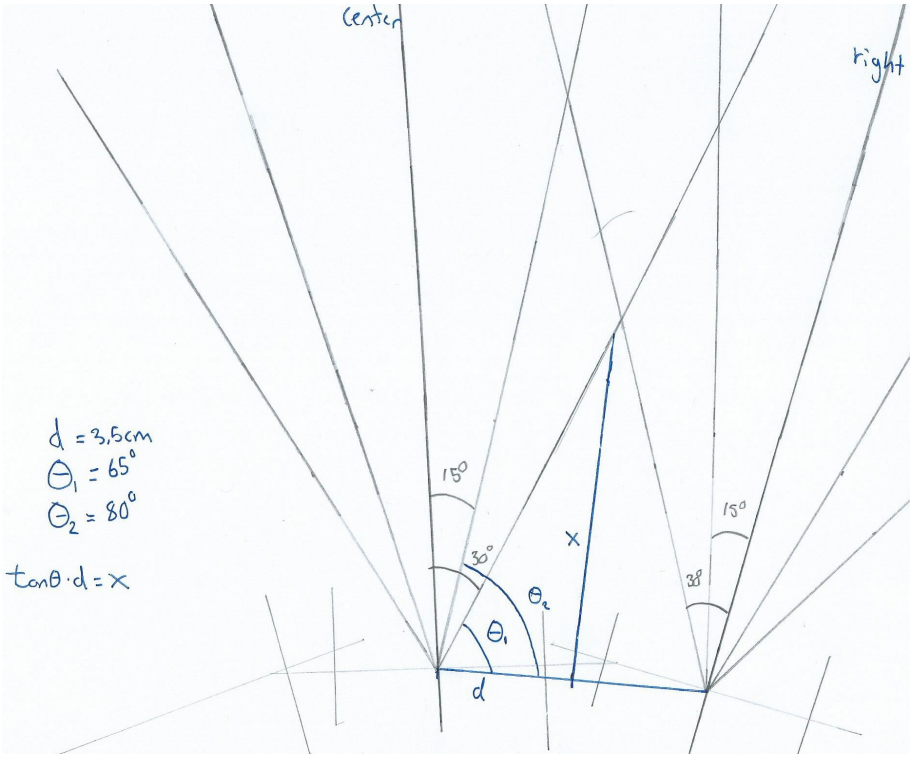


Figure 4.21: The diagram used for the calculations

The blind spot can be determined by finding the length x on the figure above. The triangles with d as their bottom line indicate the area of which the ultrasonic sensors covers, for both the 15° and 30° measuring angles. Since some versions of the ultrasonic sensor perform best at an 15° angle, two different calculations have been made to help determine where the blind spots for this particular setup are located. The length x marked on the sketch above indicates where the blind spot starts. To find the length x, the equation $tan(\Theta)d = x$ can be used to solve the problem. The length d is found by taking half of the length between the center of each ultrasonic sensor and it this length stays constant for both measuring angle cases. Θ is found my measuring the angle between the length d and the measuring angle of the ultrasonic sensors. Since the ultrasonic sensors measuring angle is different for the two cases, the value of Θ changes depending on the ultrasonic sensors operates at 30°(Θ_1) or 15°(Θ_2).

$\Theta_1 = 65°$ $\quad d = 3.5cm$ $\quad tan(65°)3.5cm = 7.506cm$
$\Theta_2 = 80°$ $\quad d = 3.5cm$ $\quad tan(80°)3.5cm = 19.849cm$

The rover has two theoretical blind spots on either side of the center mounted ultrasonic sensors at a measuring angle of 15° the blind spot is $34.7cm^2$ and at 30° the blind is $13.1cm^2$.

During testing these measurements will be used to figure out what the specific measuring angle is of the ultrasonic sensors the rover is equipped with, to help identify possible issues that might occur with how it operates.

```
    while True:
        #The time between each cycle (mininum recommended is 60ms)
        time.sleep(0.200)
        #Measuring the distances
        dist_left = getdist(TRIGGER_LEFT, ECHO_LEFT)
        dist_right = getdist(TRIGGER_RIGHT, ECHO_RIGHT)
        dist_center = getdist(TRIGGER_CENTER, ECHO_CENTER)
        #Moves forward (As a default)
        forward()
        print("Going forward")

        #Checking the different thresholds
        if dist_left < avoid_at:
            right()
            print("dist_left ("+str(dist_left)+") < " + str(avoid_at) + "
    -> Turning right")
            time.sleep(turn_time)
        elif dist_right < avoid_at:
            left()
            print("dist_right ("+str(dist_right)+") < " + str(avoid_at) +
    " -> Turning left")
            time.sleep(turn_time)
        elif dist_center < avoid_at:
            print("dist_center (" + str(dist_center)+ ") < " +
    str(avoid_at) + " --> Detecting optimal direction")
            if dist_left < dist_right:
                print("dist_left < dist_right -> Turning right")
                right()
                time.sleep(turn_time)
            else:
                print("dist_left > dist_right -> Turning left")
                left()
                time.sleep(turn_time)
```

main

The main idea behind the current algorithm, is that every loop of the code, the three ultrasonic sensors mounted on the front of the vehicle, all gather three separate measurements. Then afterwards the measurements are compared in three different statements that will determine the next direction the rover will be heading in. If the distance recorded by the left most facing ultrasonic sensor is less than the threshold distance set as the variable *avoid_at*, the rover will then turn towards the right for the amount of seconds that the *turn_time* variable is set to. The same thing applies to the ultrasonic sensor facing to the right, but instead the rover turns to the left when the threshold is exceeded. If the threshold distance is exceeded at the center ultrasonic sensor, it determines whether the optimal path is to turn left or right depending on which direction has the largest distance measurement. This means that if an object is detected by the center ultrasonic sensor, the rover

will then determine whether the distance measured to the left is greater than the distance measured to the right. The rover will then turn in the direction in which the distance measurement is greatest, since this in theory means that the rover will have the capability of travelling for a longer distance until meeting a new object that needs to be avoided or maneuvered around.

```
def getdist(trigger, echo):
    #Sending the 10microsecond pulse nedded
    GPIO.output(trigger, True)
    time.sleep(0.00001)
    GPIO.output(trigger, False)
    #When echo goes high the time starts
    while GPIO.input(echo)==0:
        pass
    start = time.time()
    #When echo goes low the time stops
    while GPIO.input(echo)==1:
        pass
    stop = time.time()

    #The time difference * speedofsound
    #Divided by two since the since the distance meassured is back and
    forth
    distance = ((stop-start) * 34000)/2

    return distance
```

getdist()

At the start of the navigation loop seen in the *main* code, the *getdist()* function is run three separate times with different arguments for each of the ultrasonic sensors. Measurements are gathered every 200ms, to allow the rover to move a short distance before re-calculating its direction. The datasheet for the ultrasonic sensors recommend a measurements cycle of at least 60ms, to ensure that the ultrasonic bursts do not overlap. In the appendix the full code can be found, where the arguments are clearly labelled to help determine what measurement is coming from what ultrasonic sensor (Appendix 9.1).

4.10 Stepper motor

Stepper motors are electrical motors that can be precisely controlled. They divide a full rotation into a discrete amount of steps, each proportionally sized.

Stepper motors are usually used with a specific circuit to control them, which is called a stepper-motor controller. Stepper-motor controllers allow for micro-stepping, which further divides a step into smaller pieces. We used 8 micro-steps per step.

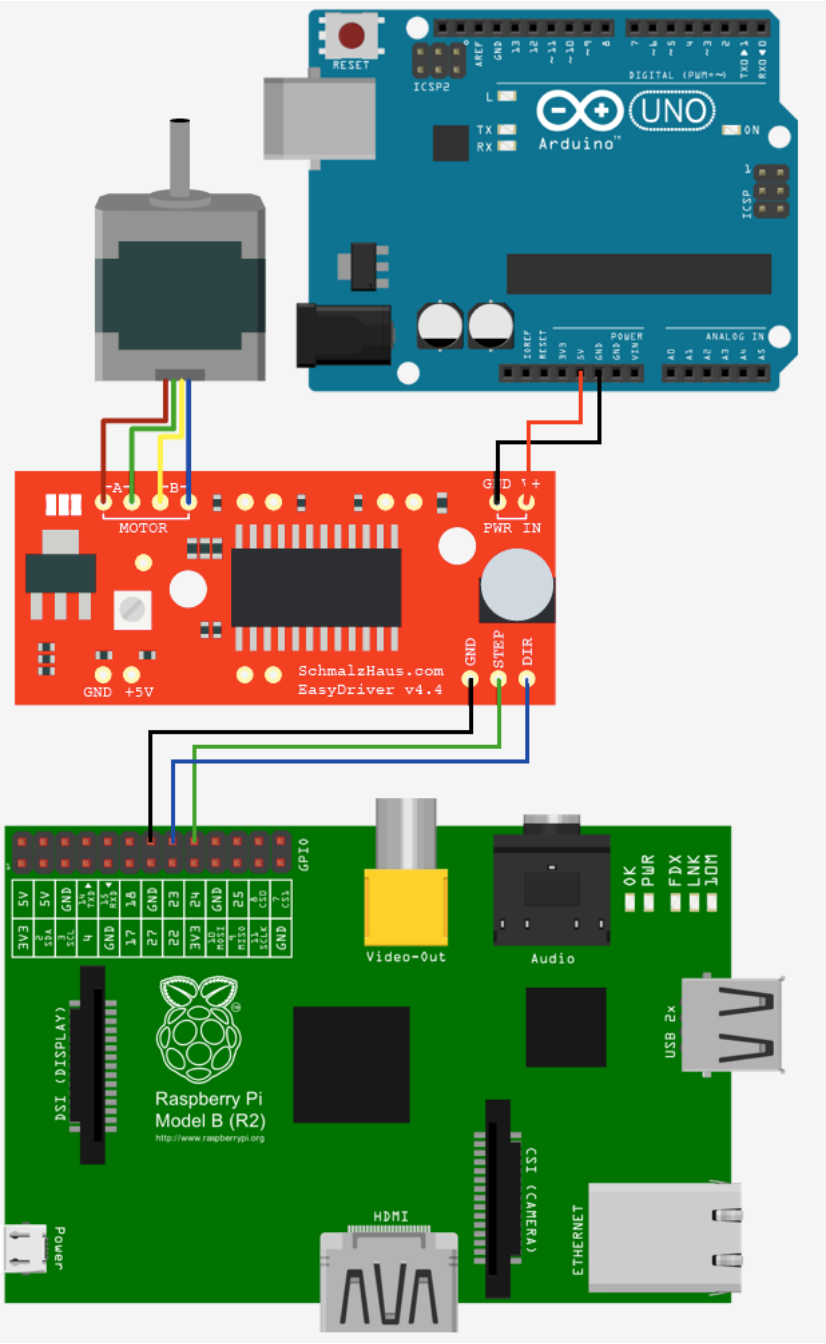


Figure 4.22: Connection chart for the stepper-motor

Figure 4.22 depicts how we connected the Pi to the stepper-motor, using a motor controller and an Arduino in the process. The only use of the Arduino in this picture is to serve as a 5V source, as we were already using the Pi to power the laser.

The stepper motor we used in our project is a NEMA-17 Bipolar Stepper Motor [80]. The stepper motor controller we used is an EasyDriver [81].

```
int dirpin = 2;
int steppin = 3;

void setup()
{
    pinMode(dirpin, OUTPUT);
    pinMode(steppin, OUTPUT);
    digitalWrite(dirpin, LOW);
}
void loop()
{
    // This LOW to HIGH change is what creates the
    digitalWrite(steppin, LOW);
    // "Rising Edge" so the easydriver knows when to step.
    digitalWrite(steppin, HIGH);
    /*This delay time is close to top speed for this particular motor.
    Any faster, the particular motor stalls.*/
    delayMicroseconds(500);
    delay(10);
}
```

Code 4.10

Since the stepper motor we use divides a full cycle into 200 steps, this means that the resolution is 1.8° per step. With this, the we can calculate the minimum distance we can resolve.

$$\frac{x}{sin(1.8°)} = \frac{1m}{sin(89.1°)}$$
$$x = \frac{sin(1.8°)}{sin(89.1°)}m$$
$$x = 3.14cm$$

At a distance of 1m, 1.8° can resolve a distance of ≈3.1cm, and this increases linearly with a larger distance (Code reference 4.10).

4.11 Configuring ROS for mapping

During the installation of ROS on Ubuntu, a new package named *Hector-slam* was discovered. This package allows ROS to generate a map without the use of the GMapping package, which was one of the packages that caused issues during the installation process on Raspbian. Its biggest advantage is that it does not rely on odometry to know whether the reference frame has changed. Odometry is the use of data from sensors to estimate the difference in position over time. It is used by robotic systems to estimate their position relative to a starting point. It is also important to highlight that is does not determine, but estimates this position in space and time.

4.11.1 How ROS works

During our development, we gained a deep understanding of how all the moving parts in ROS work well together. The following diagram showcases the general flow of how the ROS system that we built up works.

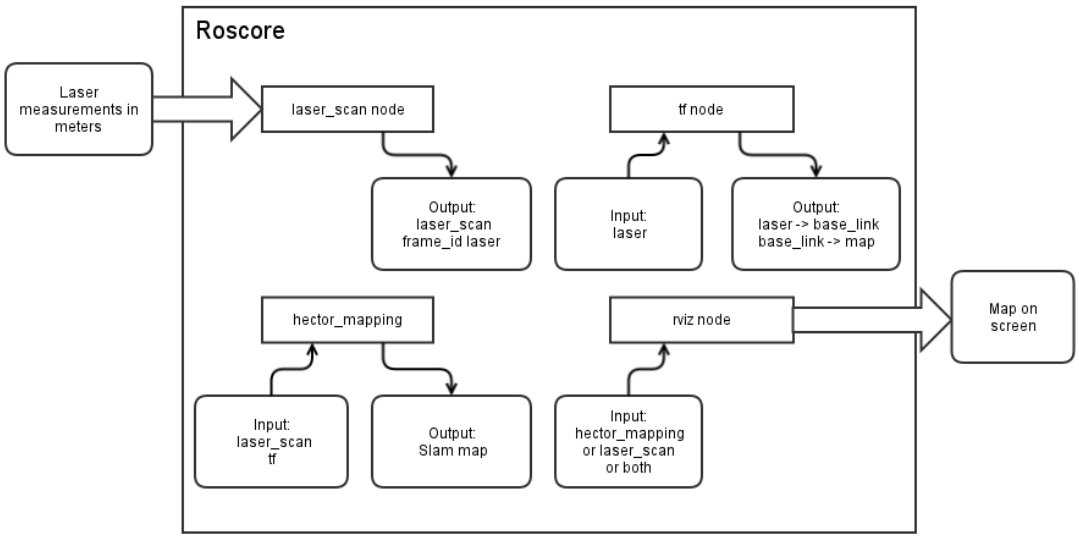


Figure 4.23: ROS flow diagram, showcasing the map generation from sensory readings

Roscore is the main process behind ROS and it allows for all running nodes to talk to each other, although by itself it does nothing, it just keeps running in the background.

Nodes are the smallest parts of ROS that are put together to form a full application. They are run as separate processes, all with their own executable files. The only way nodes communicate is through the roscore. To make a node, it is necessary to use or create a ROS library, written in any desired programming language.

Roslaunch is the ROS command that allows starting multiple nodes, and also allows giving parameters to nodes. Roslaunch executes nodes that are specified in an XML file you give it, called the launchfile.

Bagfiles are used for storing ROS message data, transmitted over any topic visible to list by calling *rostopic list* command. Bags are typically created by a tool like rosbag, which is subscribed to one or more ROS topics, and store the serialized message data in a file as it is received. These bag files can also be played back in ROS to the same topics they were recorded from, or even remapped to new topics. Once the current node setup has been established and the correct information is transmitted over the topics, the data will be stored in a .bag file.

4.11.2 Developing for ROS

During this stage of development, the gathering of the data happens on the Raspberry Pi, while a map is generated using ROS. The Raspberry Pi is accessed through SSH and the generated map is then broadcasted to another computer through that SSH connection, for visualization the tool called rviz is used.

To achieve this, we had to set up a direct connection between one of our developer computers running Ubuntu and the Raspberry Pi using a CAT5 Ethernet cable. Then we created a new network settings, set the IPv4 method to be 'Shared to other computers' which bridged the two network adapters and transmitted Internet packages to the Raspberry Pi from our own computers, connected to the Internet via Wi-Fi. Then we located the leases and found the IP address assigned to the connected device. To connect via ssh we used the following command line:

```
>> \$ ssh -Y username@ip-address
```

This enabled us to connect to the Raspberry and by using the *-Y* flag, it allowed us to stream the graphical application using X11 forwarding.

Although, this seemed to be working quite well, this operation required quite a lot of memory and processing power, therefore some issues were experienced:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3698	**ubuntu**	**20**	**0**	**15396**	**9136**	**2728**	**R**	**96.2**	**1.0**	**1:32.86**	**sshd**
3847	ubuntu	20	0	285196	69080	37360	S	23.8	7.3	0:35.16	rviz
3848	ubuntu	20	0	197328	98360	7120	S	5.0	10.4	0:28.83	hector_mapping
3	root	20	0	0	0	0	S	2.3	0.0	0:01.79	ksoftirqd/0
3874	ubuntu	20	0	89372	7748	6728	S	2.3	0.8	0:07.39	hector_trajecto
3283	ubuntu	20	0	43064	13728	4824	S	0.7	1.5	0:02.21	roscore
3308	ubuntu	20	0	69768	6696	5956	S	0.7	0.7	0:01.51	rosout
4084	**ubuntu**	**20**	**0**	**2728**	**1652**	**1288**	**R**	**0.7**	**0.2**	**0:00.29**	**top**
913	root	20	0	0	0	0	S	0.3	0.0	0:00.84	kworker/0:0
3295	ubuntu	20	0	86780	12748	4596	S	0.3	1.3	0:04.51	rosmaster

Figure 4.24: The resource usage

Looking at figure 4.24, we can observe several things by running the mapping operation: by generating the actual map directly on the Raspberry Pi required too

many resources from the CPU, and that the rendering of the visualization tool, rviz, channeled through the SSH connection to the computer screen required more bandwidth than there could be supplied. This caused a terrible lag and a bad experience both for us developing, but for the end result too.

To reduce the amount of resources used by the Raspberry Pi, the map needs to be generated and visualized using separate computing unit. ROS is built to have separate nodes with a main computing unit.

To copy files to and from the Raspberry, we used the *scp* utility, and therefore no version control setup was not necessary on the controller. The command line was used to transfer .bag files and source code:

```
>> \$ scp username@from-ip-address:~/file
 /home/mylocaluser/to-location/
```

When the custom laser node written by us has been started, it starts publishing laser readings in form of a LaserScan message, to the */scan* topic. By using the following command, we could listen to it and see the outcome of our scanning:

```
>> $ rostopic echo /scan
>> $ header:
      seq: 1
      stamp:
      secs: 1432290470
      nsecs: 627695559
      frame_id: laser
    angle_min: 4.71238803864
    angle_max: 6.28318405151
    angle_increment: 0.0314159207046
    time_increment: 0.0399999991059
    scan_time: 0.019999999553
    range_min: 0.10000000149
    range_max: 40.0
    ranges: [0.3100000023841858, 0.30000001192092896, 0.30000001192092896,
     0.3199999928474426, 0.30000001192092896, 0.3100000023841858,
   0.33000001311302185, 0.3199999928474426, 0.33000001311302185,
   0.3400000035762787, 0.3400000035762787, 0.3499999940395355,
   0.3400000035762787, 0.3400000035762787, 0.3700000047683716,
   0.38999998569488525, 0.4000000059604645, 0.38999998569488525,
   0.44999998807907104, 0.46000000834465027, 0.46000000834465027,
   0.47999998927116394, 0.5, 0.5099999904632568, 0.5400000214576721,
   0.5899999737739563, 0.6299999952316284, 0.6200000047683716,
   0.6000000238418579, 0.5899999737739563, 0.6499999761581421,
   0.6600000262260437, 0.6700000166893005, 0.6899999976158142,
   0.7099999785423279, 0.6899999976158142, 0.7099999785423279,
   0.7200000286102295, 0.1899999976158142, 2.549999952316284,
   2.549999952316284, 2.549999952316284, 2.549999952316284,
```

```
2.549999952316284, 1.149999976158142, 1.159999966621399,
1.2300000190734863, 1.2400000095367432, 1.2400000095367432, 1.25]
 intensities: []
```

../code/sampleOutput.txt

As the output shows, the number of readings are being stored in an array that is published to */scan* and then used by ROS to visualize or transform it later on. These readings are represented in a floating-point number in meters. Range is also represented in meters, but they are hard-coded, as the minimum and maximum value of the range does not change, but rather set by the manufacturer. The messages are time-stamped, which turned out to be very important when ROS transforms them for mapping and there is also a scan time, which tells the time between the scans in seconds. The angle values are radians of the start and end angle of the scan, while the angle increment is the angular distance between measurements. Our laser does not provide intensity-data, therefore the array has been left empty.

With this, we set up everything for starting to create a map in 2D using Hector-slam.

To run the code written by us, we had to place it in the ROS workspace, use the command catkin_make to build our code into a package that ROS also understands and run the following command:

```
>> \$ rosrun lidarlite_laser laser_scan
```

In order to access the I^2C interface on the Raspberry with no root privileges, we had to give the local user full permissions to */dev/i2c-1*.

4.12 Hector SLAM

During development we decided to use the Hector-slam package for ROS to make a 2D SLAM map. Hector-slam is a ROS library that we can use without odometry [82]. This allowed us to skip using the optical flow sensor for publishing movement on the x and y plane. Hector mapping uses reference points in its surroundings to keep track of movement in the xy plane. It is also possible to give hector mapping information about movement in z scale, tilt and yaw. This is beneficial for a robot that travels in terrain with big changes or even when controlling flight for a drone or quadcopter.

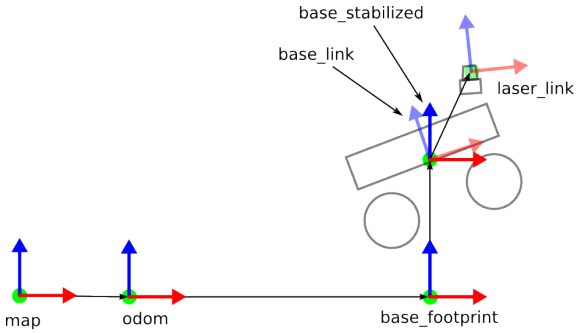


Figure 4.25: transform of robot points. [83]

Hector-slam needs a laser input published over the laser scan node, and also *tf* (transform) information about the robot. The *tf* represents the difference in height of the laser from the ground or an offset in x, y and z coordinates. A robotic arm with multiple joints would have a transform on each of them so the system could calculate movement of the sensor located on the arm.

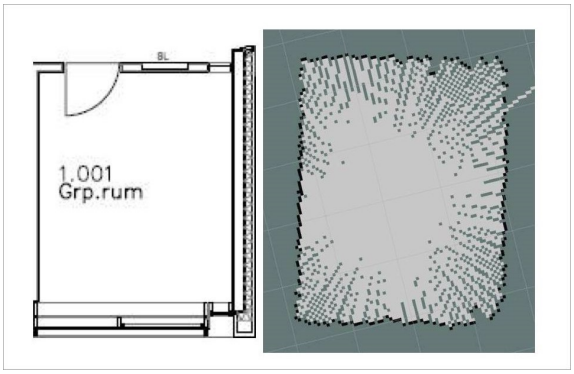


Figure 4.26: Room-plan of H101, Aalborg University Esbjerg

We took a single measurement using the laser and Hector-slam, which created the map shown to the left on the figure above. The university gave us a picture of group room H101 at Aalborg University Esbjerg, which we used to compared the map from Hector-slam with the floor plan.

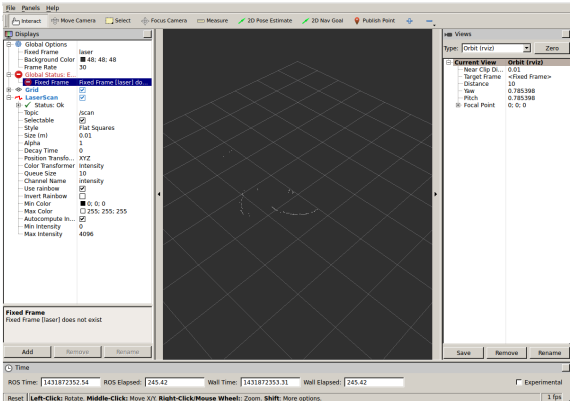


Figure 4.27: Rviz visualization tools package in ROS

The map is a screen shot taken from rviz. Rviz is a visualization tool included in ROS, which allows the user to see the laser measurements and 2D mapping in real-time. [84]

Hector-slam takes the information given from the transform and laser measurement and makes a 2D map out of it. The following figure is a flow-chart explaining these steps.

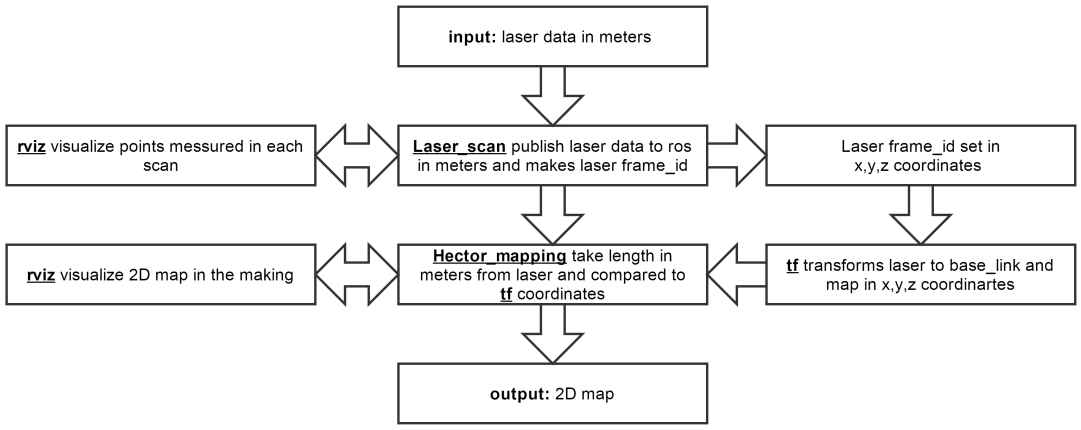


Figure 4.28: Hector-slam flowchart

We made a .bag file recording of the */scan* topic on the Raspberry Pi, then the file was copied to a developer computer running Ubuntu and ROS. The bag file was played on the laptop and hector-slam with transform published, was run. The outcome was a hector map.

```
<launch>
    <param name="/use_sim_time" value="true" />
    <node pkg="hector_mapping" type="hector_mapping"
   name="hector_mapping" output="screen">
        <param name="base_frame" value="base_link"/>
        <param name="odom_frame" value="base_link"/>
        <param name="pub_map_odom_transform" value="true"/>
        <param name="scan_subscriber_queue_size" value="25"/>
        <!-- Map size / start point -->
        <param name="map_resolution" value="0.050"/>
        <param name="map_size" value="2048"/>
        <param name="map_start_x" value="0.5"/>
        <param name="map_start_y" value="0.5" />
        <param name="map_multi_res_levels" value="2" />
    </node>
    <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch">
   </include>
    <node pkg="tf" type="static_transform_publisher"
   name="map_nav_broadcaster" args="0 0 0 0 0 0 b$
```

```
    <node pkg="tf" type="static_transform_publisher" name="map_to_base"
      args="0 0 0 0 0 0 /map$
    <node pkg="tf" type="static_transform_publisher" name="map_to_frame"
       args="0 0 0 0 0 0 /ma$
</launch>
```

main.launch

Chapter 5

Testing

To ensure that our prototype fulfills all the requirements we set up in Chapter 3, we set up some testing scenarios, which we used for testing the functionalities of our prototype. These scenario are made out of several locations and setups:

- Room H101, H-building, AAU Esbjerg
- Hallway, H-building, AAU Esbjerg
- Outside court, between H-building and main building, AAU Esbjerg
- Cantina, Main building, AAU Esbjerg
- Reception Area, Main building, AAU Esbjerg

We will not cover each individual test at all of the above listed location, but we will include the ones that are interesting or have dissatisfied our requirements.

The table below displays how well we achieved the testing goals set by us during Chapter 3. We have only managed to achieve 1 out of the 3 original requirements.

Requirement	**Test**
The rover can be controlled using a remote device.	FAILED
The rover should navigate autonomously through a maze.	FAILED
The laser sensor should make a usable 2D map of its surroundings.	PASSED

5.1 Close proximity navigation test

The first test that was conducted on the rover, was to figure out what the measuring angle of the ultrasonic sensors are. We used the calculated blind spot distance to place an object on a piece of paper with a ruler, which was then placed beneath the rover as a guide.

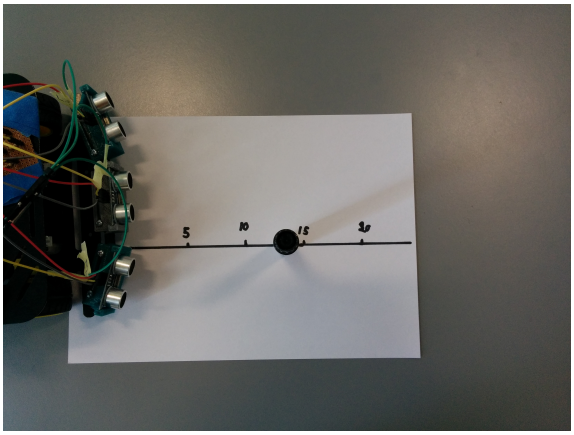
```
def getdist(trigger, echo):
    #Sending the 10microsecond pulse nedded
    GPIO.output(trigger, True)
    time.sleep(0.00001)
    GPIO.output(trigger, False)
    #When echo goes high the time starts
    while GPIO.input(echo)==0:
        pass
    start = time.time()
    #When echo goes low the time stops
    while GPIO.input(echo)==1:
        pass
    stop = time.time()

    #The time difference * speedofsound
    #Divided by two since the since the distance meassured is back and
    forth
    distance = ((stop-start) * 34000)/2

    return distance

print("Left: " + str(getdist(TRIGGER_LEFT, ECHO_LEFT)))
print("Right: " + str(getdist(TRIGGER_RIGHT, ECHO_RIGHT)))
print("Center: " + str(getdist(TRIGGER_CENTER, ECHO_CENTER)))
```

triple_ultrasonic_test.py

The code above was used to test the distance measurements from the ultrasonic sensor. As shown on figure 5.1a and 5.1b, an object on the ruler was moved back and forth to find out where the point was just before the ultrasonic sensor would pick up the object.

```
pi@zero ~ $ sudo python triple_test.py
Program started
Pin def succes
Left: 151.825666428
Right: 163.218975067
Center: 154.002189636
pi@zero ~ $ sudo python triple_test.py
Program started
Pin def succes
Left: 153.698205948
Right: 17.5621509552
Center: 15.4707431793
pi@zero ~ $ sudo python triple_test.py
Program started
Pin def succes
Left: 153.714418411
Right: 163.251399994
Center: 154.034614563
```

(a) Blind spot readings



(b) Measuring the blind spot

Figure 5.1: The blindspot testing

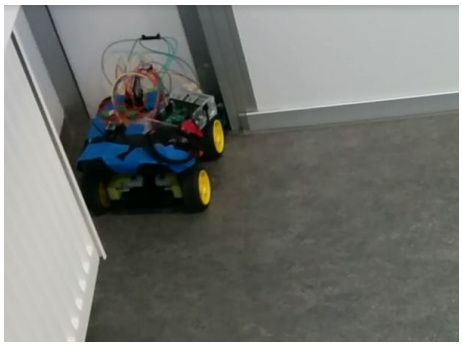
Through testing we determined that the tipping point for the blind spot was at around 15cm. If the object used for the testing had been smaller in diameter, it would have been possible to get closer to the theoretical calculation (which is 19.8cm). Since the tipping spot was at length greater than 7.5cm (from figure 4.21, this then means that the ultrasonic sensors used for the particular project has a measuring angle of 15°.

Figure 5.2: The testing course

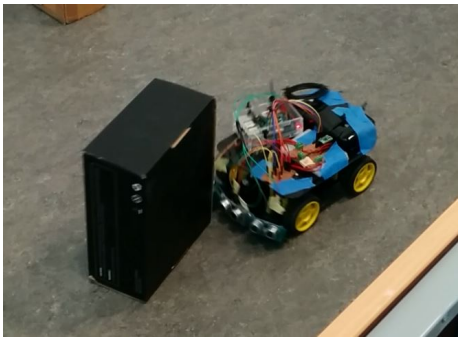
The testing course had been created arbitrarily, with the purpose of placing some random objects for the rover to detect, avoid and navigate around.

The rover was tested using different threshold distances, which means that the rover will sense objects and walls according to that set threshold. When the distance was set to a lower length (e.g. 20cm), the rover would get really close to objects before

determining a new path, this occasionally also led to the rover hitting some of the objects or the walls of the test course. If the distance threshold is set to a higher value (e.g. 50cm), the rover has an easier time navigating/avoiding objects in its path, the downside is though that with such a great distance, the rover never reaches certain parts of the course, because it often detects too many nearby objects and therefore never approaches certain areas. The rover would also sometimes get stuck in the middle of an open space, oscillating left and right whilst slowly moving forward, the cause of this is because each time the the rover turned in one direction the rover detected a new object within a short distance, and immediately turns again - continuing in an endless loop.



(a) Rover stuck in a corner



(b) Bad angle for measurements

With the current algorithm, the rover gets stuck in a loop if the rover drives directly into a corner(figure 5.3a). The reason for the looping is that the distance towards to wall of either side of the rover are equal, so it can not determine which direction it should steer towards.

Because of the skewed angles of the object in the test course, the rover would sometimes hit them. Figure 5.3b clearly shows the rover hitting the object face on, because the ultrasonic sensors were not able to read the distance to the walls of the box because they are at a bad angle.

When the rover would turn to avoid objects or change its direction, it would skid across the floor. This resulted in the rover not turning on the spot, which reduces the accuracy of the steering.

5.2 2D Mapping

5.2.1 Indoor test

In order to test how well our mapping can keep the relation between objects, we made a small setup on a table in room H101. We placed the laser on the edge of the table, and set a laptop, a small box, and two sponges, at varying distances from the center of the laser, trying to keep their faces perpendicular to the laser. We then roughly measured the distances.

To gather data from the laser, it was roughly run for 2 minutes or 4 full rotations, since it took 30 seconds to collect 360°'s worth of measurements.

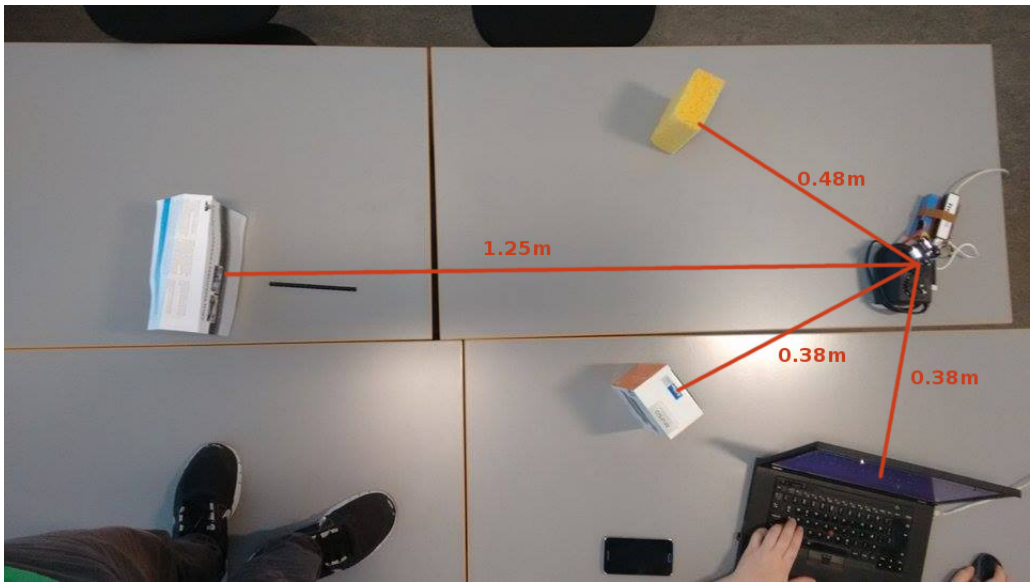


Figure 5.4: A photo of room H101 that was mapped

Figure 5.4 shows an overhead view what the setup was, with the addition of some distances. For this particular testing location we also made sure that the laser would not hit any windows, since those are considered specular surfaces and it would be impossible or at least very difficult for the LIDAR-Lite laser to detect it.

We let the laser gather enough data, such that there was no noticeable interference from us moving around inside of the room. We then took a screenshot of the map that had been produced, which the was compared it to figure 5.4. First by estimating distances on the picture, then by digitally overlaying the two images.

Figure 5.5: A map of room H101 with some small obstacles

Figure 5.5 shows the map generated by the laser, roughly overlayed with the picture from the test course in figure 5.4. From the comparison its clear that the generated map keeps the same aspect-ratio of the physical world that it is mapping. The data used for this particular map is purely point data.

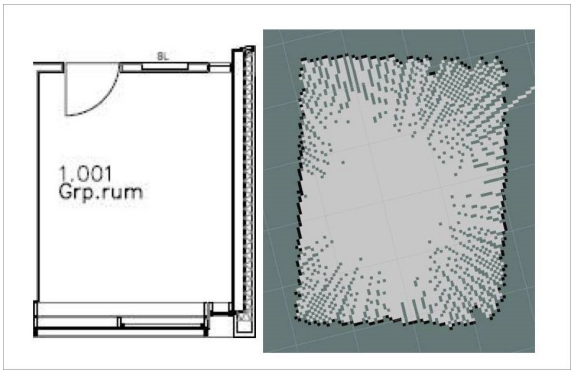


Figure 5.6: Room-plan of H101, H-building, AAU Esbjerg

Figure 5.6 shows another test using hector slam, instead of pointing out data from the previous test. This test was conducted without the use of obstacles, since we had a floor plan supplied by Aalborg University Esbjerg to compare it to. The room H101 in H-building, has a feature in the bottom right corner. This feature is also noticeable on the map created using hector slam.

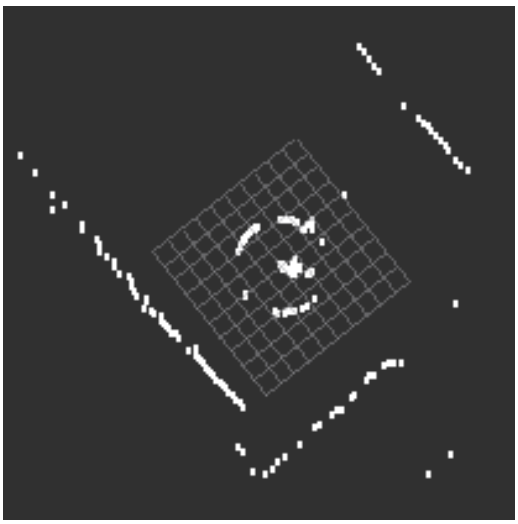
The map has a single point which is traced out of the picture, this is from the laser reflecting poorly on a single spot along the aluminium from on blackboard, which was mounted in room H101.

5.2.2 Outdoor test

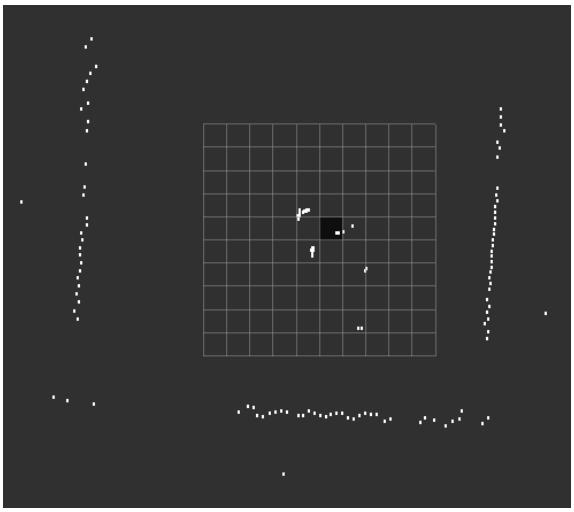
The outside tests were conducted between H-building, the main building and Idea House, located on the Esbjerg campus of Aalborg University. The laptop with the laser module was placed on the benches between the above mentioned area and a scan was run for approximately 4 minutes.

Figure 5.7: The site for the outdoor testing

While running testing outside in areas with distances longer than 40m the laser outputted a measurement of 255cm. The reason behind this is that if the laser does not get a signal back it prints a error message with the integer 255. If that is not dealt with in the code the */scan* topic will output a measurement of 2.55m. We included a if statement that write 0 if a message with 255 is received, to temporarily fix the issue and get a correct 2D map generated.



(a) Circular error present



(b) No more circular error, dots representing people, objects in the middle

Chapter 6

Discussion

During the initial stages of the project, the goal was that we wanted to develop an autonomous mapping vehicle. We decided that ROS would be the core of our prototype, because at the time, it seemed like the most optimal and most powerful choice for our main operating system. One of the reasons being, is that there is a lot of documentation and resources about ROS, and people had also previously successfully run ROS and created maps using SLAM on a Raspberry Pi [85] [86].

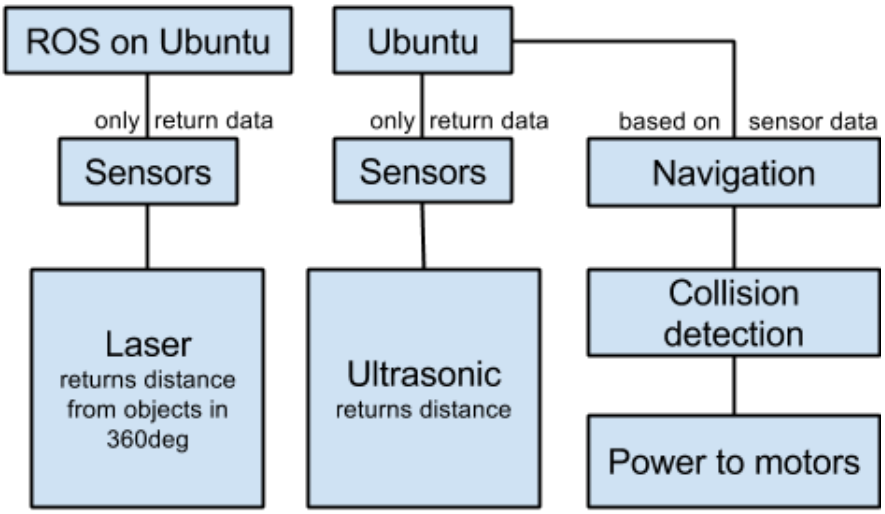


Figure 6.1: Diagram of the modules we have developed

Figure 6.1 represents the actual implementation of the mapping and navigation system that we built during the development phase of the project. Compared to the initial plans, the *optical flow* sensor was not implemented due not being necessary when using Hector-slam for reference points, as it does its own reference points out of the box. The *remote* module was completely left out of the development, due to the limited time we had to develop it.

Due to an overwhelming time spent on ROS and the laser sensor, we did not manage to create a fully autonomous navigation system, instead we only implemented

collision detection, and therefore we decided to split the prototype into two separate parts:

- Navigation system
- Mapping system

The ROS installation process proved to be quite the challenge, which then lead to a setback in terms of development. Many of the ROS packages caused different conflicts with the Raspbian operating system, even though these packages should have been compatible with the system by default. After spending too much time on fixing the many errors and getting it to work on Raspbian, we decided to switch to Ubuntu ARM, where the installation process was much easier compared to the previous one.

For the prototype, we used a CAD design we found on the Internet as a case and rotation platform for the laser. This design was good enough for a prototype, but one of the design-faults we observed was the jittering effect of the gears when rotated by a stepper motor. This forced us to do small step-based calculations using the laser, because it would not take precise microsteps, and therefore our measurements would be off. To overcome this, we needed to speed up the rotation. We had separate code for rotating the laser and reading measurements from it, running asynchronously, which made it hard to synchronize the measurements with the rotation , and therefore the maps plotted were too off. We calculated how much time it took to gather readings and tried adjusting the rotation speed accordingly, which in the end turned out to fix our issues, but the speed was too slow compared to our expectations. It took approximately 5 seconds to take a full circle while gathering readings at every single step the stepper-motor took. Neither the wiringPi or lidarLite libraries, used for communicating with the laser through I^2C, were well optimized for the application we wanted to use them in, namely faster data acquisitions. We think this was caused because of a mixture of async- and synchronous code executed in these libraries when writing and reading through the I^2C registry for acquiring new laser data. Also, the laser rotation platform was pretty well thought out, but if our resources would have allowed us, we would have designed our own to fit everything into it and to potentially design better gears.

Hypothetically, if the mapping device would be mounted to the rover, which would be able to navigate areas whilst the mapping device creates a map. However, due to the low-intelligence of the rover, fast speed of movement and the slow rotation and reading of the laser, it would not be able to distinguish unknown places from known places. It would function purely by avoiding an object and thereafter determining an optimal direction away from said object, not navigating based on the map.

The close proximity detection system, is a low-intelligence autonomous navigation system for the rover. If we succeeded in completing our original scope, the autonomous navigation would have needed to interface with sensors to ensure that it avoided objects at a height that is not recorded by the LIDAR-Lite laser and mapping algorithms. So what we have currently achieved is a navigation system that is able to successfully navigate between obstacles by avoiding them and recalculating

its direction according to where it detected the obstacles.

During testing we observed a couple of key-flaws in terms of the close proximity detection systems performance. As described in the testing chapter, the rover would at times get stuck in corners due to the equality of the measurements recorded by the ultrasonic sensors. Whenever the ultrasonic sensors mounted on either side of the rover would have had their distance threshold exceeded the rover would continuously try to turn to left, hit the one wall of the corner, and then immediately attempt to turn right, hitting the other wall. This can be fixed by changing the code, so that it reverses the rover when the threshold for the side mounted sensors is exceeded. Currently the rover takes three measurements, and then afterwards compares them in the same sequence: *Left -> Right -> Center*. This unfortunately always prioritizes turning left, compared to other choices. Optimally the code should be structured to that the navigation algorithm consists of multiple phases, so that the rover takes measurements in the first phase and before making a directional decision it compares the measurements to each other in the second phase. When the measurements have been compared, the rover then knows what direction would be the optimal to turn towards, in the third phase the measured distance would be compared to a threshold distance, depending on the return from the comparison the rover would then adjust its direction.

Changing the algorithm to something more similar to the one stated above, would add another level of intelligence to the rover, making it more flexible in terms of navigation and decision-making in other situations.

The rover is currently operated by using four DC motors with regular rubber wheels mounted to them. Currently we do not have precise control over how far the rover moves each time it is powered on, although we have encoders mounted on the two front wheels. To increase the accuracy of the distance moved, encoders for the DC motors could have been used to determine the distance travelled. The rover also does not turn consistently each time, which is either an issue with uneven weight distribution on the rover or because of inaccuracy with the signalling. In terms of improving how well the rover turns, the rover needs to be balanced and the amount each wheels turns by cycle needs to be determined. Another possible solution for the turning inconsistency is using tracks for movement instead of four independently operated wheels. Tracks would have a larger guarantee of on-site turning.

Too much time during this project was spent chasing the idea of getting SLAM to work. It was a bad combination of tools, because it seems like that ROS was made for desktop computing and its only recently since embedded systems became powerful enough and compatible to run ROS.

Chapter 7

Conclusion

Based on the test results, we can conclude that our hypothesis was not covered completely, although the prototype is capable of creating a 2D map of its surroundings, but it does not use that data for navigating the maze and finding its path from A to B. Our hypothesis still stands correct and it is doable with more time and resources at hand.

The laser of choice for the prototype seemed to be a good fit, but it does come with certain limitations compared to a higher-end model, like USB support for faster data transfer, better rep rate and acquisition rate. We have to keep in mind, although that it comes with limitations, it does its job in creating a 2D map at quite an inexpensive cost.

Mapping based on multiple sets of measurements, taken from different locations, is an incredibly difficult thing to do. It makes no sense to try and reinvent the wheel, as there are plenty of mapping libraries that are already usable. We choose to use SLAM because it is simultaneous, meaning that the processing of data, mapping and navigating, all go hand in hand. This is the best suited approach for mapping unknown terrains. However, our particular choice of SLAM library did not go very well with our choice of hardware, at least in the time frame we had for the project.

After realizing that getting the mapping part of our prototype to work required more effort than first expected, the scope for the rover navigation system was made much smaller. Instead of a highly intelligent autonomous system, the autonomous navigation turned out to become a close proximity detection system. This system adds a low level of intelligence to the rover, which makes it to avoid objects in its path and change the direction accordingly.

Breaking down the prototype into two separate modules did help us to progress faster with the development, and these could be used as building blocks towards another future project that would take this topic into the next level, where the SLAM technique could be used on a moving rover and processing that information for navigational purposes. Even though we might have not covered all the points in our hypothesis, we did succeed to generate a 2D map using SLAM with a stationary camera and therefore meeting our main goal of creating a map.

Chapter 8

Perspective

Due to our big scope, and the issues we ran into during development, we did not achieve one full prototype. Rather, we achieved two separate prototypes that showcase separate aspects of our scope. The first thing that is necessary for the continuation of development is to combine these two prototypes. Only then, will we have a clear idea of what our prototype is capable of, and what would need to be worked on for the full product.

The laser we built works as a proof-of-concept. Before putting the two prototypes together, it is necessary to vastly increase the turning speed of the laser. The laser can take up to 255 measurements per second but at the current setup we went for precision over speed. Hector-slam was made for faster turning lasers. From our current state, we would need to increase the speed and keep the precision. Once that is done, we can attempt to get SLAM to work on making continues 2D map of its surroundings.

With SLAM working, it is quite simple to put the two prototypes together and get a much better proof-of-concept for our scope. Of course, with the two prototypes together, many adjustments should be made to better work together, such as the navigation system reading the map to know where it has already gone, and where it should try to map next. The rover should have multiple navigation algorithms set up for different environments and tasks.

This proof-of-concept is still only designed for 2D-mapping, and we would like to use this technology on other planets and unmapped territory on Earth. For that task, 3D-mapping is a much better fit. In order to expand our combined prototypes into a 3D capable mapper, we would need a combination of stereoscopic vision, barometric sensors, or any sort of other sensor that can give a height reading. Furthermore, our laser is mostly made for mapping of surfaces that are normal to the direction of the laser, this would not work very well on sloping hills that is found on most uninhabited places.

Hector-slam uses landscape extraction to find reference points in a 2D scan. Using a similar system we could use a 2D spinning laser for obstacle avoidance and navigation through unknown terrain. Our prototypes takes a cheap laser module, that goes for

around 600 DKK [87] plus the costs of the 3D printed case and stepper motor, that can give the same output, and in some cases even better in terms of range and refresh rate, as a 2700 DKK [88] LIDAR model. Implementation of a low cost LIDAR laser system like this combined with small and low cost control unit, like the Raspberry Pi, really opens up possibilities for 2D and 3D mapping of our surroundings and even extend that to other planets.

Bibliography

[1] Thomas Forner Alexander Wolodtschenko. *Prehistoric and Early Historic Maps in Europe: Conception of Cd-Atlas.* e-Perimetron, Vol. 2, No. 2, Spring 2007 [114-116], 2007.

[2] BBC FOCUS Science Lizzy Daw and Technology. Has all of the earth been mapped? http://www.sciencefocus.com/qa/has-all-earth-been-mapped, July 2009.

[3] Mysterious Universe Tom Head. Earth is still an alien planet: 5 habitats we haven't explored. http://mysteriousuniverse.org/2014/09/earth-is-still-an-alien-planet-5-habitats-we-havent-explored/, September 2014.

[4] National Ocean Service. How much of the ocean have we explored? http://oceanservice.noaa.gov/facts/exploration.html, June 2014.

[5] National Aeronautics and Space Administration. Mariner 2. http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1962-041A, August 2014.

[6] National Aeronautics and Space Administration. Venera 3. http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=1965-092A, August 2014.

[7] NASA. Curiosity has landed. http://www.jpl.nasa.gov/video/details.php?id=1103, August 2012.

[8] Planetary Habitability Laboratory University of Puerto Rico at Arecibo Abel Mendez. The potentially habitable universe around us. http://phl.upr.edu/press-releases/thepotentiallyhabitableuniversearoundus, December 2012.

[9] Planetary Habitability Laboratory University of Puerto Rico at Arecibo. http://phl.upr.edu/projects/earth-similarity-index-esi. http://phl.upr.edu/press-releases/thepotentiallyhabitableuniversearoundus, 2011.

[10] Alexis C. Madrigal. Inside the drone missions to fukushima. http://www.theatlantic.com/technology/archive/2011/04/inside-the-drone-missions-to-fukushima/237981/, April 2011.

[11] New Yersey Scuba Diving Rich Galiano. Bottom composition. http://njscuba.net/biology/misc_bottom.php, May 2015.

[12] Stuart Russel and Peter Norvig. *Artificial Intelligence, page 44, section 2.3.2.* Pearson, 2010.

[13] NOAA Ocean Explorer. What is ocean exploration and why is it important? http://oceanexplorer.noaa.gov/backmatter/whatisexploration.html, July 2014.

[14] NASA. Why we explore. http://www.nasa.gov/exploration/whyweexplore/why_we_explore_main.html, September 2013.

[15] NASA. Risk and exploration. http://www.nasa.gov/missions/solarsystem/Why_We_02.html, November 2007.

[16] The Library of Congress. What is a gps? how does it work? http://www.loc.gov/rr/scitech/mysteries/global.html, June 2011.

[17] Invention from NewScientist Blogs. Underwater gps. http://www.newscientist.com/blog/invention/2007/03/underwater-gps.html, March 2007.

[18] Global positioning system. http://en.wikipedia.org/wiki/Global_Positioning_System.

[19] What is a gps? how does it work? http://www.loc.gov/rr/scitech/mysteries/global.html, June 2011.

[20] Dead stars 'to guide spacecraft'. http://www.bbc.com/news/science-environment-17557581, March 2012.

[21] Photogrammetry. http://en.wikipedia.org/wiki/Photogrammetry.

[22] The basics of photogrammetry. http://www.geodetic.com/v-stars/what-is-photogrammetry.aspx.

[23] Georeferencing: a review of methods and applications. http://www.tandfonline.com/doi/full/10.1080/19475683.2013.868826#abstract, January 2014.

[24] et al Mario N. Berberan-Santos. On the barometric formula. http://web.ist.utl.pt/ist12219/data/43.pdf, March 1996.

[25] Ralf D. Tscheuschner Gerhard Gerlich. On the barometric formulas and their derivation from hydrodynamics and thermodynamics. http://arxiv.org/pdf/1003.1508.pdf, March 2010.

[26] Figure for the complex rangefinder. https://electronics.stackexchange.com/questions/83328/real-time-short-distance-range-finder-and-visual-output.

[27] James C. Owens. Optical refractive index of air: Dependence on pressure, temperature and composition. http://www-mipp.fnal.gov/RICH/refractivityOfAir.pdf, January 1967.

[28] E. A. Dean. Atmosphere effects on the speed of sound. http://www.dtic.mil/get-tr-doc/pdf?AD=ADA076060, August 1979.

[29] K. B. Wolf. *Geometry and dynamics in refracting systems.* European Journal of Physics 16: 14–20, 1995.

[30] Figure for refraction of light. http://spmphysics.onlinetuition.com.my/2013/07/refraction-of-light.html.

[31] Reflectivity. http://kb.pulsedlight3d.com/support/solutions/articles/5000554104-reflectivity, February 2015.

[32] RobotWorx. What are autonomous robots? http://www.robots.com/articles/viewing/what-are-autonomous-robots, November 2014.

[33] Sriram Emarose. How to make a simple autonomous vehicle. http://letsmakerobots.com/blog/sriram-emarose/how-make-simple-autonomous-vehicle, July 2013.

[34] Margaret Rouse. Robot (insect robot, autonomous robot). http://whatis.techtarget.com/definition/robot-insect-robot-autonomous-robot, April 2007.

[35] Figure for the a* description. http://www.policyalmanac.org/games/aStarTutorial.htm.

[36] Patrick Lester. A* pathfinding for beginners. http://www.policyalmanac.org/games/aStarTutorial.htm, July 2005.

[37] Tony Stentz. Real-time replanning in dynamic and unknown environments. http://www.extremetech.com/extreme/159347-pathfinding-is-the-key-to-putting-robot-pack-mules-in-the-field, May 2000.

[38] Anthony Stentz. Optimal and efficient path planning for partially-known environments. http://www.extremetech.com/extreme/159347-pathfinding-is-the-key-to-putting-robot-pack-mules-in-the-field, May 1994.

[39] Ibrahim Kamal. Line tracking sensors and algorithms. http://www.ikalogic.com/line-tracking-sensors-and-algorithms/, February 2008.

[40] 5D Robotics. Mapping + localization. https://vimeo.com/44059341, June 2012.

[41] Graham Templeton. Pathfinding is the key to putting robot pack mules in the field. http://www.extremetech.com/extreme/159347-pathfinding-is-the-key-to-putting-robot-pack-mules-in-the-field, June 2013.

[42] Jay Chakravarty. Simulated laser range finder readings using python and opencv. http://www.jaychakravarty.com/?p=75, April 2012.

[43] Louie Huang. Path finding mobile robotics. http://www.seas.upenn.edu/sunfest/docs/slides/HuangLouie05.pdf, January 2015.

[44] Elizabeth Howell. The inner and outer planets in our solar system. http://www.universetoday.com/34577/inner-and-outer-planets/, April 2014.

[45] European Planet Network. Outer planets. http://www.europlanet-eu.org/project/5-why/454-outer-planets.

[46] Lunar and Planetary Institute. Solar system temperatures. https://solarsystem.nasa.gov/multimedia/display.cfm?IM_ID=169, November 2012.

[47] Forsvaret for danmark - aim and tasks. http://forsvaret.dk/FKO/ENG/FACTS%20AND%20FIGURES/AIM%20AND%20TASKS/Pages/default.aspx.

[48] About the police - the national police. https://www.politi.dk/en/About_the_police/national_police/, August 2007.

[49] About the department of electronics, aau. http://www.es.aau.dk.

[50] Forsvaret for danmark - ammunitionsrydning. http://www2.forsvaret.dk/viden-om/indland/ammunitionsrydning/Pages/Ammunitionsrydning2.aspx, December 2013.

[51] Hc-sr04 ultrasonic distance sensor - datasheet. https://docs.google.com/document/d/1Y-yZnNhMYy7rwhAgyL_pfa39RsB-x2qR4vP8saG73rE/edit.

[52] Ros.org. http://www.ros.org/install/.

[53] Raspberry pi specifications. https://www.raspberrypi.org/products/raspberry-pi-2-model-b.

[54] Figure for raspberry pi gpio layout. http://www.policyalmanac.org/games/aStarTutorial.htm.

[55] Raspberry pi installation. https://www.raspberrypi.org/documentation/installation/installing-images/README.md.

[56] Ros - installation of indigo. http://wiki.ros.org/indigo/Installation/Source.

[57] Ubuntu arm - installation. http://www.ubuntu.com/download/server/arm.

[58] Ubuntu arm - installation. http://wiki.ros.org/indigo/Installation/UbuntuARM.

[59] Optical flow using color information: Preliminary results. http://www.dca.ufrn.br/~adelardo/artigos/SAC08.pdf.

[60] High-performance optical mouse sensor (adns-3080). http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2009/ncr6_wjw27/ncr6_wjw27/docs/adns_3080.pdf.

[61] Figure for the opticalflow. http://www.singahobby.com/files/images/Arduino-Optical-Flow-Sensor-3.jpg.

[62] Lidar-lite datasheet. http://kb.pulsedlight3d.com/helpdesk/attachments/5007466446.

[63] Block of the lidar-lite. http://kb.pulsedlight3d.com/support/solutions/articles/5000548641-lidar-lite-block-diagram.

[64] What is the spread of the laser beam? http://kb.pulsedlight3d.com/support/solutions/articles/5000566352-what-is-the-spread-of-the-laser-beam-, May 2015.

[65] Technology and system hardware overview. http://kb.pulsedlight3d.com/support/solutions/articles/5000548635-technology-and-system-hardware-overview, February 2015.

[66] How can i get faster acquisition than 50hz? http://kb.pulsedlight3d.com/support/solutions/articles/5000583786-how-can-i-get-faster-acquisition-than-50hz-, March 2015.

[67] Gordon Henderson. The story of wiringpi. https://projects.drogon.net/raspberry-pi/wiringpi/, May 2013.

[68] lidarlite github repository. https://github.com/answer17/lidarLite.

[69] Configuring i2c. https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c, May 2015.

[70] Laser-safety. http://kb.pulsedlight3d.com/support/solutions/articles/5000548623-laser-safety, February 2015.

[71] Lidar lite product details. https://www.sparkfun.com/products/13167.

[72] 360 degree lidar-lite scanner. https://hackaday.io/project/4087-360-degree-lidar-lite-scanner.

[73] Slip ring datasheet - snm022a-06. https://cdn.sparkfun.com/datasheets/Robotics/SNM022A-06%20update.pdf.

[74] Figure for the slipring. http://img.alibaba.com/img/pb/563/889/871/871889563_606.jpg.

[75] How a slipring works. http://www.electro-miniatures.com/HowSlipRingWorks.shtml.

[76] Hc-sr04 ultrasonic distance sensor - timing chart. http://www.ezdenki.com/graphics/hc-sr04-timing-chart.png.

[77] Hc-sr04 ultrasonic distance sensor. http://arcbotics.com/products/hc-sr04-ultrasonic-distance-sensor/.

[78] Drv8833 picture. https://www.pololu.com/product/2130.

[79] Pololu drv8833 datasheet. https://www.pololu.com/file/download/drv8833.pdf?file_id=0J534.

[80] Nema-17 bipolar stepper motor datasheet. http://www.robotshop.com/media/files/pdf/datasheet-324.pdf.

[81] Easydriver stepper motor controller datasheet. https://cdn.sparkfun.com/datasheets/Robotics/A3967-Datasheet.pdf.

[82] Hector mapping. http://wiki.ros.org/hector_mapping.

[83] Transform points on a robot. http://wiki.ros.org/hector_slam/Tutorials/SettingUpForYourRobot.

[84] Information on rviz. http://wiki.ros.org/rviz.

[85] Robot cartography: Ros + slam. http://www.pirobot.org/blog/0015/, November 2010.

[86] Ros by example indigo - volume 1. http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-indigo-volume-1/ebook/product-22015937.html, January 2015.

[87] Lidar lite. http://www.robotshop.com/en/lidar-lite-laser-rangefinder-pulsedlight.html.

[88] 360 lidar. http://www.robotshop.com/en/rplidar-360-laser-scanner.html.

Chapter 9

Appendix

9.1 Code for close proximity detection

```
#!/usr/bin/python
import time
import RPi.GPIO as GPIO
import argparse
#Ability to read arguments
parser = argparse.ArgumentParser()
parser.add_argument("-d","--distance",help="Add the set distance (def.
    50cm)")
args = parser.parse_args()

#Setting the GPIO library to use the BCM layout
GPIO.setmode(GPIO.BCM)
print("Program started")

#Ultrasonic sensor pins
TRIGGER_LEFT = 4
ECHO_LEFT = 17
TRIGGER_RIGHT = 27
ECHO_RIGHT = 22
TRIGGER_CENTER = 23
ECHO_CENTER = 24
#Motor pins
MOTOR_A1 = 5
MOTOR_A2 = 6
MOTOR_B1 = 13
MOTOR_B2 = 26
#Preset dist
dist_left = 0
dist_right = 0
dist_center = 0
```

```
#Checking for an argument
if args.distance:
    avoid_at = int(args.distance) #cm
else:
    avoid_at = 50 #cm (default)

#How long the rover turns
turn_time = 0.5 #seconds

print("Pin def succes")

#Setting up the GPIO pins
GPIO.setup(TRIGGER_LEFT,GPIO.OUT)
GPIO.setup(ECHO_LEFT,GPIO.IN)

GPIO.setup(TRIGGER_RIGHT,GPIO.OUT)
GPIO.setup(ECHO_RIGHT,GPIO.IN)

GPIO.setup(TRIGGER_CENTER, GPIO.OUT)
GPIO.setup(ECHO_CENTER, GPIO.IN)

GPIO.setup(MOTOR_A1, GPIO.OUT)
GPIO.setup(MOTOR_A2, GPIO.OUT)
GPIO.setup(MOTOR_B1, GPIO.OUT)
GPIO.setup(MOTOR_B2, GPIO.OUT)
print("Pin setup success")

#####################################
#####################################

#function for retrieving the distance
def getdist(trigger, echo):
    #Sending the 10microsecond pulse nedded
    GPIO.output(trigger, True)
    time.sleep(0.00001)
    GPIO.output(trigger, False)
    #When echo goes high the time starts
    while GPIO.input(echo)==0:
        pass
    start = time.time()
    #When echo goes low the time stops
    while GPIO.input(echo)==1:
        pass
    stop = time.time()

    #The time difference * speedofsound
    #Divided by two since the since the distance meassured is back and
    forth
    distance = ((stop-start) * 34000)/2
```

```
    return distance

#The functions for the movement
def forward():
    GPIO.output(MOTOR_A1, True)
    GPIO.output(MOTOR_A2, False)
    GPIO.output(MOTOR_B1, True)
    GPIO.output(MOTOR_B2, False)
    print("Forward.")
    return


def left():
    GPIO.output(MOTOR_A1, False)
    GPIO.output(MOTOR_A2, True)
    GPIO.output(MOTOR_B1, True)
    GPIO.output(MOTOR_B2, False)
    print("Turning left.")
    return

def right():
    GPIO.output(MOTOR_A1, True)
    GPIO.output(MOTOR_A2, False)
    GPIO.output(MOTOR_B1, False)
    GPIO.output(MOTOR_B2, True)
    print("Turning right.")
    return

def stop():
    GPIO.output(MOTOR_A1, False)
    GPIO.output(MOTOR_A2, False)
    GPIO.output(MOTOR_B1, False)
    GPIO.output(MOTOR_B2, False)
    print("Stopping.")
    return

def reverse():
    GPIO.output(MOTOR_A1, False)
    GPIO.output(MOTOR_A2, True)
    GPIO.output(MOTOR_B1, False)
    GPIO.output(MOTOR_B2, True)
    print("Reversing")
    return

#####################################
#####################################

#Setting up the sensor (0.1second to 'load' the module)
```

```
GPIO.output(TRIGGER_LEFT, False)
GPIO.output(TRIGGER_RIGHT, False)
time.sleep(0.1)

#Getting an inital distance
dist_left = getdist(TRIGGER_LEFT, ECHO_LEFT)
time.sleep(0.1)
dist_right = getdist(TRIGGER_RIGHT, ECHO_RIGHT)
time.sleep(0.1)
dist_center = getdist(TRIGGER_CENTER, ECHO_CENTER)
time.sleep(0.1)

print("Inital distance complete")
try:
    print("Navigation loop started")
    while True:
        #The time between each cycle (mininum recommended is 60ms)
        time.sleep(0.200)
        #Measuring the distances
        dist_left = getdist(TRIGGER_LEFT, ECHO_LEFT)
        dist_right = getdist(TRIGGER_RIGHT, ECHO_RIGHT)
        dist_center = getdist(TRIGGER_CENTER, ECHO_CENTER)
        #Moves forward (As a default)
        forward()
        print("Going forward")

        #Checking the different thresholds
        if dist_left < avoid_at:
            right()
            print("dist_left ("+str(dist_left)+") < " + str(avoid_at) + "
   -> Turning right")
            time.sleep(turn_time)
        elif dist_right < avoid_at:
            left()
            print("dist_right ("+str(dist_right)+") < " + str(avoid_at) +
   " -> Turning left")
```

getdist()

9.2 Code for laser sensor module

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>
#include <iostream>
#include <wiringPi.h>
#include <stdlib.h>

extern "C" int lidar_init(bool);
extern "C" int lidar_read(int);

#define PIN16 23
#define PIN18 24

//PIN16 is the pin #16 on the pi diagram
// use for setting up the motorcontroller

//PIN18 is the pin #18 on the pi diagram
// use for controlling the steps

void setUpPins(){
    system("gpio export 23 out");
    system("gpio export 24 out");
    system("gpio -g write 23 1");
}

void step(int steps){
    for(int i = 0; i<steps; i++){
        system("/home/ubuntu/gpioThing.sh");
    }
}

int main(int argc, char** argv){
    int fd = lidar_init(false);//set to true for printouts
    setUpPins();
    // If init failed
    if (fd == -1) {
        printf("initialization error\n");
    }

    ros::init(argc, argv, "laser_scan");

    ros::NodeHandle n;
    ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>("scan",
    10);

    const unsigned int num_readings = 200; //number of readings in one
    packet.
```

```
    const unsigned int num_steps = 1;
    const double laser_frequency = 0.5/*MIGHT CHANGE*/; //how many spins
   in one second
    const unsigned short circle_segments = 1;
    const double pi = 3.141592;

    int i = 0;
    while(n.ok()){
        ros::Time scan_time = ros::Time::now();

        //sets up the relevant info about the laser
        sensor_msgs::LaserScan scan;
        scan.header.frame_id = "laser";
        scan.angle_min = i*((2.0/circle_segments)*pi); // angles in
   radians
        scan.angle_max = (++i)*((2.0/circle_segments)*pi);
        scan.angle_increment = 2*pi / (200/num_steps);
        scan.time_increment = (1 / laser_frequency) / (num_readings);
   //time between measurements in seconds
        scan.range_min = 0.1;//the distance range. in meters
        scan.range_max = 40.0;//set to 40 or less for our laser
        scan.scan_time = 0.02;
        scan.header.stamp = scan_time;

        //publish the data to /scan
        scan.ranges.resize(num_readings);
        for(unsigned int i = 0; i < num_readings; ++i){
            step(num_steps);
            scan.ranges[i] = lidar_read(fd)/100.0;
        }

        scan_pub.publish(scan);
    }
}
```

main.cpp

9.3 Libraries used for the laser sensor module

```
#include <stdio.h>
#include <wiringPiI2C.h>
#include <stdbool.h>
#include <time.h>

#define LIDAR_LITE_ADRS 0x62

#define MEASURE_VAL 0x04
#define MEASURE_REG 0x00
#define STATUS_REG  0x47
#define DISTANCE_REG_HI 0x0f
#define DISTANCE_REG_LO 0x10
#define VERSION_REG 0x41

#define ERROR_READ -1

// Status Bits
#define STAT_BUSY               0x01
#define STAT_REF_OVER           0x02
#define STAT_SIG_OVER           0x04
#define STAT_PIN                0x08
#define STAT_SECOND_PEAK        0x10
#define STAT_TIME               0x20
#define STAT_INVALID            0x40
#define STAT_EYE                0x80

int lidar_init(bool);
int lidar_read(int);
unsigned char  _read_byte(int, int);
unsigned char  _read_byteNZ(int, int);
unsigned char  _read_byte_raw(int, int, bool);
unsigned char lidar_version(int) ;
unsigned char lidar_status(int);
```

lidarLite.h

```
#include "lidarLite.h"

bool _dbg;

// Initialize wiring I2C interface to LidarLite
int lidar_init(bool dbg) {
    int fd;
    _dbg = dbg;
    if (_dbg) printf("LidarLite V0.1\n\n");
    fd = wiringPiI2CSetup(LIDAR_LITE_ADRS);
    if (fd != -1) {
        lidar_status(fd);  // Dummy request to wake up device
        delay (100);
        }
    return(fd);
    }

// Read distance in cm from LidarLite
int lidar_read(int fd) {
   int hiVal, loVal, i=0;

   // send "measure" command
   hiVal = wiringPiI2CWriteReg8(fd, MEASURE_REG, MEASURE_VAL);
   if (_dbg) printf("write res=%d\n", hiVal);
   delay(20); // DO NOT CHANGE! Might get inconsistent readings from the
    sensor

   // Read second byte and append with first
   loVal = _read_byteNZ(fd, DISTANCE_REG_LO) ;
   if (_dbg) printf(" Lo=%d\n", loVal);

   // read first byte
   hiVal = _read_byte(fd, DISTANCE_REG_HI) ;
   if (_dbg) printf ("Hi=%d ", hiVal);

   return ( (hiVal << 8) + loVal);
}

unsigned char lidar_version(int fd) {
    return( (unsigned char) _read_byteNZ(fd, VERSION_REG) );
}

unsigned char lidar_status(int fd) {
    return( (unsigned char) wiringPiI2CReadReg8(fd, STATUS_REG) );
}

void lidar_status_print(unsigned char status) {
    if (status != 0) printf("STATUS BYTE: 0x%x ", (unsigned int) status);
    if (status & STAT_BUSY) printf("busy \n");
```

```
    if (status & STAT_REF_OVER) printf("reference overflow \n");
    if (status & STAT_SIG_OVER) printf("signal overflow \n");
    if (status & STAT_PIN) printf("mode select pin \n");
    if (status & STAT_SECOND_PEAK) printf("second peak \n");
    if (status & STAT_TIME) printf("active between pairs \n");
    if (status & STAT_INVALID) printf("no signal \n");
    if (status & STAT_EYE) printf(" eye safety \n");
}

// Read a byte from I2C register.  Repeat if not ready
unsigned char  _read_byte(int fd, int reg) {
    return _read_byte_raw(fd, reg, true);
}

// Read Lo byte from I2C register.  Repeat if not ready or zero
unsigned char  _read_byteNZ(int fd, int reg) {
    return _read_byte_raw(fd, reg, false);
}

// Read byte from I2C register.  Special handling for zero value
unsigned char  _read_byte_raw(int fd, int reg, bool allowZero) {
    int i;
    unsigned char val;
    delay(1);
    while (true) {
        val = wiringPiI2CReadReg8(fd, reg);
        // Retry on error
        if (val == ERROR_READ || (val==0 && !allowZero) ) {
            delay (20) ;      // ms
            // if (_dbg) printf(".");
            if (i++ > 50) {
                // Timeout
                printf("Timeout");
                return (ERROR_READ);
            }
        }
        else return(val);
    }
}
```

lidarLite.c

WiringPi library to be found here: https://github.com/WiringPi/WiringPi.

9.4 General instruction for project

This meant to be a general guideline, an abstract overview of the necessary steps to be taken to replicate the project described in the report:

- Install Ubuntu ARM on a Raspberry Pi 2 Model B (or newer, similar hardware configuration).
- Install ROS Indigo distribution for Ubuntu ARM with desktop tools.
- Clone code, install libraries, dependencies.
- Setup nodes, run roscore, start necessary nodes.
- Using rviz, visualize the generated map.