COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DESARROLLO DE UN ROVER SEMIAUTÓNOMO PARA EXPLORACIÓN URBANA CON NAVEGACIÓN POR RED NEURONAL ENTRENADA EN SIMULACIÓN

Autor: Jorge Huete Solís

Director: Pedro Olmos González

Madrid

Junio de 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título:

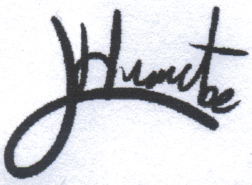
"Desarrollo de un ROVER semiautónomo para exploración urbana con navegación por red neuronal entrenada en simulación"

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2019/2020 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada

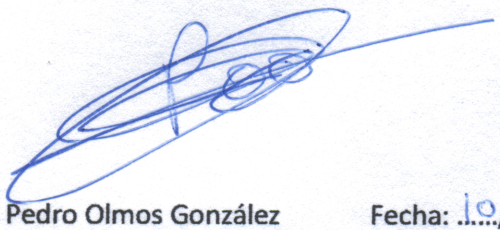
de otros documentos está debidamente referenciada.



Fdo.: Jorge Huete Solís Fecha: 09/ 07/ 2020

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Fdo.: Pedro Olmos González Fecha: 10/ 7/ 2020

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DESARROLLO DE UN ROVER SEMIAUTÓNOMO PARA EXPLORACIÓN URBANA CON NAVEGACIÓN POR RED NEURONAL ENTRENADA EN SIMULACIÓN

Autor: Jorge Huete Solís

Director: Pedro Olmos González

Madrid

Junio de 2020

DESARROLLO DE UN ROVER SEMIAUTÓNOMO PARA EXPLORACIÓN URBANA CON NAVEGACIÓN POR RED NEURONAL ENTRENADA EN SIMULACIÓN

Autor: Huete Solís, Jorge.
Director: Olmos González, Pedro.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN DEL PROYECTO

Desarrollo hardware y software para crear un robot que recorre autónomamente su entorno. Se examinan los requisitos de hardware para construir un sistema capaz de moverse y construir un mapa con un sensor LIDAR. Se lleva a cabo también una combinación de algoritmos de visión artificial, búsqueda de ruta y evasión de obstáculos para reconocer y navegar el entorno. Esto forma una base para una red neuronal que por aprendizaje por refuerzo controla el robot en situaciones complejas. Se muestran y discuten los resultados en un entorno de 80m^2.

Palabras clave: Robótica, LIDAR, Deep Q-Learning, Visión Artificial, Algoritmos de Navegación, Red Neuronal, Machine Learning

1. Introducción

Ciertos entornos, como aquellos contaminados por materiales radiactivos o tóxicos, son incompatibles con la vida humana. Es común utilizar en estos entornos herramientas por control remoto. No obstante, las condiciones de la conexión pueden hacer esto poco práctico o incluso imposible. Por ejemplo, el agua del mar obliga a unir vehículos subacuáticos y pilotos por un cable que limita el alcance y prestaciones del robot. Los enlaces interplanetarios tienen latencias altas que imposibilitan el control directo. La tarea a realizar también puede hacer inviable el uso de un piloto humano. Tareas repetitivas como vigilancia o exploración crean situaciones de baja atención y aumentan el riesgo de accidentes u omisiones. Este es de hecho uno de los principales argumentos a favor de la automatización de vehículos en carretera.

En estas situaciones, un sistema autónomo es preferible o incluso imprescindible. Un sistema capaz de recibir instrucciones, crear una cola de tareas y reportar cuando sean realizadas. La utilización de redes neuronales para robótica y navegación han abierto las puertas a la creación de sistemas capaces de aprender a interactuar con su entorno de una manera innovadora [1].

Para ello se describe en este trabajo el proceso de desarrollos hardware y software del sistema necesario. Todos los archivos e información pertinente se pueden encontrar en un repositorio referenciado.

2. Definición del proyecto

Este proyecto está dividido en dos partes principales: hardware y software. La parte hardware es una plataforma robótica alimentada por baterías capaz de moverse gracias a una seria de motores y orugas. Para alimentar los distintos componentes, la placa principal contiene dos convertidores reductores con la consideración de mantener una alta eficiencia. Dado que es necesario mover los motores en ambas direcciones, se utilizan dos puentes H MOSFET.

La placa principal está también equipada con un sistema de monitorización analógico. Este sistema es utilizado para medir la tensión de múltiples buses en la placa y comprobar el correcto funcionamiento. Esto también permite desconectar subsistemas en caso de fallos eléctricos o para disminuir consumo. Una batería de reserva asegura alimentación al procesador para garantizar la comunicación en situaciones de subtensión.

La sección software supone la mayor parte de este proyecto. El sensor LIDAR requiere un controlador dedicado para poder operar a una alta frecuencia de muestreo. Debido a las limitaciones del protocolo de comunicación, se ha requerido un esfuerzo significativo para reducir el tamaño de las muestras y recuperar la información. Se ha utilizado un protocolo sobre TCP para conectar el robot (servidor) con un controlador. A través de esta conexión, el robot puede recibir órdenes, configuración y devolver resultados y telemetría.

La mayor parte del desarrollo software está basada en la aplicación de técnicas de visión artificial sobre datos del sensor LIDAR. Esto incluye por ejemplo la transformada de Hough, utilizada extensamente en la detección de muros y obstáculos. Estas técnicas son parte esencial de un algoritmo para la detección de desplazamientos y rotaciones indeseadas del robot. El algoritmo de clustering DBSCAN es utilizado en varias ocasiones para la identificación de errores y reconocimiento de patrones en estos datos.

El robot crea y actualiza un mapa a lo largo del proceso de exploración. El sistema emplea un algoritmo de reputación para reconocer información desactualizada e incorrecta. Esto permite utilizar el algoritmo de navegación A* para calcular de forma eficiente que puntos son alcanzables y una ruta aproximada.

Por último, se entrena una red neuronal para evitar obstáculos cercanos. Se diseña un entorno simulado y funciones de recompensa para el entrenamiento. La exhaustiva exploración de hiperparámetros de este proceso se realiza en paralelo en la nube.

3. Resultados

Hardware

La placa principal cumple satisfactoriamente con todos los requisitos. Tiene una eficiencia de batería a cargas superior al 80%. Operando el ordenador de abordo y sensor, tiene un tiempo de vida superior a las 26 horas sin contar la reserva. En modo de reposo, este periodo se podría extender hasta 2 días. No obstante, las facultades y precisión del sistema se podrían aumentar significativamente añadiendo un microcontrolador. Además incluye un sistema capaz de monitorizar multiples tensiones en la placa para comprobar el correcto funcionamiento.

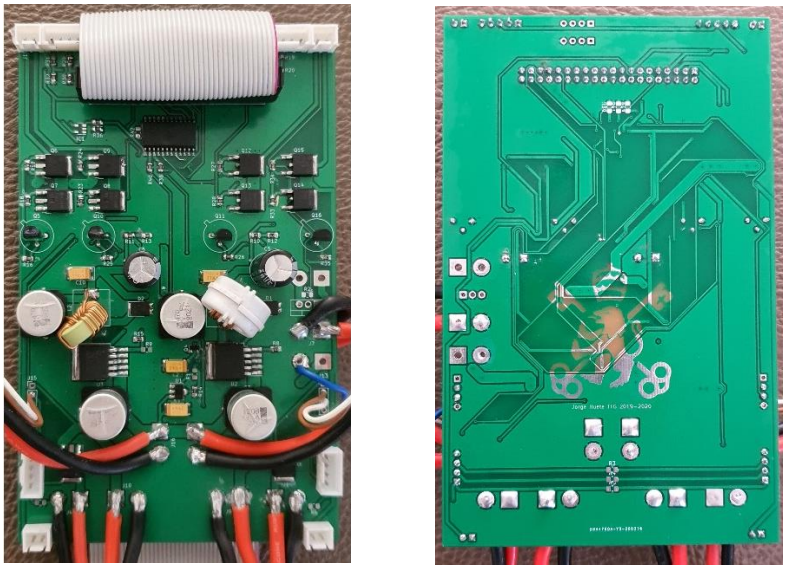


Figure 1: Placa principal. Frontal (izqd) y dorsal (drch)

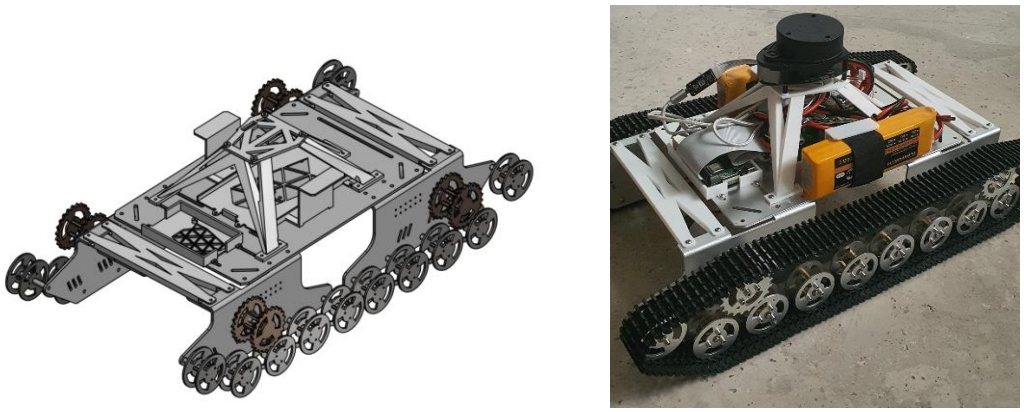


Figure 2: Plataforma robótica. Montaje en Inventor (izqd) y fotografía (drch)

Las orugas de la plataforma resbalan excesivamente sobre el terreno. La elección de orugas en lugar de ruedas tenía como objetivo hacer el movimiento más predecible. Sin embargo, debe realizarse una considerable corrección por software para compensar este rozamiento. Los

componentes son altamente modulares y el montaje está diseñado para simplificar el acceso de forma independiente.

Software

Los resultados del software son en gran medida satisfactorios, pero con evidente margen para mejora. La siguiente figura muestra el mapa resultante de la exploración de un espacio de 80m^2. Aunque se asemeja al plano de referencia, se producen ciertas distorsiones derivadas de los problemas de posicionamiento heredados en parte del hardware.

La evasión de obstáculos por red neuronal es funcional en situaciones simples. No obstante, los resultados satisfactorios del entrenamiento en simulación no se traducen a la vida real en situaciones más complejas.

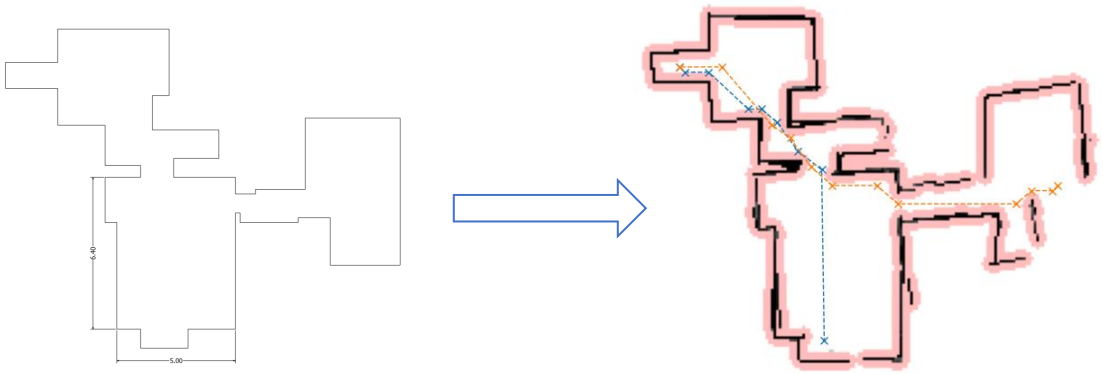


Figure 3: Plano de un edificio residencial (izqd) y mapa producido por el robot (drch)

4. Conclusión

Este proyecto ha logrado el desarrollo de un robot con un alto grado de autonomía capaz de reconocer su entorno y descubrir rutas para alcanzar objetivos establecidos. Esto permite la navegación en entornos remotos o no aptos para seres humano sin necesidad de información previa.

Para lograrlo se ha diseñado una plataforma robótica modular, alimentada por baterías con gran autonomía. Cuenta con múltiples sistemas de seguridad y monitorización para asegurar su correcto funcionamiento. El software diseñado utiliza los datos del sensor principal para identificar muros y obstáculos en el entrono y trazar un mapa. Estos datos también son utilizados para corregir los errores en el movimiento y navegación. La combinación de

algoritmos de búsqueda a gran escala y redes neuronales a pequeña escala permite una navegación a la vez eficaz y adaptativa.

Aunque los resultados son mayoritariamente satisfactorios, existen ciertas áreas de mejora tales como optimizar la fiabilidad del posicionamiento y movimiento. La navegación por red neuronal también podría beneficiarse de una mayor complejidad como la encontrada en redes "en duelo" [2] [3].

5. Referencias

[1] J. Bongard, V. Zykov and H. Lipson, "Resilient Machines Through Continuous Self-Modeling," *Science,* vol. 314, no. 5802, 2006.

[2] G. Khan, A. Villaflor, B. Ding, P. Abbeel and S. Levine, "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation," 2017.

[3] H. Surmann, C. Jestel, R. Marchel, F. Mesberg, H. Elhadj and M. Ardani, "Deep Reinforcement learning for real autonomous mobile robot navigation in indoor environments," *Cornell University,* 2020.

[4] S. Bhattacharjee, K. Kabara and R. Jain, "Autonomous drifting RC car with reinforcement learning," *University of Hong Kong,* 2018.

[5] N. Kohl and P. Stone, "Machine Learning for Fast Quadrupedal Locomotion," *Univerisy of Texas,* 2004.

DEVELOPMENT OF A SEMIAUTONOMOUS ROVER FOR URBAN EXPLORATION WITH NEURAL NET NAVIGATION AND SIMULATED TRAINING

Author: Huete Solís, Jorge.
Supervisor: Olmos González, Pedro.
Collaborating Entity: ICAI – Comillas Pontifical University

ABSTRACT

Hardware and software developments for an autonomous robot cable of mapping and exploring its environment. The hardware requirements for a system capable of moving on its own and creating a map using a LIDAR sensor are examined. A series of algorithms for computer vision, pathfinding and obstacle avoidance is utilized. This is used as a platform for a neural network trained by reinforcement learning to navigate complex scenarios. Results for an $80m^2$ environment are showcased and discussed.

Keywords: Robotics, LIDAR, Deep Q-Learning, Computer Vision, Pathfinding, Neural Network, Machine Learning

1. Introduction

Some environments are not suitable for human life, such as those contaminated by radioactive or toxic materials. It is common to use remotely operated tools in such scenarios. However, the link's limitations may make this impractical or even unfeasible. For example, seawater forces the use of tethers to connect underwater robots to human pilots on the surface. This limits the range and reduces overall flexibility. A similar example would be interplanetary link latency. Long delay between pilot and robot makes direct control impossible. The assigned task can also make the use of human pilots undesirable. Repetitive tasks such as security patrols or exploration are highly monotonous and greatly increase the risk of accidents or oversights. This is in fact one of the major arguments for the use of artificial intelligence in road vehicles.

In this sort of situations, an autonomous system is preferable or even a requirement. A system capable of receiving and queuing orders and reporting back once they are completed. The use of neural networks for robotics and navigations has opened the doors to the creation of systems capable of learning and interacting with their environments in an innovative way. [1]

This project describes the hardware and software development processes for such a system. All files and relevant information are available in a referenced repository.

2. Project definition

This project is divided in two main parts: hardware and software. The hardware section is a robotic platform powered by batteries and capable of moving by means of a series of motors and continuous tracks. To power the multiple components in the system, a main board housing two step-down converters was designed and manufactured with efficiency as one of the main concerns. To accommodate the need of bidirectional motion of the motors, they are powered by two MOSFET H-Bridges.

The main board is also to be equipped with an analog monitoring system. This system is used to sample multiple voltage levels on the board buses to verify appropriate operation. This also allows switching down subsystem in case of electrical issues or to save power. A backup battery system ensures CPU power for communication in case of undervoltages.

The software development represents the largest part of this project's development. The LIDAR sensor required a dedicated driver to operate at a high sampling frequency. Given the limitations of the communication protocol, a significant effort to reduce the size of samples must be done. This also required a more complex data recovery process. A protocol over TCP is utilized to connect robot (server) and controller. Through this connection it is possible to send orders, configuration to the robot and receive back data.

The main part of the software development is based on the application of computer vision to LIDAR sensor data. This includes the extensive use of Hough's transform for the detection of walls and obstacles. These techniques are an essential part of an algorithm to detect the undesired rotations and translations of the robot. The DBSCAN clustering algorithm is repeatedly used to correct error and identify patterns in this data.

The robot generates and maintains a map of its environment along the exploration process. The system employs a reputation algorithm to recognize stale or otherwise incorrect information. This allows the use of the A* pathfinding algorithm to efficiently calculate which points are reachable and an approximate route.

Lastly, a neural net is trained to avoid nearby obstacle. A simulated environments and reward functions are designed for this purpose. The exhaustive exploration of training hyperparameters is computed in parallel in the cloud.

3. Results

Hardware

The main board adequately meets all the minimum requirements. It has an efficiency of over 80% from batteries to load. Operating both the onboard computer and sensor, it has a battery life of over 26 hours without accounting for reserve power. In a low-power state, this time is realistically extended to 2 days. However, the capabilities and precision of the system could greatly be improved by adding a microcontroller. Furthermore, it includes monitoring system capable of measuring multiple voltage points along the board to verify the correct operation.

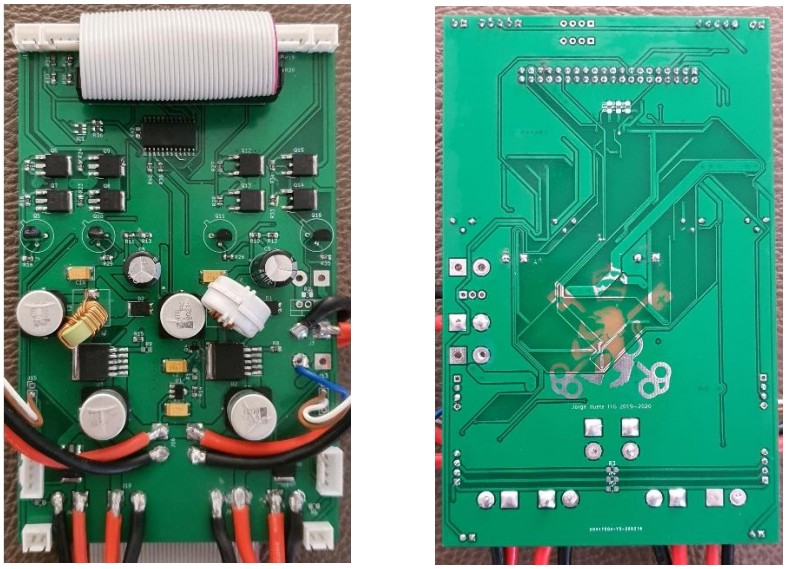


Figure 4: Main PCB. Front (left) and back (right)

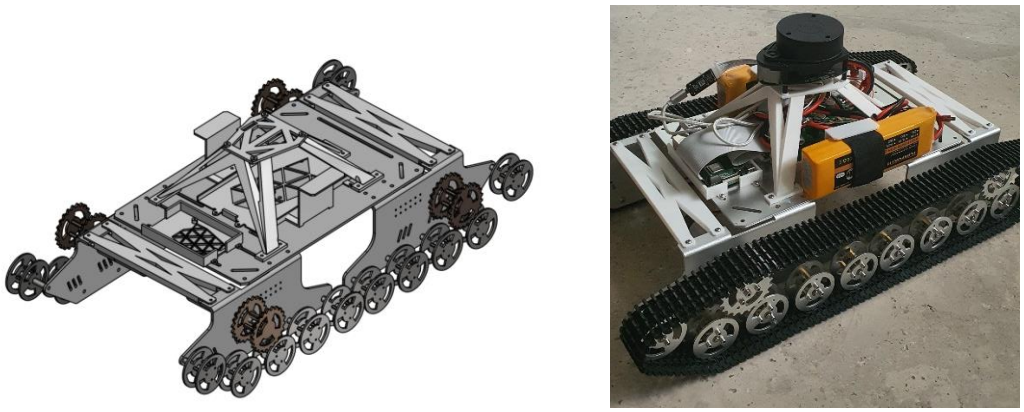


Figure 5: Robotic platform. Inventor assembly (left) and photograph (right)

The tracks slide noticeable over the terrain. The choice of continuous tracks instead of more traditional wheels was taken to make the movement more predictable. However, significant

software correction must be made to compensate for this sliding. The components are highly modular, and the assembly is designed to grant independent access to them.

Software

The software results are largely satisfactory. However, there is still large scope for improvement. The following figure shows the map resulting from de exploration of an 80m^2 space. Although it resembles the reference floor plan, significant positioning issues, partly inherited from the hardware, create issues of distortion.

The neural network obstacle avoidance is functional in simple situations. However, the satisfactory results in the simulated environment does not traduce to real life in more complex situations.

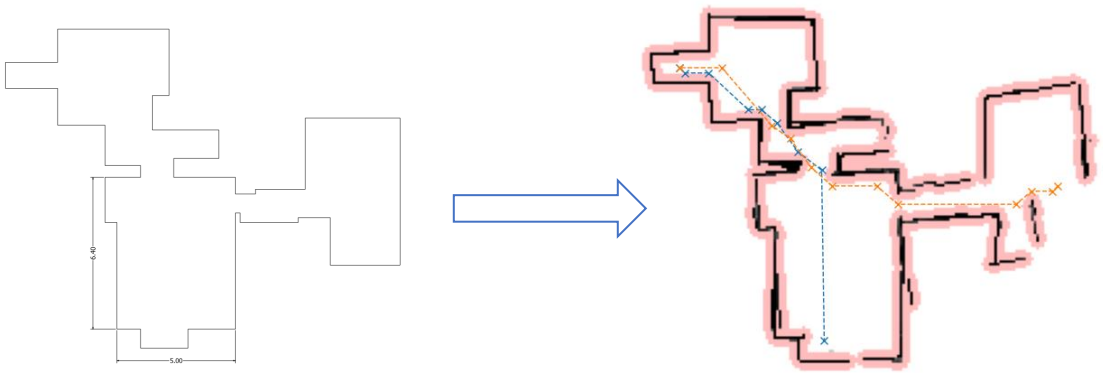


Figure 6: Floor plan of a residential building (left) and exploration result (right)

4. Conclusion

This project has achieved the development of a robot with a high degree of autonomy capable of recognizing its surroundings and discovering routes to reach established waypoints. This allows navigation in environments that are remote or unfit for humans without the need of prior information.

To achieve this, a modular robotic platform powered by batteries with long battery life was designed and constructed. It counts with multiple safety and monitoring systems to ensure correct operation. The designed software uses data from the main sensor to identify walls and obstacles in the environment and create a map. This data is also used to correct errors in movement navigation. The combination of search algorithms at large-scale and neural network at small-scale allow for a navigation that is both efficient and adaptive.

Although the results are mostly satisfactory, there are certain areas for improvement such as optimizing the reliability of positioning and movement. Neural network navigation could also benefit from greater complexity such as that found in dueling networks [2] [3].

5. References

[1] J. Bongard, V. Zykov and H. Lipson, "Resilient Machines Through Continuous Self-Modeling," *Science,* vol. 314, no. 5802, 2006.

[2] G. Khan, A. Villaflor, B. Ding, P. Abbeel and S. Levine, "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation," 2017.

[3] H. Surmann, C. Jestel, R. Marchel, F. Mesberg, H. Elhadj and M. Ardani, "Deep Reinforcement learning for real autonomous mobile robot navigation in indoor environments," *Cornell University,* 2020.

[4] S. Bhattacharjee, K. Kabara and R. Jain, "Autonomous drifting RC car with reinforcement learning," *University of Hong Kong,* 2018.

[5] N. Kohl and P. Stone, "Machine Learning for Fast Quadrupedal Locomotion," *Univerisy of Texas,* 2004.

1. Table of Contents

2. Table of Figures

3. Introduction

3.1 Project Motivation

There are some situations where human presence is not possible or desirable. These scenarios are normally created by conditions dangerous to human life such as the presence of radioactive, or toxic material. They can also be created by space constraints. Collapsed buildings or very tight spaces need to be cleared before a human can work. Some activities would require sending a human impractically or impossibly far, such as space exploration.

In these circumstances, the use of remotely operated vehicles is common. However, if this vehicle requires a human pilot, some other constraints arise. Tasks can be very monotonous, creating situations where the human pilots become tired or distracted. A human pilot operating a vehicle for security patrols for example, would likely become complacent and absent-minded. This is, in fact, the most cited desirability of self-driving cars. Other constraints can also arise from the nature of link between pilot and vehicle. Due to the properties of water, underwater vehicle requires long and expensive tethers to control, limiting its range and flexibility. Another example would be space exploration vehicles that cannot be controlled manually due to the long latency of interplanetary links.

In these circumstances, a human pilot must be substituted by an autonomous system. These systems schedule and execute orders and waypoints to later return results to a controller.

These autonomous systems, however smart, might find new situations that they were not specifically programmed to deal with. There has been a significant use of machine learning algorithms in robotics to give robots the ability to learn a new task [1].

3.2 Project Goals

For the system to explore its environment, it needs to be moveable, and therefore, powered by batteries. It is also of importance that it is highly efficient. Higher efficiencies would allow the robot to explore further and train for longer periods. Since the connection to the controller could be interrupted for long periods of time, the system can enter a low-power state until polled.

Given that the main goal is independent exploration, the robot needs to create a map of its environment. This task will require many subsystems and algorithms to perform. In order to situate new data, the robot

requires precise positioning and movement. While this can be done to a certain degree with simple rotary encoders, as it is discussed at length below, it requires more complex algorithms to correct for errors.

In order to create a distinction between obstacles and walls, a significant amount of point cloud data processing must be done. Even a simple task, such as identifying straight lines in the data, has multiple smaller problems than need to be solved. This will be done by applying multiple computer vision and clustering algorithms on discretized LIDAR data.

Identified walls and obstacles must be used to create a map of the environment. This map is necessary for pathfinding purposes and will be considered the principal output of the system. It requires a reputation system to identify and correct stale or incorrect information.

A certain degree of autonomy is also expected from the system. Although the main goal is for the system to follow commands received from the controller, it should also be able to independently explore its environment until the completeness of the map has been determined.

To reach a designated waypoint, the system will apply a combination of long-range navigation via a pathfinding algorithm, and a short-range obstacle avoidance via a neural network previously trained in a simulation. The robot's movements and responses will need to me simulated to create a suitable environment such that it reassembles the inputs and results of the real word application.

3.3 Technique used

Unlike other previous publications related to the movement of robots by neural nets and deep reinforcement learning, is the use of abstraction. Most other solutions to this problem use extraordinarily complex neural net structures and paradigms such a dueling nets [4] . In this project however, the neural net structure is a simple, low-depth, sequential network that receives heavily preprocessed information. All the extensive sensor data is processed to extract relevant information by means of computer vision algorithm such as Hough's transform. Robot movement and position tracking is also abstracted by exploiting the LIDAR data algorithmically.

4. State of the art

Although there is significant background to the integration of machine learning in robotics and navigation, this section will showcase four projects and a commercial product line. Three of these projects have significant objective overlap and deal with both reinforcement learning for robotics and navigation.

In 2018, a graduate project from the University of Hong Kong explored the use of the same reinforcement learning algorithm used in this project for controlling a radio-controlled car. The vehicle was controlled by a neural network trained by Q-learning. The reward paradigm reinforced the movement of the vehicle in oversteering situations, what is referred as drifting. In very much a similar fashion to the process followed here, the neural network was initially trained in a simulated environment that attempted to replicate the physics of the problem at hand. The second phase of training was done on the physical robot by tracking its movement with cameras and tracking dots. This paper reported positive results, where the vehicle could maintain a drifting state around a static target indefinitely. However, the paper notes difficulties translating the simulation training to the physical car citing differences between simulated and real movement. [4]

A very recent paper from Cornell University describes a similar concept to the one developed here. Researchers attempted to combine a LIDAR and RGB camera to guide a robot across a cluttered room by using deep reinforcement learning. The two main differences between this approach and this project are the scope of the use of a neural net, and mapping. In this Cornell paper, the robot's movement is completely dictated by the neural net predictions fed by the sensors. On the other hand, the approach followed here relies on the neural net only for obstacle avoidance. This project also incorporates memory in form of an environment map while the Cornell robot only uses present information. [3]

Another Cornell paper from 2017 describes a neural net guided robot using a camera as input. This neural net used the same deep reinforcement learning paradigm as the previous two as well as this project. It also describes a simulated environment used to train the agent before integrating it into the physical robot. Once again, the main difference with this project and Cornell's is the existence of environment memory. [2]

A last example that illustrates the use of machine learning for robot control is a 2004 University of Texas paper describing the use of neural nets and quadrupedal robots. The Texas project did not deal with navigation or pathfinding but purely quality of movement. The movement of these robots is significantly

more complex than the previous ones or the one described here. However, its positive results once again showcase the applicability of reinforcement learning for robotic movement. [5]

Lastly, in the last decade, autonomous vacuum cleaners have increased in capabilities and complexity significantly. The first available models used only IR or collision sensors and roamed a room by nearly blindly walking into walls. However, in the last years they have become extraordinarily complex pieces of engineering. Some of them use LIDAR sensors to map its environments and predict more efficient routes in a remarkably similar to the process used by the robot developed here.

5. Hardware

Given the nature of this project, a significant component of the development is hardware based. Files relating to this section can be found in the "hardware" subfolder of the repository. The "PCB" folder contains EDA schematics and boards layouts for the main board. The "ROVER" folder contains Autodesk Inventor parts, drawings, and STL files for all the other physical components in the prototype.

For the platform to be capable of moving on its own, it requires batteries to be used as a power source. This creates an increased challenge in the form of circuit protection, power management, power supply design, and more. Most of the hardware described here is contained in a single double-layer PCB that lies on the center of the robot's structure. However, all the processing and most of the internal logic is done in software running on a single board PC, a Raspberry Pi 4th generation.

Battery control and protection PCBs are independent from the main bord and are connected by wiring. In a similar fashion, supports and decks are designed to fit together while still being separate. This is done to ensure a modular design and ease the process of repair of upgrade.

5.1 Power Management

Relevant files: '/hardware/PCB'

The use of multicell lithium polymer batteries (LiPo) adds significant challenges. While their high capacity and low internal resistance make them suitable for both high current and long-life applications, mishandling them can have catastrophic consequences. On a similar note, multicell arrangements make these batteries more versatile but require careful cell balancing, especially during charging, to prevent a runaway reaction. [6]

This section illustrates how these issues are tackled for both safety and efficiency purposes.

5.1.1 Batteries

The system is designed to be powered by up to three 3S LiPo batteries but only one of them is required to operate. The 3S notation indicates a three-cell series arrangement. Given a 3.7V nominal LiPo cell voltage, the system's nominal input voltage is 11.1V (ranges 12.6V – 10.2V) [Figure 7].

Two of the three batteries are expected to power the system in normal circumstances. These have the same capacity of 6000mAh each and supply power to the system in parallel. They will be referred to as "main batteries" from this point forward.

The last battery in the system is used as a backup. In its current state, the system uses a smaller 2200mAh battery for this purpose. This battery does not power the system in normal circumstances and is only used in case of an undervoltage from the main batteries. The goal of the backup system is to ensure power to the CPU during faults in order to alert the controller of the issue and ensure communication.

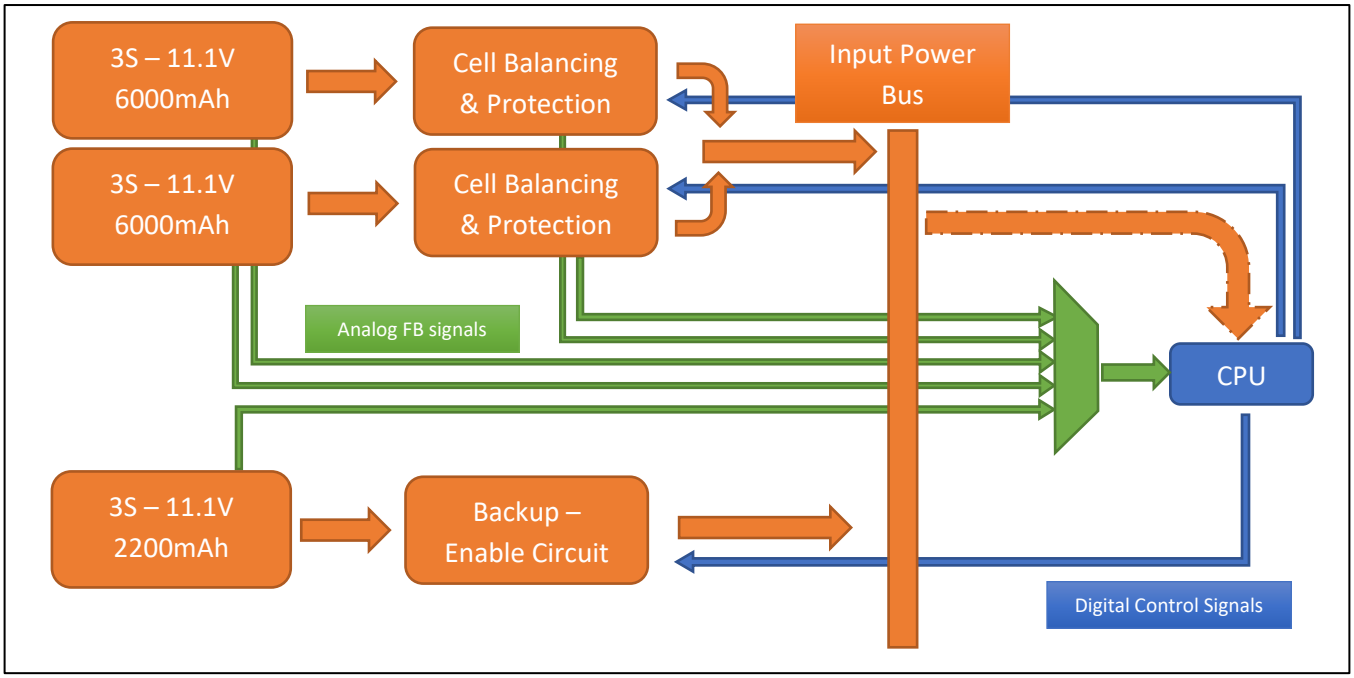


Figure 7: Power delivery and control flow diagram

5.1.2 Reverse polarity protection

When completely charged, the robot has approximately 570kJ of stored energy onboard. With very low internal resistance, connecting one of the batteries in reverse could cause irreparable damage to the robot and create a fire hazard. For this reason, most removeable batteries are fitted with polarized connectors to prevent the user from creating the aforementioned situation. While the main board uses T-plugs [Figure 8], it was deemed necessary to implement a second layer of electrical reverse polarity protection (RPP). This is even more important when considering that the circuit is commonly powered by non-polarized crocodile-type cables for testing and charging.

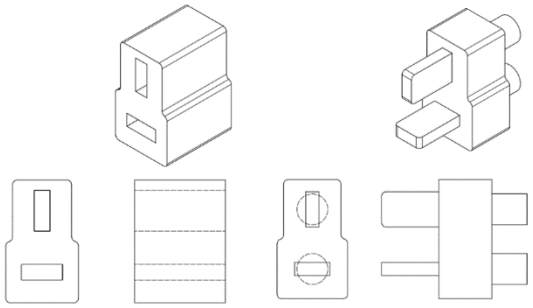


Figure 8: T-Plug polarized connector – Female Left – Male Right

The simplest way of achieving sufficient RPP is to use a diode in series with the battery connector. This prevents reverse current at the cost of a forward diode drop. In applications powered by mains AC this forward voltage might be acceptable if the diode can dissipate the resulting lost power. In battery powered applications however, diode RPP decreases overall efficiency and therefore battery life. At the nominal input voltage, a forward voltage drop of 0.6V represents a reduction of ~5.5% in efficiency. At a maximum current of 5A the diode would dissipate 3W transferring significant heat to surrounding components.

The solution employed in the final circuit is a MOSFET based RPP [Figure 9]. A P-channel FDD6685 has a gate to source threshold voltage of -4V. When the gate is at a potential 4V lower that the source it is in a conductive state. When the battery is connected to the circuit in the correct orientation, some current passes through the parasitic diode and creates a small voltage drop ($Vds \approx -1V$). At this point the source of the transistor is at approximately 10V over the gate resulting in a conducting state. [7]

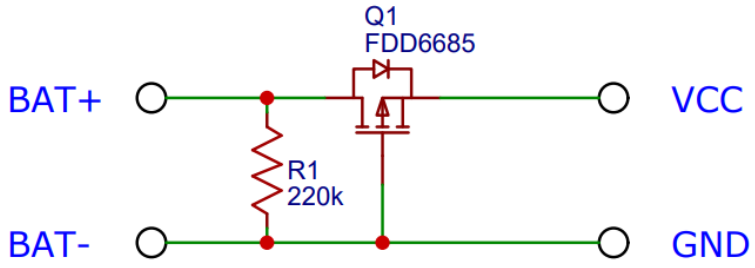


Figure 9: Simple P-MOS reverse polarity protection

In a reverse polarity scenario, the transistor's drain is at negative battery voltage. With no other voltage sources, the source cannot be a higher potential that the gate since no current can pass through the

parasitic diode. The transistor is in an open state. Other implementations of this technique use a clamping Zener diode to protect the gate in this scenario. This is not necessary in this case since the battery voltage is under the rated maximum gate-to-source/gate-to-drain voltages.

According to manufacturer's documentation, at normal gate-to-source voltage, the conductive resistance of the P-MOS is only 20mΩ. At a maximum expected current of 5A, this circuit dissipates 500mW. This indicates that this very economical and simple RPP is 6 times more efficient than a standard diode. [1]

5.1.3 Cell balancing & protection

Since the cells in a battery are not electrically identical, some of them have a different internal resistance and capacity. Simply supplying or draining a certain current will cause the cell voltages to drift apart. In most cases, and particularly low power applications, this is not a major issue but does decrease the battery's lifespan. In some cases, cells can over-discharge, causing irreparable damage [8]. During charging, a cell might over-charge and become a fire hazard. For these reasons, applications powered by multi-cell batteries should use a cell balancing circuit. Cell balancing circuit ensures that cell voltages remain equal during charging and discharging.

During charging, cells are protected from over-charging by allowing them to discharge once they reach a voltage of 4.2V. The HY2213 IC [2] offers precisely that functionality, requiring only an inexpensive external MOSFET and resistor. As shown in Figure 10, it accomplishes this by comparing the cell voltage to an internal reference. The output of this comparison is converted into a logic level control signal for the external transistor, ensuring a maximum update rate set by an internal oscillator.

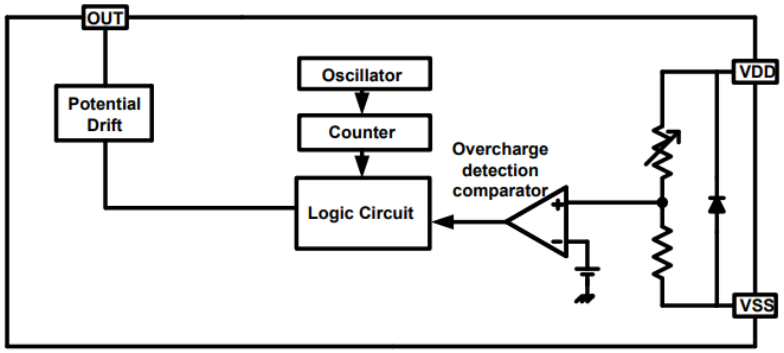


Figure 10: HY2213 block diagram. Source: manufacturer's datasheet

[1] FDD6685 Datasheet: https://www.onsemi.com/pub/Collateral/FDD6685-D.PDF
[2] HY2213 Datasheet: http://www.hycontek.com/wp-content/uploads/DS-HY2213_EN.pdf

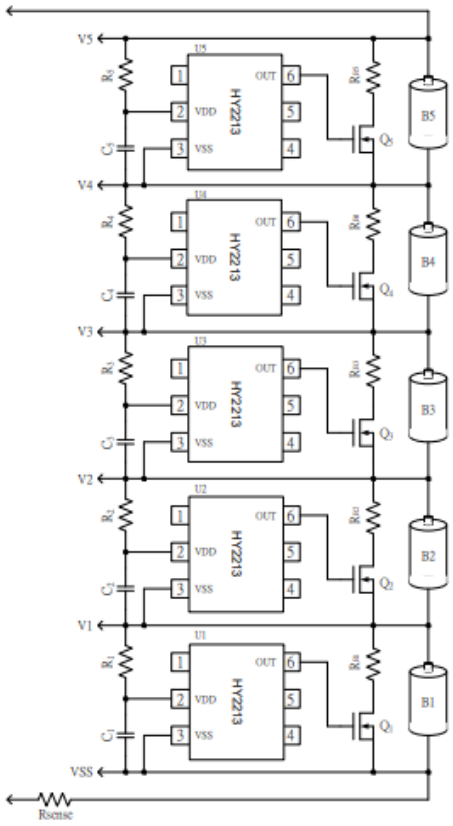


Figure 11: Example 5S arrangement using HY2213

Figure 11 shows a 5S arrangement using this IC. Since the IC is powered by the cell voltage, this arrangement can be used for any number of cells in series. Each HY2213 has a constant operating current of ~2.0µA. Compared to the quiescent current of the entire system this is negligible.

Over-discharge requires a slightly more complex protection circuit. Since the battery cells are internally connected, it is not possible to disconnect one of them from the circuit when it reaches the lower voltage threshold. For this reason, the circuit uses three DW01A[3] over-current/over-discharge ICs.

The DW01As drive two N-channel MOSFETs each to disconnect the cell from the stack. These MOSFET are opposing to make sure no current can pass through the parasitic diodes. The CPU can also disconnect the cells from the stack by setting the "OC" signal high. This signal passes through signal diodes before reaching the transistors to prevent cross talk between the Ics, since they operate at different voltages [Figure 12].

[3] DW01A Datasheet: http://escooter.org.ua/_fr/1/DW01A-DS-10_EN.pdf

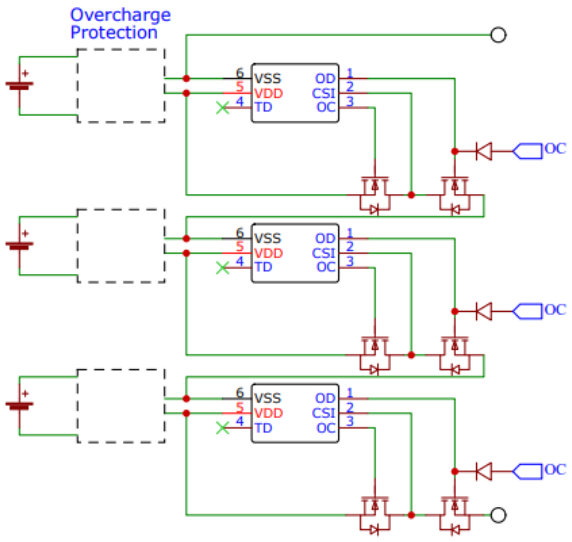


Figure 12: DW01A over-discharge/over-current protection

5.1.4 Backup battery switching

As previously mentioned, one of the batteries is used for backup and remains disconnected in normal circumstances. A N-channel MOSFET (IRF540N) is placed between the negative contact of the battery and the ground rail. The signal named “BCKP_EN” (Backup Enable) controls the state of this transistor. When the signal is low, the transistor’s gate is pulled to source and no current flows through the battery. The signal can be triggered by either the CPU or an operational amplifier used as a comparator. The op-amp circuit is currently unpopulated in the final version since large voltage ripples caused by the motors’ switching sometimes triggers it prematurely. As this battery comes online, the CPU ensures the main batteries get disconnected by triggering their protection circuits.

5.2 Power Bus Generation

Relevant files: ‘/hardware/PCB’

The main two loads in the system, the CPU and the motors, are designed to operate at lower voltages than the batteries’ 11.1V. Therefore, an efficient way of creating a 5V (CPU) and a 9V(motors) high power rails is essential. In addition, two logic level rails are also required: a 5V and a 3V3.

Due to battery life requirements and relative high loads, buck converters are used to generate the 9V and 5V power rails. Buck converters are a type switch mode power regulator that can reach very high efficiencies [9]. The core component used for this application is a TD7590 monolithic step-down controller IC [4]. The TD7590, as most switch mode regulators, only require a few accessory components to create a stable, although noisy, output voltage [Figure 13]. This IC also incorporates overcurrent protection, shutting down the output if it reaches a threshold between 6A and 6.5A.

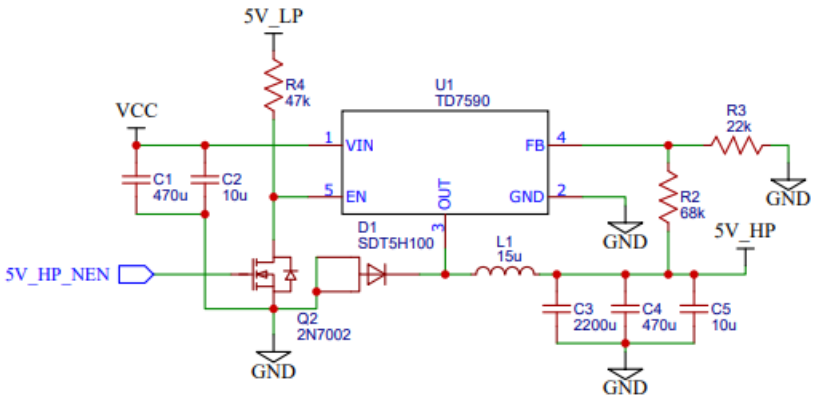


Figure 13: 5V-5A power supply

The original design used small SRP6030VA[5] 15µH, 100mΩ inductors for both power supplies. While according to manufacturer's specifications these components are capable of constantly sustaining the maximum expected load, this caused inefficiencies and extreme overheating leading to damage to the board caused by meltdown of the isolation and short circuit. This problem was solved by using much larger hand-wound inductors with a series resistance of 5 mΩ.

Multiple values of capacitors are used not for smoothing but for noise filtering. Since this type of power supply is very noisy and some components have frequency dependent impedance characteristics, multiple values are used to avoid a single resonant frequency. Placing multiple capacitors of different values together present lower impedance to more undesirable frequencies.

[4] TD7590 Datasheet: http://www.techcodesemi.com/datasheet/TD7590.pdf
[5] [Unused] SRP6030VA Datasheet: https://www.elfadistrelec.dk/Web/Downloads/_t/ds/SRP6030VA_eng_tds.pdf

As discussed in the 5.4.1 “Control Signals and Low Power State” section, these power supplies can be shut down by an external signal. This function is however unpopulated in the final prototype for the 5V rail since it would render the board inoperable until reset.

Efficiency

According to manufacturer’s specifications, power supplies using a TD7590 can reach “up to 95% efficiency” with an optimum at 1.3A of output load current. The following section the methodology used to measure the system’s efficiency and experimental results.

To measure the true efficiency of both power supplies, different loads are attached to the outputs. By measuring the input voltage and current as well as output voltage the efficiency as a function of current can be immediately calculated.

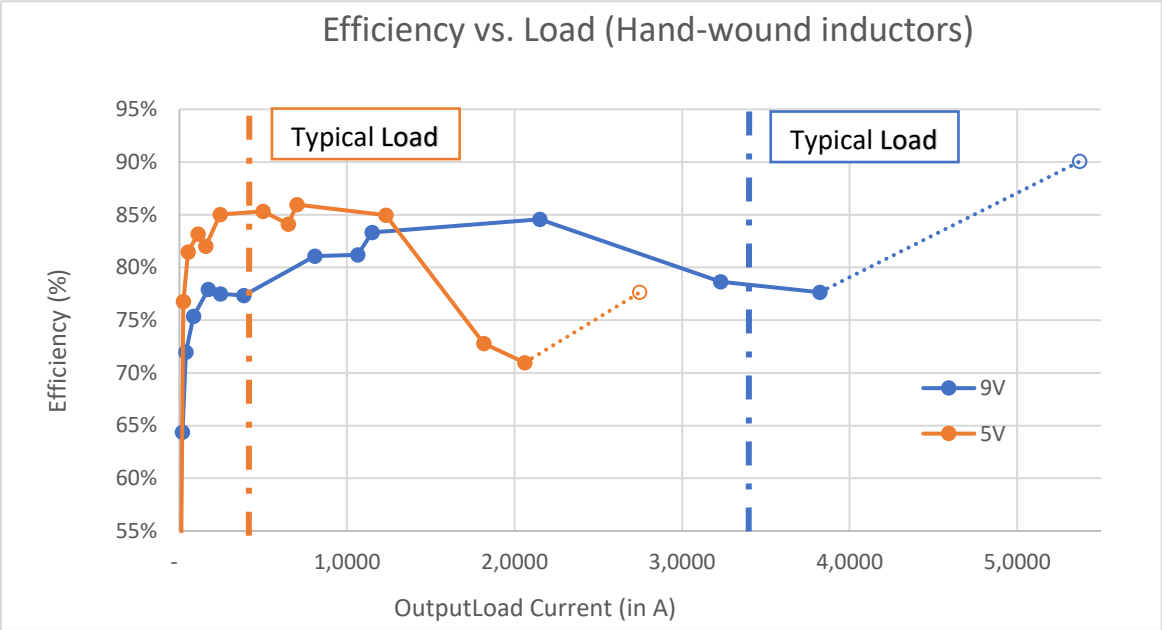


Figure 14: Efficiency of the power supplies vs load (using the bigger hand-wound inductors)

Figure 14 shows the results for both power supplies using the bigger hand-wound inductor. As expected with this type of power supply, they have a concave efficiency/load curve. They are very inefficient at low loads reaching a maximum efficiency in the mid-range. At higher loads, trace and component resistances become a source of power losses decreasing overall efficiency. At typical loads, the 5V and 9V step-down

converters operate at $\sim 85\%$ and $\sim 80\%$ respectively. Results at higher loads prove to be very unreliable, likely caused by insufficient or incorrect information about line resistance and output current.

Linear power regulators

Some parts of the circuit require logic voltages to operate. The loads on these buses are exceedingly small compared to the CPU and motors and can be supplied by linear regulators. Using linear regulators to create separate buses for logic prevents voltage spikes and drops from affecting these circuits. This is particularly important given the nature of the motors' inductive loads. When switching the motors, the 9V rail becomes very noisy due to the coils' inductance. In fact, the 5V power rail for the CPU requires large amount of decoupling to compensate for this and prevent glitches or brownouts [Figure 13].

The 5V logic bus is created by a L78L05[6]. The 3V3 bus is created by the Raspberry Pi's on-board linear regulator; it is not present in the annexed schematics. Since the input and output currents of a linear regulator are equal, the 5V and 3V3 bus efficiencies can be estimated to be 45% and 66% respectively. Fortunately, total load on these buses is around 20mA accounting for approximatively 200mW of losses.

5.3 Motor Control and Feedback

Relevant files: '/hardware/PCB' , '/hardware/ROVER/platform'*

5.3.1 H-Bridge

Since the robot's motion requires tracks to operate in both directions, the motor driver must switch the polarity of the voltage applied to the load. H-bridges can achieve this with simple hardware and high efficiency [10]. The system uses two identical MOSFET H-bridges, one for each track. The motors operating the same track are connected in parallel.

[6] L78L Datasheet: https://www.st.com/resource/en/datasheet/l78l.pdf

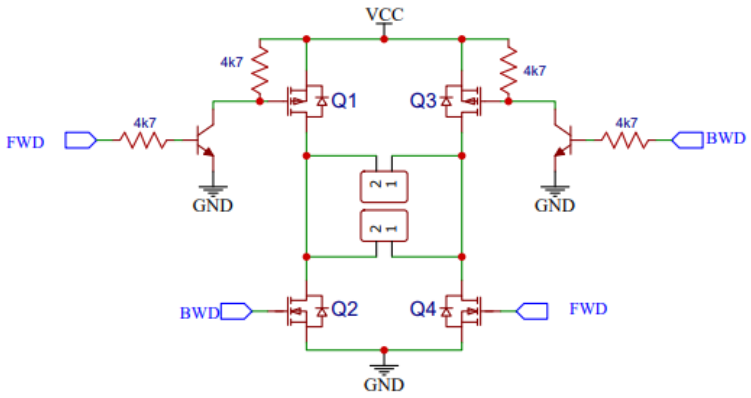


Figure 15: MOSFET H-Bridge (FWD: Forwards, BWD: Backwards)

As shown in Figure 15, Q1 and Q4 are driven using the same logic signal as well as Q3 and Q2. This is to ensure that both transistors responsible for a motor direction are in the same state. This decreases the likelihood of the system entering an unsafe state where current flow directly from power bus to ground. It is especially important to note however, that it is still possible for both forward and backward signal to be high at the same time. If this were to happen, the 9V step-down converter would only be protected by the internal overcurrent sensor.

A possible improvement for this circuit would be to add braking capability. In its current state, when both control signals are low, the motors are effectively disconnected from the rest of circuit and left in a "coasting" state. Saturating two parallel transistors, Q1 and Q3 for example, would effectively short the motors creating a braking force. As it is discussed later, this *coasting* causes issues with movement reliability and precision.

5.3.2 Motor feedback

In order to precisely move the robot, motor position feedback is essential. To achieve this, the motors axles have Hall effect rotary encoders. As Figure 16 shows, the encoder disks have four poles resulting in two pulses per revolution. This would normally be considered exceptionally low definition. However, the motors are connected to the sprockets using a 38:1 gearbox resulting in 76 pulses per revolution. The drive sprockets have a diameter of 50mm. Therefore, the final linear resolution is 76 pulses per 50 π of travel: one pulse every 2 millimeters.

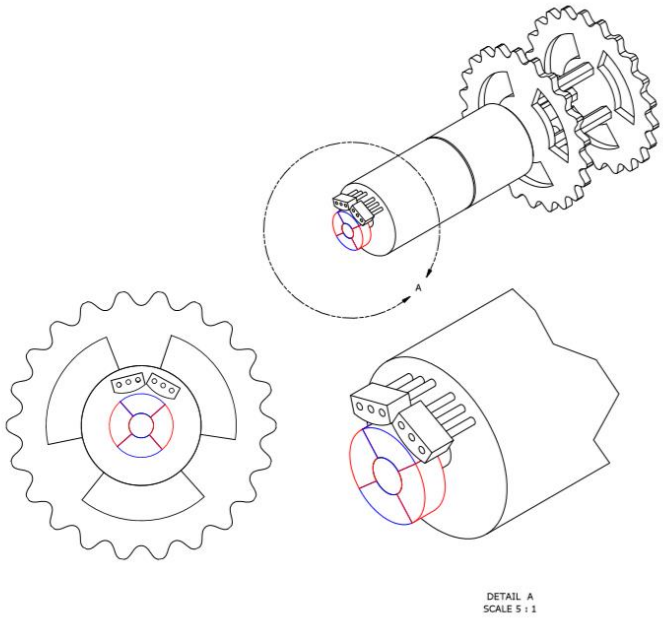


Figure 16: Motor assembly with sprocket and encoder

By using two offset Hall effect sensors for encoding, it is possible to derive the direction of rotation from their relative state at the edge. How this is used for position tracking is developed in detail in the software section [Figure 27].

Since the Hall effect encoders operate at a 5V logic level, their output must be lowered to 3V3 to prevent damage to the Raspberry Pi's GPIO hardware. This is achieved by a simple resistor divider.

5.4 Internal signals

Relevant files: '/hardware/PCB' '/controller'

5.4.1 Control Signals and Low Power State

Multiple components on the board have been setup to require an enable signal. This allows the CPU to control power usage through the Raspberry Pi's GPIO pins. The following sub-systems can be toggled by the CPU: the 9V power supply, the backup battery enable, the main battery undervoltage protection (disconnects main batteries), and USB power (turns off the LIDAR sensor). In theory the 5V power supply could also be operated in a similar fashion but it is not connected since this would turn off the CPU until manually reset.

This is particularly useful to enter a certain “sleep” state for low power consumption. Turning off all the aforementioned non-essential peripherals lowers battery current load from 460mA to 250mA, reducing the power consumption by 45%. The fourth generation Raspberry Pi can itself enter a sleep state but requires an external signal to wake-up. A possible improvement of this low power state would incorporate an external timer (like a microcontroller for example) to further reduce power consumption, only coming online periodically and poll the controller.

5.4.2 Internal Voltage Monitoring

Self-monitoring is essential to ensure safe operation and diagnostic of the on-board equipment. For this purpose, the CPU measures the value of 9 analog signals on the board. These signals correspond to individual cell’s battery voltages, power rails and backup power voltage. Since it would be impractical and expensive to use an ADC for every analog signal, the system uses a CD74HC4067[7] 16-channel analog multiplexer and a single MCP3021 10-bit ADC[8].

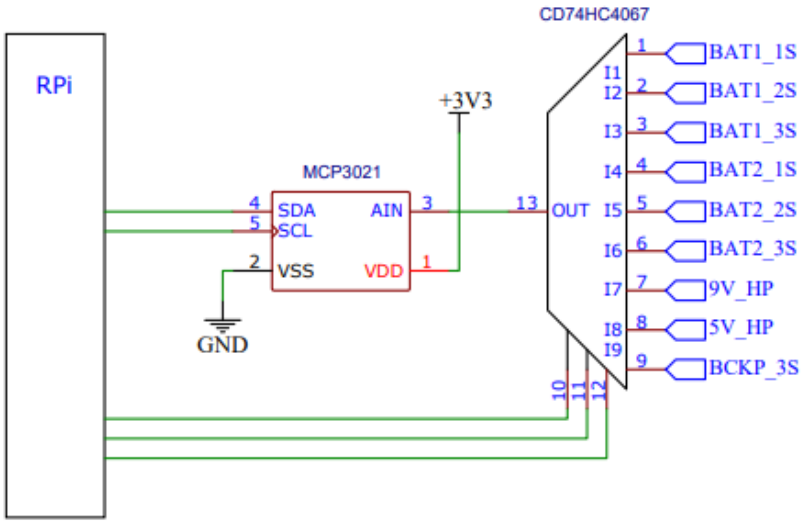


Figure 17: Analog monitoring system: ADC & Multiplexer

[7] CD74HC4067 Datasheet: https://www.ti.com/lit/ds/symlink/cd74hc4067.pdf
[8] MCP3021 Datasheet: http://ww1.microchip.com/downloads/en/DeviceDoc/20001805C.pdf

The CPU selects one of the inputs from the multiplexer by using 4 of its GPIO pins. It then waits a small period of time for the output voltage to stabilize and polls the ADC through an I2C bus [Figure 17]. According to experimental results, this setup can work up to 1ksps without losing accuracy. This polling frequency is equivalent to reading all analog signals over 100 times per second which is far over the system requirements. In fact, to reduce computational overhead and GPIO bus bandwidth use, the polling cycle is slowed down to 6 cycles per second in the current implementation.

Probably due to the many resistor divider networks required for signal level compatibility, this system requires a very repetitive calibration. Each of the 9 inputs was manually calibrated in software, adding an offset based on an external voltmeter's measurement. These results are not only used for the safety subsystems such as power supply shutdown and backup battery toggle, but they are also reported to a controller if available as telemetry. This telemetry is mostly used to update the user and logging.

5.5 PCB Layout and Metrics

The main board incorporates all the previously mentioned subsystems on a single 90mm x 150mm board. The Raspberry Pi interfaces with the board through a 20x2 ribbon cable. This cable is not only used for signal transmission but also power delivery with multiple pins being used for this purpose. This is not only important for safety and efficiency purposes but also system stability. Since no decoupling can be added on the CPU board, reducing the power line impedance is the only option for limiting power related instabilities [11] [12].

Most components are SMD. This is done mainly to minimize package sizes and therefore overall size. The manufacturing board house also provides 'Pick and Place' assembly for SMD components at nearly no other than component price[9]. This greatly reduces the total prototyping time given the large component count.

The final version is manufactured from a standard double-sided 1.5mm FR4 board using 35µm copper [Figure 18].

[9] Board manufacturer's surface mount service https://jlcpcb.com/smt-assembly

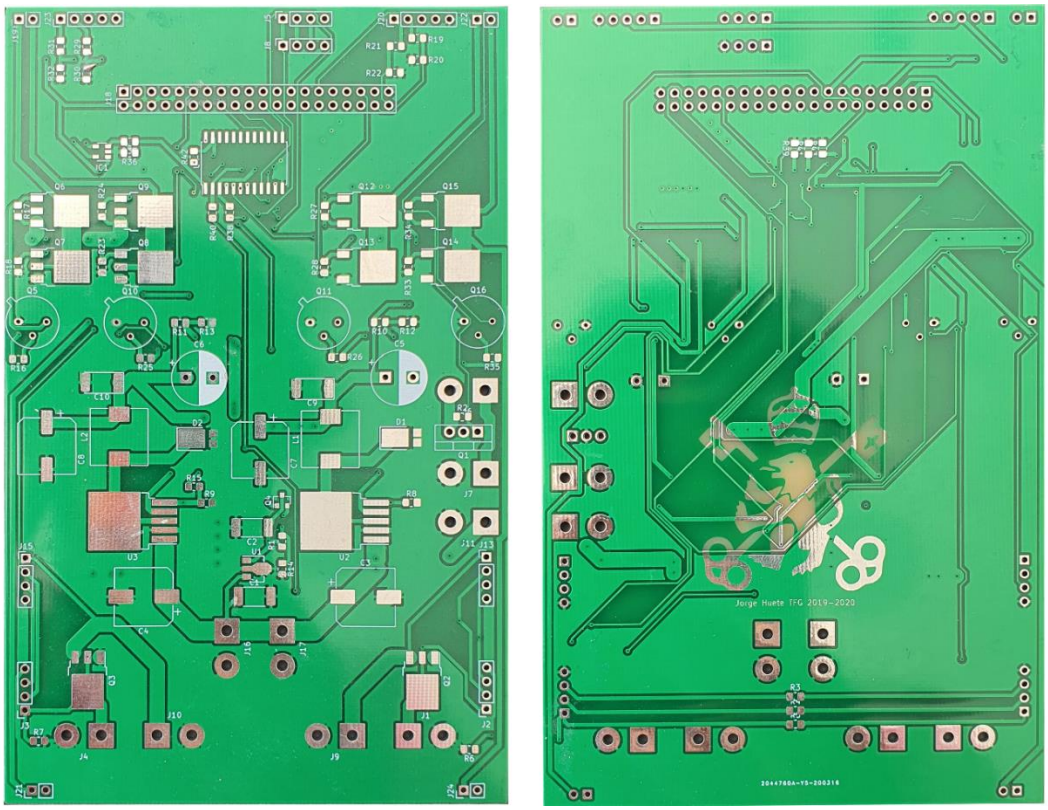


Figure 18: Unpopulated main board PCB

Note: Higher resolution and detail pictures are available in the annex

Note: While not covered by solder mask, the rear insignia is lacquered

For heatsinking purposes, all power MOSFETS, power supply ICs, and inductors have significant copper pad contacts.

According to calculations these are the following estimated resistances of major power delivery traces:

$$R = \rho_{copper} \cdot \frac{L}{35 \cdot 10^{-6} \cdot W}$$

- Main battery to step-down converters (excluding RPP MOSFET): 16mΩ
- Main battery to step-down converters (including RPP MOSFET): 36 mΩ
- 9V step-down converter to H-bridges: 9mΩ
- 5V step-down converter to CPU board (including ribbon cable): 22mΩ
- H-bridges to motors (average): 48mΩ

As the previous equation shows, these results are calculating assuming a constant temperature of 25ºC, ignoring the copper thermal coefficient. Given the expect loads of these buses, these resistances are acceptable, and the dissipated power is negligible compared to the surface area of the board.

5.6 Supports and Base

The base and tracks used in the final robot are an off-the-shelf product for prototyping and makers. Therefore, it does not warrant much detail here since it is not part of the development. Reconstructed schematics can be found in the annex for illustrative purposes. The main bed has a variety of mounting holes that are used to attach all the hardware described here.

The following support components are designed to be mating but separate. The goal for the system is to remain highly modular to ease the disassembly and diagnostic processes. In case of damage to any of the components they can be easily and cheaply fabricated and replaced.

All schematics are included in the annex. The latest version of the Autodesk files at the time of writing can be found in the repository as well. All parts are 3D printed in PLA or ABS.

PCB holders

Both the main board and Raspberry Pi are fixed to the base by holders. These holders offer physical protection as well as electrical separation from the conductive base. Although no active airflow is present, the bottom of the holders has significant open grids to allow for passive movement of hot air [Figure 19]. While the top face of the PCB is technically open, many cables and external structure block the flow of heat. This is particularly important in the case of the CPU. Since some of the software function are very CPU intensive it is prone to overheating which causes thermal throttling.

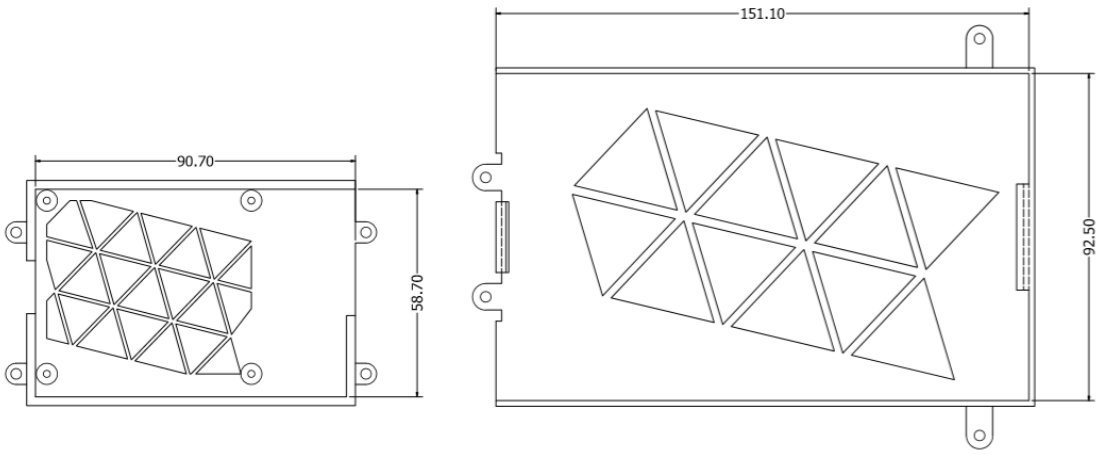


Figure 19: PCB decks with venting cutouts

LIDAR support

For the LIDAR data to be usable, the sensor must be placed above any other component of the robot. Otherwise obstruction would create blind spots that would require compensation in software. The sensor also needs to be placed in such a way that the center of rotation of the head is exactly above the center of the robot. This is done to prevent any movement of the sensor in space when a rotation is executed. Otherwise, data would not only have an offset, but also one that requires correction with every rotation.

The support for LIDAR was designed with these considerations in mind. The support rises the LIDAR 80mm above the main platform deck. Angles of the legs were also calculated to ensure the concentricity of the sensor as well remain easily printable in a 3D printer without the need of wasteful support material.

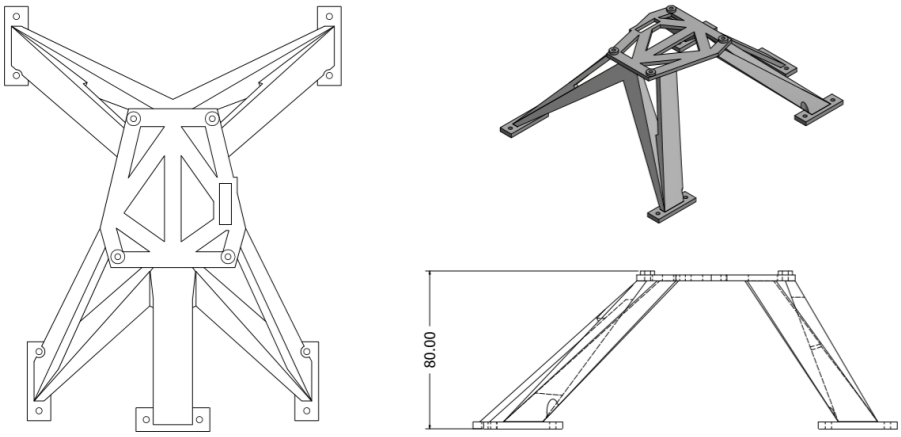


Figure 20: LIDAR support

6. Software

The software side of this project represent the largest part of the development. All files in the repository except for the “hardware” subfolder are relevant to this section. Each subsection will specify where the related files are located. Most the systems runs off of Python3 scripts.

6.1 LIDAR Driver

Relevant files: ‘/driver’

The principal sensor used for mapping, navigation, and movement feedback is a rotating LIDAR (Light Detection and Ranging). The precise sensor is an RPLIDAR A1M8 manufactured by SLAMTEC. According to documentation[10] this sensor is capable of up to 8kSa/s with a maximum range of 16m. While this rate is technically possible, the documentation regarding this mode proved to be very lackluster. The provided driver by the manufacturer at the time of writing consists of an uncompiled Microsoft Visual Studio C++ project largely dependent on deprecated functions. [13] [14] [11]

Some community efforts have been made to create a more accessible and flexible Python driver that uses the LIDAR low-rate standard scan mode [15]. However, it was not possible to locate a driver capable of utilizing the more complex “Express” mode. Nonetheless, the code published by ‘SkoltechRobotics’ proved to be an excellent base for the UART package exchange with the LIDAR.

6.1.1 UART communication and Standard Mode

The LIDAR communicates with the host using a 115200 baud rate UART. The communication uses a simple request/response protocol where requests are only emitted by the host. Every response from the LIDAR is preceded by a response descriptor containing information regarding the size, data type, and number of responses. Most request result in a single response; however, scans regenerate constant traffic of responses until halted by host [Figure 21] [16] .

[10] LIDAR Datasheet: https://download.slamtec.com/api/download/rplidar-a1m8-datasheet/2.4?lang=en

[11] Since the start of this project, SLAMTEC has now endorsed and collaborated on a community C++ SDK clearly superior to the old one.

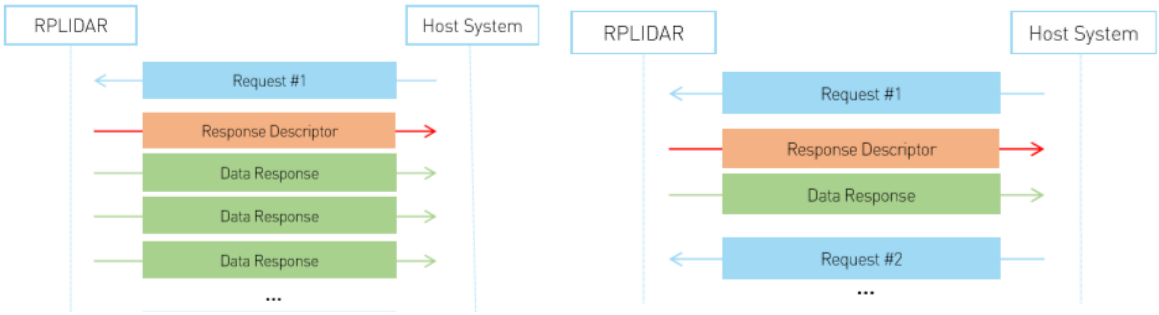


Figure 21: LIDAR communication protocol. Single (left) and multiple(right) response modes [16]

In the standard scan mode, each sample is sent in a separate response packet. Each response is encoded into 5 bytes containing signal strength, angle, and distance of the sample plus a checksum and a bit indicating that a full revolution was completed [Figure 22]. This is the mode of operation used by SkoltechRobotics' community supported driver [15].

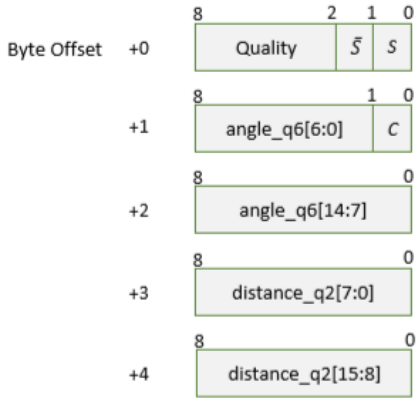


Figure 22: Standard scan response [16]

Q binary fixed number format

The sensor uses the Q number format for encoding all distance and angle values as fixed-point numbers. This format uses the notation Qn where the number n is the number of fractional bits used. The value encoded in the packet can be recovered by dividing by denominator 2^n.

For example: $angle = \frac{angle\_Q6}{64}$

Simple size per sample calculations quickly explain why, although simple, this protocol is incapable of reaching the claimed sample rates.

- $packet\ size = 5\ bytes = 40\ bits$
- $samples\ per\ packet = 1$
- $channel\ capacity = 115200\ bps$

$$maximum\ sample\ rate = \frac{channel\ capacity}{packet\ size} \cdot samples\ per\ packet$$

$$maximum\ sample\ rate = 2880\ Sa/s$$

In fact, after considering the interbyte time, the sample rate is reduced even further resulting in an experimental result of approximatively 2.5kSa/s. At a scan frequency 6 turns per second, this results in only 417 samples per turn; or a resolution of 0.9º.

6.1.2 Express Mode and compression

Although the resolution obtained by the standard scan mode could be considered acceptable for the purposes of this project, Express Mode offers a more compact way of transmitting samples and therefore increased resolution. This, however, comes at the cost of increased recovery complexity.

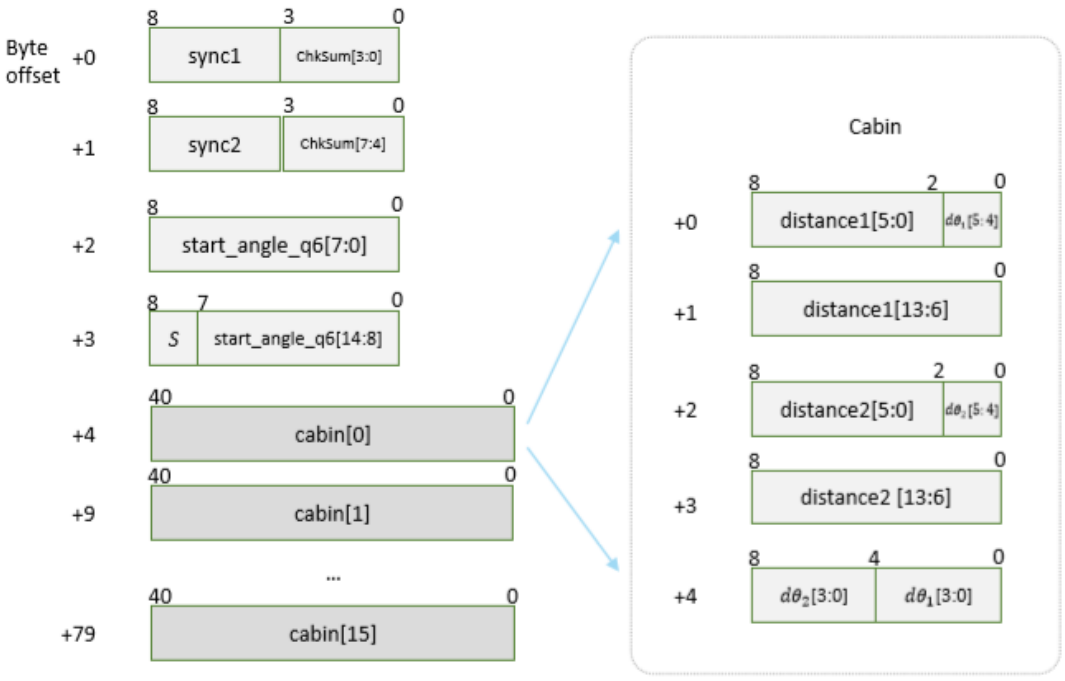


Figure 23: Express Scan response [16]

As illustrated in Figure 23, each packet contains the distance measurement of 32 samples. However, the angle measurements are not explicitly included. The packet header only contains the position of the rotating head at the beginning of the scan in Q fixed point number format. The angle of a sample is derived from the implicit equidistance of samples.

The encoding logic begins with the assumption that, given a constant rotation speed, all samples are equally spaced along the rotation of the sensor. If this assumption were to hold true, it would be possible to determine the angle of each sample as follows:

$$\theta_{n+1} = \theta_n + \Delta\theta$$

$$\Delta\theta = \frac{end\ angle - start\ angle}{N\ samples}$$

Note: in order to determine "end angle" the driver must wait for the start angle of the next scan

Although the initial assumption is close to reality, the actual rotation speed changes slightly over the course of a scan and thus the inter-sample angle ($\Delta\theta$). To compensate for this, each distance

measurement is paired with a compensation angle ($d\theta$). This allows to recover the angle magnitude by using under half the number of bits. [12]

$$\theta_n = start\ angle + \frac{end\ angle - start\ angle}{32} \cdot n - d\theta_n \quad for\ start\ angle \leq end\ angle$$

$$\theta_n = start\ angle + \frac{360 + end\ angle - start\ angle}{32} \cdot n - d\theta_n \quad for\ start\ angle > end\ angle$$

With this encoding, the transmission overhead is reduced increasing throughput and therefore the maximum sample rate:

- $packet\ size = 84\ bytes = 672\ bits$
- $samples\ per\ packet = 32$
- $channel\ capacity = 115200\ bps$

$$maximum\ sample\ rate = \frac{channel\ capacity}{packet\ size} \cdot samples\ per\ packet$$

$$maximum\ sample\ rate \approx 5485\ Sa/s$$

This increased throughput compared to the standard mode increases the angular resolution to 0.4º. Although 5.5kSa/s is still inferior to the manufacturer's specification of 8kSa/s, the protocol used by the sensor to reach such transmission speed uses a vastly more complex compression algorithm involving linear interpolation every 3 samples with distance correction [16] [17].

Given the 16m maximum range of the sensor, the maximum resolution gained by using the compressed protocol is 37mm (a centimeter at 5 meters). As will later be noted, such difference in resolution is inconsequential for this application and can even be obtained by simply scanning the same environment at lower resolution twice. Therefore, for the remaining of this project, results are obtained using the Express Mode.

[12] Total throughput can be further increased by requesting the same angular resolution without the compensation data. This relies entirely on the presumption that rotation of the head is constant.

6.1.3 Scan Results

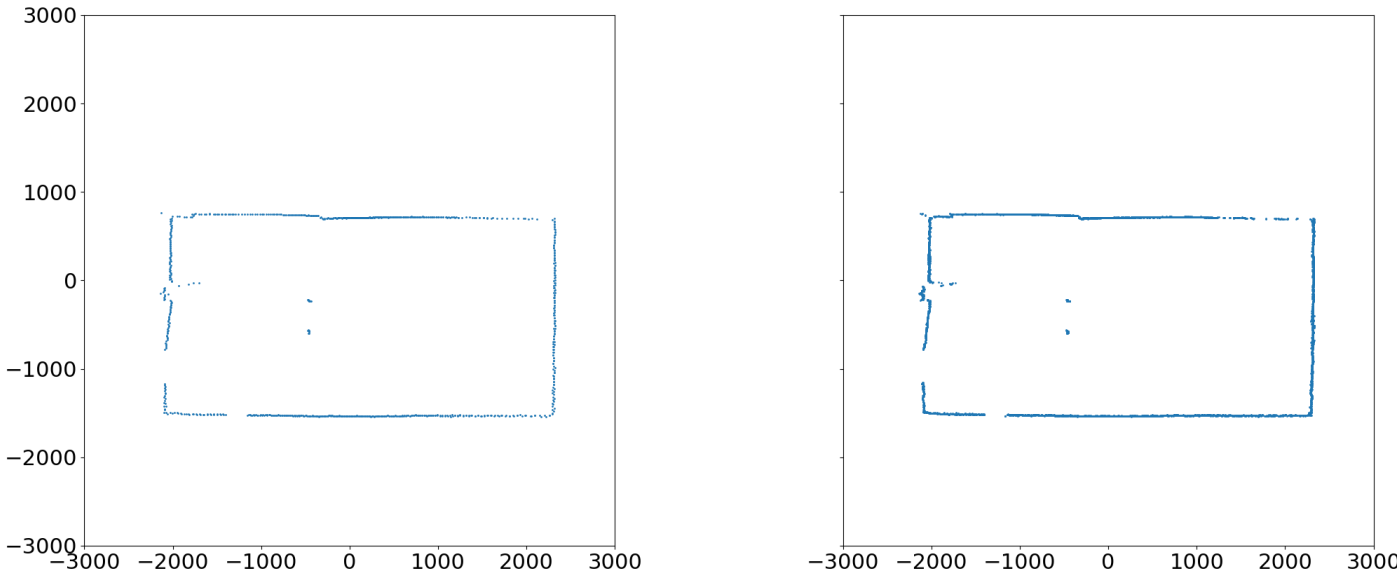


Figure 24: Scan point cloud output. Single rotation (left) and one second (right)

Figure 24 show the output of the driver when requesting a single rotation and a one second long scan in a $4.5m$ x $2.5m$ room. As expected by previous calculations, increasing the number of samples requested, and therefore turns, the resolution is effectively increased. However, single turn scans seem to provide enough information and level of detail for most navigation and mapping applications.

6.2 TCP Server and Controller

Relevant files: '/controller'

Although the robot can operate autonomously, scanning and mapping its environment, it runs a TCP server to receive orders from a controller and return data. In fact, the robot can operate at three different levels of autonomy: manual control, semi-autonomous, and fully autonomous.

In manual control mode, the movements no longer follow the algorithm described in 6.5 "Precise Maneuvering". In this configuration, the controller sends manual move orders and receives a constant flow of sensor data. This means that the current position cannot be precisely kept and therefore real-time mapping functions are unavailable. It can be considered as a form of first-person direct remote control.

The most developed mode is semi-autonomous. In this mode, the robot receives a waypoint from the controller and will attempt to reach it. To achieve this, it must regularly scan and map its environment.

During its discovery process, every time a maneuver has been completed, the robot will send the controller a full update of its findings and status. At any time during this process, the controller can update the current waypoint. The robot will keep the map in memory and attempt to reach its new destination.

Fully autonomous mode is still in an experimental state, but some results are provided bellow in the "7 Results and Analysis" section. This state simulates a complete lack of communication between robot and controller. However, the robot will still send data if possible. In this mode, the robot will attempt to scan its entire environment until a complete map has been created. Once the robot's pathfinding algorithm determines that no other possible rooms can be discovered, the system enters a low-power state awaiting orders.

6.2.1 Packet format

To ensure completeness of packets within the TCP buffer, all packets have a header and a trailer. The system exchanges two types of packet: message and data. These packets have different headers and trailers:

- Message: "-----BEGIN TFG MSG-----" "-----END TFG MSG-----"
- Data: "-----BEGIN TFG DATA-----" "-----END TFG DATA-----"

The greatest interest behind this protocol choice is not to ensure data integrity, since that is handled by lower level TCP logic, but to simplify counting. Counting the number of packets in the input buffer is as simple as counting the occurrences of both types of headers and footers. Since data is expected to maintain order, they can then be sequentially read awaiting execution.

The actual packet data is an encapsulated JSON string. This JSON has a different format depending on the type of message.

Messages

All messages exchanged by the system share a common shape:

```
{"type": string, [arguments], "priority": bool, "N": int}
```

The “type” fields indicate the nature of the message, and therefore, the expected arguments within. A brief explanation on available commands is provided below.

“Priority” is a binary field indicating where in the execution queue should this message be placed. By default, messages are added to the end of a FIFO queue. However, the controller can request an expedited execution by setting this field to “True”. If this is the case, the message is placed at the start of said queue. It is mostly used for emergency halts or shutdowns.

Lastly, each packet has a sequence number “N”. This number is used to keep track of responses by server. When the server reports back on the execution of an order, it will use the same sequence number. Since messages are not necessarily executed in order, and not all messages will get a response, the controller uses this number to pair orders and responses.

Data

Data can be one of three types: raw scan point cloud, environment map, or telemetry data. In all three cases, the information is sent as a serialized Numpy array. To determine how to interpret this, packets are always preceded by a descriptor. Data packet’s share the same ‘N’ identifier as their descriptor to ensure correct linkage. The descriptor informs the controller about the type of data and size to expect.

Raw scan data is rarely used except for debugging purposes. In contains a series of floating point bidimensional elements. Each element corresponds to a single sample. This type of data can be requested by the controller with a “**SCAN_REQUEST**” message.

Environment maps contain all the identified objects and walls in a grid, as well as the resolution of said grid and current position. Each object contains a reputation field indicating how reliable that information is.

Lastly, telemetry data contains information regarding the current state of the hardware. It informs the controller about main board voltages, computer board temperatures, and control signals states. The controller uses this data to display to the pilot. Pilot can then use this information to shutdown subsystems or configure their behavior with a “**CONFIGURATION**” message.

6.2.2 Available commands

- "type": "**CONFIGURATION**", arguments: "parameter": string, "value": float or int

This command is used for configuring many of the parameters discussed in this document. These parameters include USB power, 9V power supply state, undervoltage thresholds, backup power state, sleep mode, and more.

- "type": "**MODE_CHANGE**", arguments: "mode": string

Changes the autonomy mode to one of the aforementioned: "MANUAL", "SEMI", or "AUTO". [13]

- "type": "**MANUAL_MOVE_ORDER**", arguments: "direction": string, "duration": float

Orders the robot to move in a direction (forwards, backwards, left or right) for the determined duration in seconds. For safety reasons, the duration is capped at 2s. This message is only executed if the robot is in manual mode.

- "type": "**CORRECTED_MOVE_ORDER**", arguments: "mvt_type": string, "distance": float

Executes a maneuver according the algorithm described in 6.5 "Precise Maneuvering". Movement type can be linear or rotation, distance is in either millimeters or degrees according to type. It can be executed in either manual or semiautonomous modes.

- "type": "**SET_WAYPOINT**", arguments: "x": float, "y": float

Sets a waypoint at the specified coordinates in millimeter from the current origin. The origin corresponds to the position of the robot when it entered semiautonomous mode. The robot will then map its surroundings and apply the pathfinding algorithms described in 6.8 and 6.9.

- "type": "**SCAN_REQUEST**", arguments: "SAMPLE_SIZE": int, "MAX_RANGE": float

Requests raw samples from the LIDAR sensor. The data is guaranteed to contain at least the number of samples requested. All samples further than the specified maximum range are ignored. Particularly useful for debugging and verifying current status.

[13] This order cannot be executed in high priority. Even if requested as such, the system will wait for all previously queued movement orders to finish before switching autonomy mode. Sending a "**HALT_OVERRIDE**" right before a "**MODE_CHANGE**" bypasses this limitation.

- “type”: “**MAP_REQUEST**”, arguments: None

  Requests the current state of the map. Since the robot does not map its environment in manual mode, this is only available in semiautonomous or autonomous modes. If the robot is currently using neural net navigation (as described in 6.9), this order will be delayed until normal navigation is resumed.

- “type”: “**HALT_OVERRIDE**”, arguments: None

  Overrides all other currently executing orders to the motors and halts them. If executed in priority mode, all orders involving motor movement with a lower sequence number (previously received) are ignored. Can be executed at any time.

There are other self-explanatory message types sent by the server to controller such as position and status reports, data descriptors, and acknowledgments.

6.2.3 Server and execution diagram

The server and execution logics are separated into two different processes. Although this increases the overall complexity, given the separate memory spaces, it is necessary to achieve true parallel processing in Python. Python’s Global Interpreter Lock (GIL) is a mutex that restricts access to certain Python objects. This effectively means that although many threads might be active at a time, GIL prevent simultaneous execution [18]. By spawning multiple instances of the interpreter in memory, multiprocessing avoids the inherent limitations created by the GIL. This also allows the server process to freeze in case of a dropped connection and wait for a new one without pausing the execution of other messages or navigation or releasing its lock on the connection.

Both processes communicate through two FIFO queues: a high priority queue and a normal one. These two queues are joined into a single abstracted one where all the high priority messages are placed at the top.

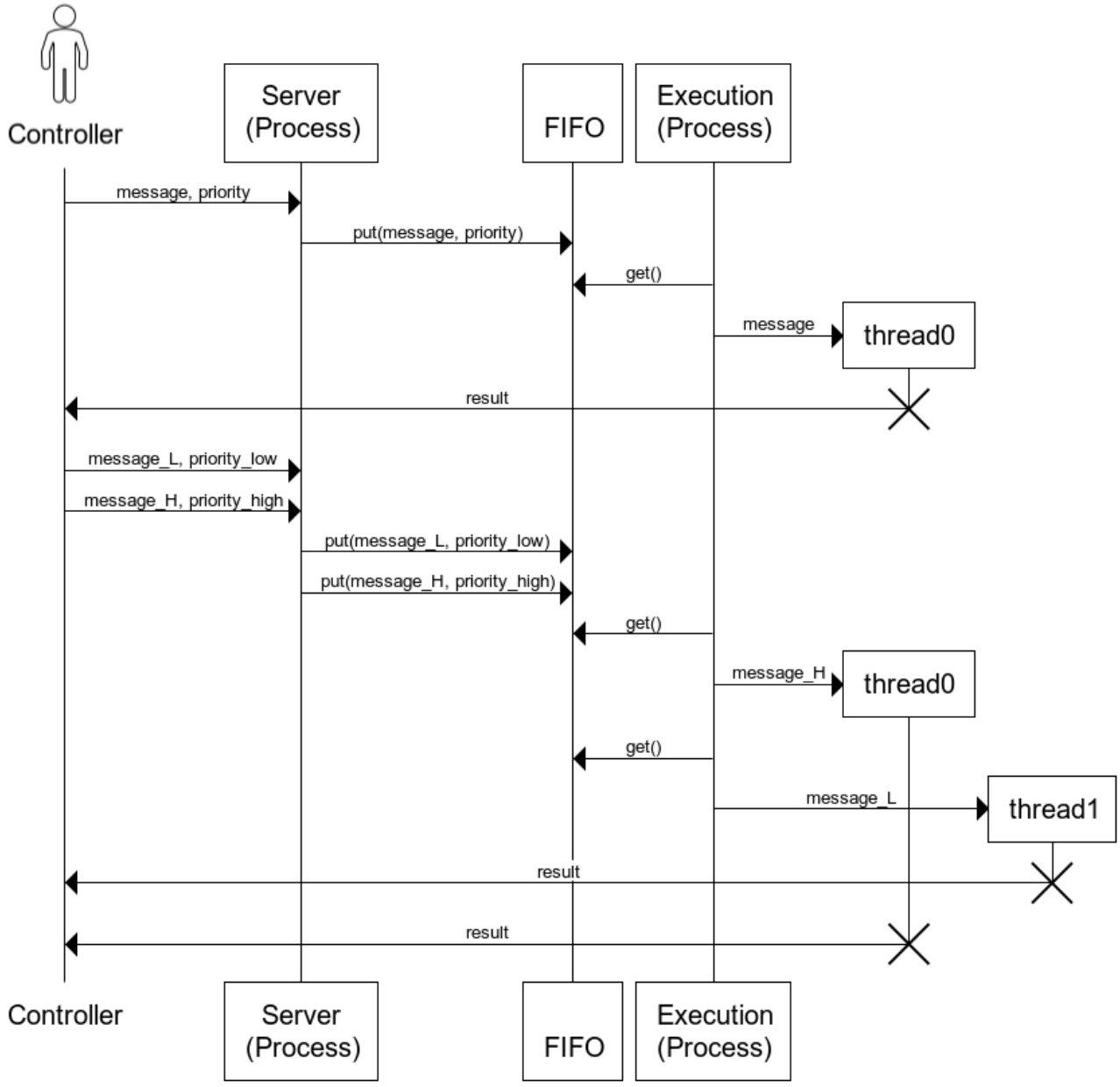


Figure 25: Server/Execution sequence diagram

As the Figure 25 sequence diagram shows, when a message is received by the server process, its contents are placed in the shared queue. The execution process attempts to read form the queue as soon as possible and spawns a new thread for each message execution. The execution process keeps track of its sons and corresponding messages in order to terminate them if necessary (halt command for example). If a response is necessary, the threads use the connection object to reply to the controller directly. Rarely does the server itself send information to the controller.

Following the second and third messages illustrates how the different priorities are handled. Although the server received the low priority message first and placed it in the queue accordingly, when the execution process requested the next message, it obtained the high priority one. This abstraction greatly simplifies the execution logic since it does not need to incorporate its own buffer.

Although not depicted in this diagram, the execution process spawn other threads, notably for navigation and position tracking.

6.3 Position Tracking I: Motor Feedback

Relevant files: '/controller'

As it was briefly mentioned in the hardware section, the position of the tracks is kept by means of rotary encoders attached to the axle of the motors. Although all four motors have rotary encoders, only one of them is used for each track [Figure 26]. Using both rotary encoders on a side does not add any precision whatsoever since they are jointly coupled to the same track. They could however be used to track motor errors and stalls. Before replacing faulty connectors, motors would disconnect from the board and halt, causing damage to the tracks and overloading the other motor. A possible improvement to this technique would be using all rotary encoder to detect failures.

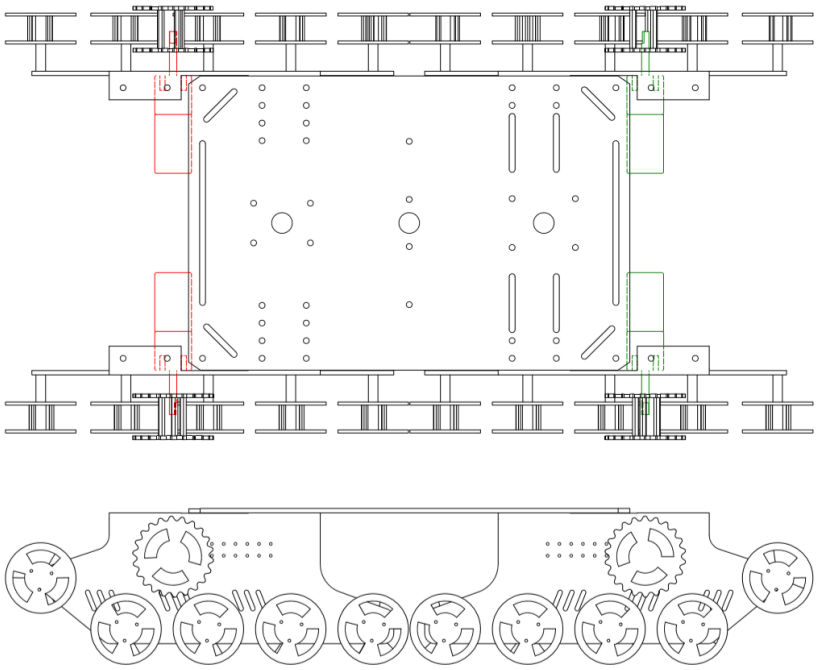


Figure 26: Platform assembly. Motors are colored: green (rear), red (front)

Although all motors are powered, only the rear (shown in green) are used for feedback purposes.

6.3.1 Pulse waveform

The encoders output two quadrature-encoded signals (A and B) of equal, and assuming constant velocity, constant frequency. When moving, there is a phase difference of 90º between the two signals, hence the use of term quadrature. The direction of rotation can be determined by the sign of the phase difference. When A is leading B (+90º phase difference), it can be assumed that the encoders are moving in the clockwise direction. When B is leading A (-90º phase difference), it can be assumed that it is moving in a counter-clockwise direction [19]. Both cases are depicted in Figure 27.

This delay between the two signals arises from the angle between the two Hall effect sensors.

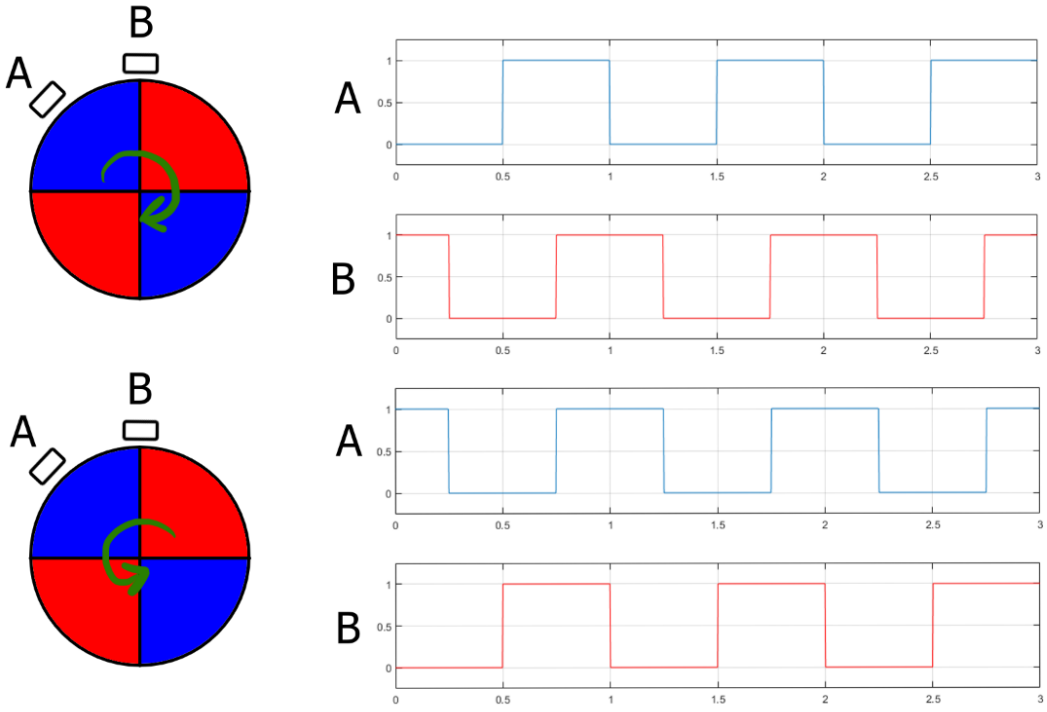


Figure 27: Encoder pole arrangement (left) and waveform (right)

When an edge is detected in one of the channels, it is possible to determine the direction of motion from the other signal's state:

- A rising edge: B = 0 → clockwise pulse
- A rising edge: B = 1 → counterclockwise pulse
- A falling edge: B = 0 → counterclockwise pulse
- A falling edge: B = 1 → clockwise pulse
- *B rising edge: A = 0 → counterclockwise pulse*
- *B rising edge: A = 1 → clockwise pulse*
- *B falling edge: A = 0 → clockwise pulse*
- *B falling edge: A = 1 → counterclockwise pulse*

In theory, this allows to update the position counter four times per cycle (two edges per channel), vastly increasing the resolution of 76 pulses per revolution mentioned in the hardware section. However, experimental results with four counts per revolution proved to be very unpredictable sometimes resulting in missed steps. [14]

6.3.2 Multiprocessing counter and interrupt routines

It is essential for the system to be able to keep track of movement while completing other tasks. For example, during a maneuver it is possible that the controller executes an emergency halt or that a new obstacle is placed or detected in front of the robot. For this reason, the track position counters are kept in a separate process from other functions. This is done to avoid memory sharing (GPIO access in particular), and more importantly prevent interruptions from halting other parts of the system.

Since pulses are frequent and require little actual processing, it would be very inefficient to poll the state of the channels constantly and halt execution for long periods. For this reason, this is done by setting interrupts linked to the corresponding GPIO pins that spawn short-lived thread to update the counters when every state change of the A channel. [20]

The following pseudocode is used to create a software-based interrupt on one of the Hall effect sensor channels. Two GPIO interrupts cannot be set for the same pin, therefore the edge type (rising or falling) must be done with the interrupt function.

[14] This very likely caused by timing issued arising from software base interrupts. It is in this issue in particular that a great deal of precision could be gained by the addition of a microcontroller to the main board.

```
import RPi.GPIO as GPIO
…
# This function runs every edge of signal A
func ch_A_thread():
	if raising_edge:
		if ch_B: counter –
		else: counter ++
	else:
		if ch_B: counter ++
		else: counter –
# Sets up the interrupts
func setup(counter):
	GPIO.add_event_detect(channelA, GPIO.BOTH, callback=ch_A_thread)
	freeze()
```

By running the setup function in its own process, the counter will update every channel A edge. This process must be repeated for the other track. Since this code runs in a different process from the main program, counters must be part of a shared memory space. Otherwise, attempting to read from the value of this variable would constantly return the initial value.

Adding a maneuver target

While the previous pseudocode is useful for keeping track of the current position, using it for moving a precise distance is impractical. It would involve reading the value of the counters in a loop and would be equivalent to polling the state of the input pin itself. To aid in precise movement, the interrupt function can be modified to include a target count. Since the interrupt code runs at every edge, this is vastly more precise and helps prevent the robot from overshooting the target due to a main process slowed down by other executions.

```
func ch_A_thread():
	if raising_edge:
		if ch_B: counter -
		else: counter ++
	else:
		if ch_B: counter ++
		else: counter -
	# Check if a target has been set and reached
	if target_flag and counter == target:
		halt_motor()
		target_flag = False
```

The “target_flag” and “target” variables are in shared memory space that the main process can use to set the target distance of the current maneuver. Once the track has reached the intended position, the motor is stopped and the flag cleared to prevent another unintended stop.

Although this code stops the motors the instant it reaches the correct position, the robot still overshoots the target. This is caused by two factors: momentum and lack of braking torque. As mentioned in the 5.3.1 “H-Bridge” section, once the motors are halted, they are effectively free rotating and the only braking torque applied to the track is the internal resistance of the gearbox and motor. Due to the mass of the robot itself, the tracks continue to move after the halt command is executed.

A remarkably interesting improvement for this motor control would be the use of a microcontroller or even an inexpensive FPGA to add a layer abstraction and more control features to the main board. The main board could have dedicated hardware that exclusively handles track motion, unhindered by other, more complex processes.

6.4 Position Tracking II: LIDAR based tracking

Relevant files: ‘/preprocessing’

The main reason behind the choice of a continuous track for the vehicle’s propulsion was the predictability of its motion. The initial assumption was that the tracks’ travel could be used to directly extrapolate the movement of the robot. Given the large surface area of the tracks compared to wheels, the expectation was to have negligible kinetic friction and to use the feedback from the rotational encoders to determine

the position with minimal need for correction using LIDAR data. This is in fact the same type of propulsion used by the first Cornell paper for simplicity [3]. However, precise positioning can also be determined with the use of wheel movements if an optical reference is added as showcased by the Hong Kong robot [4].

In linear motion, the initial hypothesis was that by moving both tracks in the same direction for the same distance of travel, this distance could be used to determine the new position of the robot.

In a similar way, in rotation, the intention was to move both tracks an equal distance but in opposite directions and approximate the rotation from this distance.

These assumptions proved to be incorrect and results remarkably unpredictable. Instead of using LIDAR data to infrequently correct for small errors, it became essential for every translation and rotation maneuvers. The following sections show the sources of error and demonstrate how the LIDAR data is used to obtain a reasonable degree of motion precision.

6.4.1 Sources of error

The two most likely sources for error are slipping of the tracks over the terrain and error in the tracking of motors. To determine the contribution of these factors to the total error a simple repeatability test was performed.

By imputing a 5-meter move order in one direction and again in the opposite direction, it is possible to compare the relative position of the tracks and body of the robot. If the CPU correctly counts the encoder pulses, since they are mechanically coupled, the tracks should return the initial position.

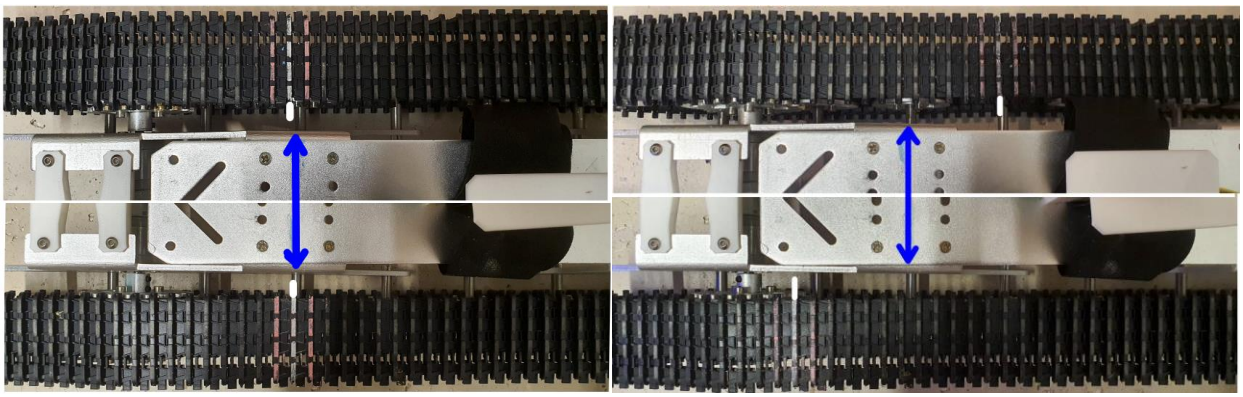


Figure 28: Track traction test results. Before (left) and after (right)

Figure 28 shows the results of this test. The tracks initial positions were marked by white and pink colored chalk and a marker line on the robot's body. The left image shows the final position of the tracks after the test.

By counting revolutions, tracks seem travel the expected full distance of 5m. At the end of the test, both tracks are close to the original position: +5.2cm (left track) and -4.5cm (right track) which correspond to about 1% error. The robot, however, did not reach the 5m distance nor returned to the original position (~55cm of error) nor rotation (~10º of error). These results lead to the conclusion that nearly all the error comes from the kinetic friction between the tracks and terrain. In fact, simple observation of the belt during operation shows visible movement between track and ground.

Unfortunately, this type of error is highly unpredictable and cannot be corrected by scaling the total travel distance. In order to carry out precise movement maneuvers scan data is used to compare the expected environment and the current scanned environment. The following sections illustrate how this is done in linear and rotational movements.

6.4.2 Detection of translation error

Since scan data can be approximated as continuous variables, it would be difficult to compare two raw scan results. Moreover, since scans can have thousands of points of data, comparing by using standard statistical methods would be extremely intensive increasing the time necessary for every maneuver.

To correct for this, scan data is discretized. This is done by creating a grid where each space has a binary value: 255 or 0. If the grid space contains any number of samples, it is assigned a value of 255 [Figure 29]. When establishing a grid, two main variables are used: maximum range and resolution division. Maximum range corresponds to the size in millimeters of the grid (grids are square) and resolution division corresponds to the size in millimeters of one grid cell.

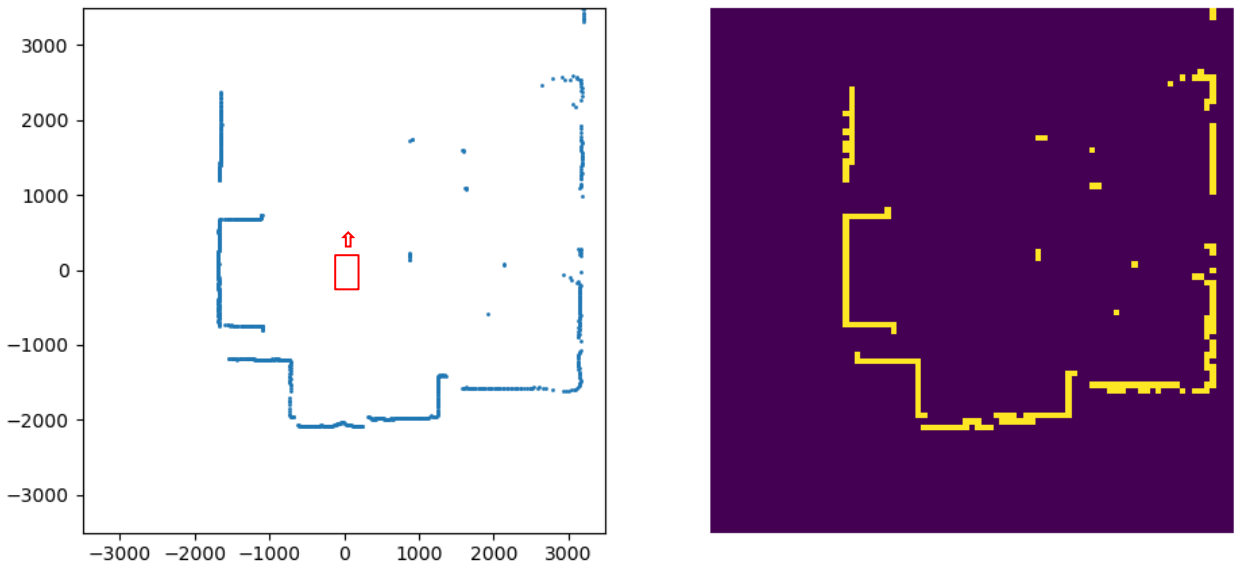


Figure 29: Discretization of scan data. 7m x 7m, resolution = 80mm/pixel (right)

Note: this resolution division is higher than normally used. This is done for illustrative purposes for the remainder of this section

This scan grid is aligned with the direction of travel on the vertical axis. This means that when the robot moves forward, the expected result on the grid is a vertical scrolling where new data appears at the top and disappears at the bottom. This is certainly helpful for predicting the result at the end of a maneuver.

By simply removing certain rows from the bottom and appending empty cells at top of the grid, the obtained result is a prediction of what a scan should look like from a position ahead of the current one [Figure 30]. The relationship between the number of rows and distance is simply:

$$row\ count = distance\ //\ \text{resolution}\ division$$

Note: "//" indicates integer division

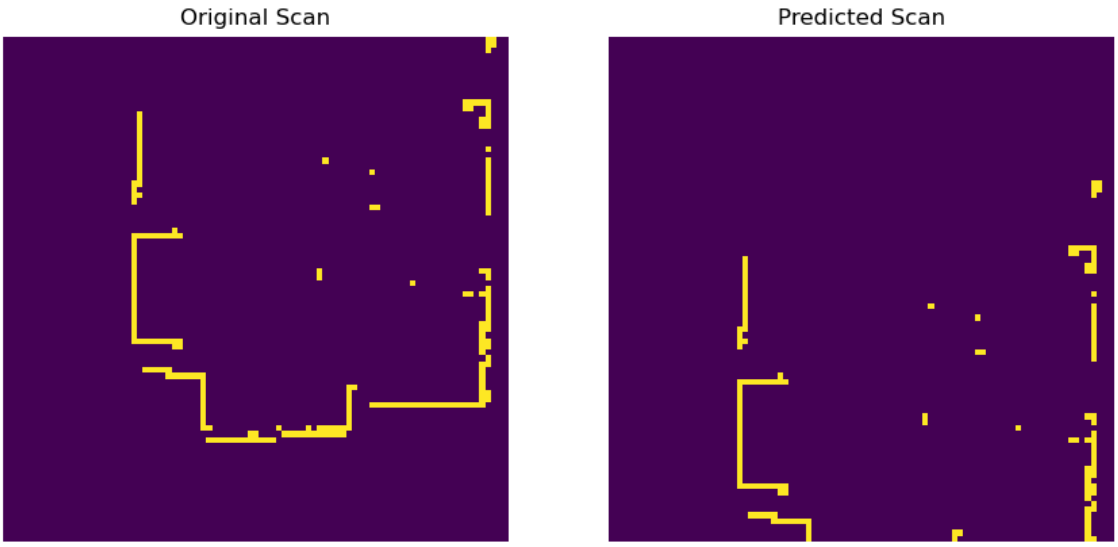


Figure 30: Predicted scan result after a 2m (25 pixels) movement forwards

Figure 30 shows the result of such a prediction. One notable issue with this prediction is the lack of the new information that would be discovered by new data. This is represented by noticeably large empty space at the top of the right image.

By scanning and discretizing after moving said movement, the result is a grid similar to the predicted one but with a certain translation. Subtracting both grids gives a quantitative similitude of both images. If the linear movement were perfectly predicted, this error would be minimal. It is then possible to iteratively *slide* the result image up and down in both directions (currently assuming a maximum of 25% in any direction). At the end of this loop, the translation with minimum difference is established as the error committed during the linear movement maneuver.

Once both images are ideally superposed, the subtraction of their cell-by-cell values would be minimal, since positively identified samples would cancel out. This however relies on the assumption that the only transformation applied to the image is a vertical translation. Horizontally translated or rotated images would not significantly cancel out since large sections of walls would not match.

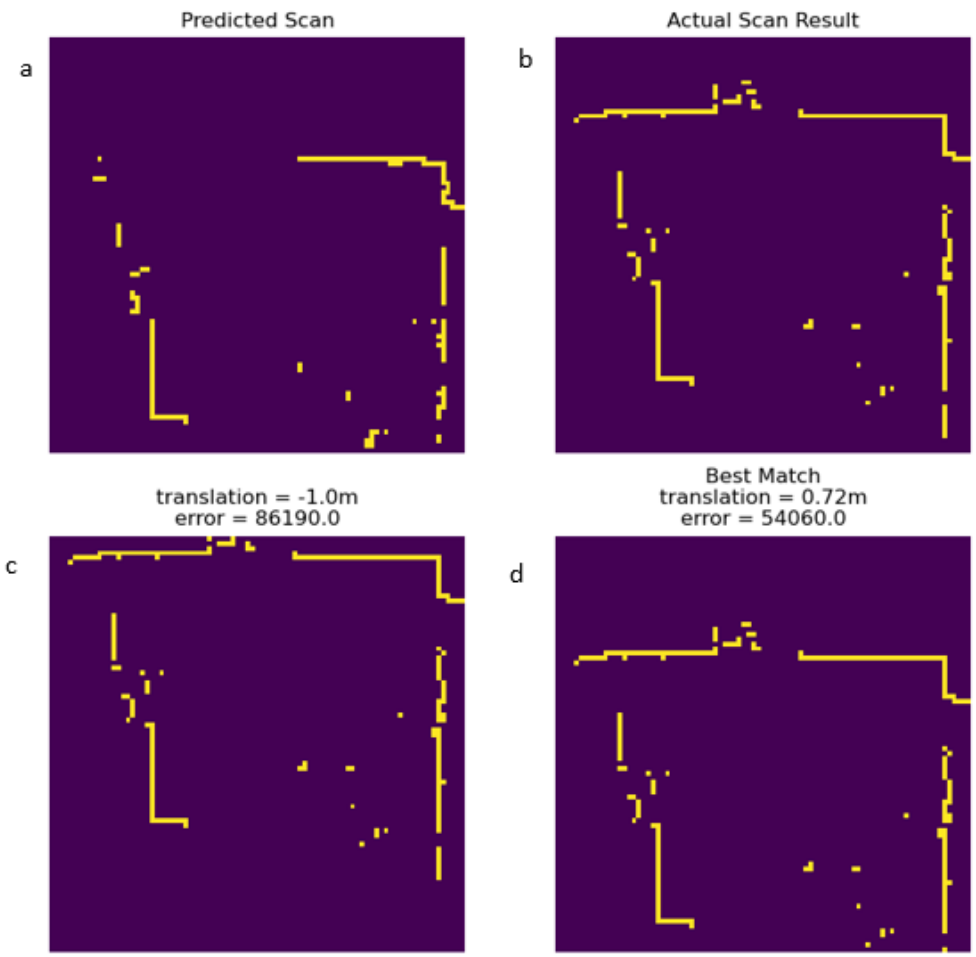


Figure 31: Result of the translation error analysis

Figure 31 shows the results of this algorithm for a 2m prediction but moving the robot only ~1.2m. Subfigure 'a' shows the prediction made from the original position. It is apparent, that the actual scan result 'b' after the movement does not match the expected result, it is shifted up in the vertical axis.

As expected, the algorithm detects a translation of 0.72m that better matches the prediction 'd' (minimum error). This result will be interpreted by system as an error in the maneuver and correct for it. Subfigure 'c' shows a possible translation with higher error, therefore discarded.

This method is accurate but prone to errors due to two main factors:

The first factor leading to error is the new information from the recent scan. As the robot moves, more objects become visible. These cells add to the total error and make the result more unpredictable,

especially for long distance maneuvers. The cells at the top of Figure 31.b are a clear example of this, since part of the top wall is missing from Figure 31.a. It can easily be fixed by deleting this data from the problematic grid. This does not remove information for correction purposes since it cannot be used anyways.

The second factor is small rotations. Small difference in the tracks' movement leads to slight rotations and translations over the horizontal axis of the grid. This leads to very small, sometimes acceptable, errors. However, since the grid is binary, small horizontal movements lead to large difference between predictions and scans. To compensate for this, a gaussian blur can be applied [Figure 33]. If no such errors were committed, the algorithm still finds the same optimal translation. However, this adjustment greatly improves predictability and reliability.

6.4.3 Detection of rotation error

Rotation error detection follows a process comparable to one used in linear error. The same prediction, maneuver, and comparison algorithm is used. Although the lack of movement in space simplifies some aspects of this process, it also creates issues of proportionality, grid binarity and conservation of the data.

In order to rotate the grid, a rotation matrix is created for the expected angle. This rotation matrix is then applied to center coordinates of cells and the data is moved to the output grid's corresponding cell. This process is very similar to the one used for image rotation. However, since the rotated coordinates do not match an actual integer value, standard image rotation approximates the value of pixel by 'mixing' the surrounding data. This destroy the binarity of the grid as show in Figure 32. [21]

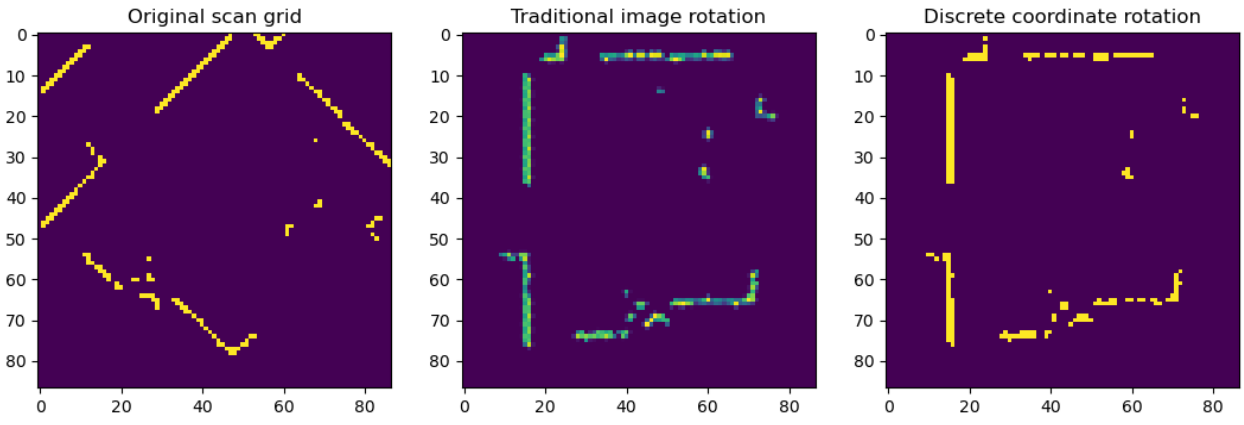


Figure 32: Comparison between standard rotation and discrete rotation of scan data

As described, this rotation process conserves the binarity of the data. This however proved to be a non-issue for the purposes of error detection since the actual resolution division is much higher and data is passed through a Gaussian smoothing filter.

In linear error detection, small rotations were not a significant issue since most long wall would still match after translation. In rotation error detection however, small translations during the rotation maneuver proved to be very detrimental to this process. In fact, translation as small as 15cm over the scan of a 5m room rendered this algorithm ineffective. This is more detrimental as resolution is increased due to the reduced apparent thickness of walls. A wall is generally only represented by a line one or two pixels thick, however, a small translation can be multiple pixels long at high resolutions.

To minimize the effects of this type of error, a Gaussian filter is applied to this data. Its standard deviation scales with the resolution division as to always have the same apparent radius for any resolution. This proves to be highly effective at mitigating the effects of translation and distortions during the rotation. After filtering, walls are no longer represented by a single, pixel-wide line. If two lines were to be separated by a noticeably short distance, when subtracting both grids, most of the weight of the lines would cancel out since the intersection of their shadows is significant.

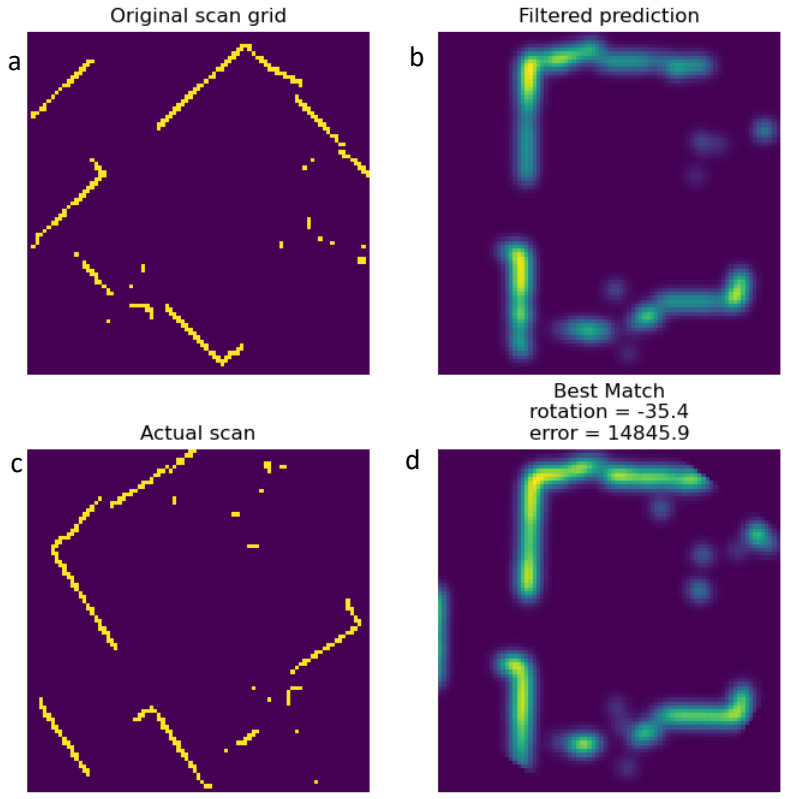


Figure 33: Result of the rotation error analysis

Legend: a: Scan before movement; b: Expected scan result (blurred); c: Scan after movement; d: Scan after movement rotated to match prediction (b)

The process then follows the same increasing tuning of the scan result until the minimum difference with the prediction is found. Figure 33 shows the results of a 45º rotation correction where the robot was made to overshoot its target. As it can be ascertained by comparing subfigures 'a' and 'b', the robot realized a rotation of nearly 90º. The algorithm correctly matched this scan data to the reference prediction by rotating it by 35.4º clockwise (negative rotation).

6.5 Precise Maneuvering

Relevant files: '/controller', '/preprocessing'

This section describes how the asynchronous encoder counters described in 6.3 and the LIDAR based error correction described in 6.4 are combined to move the robot with relative precision. This process relies in repeatedly moving a certain distance, determining the error committed, and attempting to fix it until an acceptable error in reached.

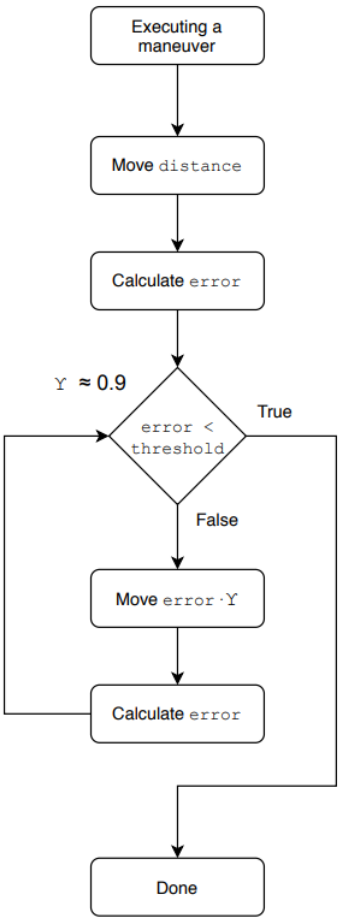


Figure 34: Precise maneuvering algorithm flowchart

6.5.1 Algorithm flowchart breakdown

When a move order is received, its entirety is running in parallel and in its own thread. This is to ensure that the robot not become unresponsive for long periods of time and is still capable of performing its order functions. Since the coordinates are constantly changing during movement, the variables storing their value are locked during this process. With position variables locked, all other functions relying on them are forced to halt their operation until the end of this execution.

Move `distance`

This part of the algorithm relies on target-based halting described in “Adding a maneuver target” from the 6.3.2 section. It uses the target flag and target count to halt the motors as soon as they reach the intended destination. It also checks periodically for this process to end according to the following

pseudocode:

```
# This function must run asynchronously
func linear_move(distance):
	target_L = position_L + distance # Update left track target
	target_R = position_R + distance # Update right track target
	target_flag = True               # Enable the interrupt halting
	forward()

	while target_flag:
		sleep(0.1)

	send_report()  # Report to controller the new coordinates
```

The periodical sleeping has a double function. It pauses the thread as to not check the shared memory space too often which could hinder the interrupt functions and it helps mitigate the reporting of stale information. As previously stated, the robot has certain inertia and tracks will continue to move slightly after being disconnected from power.

The function also utilizes the connection object to the controller to report the end of this move and the current updated position of the tracks. This information is however only temporary and will be corrected as such. This report is different from the final report after the error has been corrected. These partial reports can be disabled in configuration as to not saturate the link.

Although not represented here, the function for rotation is almost identical, with the exception of the addition sign on the left track counter and forward function. In case of a rotation, since tracks are moving in opposite direction, the target of one the tracks will be inferior to the current position and vice versa for the other. Lastly, the forward function is replaced by two different functions setting the correct direction of travel.

Calculate error

This part of the algorithm follows the processes described in the previous section (6.4). Only the error corresponding to the current maneuver is evaluated. Otherwise stated, during a linear movement maneuver, only translation error is checked but not rotation, and respectively for rotation maneuvers. Although these errors are small every maneuver, the add up over the course of an experiment as it will be discussed in the results section.

Error correction in both translation and rotation would require a more complex error detection that the current implementation.

Threshold and movement dampening

Since the precision of the motor movement is limited, reaching an exact position is mostly unfeasible with the current controls. Further fine control could potentially be achieved with a functional implementation of pulse-width modulation of the H-bridge control signals described in 5.3.1. This, however, is not feasible with the current use of the Raspberry Pi's GPIO library. Since the Raspberry Pi lacks actual PWM hardware, it can only be implemented using software-based interrupts, but these are necessary for other functions of the system, specifically the encoder position counter [20].

Every maneuver is considered as acceptable if it reaches a position within 2º and 5cm of the intended target

As the algorithm workflow shows [Figure 34], the calculated error from the movement is multiplied by a certain coefficient ϒ or 'dampening factor'. This is a remedy to the aforementioned inertia of the motors and platform. Attempting to correct small errors, with all certainty, causes over correction and therefore perpetual oscillation around the target distance. For this reason, the error is multiplied by a ϒ inferior to 1. The value of this coefficient was determined purely experimentally; a dampening factor or 0.6 for linear correction and 0.7 for angular correction yielded the best results. With these settings, the robot seems capable of reaching an acceptable error.

6.6 Wall and Obstacle Identification

Relevant files: '/preprocessing'

It has previously been mentioned that LIDAR samples are extremely hard to use in their raw state for any purpose. This is most apparent for navigation purposes. Traditional pathfinding algorithms operate mostly in graphs and discrete maps with clearly define weights and legal or illegal positions [22] (i.e.: walls or obstacles). Therefore, for any usable navigation to take place, it is necessary to first extract and differentiate objects from the point clouds of the LIDAR data.

Although technically the robot could operate entirely on the assumption that all samples belong to obstacles, this system differentiates between walls and other obstacles. Sections 6.8 "Long Range

Navigation: A*" and 6.9 "Short Range Navigation: Neural Network" explain the different considerations around these two types of objects.

6.6.1 Wall identification: Hough Transform

The system's wall identification relies heavily on the Hough Transform. The Hough transform is an image analysis and computer vision technique used for feature extraction. In other words, it provides a way of finding instances of certain shapes in an image [23]. Although originally conceived for the identification of lines, the Hough Transform can theoretically be used for any shape that can be defined mathematically. It is commonly used for finding circles or ellipses and is a base for many other computer vision algorithms [24].

It is based on a voting system where data points vote for the existence of a particular shape in data. When a shape is checked against an image for example, if a pixel falls withing that shape's borders, it will vote for it. Shapes with sufficient votes are considered existent.

Hough transform example: line

Since the Hough transform is used for identifying straight walls, this section focuses on illustrating the premise of the algorithm for straight, infinite lines. As previously stated, the Hough transform can be used for any shape mathematically definable. A straight line can be defined by the equation:

$$y = m \cdot x + b$$

This equation however poses many problems for vertical or nearly vertical lines since m approaches infinity and becomes very hard to iterate over. To remedy this issue, the polar form is used instead:

$$r = x \cdot \cos\theta + y \cdot \sin\theta$$

Assuming that only the first quadrant is used ($x > 0$; $y > 0$) is used, all possible infinite lines can be evaluated by iterating over the ranges:

$$0 \leq \theta \leq \frac{\pi}{2}$$

$$0 \leq r \leq \max(x_{max}; y_{\max})$$

Once the range and equation are defined, the next step is the voting phase. For illustrative purposes, the following example uses 3 data points and a vote threshold of 3; all points must vote for a particular line

for it to be considered valid. The following graph [Figure 35 a] contains the following points: (1, 5), (4, 2), and (2.5, 3)

Although this process repeats for every data point, only lines passing through P0 will be illustrated.

A set of lines within the previously mentioned ranges are generated and the distance to other data points is evaluated. If this distance is smaller than a certain threshold, a vote is cast for that point in favor of this line.

If a line reaches 3 votes, it is considered as existent in this data set.

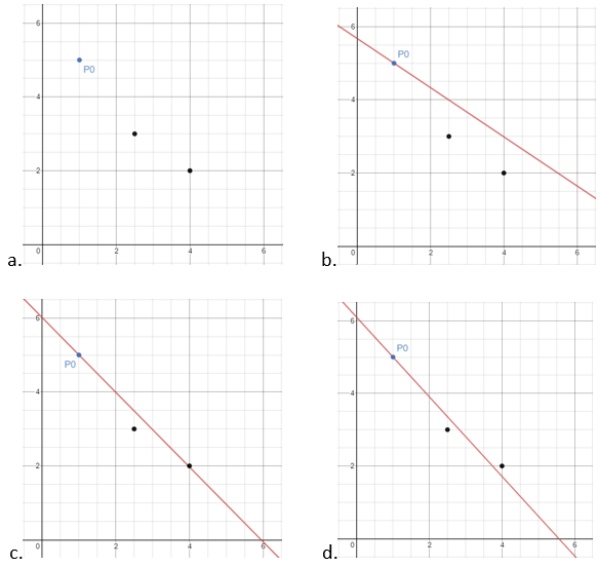


Figure 35: Hough line algorithm. A: base points; b: r= 4.71 ϑ=0.98; c: r= 4.23 ϑ= 0.78; d: r= 4.11 ϑ= 0.74

Figure 35 shows three possible lines to be evaluated by the algorithm. No other point besides P0 is close to the line in subfigure 'a'. This line will only obtain one vote and therefore be discarded. The line in subplot 'b', for this purpose, intersects another point and will surely obtain its vote but is too far from the third point to get a third vote; it is also discarded. Lastly, although the line in subfigure 'c' does not intersect any other point, it is close enough to get voted by the other two points; this line is found by Hough transform to exist.

This process might resemble linear regression, but the vital difference is that it can find any number of lines in the data and is applicable for any regular shape.

6.6.2 Probabilistic Hough Transform

A significant improvement can be made to this wall identification algorithm by using the Probabilistic (or Randomized) Hough Transform (PHT or RHT) or. PHT is a probabilistic variant of SHT where instead of iteratively polling every point for votes, only a subset of points is used. When verifying the existence of a particular line for example, since the algorithm already knows which points *should* vote positively, verifying all other null points becomes pointless. This has two main improvements over Standard Hough Transform (SHT): efficiency and flexibility. [25]

Since only a subset of points is polled for every line, this becomes greatly more efficient. This algorithm is also more flexible. As previously described, SHT could only detect infinite lines, using the border of the scan domain to set the final dimension. This is only useful for rooms formed by four walls. Although not immediately obvious, most floor plans have more complex shapes that this and identifying finite length walls returns more useful information.

Since the limits of the subset of points used for PHT can be chosen based on voting results, PHT can identify start and end points for a line.

Hough transform: LIDAR data

Although it would be possible to operate over the raw scan data, this data normally contains over a thousand data points. Since every iteration requires to check all other points, it has a quadratic ($O(n^2)$) complexity and quickly scales in computing requirements. For his reason, the current implementation uses a modified version of the discretization process from 6.3 is used [Figure 29]. As Figure 36 shows, it varies from the aforementioned discretization by cropping all empty cells. This is done to better predict the right vote threshold.

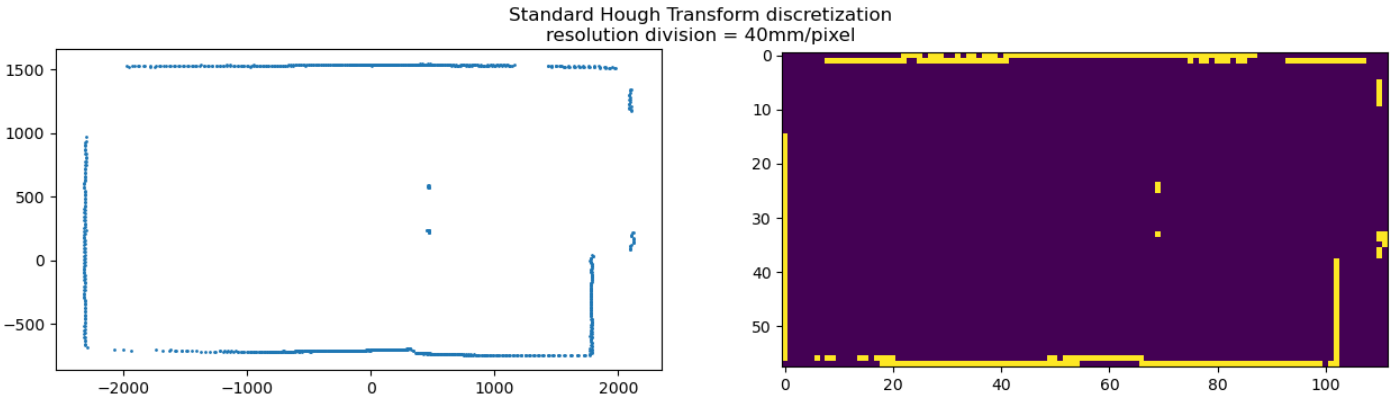


Figure 36: Hough transform data discretization

Since the number of votes a line would get depends directly on the total number of data points, the threshold must be calculated from the total numbers of positive cells (containing at least one sample) in the system. If this number is otherwise hardcoded, the number of lines would greatly depend of resolution and environment clutter. In the current implementation, at least 10% of positive cells must vote for this line. This number was chosen estimating that realistically, only 4 walls could normally be found. This number is, however, low enough that in some cases redundant walls will be found; the next subsection addresses said issue.

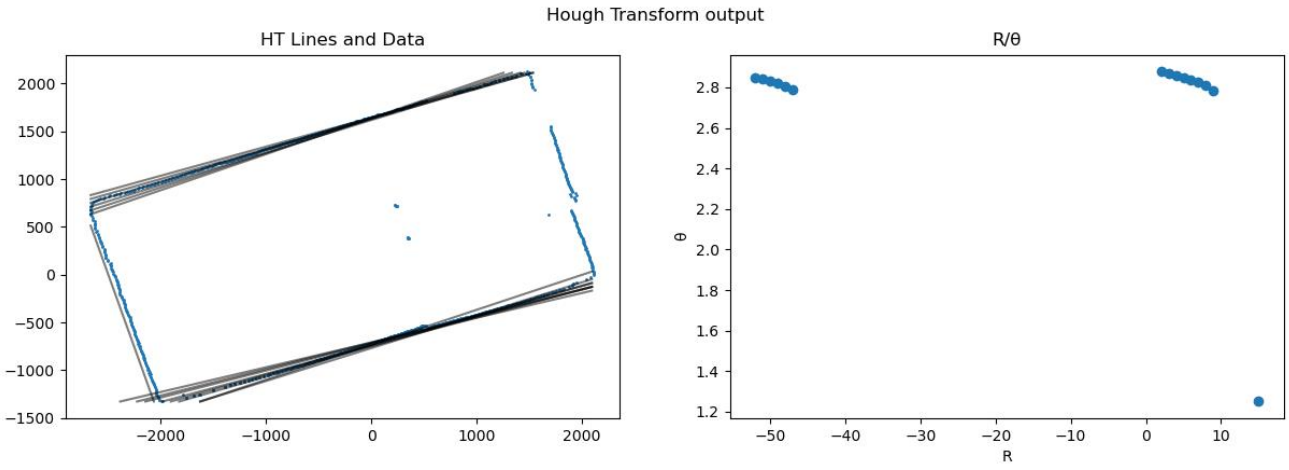


Figure 37: Hough transform unfiltered output (left). Scatter plot of distances and angles (right)

Figure 37 shows the output of the line Hough Transform. The right subplot shows all the lines returned by algorithm. These lines all obtained at least 10% off all possible votes. It is immediately apparent that a low threshold this low causes many redundant and less accurate walls to be found. However, reducing any further ignores the short wall in awkwardly shaped or elongated rooms. Thankfully, as the left subplot illustrates, the distance and angles of these lines form three clearly defined clusters corresponding to the three walls. It will be possible to identify redundant lines using clustering and averaging them.

6.6.3 DBSCAN

In order to identify wall clusters, the system uses density-based spatial clustering of applications with noise (DBSCAN). DBSAN is a density-based clustering algorithm that packs together high-density areas and labels low-density ones as outliers.

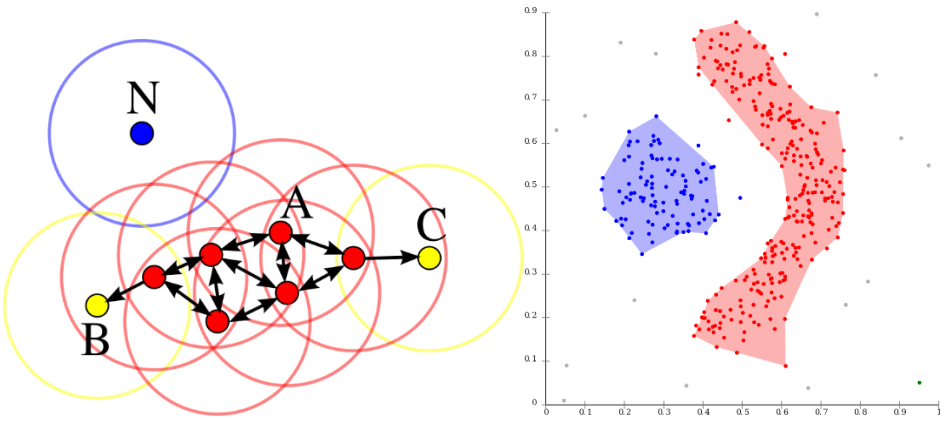


Figure 38: DBSCAN algorithm. Color coded neighborhood examples. (Source: Wikipedia)

DBSCAN uses the distance between points to find neighbors. For every point in the data, DBSCAN finds and counts all other points within a certain distance ε (sometimes called eps, radii in Figure 38 left). When a point finds a neighbor, all their neighbors are combined into a neighborhood. In Figure 38 all red and yellow points form a neighborhood. At end, neighborhoods containing more than a certain threshold of points (min_samples or min_points) is considered a cluster, all others are outliers. In Figure 38, point N's neighborhood is considered noise for any min_points > 1 [26]

The two main advantages of DBSCAN critical for the applications in the system are undefined cluster number and shape independence. For both line redundancy and obstacle finding, the total number of clusters is unknown. Unlike K-Means, using DBSCAN does not require to know this information before clustering. Although irrelevant for line grouping (see Figure 37), the shape of obstacles may make clustering difficult for many algorithms, DBSCAN can however identify non-linearly separable clusters. The clusters represented in Figure 38 left are an example of clusters that would not be adequately identified by K-Means or Gaussian Mixture for example.

Removing redundant lines

DBSCAN can be used for identifying redundant lines by correctly setting the ε and min_samples parameters. Since walls may only be identified once by the Hough Transform, min_samples is set to 0. This is a very exceptional use of DBSCAN where all points will belong to a cluster and the only determining factor of shape and size is the minimum distance ε. Neighbor range is set to ε = 2. This number was chosen experimentally but must be increased when the Hough Transform resolution is set to very high values.

Since distances in the 'θ' axis are going to be much shorter than the distances in the 'r' axis, this process would be very prone to errors, clustering together lines with a similar distance but very different angles. To correct for this issue, the angle of identified line is multiplied by a factor of 20. This gives similar ranges to both axes, greatly increasing the precision of the output.

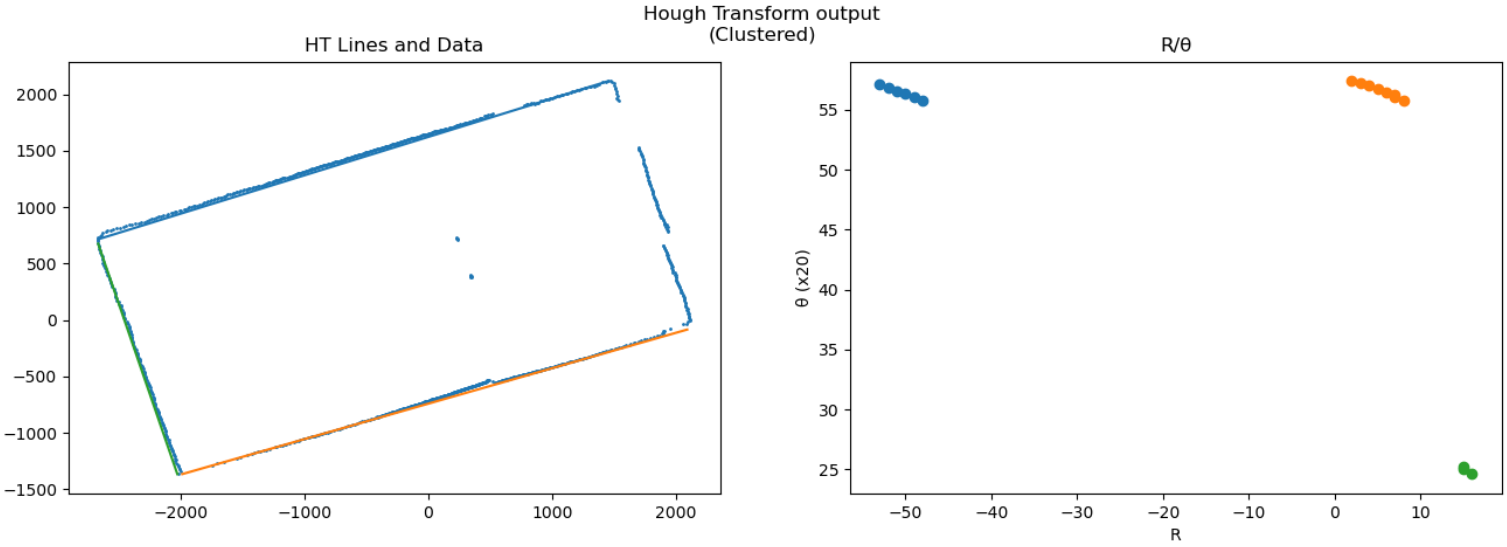


Figure 39: Clustered and filtered output of Hough Transform

Identified clustered are then represented by a single line by averaging all the corresponding points. Figure 39 shows the result of this process for the same room used to obtain the Figure 37 scan. DBSCAN correctly identified three clusters. The average line used for all clusters seems to correctly fit the wall point cloud.

It is important to note that this process in prone to failures in some scenarios. It is particularly vulnerable when the robot is near a corner between two straight walls. Another complicated situation is created when the robot is very close to an approximately flat obstacle. In these cases, lines are identified at odd angles or across a room. This creates a need for a higher abstraction of this data in order to correct for it. Section 6.7 describes such logic in greater detail.

Applying the previously described process to more complex rooms returns mixed results. Rooms with more than four walls have inside corners that cannot be defined by intersecting infinite lines. Rising the threshold for line identification, ensuring partial walls are not identified as infinite ones, leads to some walls being misidentified as obstacles. Although not critical, this slows down navigation significantly. On the other hand, lowering this threshold causes wall to be identified across a room. This misinformation is normally corrected by the reputation system described later. However, this once again slows down the mapping process.

In these scenarios, PHT data becomes more reliable and scans return a closer representation of the environments. The following figure illustrates such a scenario. This room has a more challenging geometry and a noisier environment. Most of the noise comes in the form of a large indoor plant that causes the left wall readings to be interrupted.

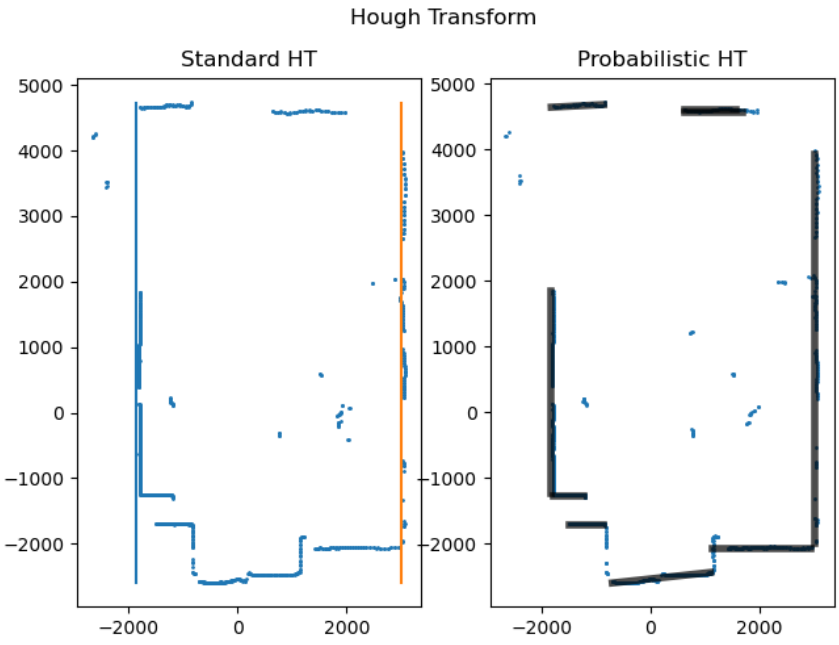


Figure 40: Hough transform comparison. Standard (left) and probabilistic (right)

As Figure 40 illustrates, in these environments, PHT wall identifications are more useful than SHT's. Setting the correct minimum wall length allows the algorithm to successfully identify staggered walls correctly.

6.6.4 Obstacle Identification

Obstacles are defined as all other detected object in the real world that are not a wall. The reason for this distinction, as will become clearer during the 6.9 “Short Range Navigation” section, is that the robot uses a different technique to avoid them compared to walls.

Firstly, all points belonging to wall must be removed. To do this, points within a certain distance to an identified line are removed. This distance is normally set to the resolution of the discretization but may be modified. The resulting point cloud data should only contain samples corresponding to object that are not walls. They can then be cluster by using DBSCAN. Points within 20mm of each other will be clustered together as long as the cluster contains more than 2 samples. Otherwise they will be considered noise.

6.7 Environment Map

Relevant files: ‘/preprocessing’

Navigation requires an accurate map of the environment storing walls and obstacles. The generation of said map requires the combination of all the systems described above. Precise movement and position feedback are necessary to track the current position of the robot. This is essential since it is used to place the newly acquires LIDAR data on the map. LIDAR data also needs to be processed to identify walls and obstacles since they are treated differently for navigation purposes. Lastly, since some of the aforementioned systems can introduce temporary error (nonexistent wall and obstacles, position and rotation error, etc.), the map needs to be able to update old or erroneous information as new data comes in.

The map created by the system is a grid divided into cells. Each cell corresponds to a square region of the real environment the robot is placed in and can be a wall, an obstacle, or empty space. The resolution and size of said grid can be configured at the start of the learning process.

6.7.1 Adding data to the grid

The first step to add scan data to the world grid is to correctly transform the coordinates from the scan to the world. The scan data is centered around the LIDAR sensor and can be considered to have the same rotation as the sensor itself. The world grid, however, is centered around an arbitrary origin. In normal

circumstances, this origin is set to the starting position of the robot. Since the grid is a discrete arrangement of data and has certain resolution, there is also a scale transformation.

Figure 41 shows scan data obtained from the LIDAR sensor placed at the marked spot on the map. It is clearly apparent by the direction of the straight lines in the scatter plot, that the scan data and map do not have the same orientation either. The sensor coordinates are noted by 't_X' and 't_Y', and its rotation by θ.

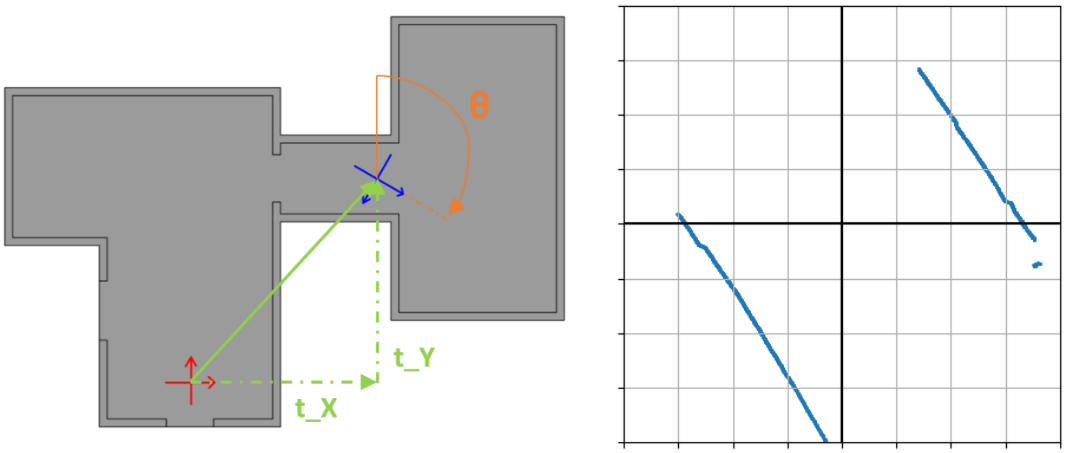


Figure 41: Scan data (right) obtained by sensor placed at the marked position and rotation (left)

To correctly place this data in the same reference as the map, a translation and a rotation must be applied. Given a point ' P ' from the scan point cloud, its corresponding coordinates on the world map, ' P' ' can be calculated by a multiplying it by a rotation matrix 'R' and a translation matrix 'T'.

$$P = \begin{bmatrix} P\_x \\ P\_y \\ 1 \end{bmatrix} \quad T = \begin{bmatrix} 1 & 0 & t\_X \\ 0 & 1 & t\_Y \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P' = T \cdot R \cdot P$$

$$P' = H \cdot P$$

$$H = \begin{bmatrix} \cos\theta & \sin\theta & t\_X \\ -\sin\theta & \cos\theta & t\_Y \\ 0 & 0 & 1 \end{bmatrix}$$

It is important to note the order of the transformation. The rotation must be applied before the translation since the data points rotate around the LIDAR origin and not the grade origin. The H matrix is then the total transformation matrix used at the beginning of the data import process.

This step requires the precise position and rotation of the robot. These values are updated every time a maneuver is executed. Any error in this data, caused by external forces for example, adds up over time and increases the total error in the grid map. As previously mentioned in the "Precise Maneuvering" section, every translation or rotation has a certain acceptable error '*e* '. The values of position and rotation are updated as such:

- Linear maneuver i: advance ΔL_i forward

$$t\_X' = t\_\mathrm{X} + \Delta x_i + e_{xi};\ \ \Delta \mathrm{x_i} = \sin\theta_i \cdot \Delta L_i$$
$$t\_Y' = t\_\mathrm{Y} + \Delta y_i + e_{yi};\ \ \Delta \mathrm{y_i} = \sin\theta_i \cdot \Delta L_i$$
$$\theta' = \theta + e_{\theta i}$$

  Although small, the robot also experiences a small rotation during linear maneuvers, hence the existence of $e_{\theta i}$

- Rotation maneuver i: rotate by $\Delta\theta_i$

$$t\_X' = t\_\mathrm{X} + e_{xi}$$
$$t\_Y' = t\_\mathrm{Y} + e_{yi}$$
$$\theta' = \theta + \Delta\theta_i + e_{\theta i}$$

  In rotation maneuver, translation errors are mostly negligible but still existent.

Therefore, at any time during the robot's discovery cycle, a certain error is committed when transferring data to the grid. This error trends infinitely upwards. This causes a certain blur effect in the grid as many redundant objects are placed in it with a small translation.

6.7.2 Wall discretization

Since a grid cell containing data can either be a wall or an obstacle, to update a wall's corresponding cells, a process to obtain these coordinates is necessary. This can be considered analogous to point discretization described in 6.4. However, since a wall likely spans multiple cells, this process requires us to calculate all the involved coordinates.

As mentioned in 6.6.1, lines are defined by their polar coordinates (r, θ) such that:

$$r = x \cdot \cos\theta + y \cdot \sin\theta$$

Therefore, the center coordinates (x_0, y_0) are defined by:

$$x_0 = r \cdot \cos\theta \;\; ; \;\; y_0 = r \cdot \sin\theta$$

And the start (x_1, y_1) and end $(x_2,\ y_2)$ coordinates are defined by:

$$x_1 = x_0 - d \cdot \cos\theta \;\; ; \;\; y_1 = y_0 + d \cdot \sin\theta$$

$$x_2 = x_0 + d \cdot \cos\theta \;\; ; \;\; y_2 = y_0 - d \cdot \sin\theta$$

Where d is a distance calculated so the resulting line intersects the limits of the Hough transform cropped input (see "Wall identification: Hough Transform").

These coordinates can now be directly transformed into the corresponding grid index by simple integer division with the resolution divider. The wall spans across an imaginary line between cells at indices `I_start` and `I_end`.

The other line indices are calculated by using a slightly modified Bresenham's line algorithm [27]. Bresenham's is a line rasterization algorithm used to draw lines in n-dimensional raster. Its normal method involves calculating the distance of the cells above and below the actual line and selecting the more accurate one.

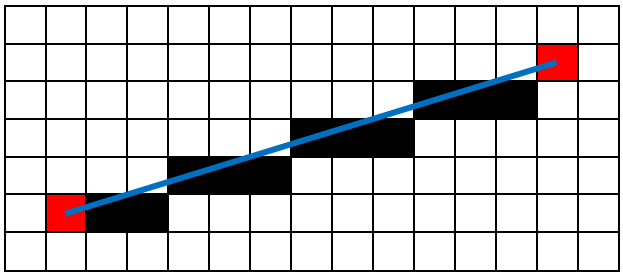


Figure 42: Rasterization of a finite line

The implemented variant calculates the steps in both dimensions by assuming that the ratio between them is constant. Once this ratio has been calculated, it is unnecessary to continuously calculate the right pixels that belong to the line in question. In the example depicted in Figure 42, this ratio is 3:1; three pixels in the x direction for each unitary step in the y direction. This pattern repeats until the end is reached.

6.7.3 Reputation system

Since an incorrect object can be placed in the map grid, a system for removing errors is necessary. Without such system, the robot might identify an inexistent wall across a room, leading to an impossible path-finding problem. Objects and people in a room can also move leading to all previous information regarding their position becoming stale.

The presented solution is a reputation system that assigns a certain reputation to every wall and obstacle cell. This reputation is updated every time new information regarding the cell contents is received. If a new scan confirms the contents of the cell, the reputation increases. If, on the contrary, the new scan does not find the expected object in the cell, the reputation decreases.

Special consideration must be made between the nature of obstacles and walls. A wall can easily be misidentified as an obstacle if the wall is cropped by the LIDAR's maximum range, other obstacles, or simply too close or far to the scanner. Therefore, identifying an obstacle in a cell previously identified as a wall decreases its reputation less than identifying it as empty space.

Calculating reachable cells

Naturally, not all cells will be updateable every scan. The LIDAR sensor has a maximum range of 16m and can only detect objects within its line of sight. The previously described algorithm requires information regarding what cells are visible to the sensor. Otherwise, updating all cells would lead to a mostly empty map.

The first condition to meet for a cell to be reachable is distance to the sensor. This condition can be asserted in two ways. The simplest implementation is an iterative one where the distance to the scanner coordinates is calculated for every cell. However, given the size of the grid this would be extremely intensive as the system would be calculating the distance to between two points hundreds of thousands of times every time a new scan is perform. The much more efficient approach is to calculate all the reachable indices in advance. To do this, the system first calculates the indices in a square around the (0,0) origin with a side equal to the scan radius. From these indices, all those outside the radius are removed. This array can now be offset by the current position to determine all the cells within range.

The second condition to meet for a cell to be reachable is obstruction. Since the LIDAR is a line of sight based sensor, it cannot detect object behind any sort of obstruction. Therefore, cells behind a wall or

obstacle must be removed from the update process. Otherwise, walls behind an obstacle would lose reputation due to misidentification as empty space.

This condition is checked by verifying if any data was found on the straight line between cell and center of scan. To illustrate this process, observe the following scenario where an obstacle is placed between sensor and a wall [Figure 43]. In this situation, the grid (Figure 43 middle) is already populated and the system is updating the grid with new scan data (Figure 43 right).

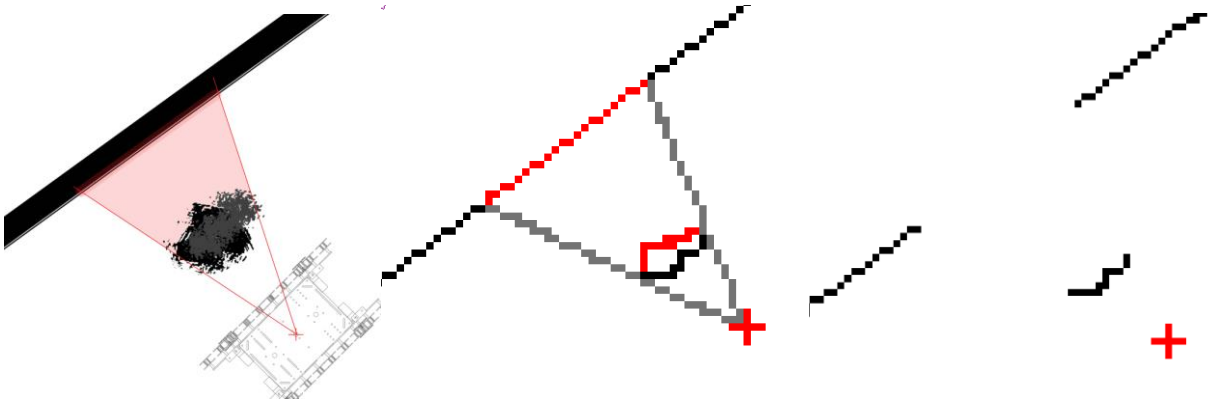


Figure 43: Results of a scan limited by obstacle occlusion. Scenario (right), occluded grid cells (middle), and scan data obtained (right)

Without proper checks, the new scan would erroneously reduce the red cells reputation since they are not present in the new data. However, since they are 'hidden', updating their reputation would be against the objective of producing an up-to-date map.

To prevent this, all cells within update range check the path to the center of data. This is done by creating an index array of all cells in a straight line between it and the center using the same rasterization described in 6.7.2

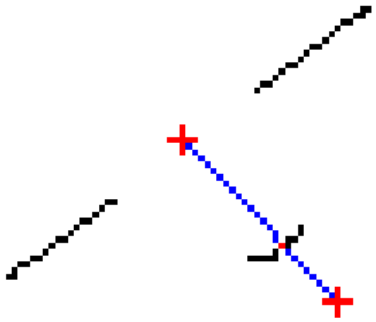


Figure 44: Identification of a hidden cell by rasterized line method

Figure 44 illustrates this process with the previously established scenario. Since the path to the center of the scan (blue line) coincides with at least one non-empty grid, this part of the data will not count towards reputation update.

6.8 Long Range Navigation: A*

Relevant files: '/navigation', '/preprocessing'

The robot's navigation is in two parts, long range navigation and short range navigation. When no obstacles are detected near the robot, it operates under the assumption that the path is clear. Using the neural net in these circumstances would be extremely slow and would require a much more complex development. For this purpose, the system relies on the path search algorithm A* ('A-star').

A* is commonly used for many applications for its flexibility and high efficiency. It was created as part of the Shakey project [28] that aimed to create a semi-autonomous mobile robot not unlike the one developed here.

6.8.1 Algorithm

Although normally represented as a routing algorithm with graphs, this section will explain and illustrate the A* algorithm on a grid. This grid is binary with a value of 0 if the space is allowed (walkable) or 1 if it is not (a wall). It can be interpreted also as a very dense graph were every open grid space would correspond to a node, and walls would correspond to the absence of one. Figure 45 shows two equivalent representations of the problem used for demonstration. The goal for the algorithm is to find the shortest route from A to B.

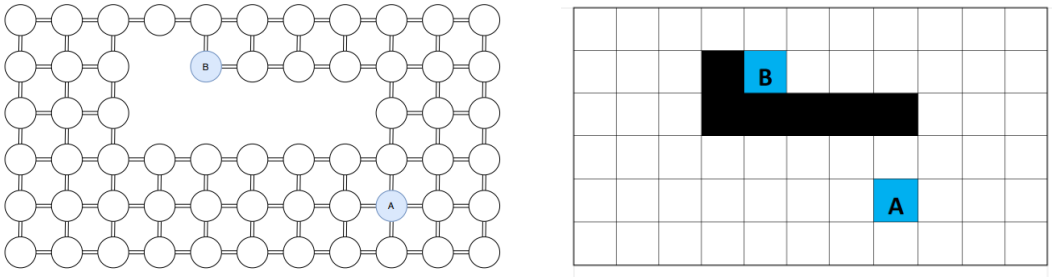


Figure 45: Demonstrated pathfinding problem. In graph (left) and grid (right) representations

Since the objective is to reduce travel distance, the cost of visiting a node will be calculated based on the geometric distance to both A and B. For any node n, the **total distance of the path** to starting node A is noted $g(n)$, and the **distance (Euclidean)** to end node B is $h(n)$. As previously mentioned, A* is very flexible, by selecting the pertinent metrics for $g(n)$ and $h(n)$, the algorithm can be repurposed for many routing applications. For example, driving time can be used as a metric for satellite navigation, or packet transfer time for low latency networking.

$$g(n) = \sum_{i=0}^{n} g\,(i) + dist(n, i)$$

$$h(n) = dist(n, B)$$

$$f(n) = g(n) + h(n)$$

$g(n)$ $h(n)$

$f(n)$

Note: for the following diagrams, grid cell costs will be represented as shown above (right)

In the first iteration, A* will calculate these metrics for all the adjacent cells to A. In the problem example, the distance between two adjacent cells in the vertical or horizontal direction is set to 10. Therefore, the distance to an adjacent but diagonal cell is set to $10 \cdot \sqrt{2} \approx 14$. All nodes that have been evaluated and are within reach are called 'open' and colored green in the following figures. The first iteration ends by checking all the open nodes and selecting the one with the smallest cost.

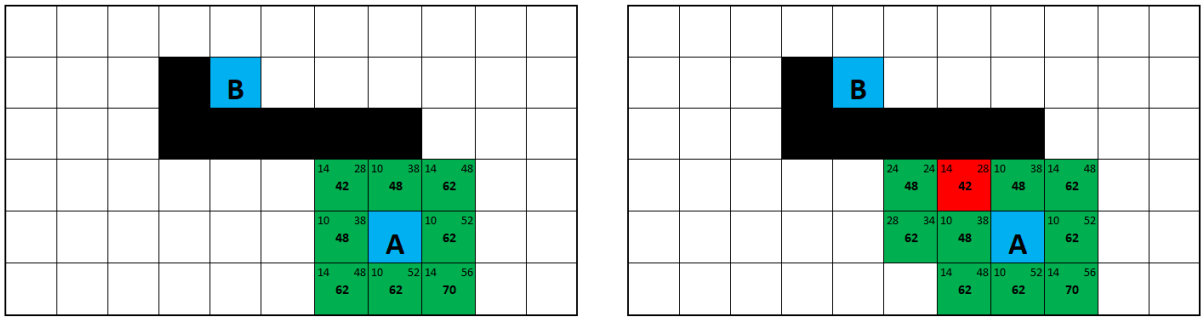


Figure 46: First iteration of A algorithm*

Since the top-left cell to A has the lowest total cost, it is selected as the next step in the path. All already visited nodes are called 'closed' since there cannot be a better path by revisiting them. They are colored red. At the beginning of the next iteration, three cells share the same lowest f cost of 48. Therefore, the algorithm continues onto the smallest g cost.

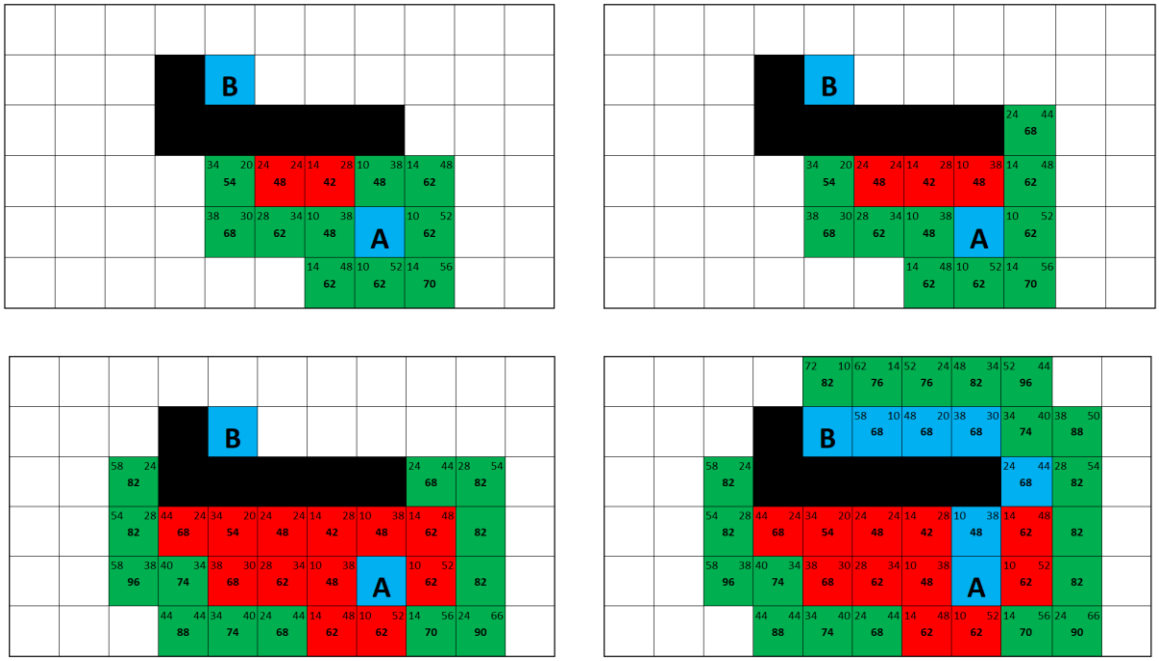


Figure 47: A algorithm state at iterations: 2, 3, 12, and 17 (end)*

The algorithm ends when the end node becomes closed [Figure 47] or no more open nodes are available. To recover the path, the algorithm then returns from end to start by picking the nodes with lowest g cost.

6.8.2 A* in the world Environment Map

For the following application and demonstration of the A* algorithm on real world data, the map shown in Figure 48 is used. This map was generated by the robot following the processes described in 6.7 "Environment Map". It uses a resolution 10cm per division and covers an area of approximatively 65m^2. Since this is not used for continuous learning, reputation of cells is currently ignored, and all discovered walls are considered as such.

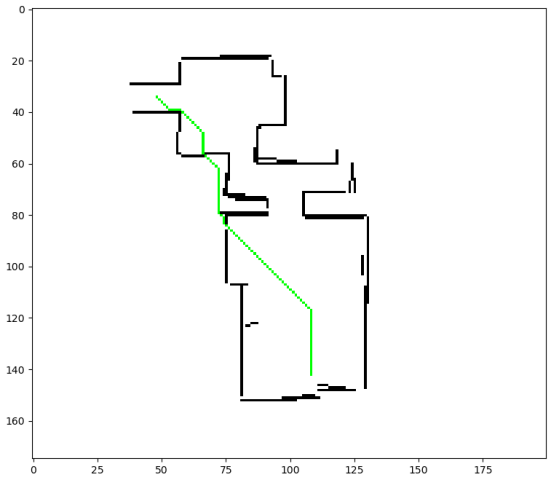


Figure 48: Application of A algorithm on a wall map*

In Figure 48, the green trace displays the output of A*. This result is clearly impossible as it would send the robot across walls to reach the destination. This is caused by discontinuous walls, a small error introduced by the Probabilistic Hough Transform used for wall identification.

During normal continuous operation, the robot would approach the wall attempting to follow the A* path. In doing so it correctly identifies that the wall is in fact continuous and calculate a new path in the updated environment map. However, this slows down navigation and it can mostly be mitigated by adding a cautionary boundary around all wall. By setting all cells around walls as unavailable to A*, it calculates a route that avoids getting too close to them.

This is also useful to ensure that the robot does not collide with walls. As Figure 48 and Figure 49 left show, the path tends to get very close to corners and walls since that is in fact the shortest route. However, the robot is not a single point in space and has significant width and length. Adding this cautionary boundary ensures the path is fact doable.

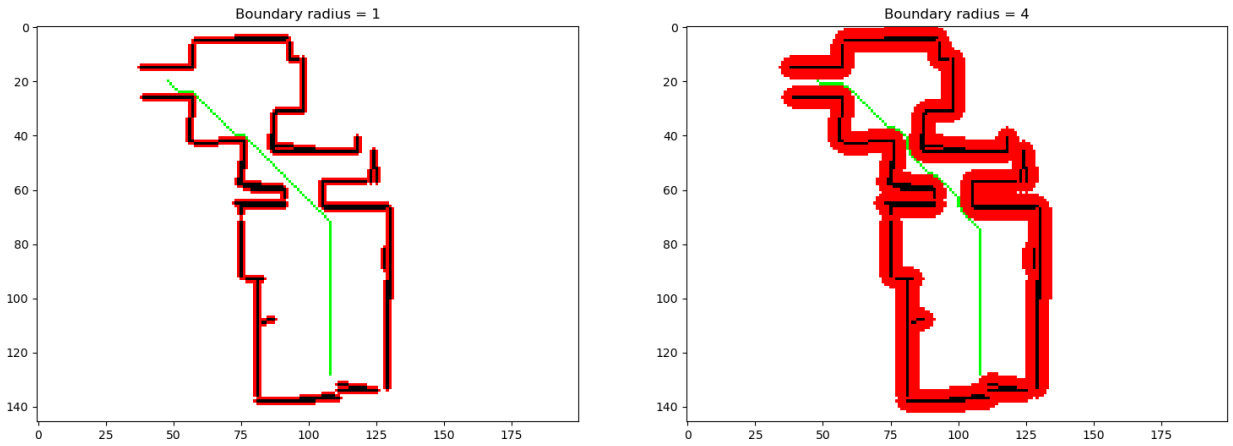


Figure 49: Application of A algorithm on a wall map with wall boundaries. Radius = 1 (left) and = 4 (right)*

Choosing the right wall boundary is complicated and a precise adaptative algorithm was not implemented. It is still a configurable parameter than can be set by the controller. Choosing boundary too thick renders the problem impossible but this is indistinguishable from a close room situation from the A* algorithm's perspective.

6.8.3 Using A* for navigation

Although the previous section showed the method used to generate a path using A*, following it would involve execute many maneuvers. As section 6.5 explained, every time a maneuver is executed, the robot attempts to calculate the error committed and cancel it. This adds error to the current position accumulators and takes significant time compared to travel itself. Therefore, executing a different maneuver for every cell in the path would be extremely time consuming and produce a very noisy and deformed map.

The solution to this problem is to convert the path into fewer and more manageable maneuvers. Since the robot is only capable of moving in a straight line or rotating over itself, this simplification resembles piecewise linearization of the path [29]. Straight lines in the path can be easily be identified if they are diagonal, horizontal or vertical. Therefore, a turn can be identified by change in direction. The following pseudocode does precisely this.

```
# path is the output of A* pathfinding algorithm
previous_direction = [0, 0]
turns = []
for i = 0 to size(path)-1:
    direction = path[i] - path[i+1]
    # if direction changes add this point to turns
    if direction != previous_direction:
        turn.append(path[i])
    previous_direction = direction
```

Figure 50 (a) shows the resulting identified turns and straight parts of the A* path. This plot also reveals that most of the turns are concentrated in clumps that can be grouped together to simplify the path into just a few turns. To accomplish this the clustering algorithm described in 6.6.3, DBSCAN, is used. Turns within a certain distance are grouped together and substituted by only the first and last turns.

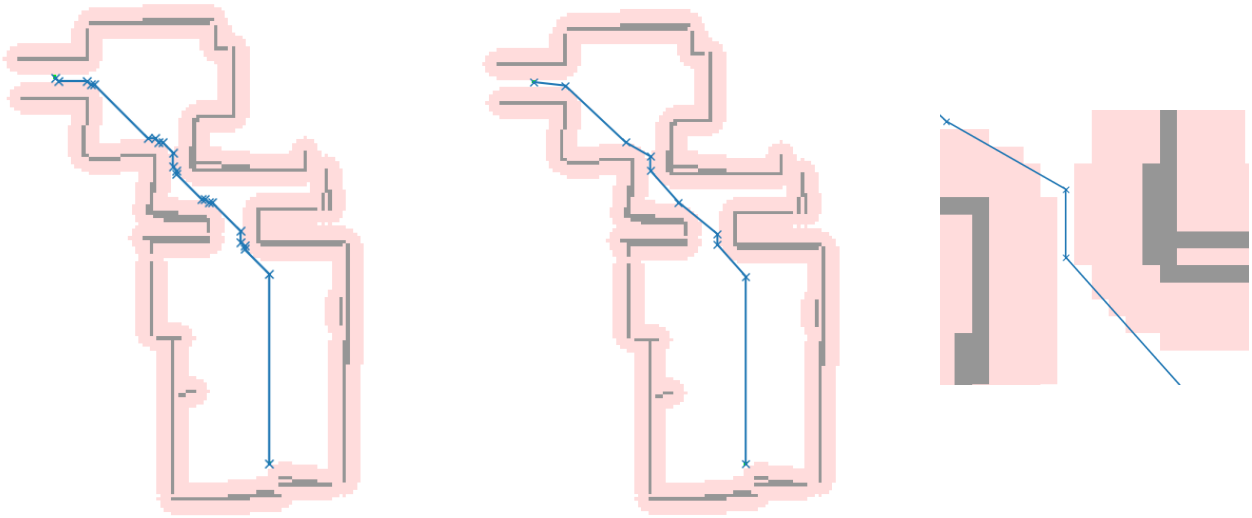


Figure 50: A to maneuvers. Unclustered (left), Clustered (center), and detail (right)*

The final route that the robot will attempt to follow is depicted in Figure 50 (center). The original path was interpreted at first as 45 maneuvers, 22 rotations and 23 linear. The final clustered and simplified route contains only 17.

This simplification sometimes intersects the safety boundary established for pathfinding purposes (Figure 50 right). Experimentally, however, this never seemed to cause a problem as long as the original boundary was large enough.

It is important to note that new information added to the environment map might make this route impossible. When a new wall is inputted, the system checks if it interferes with the current A* path. If the current path is now blocked, this triggers a new path to be calculated.

6.9 Short Range Navigation: Neural Network

Relevant files: '/navigation', '/models'

Whenever an obstacle is detected near the robot, A* navigation pauses in favor of a neural network (net) controller. This threshold is normally at 1.5 meters but can be reconfigured. This neural net was trained in a simulated environment before being loaded into the system. However, its training continues during normal operation learning from the results of its decisions.

This type of navigation is noticeably less precise than the previously described one. This is due to a lack of correction in movement error. It uses the same error recognition process described in 6.4 "Position Tracking II: LIDAR based tracking" but does attempt to correct them. In other words, it is error-aware but not error-correcting. For this reason, maps become somewhat deformed and blurred by using this technique.

6.9.1 Shape, Inputs and Outputs

The neural net uses a Keras/TensorFlow sequential model with one or more hidden layers. It Is therefore a deep feedforward network where the nodes are arranged in layers. It is called feedforward because the information flow from input to output interacting with each layer only once, not forming any cycles. It is in fact one of the most common and simple neural net designs.

Inputs

Multiple neural net input combinations and shapes were tested during the development process. However, the general input arrangement is always similar. Four of the inputs are constant: distance and course to closest obstacle and distance and course to the next maneuver node calculated by the applying the A* algorithm: the *goal.*

Figure 51 shows an example of how these inputs are obtained from world data. The green square represents the goal; therefore, the angle and length of the green line are inputs values to the neural net. Similarly, the red square in the closest discovered obstacle point.

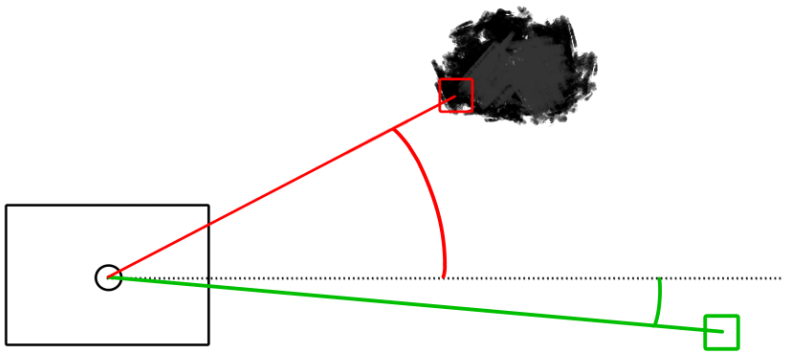


Figure 51: Robot's environment during neural net navigation. Closest obstacle (red) and target (green)

Since there can be any number of walls identified around the robot, it would be highly impractical to use a similar logic as for obstacles and goals. A possible solution that was evaluated for this project is the use of vision lines. Vision lines can be interpreted as unidirectional LIDARs for the neural net. They represent the distance to a wall in a particular direction. Figure 52 shows an example containing 3 vision lines oriented at 0º, 45º and -45º. It is important to note that unlike the obstacle and goal inputs, vision lines do not report angle since it is always constant.

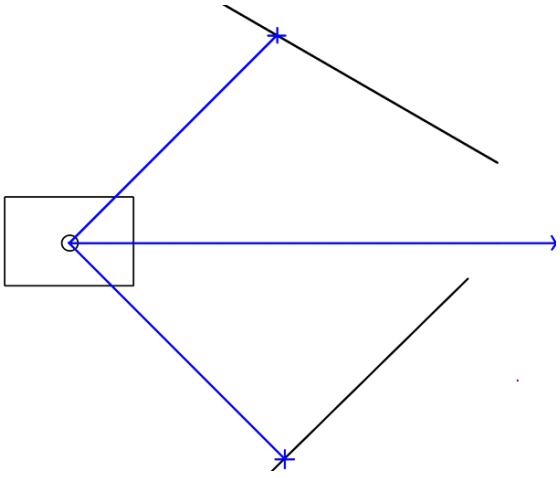


Figure 52: Robot's vision lines and walls

Neural net neuron activation can become unpredictable when the values of inputs are vastly different. For this reason, it is commonly recommended to scale these values to bring them close to the [-1; 1] interval. To do this, all angle values are given in radians and distances are divided by the maximum range of the neural net navigation. This limits angular inputs to be between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, and distance inputs to be between 0 and 1.

In theory, the neural net could receive any number of inputs. It would be possible to feed information regarding multiple obstacles and dozens of vision line. However, as will be shown in later results, increasing the amount of inputted information past a certain point did not seem to improve performance but significantly slowed down execution and training. In fact, adding too many vision lines (> 8) caused training to crash and resulting model to be unusable.

Outputs

The outputs of a neural net indicate the possible predictions for the application environment. For a classifying net, the outputs would correspond to the possible classes the input could be classified as. Therefore, for the navigation neural net, the outputs correspond to the possible movements the robot can make. There are in total four possible movements: move forwards, move backwards, rotate right, and rotate left.

These movements can change in magnitude. The system can be configured to interpret a “move forwards” decision as a 1cm maneuver or a 1m one. The results vary enormously between simulation and real-life application depending on how fine or coarse the movement is set. As will be expanded later, the simulated robot has no problem executing infinitesimal movements and keeping track of its position at the end. Attempting too fine of a movement in real-life however, leads to a higher error by the time long range navigation is reestablished.

Hidden layers shape and depth

As previously mentioned, input and output layers are not directly connected. Instead, there are number of hidden layers in between. This in theory makes the neural net a deep neural network (DNN). However, this depth is very limited. The tested neural nets had between one and 5 hidden layers. Higher depth did not seem to improve the results past two hidden layers. Very deep neural nets did not seem to acquire any new desirable behaviors. Other more complex structures of neural networks have been tried for

similar purposes. These however used dueling nets and similar processes that were not tested here. [4] [3] [2]

Layers also have a width dimension. Input and output layers are constrained to have the same size as the number of inputs and outputs, but hidden layers can have any width whatsoever.

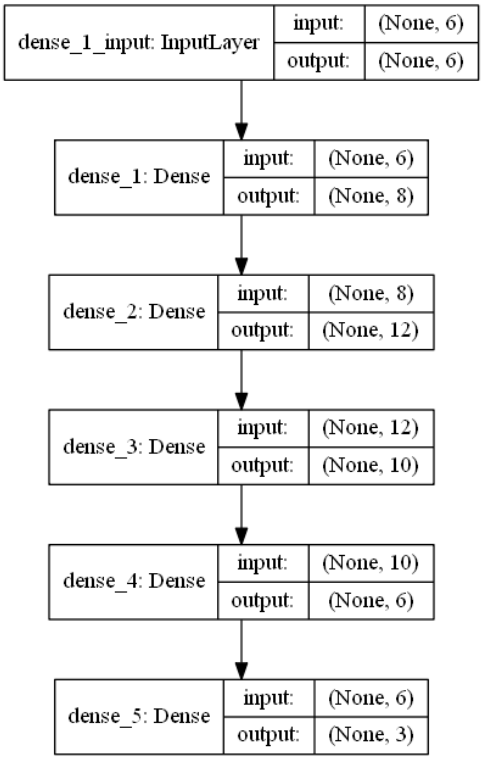


Figure 53: Keras neural net structure

6.10 Deep Q-Learning and Cloud Training

6.10.1 Reinforcement learning: Deep Q Learning

Q-Learning is a reinforcement learning algorithm used to teach an agent to decide the best action according to its environment. In Q-Learning, the agent is placed inside an environment it can interact with. The agent is expected to make an action decision (A_t) based on the current state of the environment (S_t). When the agent acts upon its environment, a reward (R_{t+1}) is calculated based upon how desirable the new state (S_{t+1}) is. [Figure 54] The coefficients used by the agent to predict the correct action are called the **Q-values**. In the deep learning implementation of this algorithm, Deep Q-Learning, these values correspond to the weights of the neurons. [30]

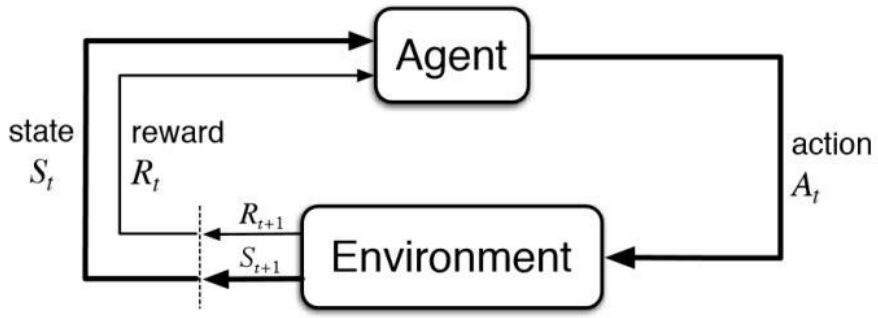


Figure 54: Q-Learning agent/environment interaction loop

For example, consider a mouse inside a unidimensional maze. In one direction, a piece of cheese is placed while in the opposite direction lies a snake. In this scenario, the mouse represents the agent, and the cheese, maze, and snake represent the environment [Figure 54]. The mouse therefore has two options, move to the left, closer to the cheese, or move to the right. The action taken by the mouse is then evaluated based on the new state of the environment.

Figure 55: Example of an agent (mouse) and a its environment

Assuming the mouse moved to the left [Figure 55], and therefore closer to the cheese, this state is more favorable for the agent and should be positively rewarded. Once the mouse reaches either the cheese or the snake, it has reached a "terminal state". Since terminal states are the end of the exercise and indicate the outcome of the simulation, they normally result in either very high or very low rewards. In the above described scenario, the mouse would receive the highest total reward by continuously walking towards the cheese.

Note that most environments could result in an infinite loop, for example, the mouse could alternate between walking left and right infinitely. To prevent this, a maximum number of iterations is commonly

placed. If the agent does not reach a terminal state before this number of moves, it is considered a negative terminal state and negatively rewarded.

Reward propagation

One of the most important aspects of Q-Learning is reward propagation. As previously mentioned, when the agent reaches a desirable terminal state (reaches its goal), it receives a high reward. However, reaching the goal involves much more than just the last step. It is the result of a series of correct decisions made by the agent. When updating Q takes this into account by propagating the reward to previous states according the following progression where γ is the "discount factor" and α is the "learning rate".

$$Q'(s_t, a_t) = Q(s_t, a_t) + \alpha \left(r_t + \gamma \cdot maxQ(s_{t+1}, a) - Q(s_t, a_t)\right)$$

$$0 < \alpha \leq 1$$

$$0 \leq \gamma \leq 1$$

$Q'(s_t, a_t)$ are the new 'trained' Q-values and $Q(s_t, a_t)$ are the previously 'untrained' Q-values. If the learning rate α is close to zero, the new Q values are very similar to the old ones and this process barely modified the behavior of the agent. If on the other hand, α is closer to 1, this process causes a large change in the Q-value. This is normally highly undesirable as it leads to overfitting and suboptimal behaviors getting to lock up the training.

Since the new value Q-values depend not only on the reward obtained at this state s_t but also on the best possible future prediction $maxQ(s_{t+1}, a)$, this algorithm takes into account future rewards:

$$Q'(s_t, a_t) \leftarrow \gamma \cdot Q(s_{t+1}, a) + \gamma^2 \cdot Q(s_{t+2}, a) + \cdots + \gamma^n \cdot Q(s_{t+n}, a)$$

$$where\ n\ is\ the\ terminal\ state$$

If the discount factor γ is set to 1, all the future rewards resulting from this decision, carry as much wait as the reward obtained from this single action a_t. If on the other hand, the discount factor is set low, the agent becomes more and more myopic, ignoring future rewards in favor of an immediate, smaller one.

6.10.2 Environment and Reward Function

Training a neural net using Deep Q-Learning requires an environment that simulates the problems to be faced in real life by the robot. It also requires an efficient way of evaluating the state of the robot at any time. To do this, an environment was programmed capable of generating a scenario emulating the

problem described in 6.9. Figure 56 shows a render of this training environment. These renders use the same notation as figures described in 6.9.1.

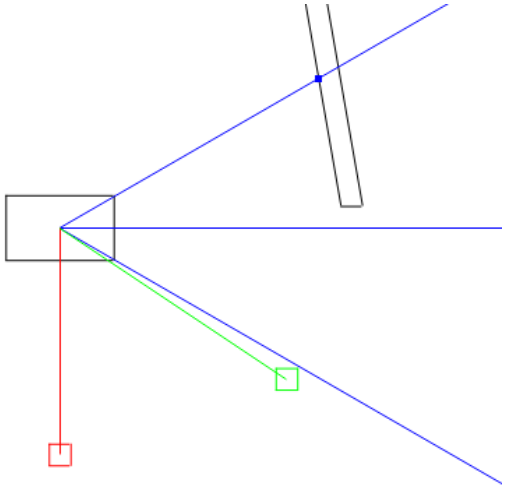


Figure 56: Training environment render

It is important to note that although the environment is omniscient to the current scenario, the state information it feeds to agent is very limited. As previously described, the environment calculates the agents vision lines, goal and obstacle vectors, simulating the information used by the neural net in real-life scenarios.

Variable difficulty training

Although in theory it could be possible to train the neural net to solve complex scenarios with a slow learning rate, the neural net is trained using a stepped difficulty. To accomplish this, the environments are randomly generated with a difficulty setting: low, medium, and hard [Figure 57]. The low difficulty setting is meant to train the basic agent-goal and agent-obstacle interactions. Medium difficulty builds up by adding some simple interaction with walls and tougher obstacle-goal positions. Finally, the hard difficulty is intended to train complex interaction with wall, requiring the agent to move around walls using limited information.

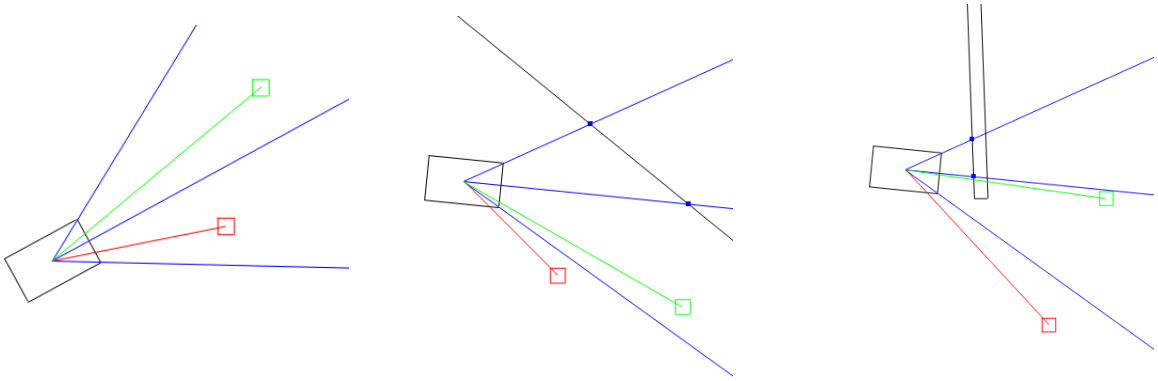


Figure 57: Examples of environments of easy (left), medium (middle), and hard (right) difficulties

Easy and medium difficulty environments proved to be suitable for this neural net's training. Hard difficulty however, lead to unpredictable results. More vision lines improved the results obtained but not significantly. It is uncertain at this point the relationship between the width of the neural net and its fitness for problem of varying difficulties.

Reward function

The reward function is the most important aspect of reinforcement learning. Agent evolution will trend to maximize the reward obtained. Therefore, a well-defined reward function will speed training and lead to the desired behavior.

The simplest type of reward function is a flat reward at terminal states; any intermediary step between the starting point and the final states would receive a null reward. When the agent reaches the goal, it would receive a positive reward. If it collides with a wall or hazard or exceeds the maximum number of steps it would be penalized.

Since Deep Q-Learning implements reward propagation, over time, the neural net will evolve towards the desired outcome. However, this propagation is slow and may introduce more undesirable behaviors. To mitigate these issues, a proportional reward is considered better for this scenario [31]. A proportional reward is given at the end of every episode based upon how desirable the new state is. A good metric for example would be distance to the goal:

$$r_t = 1 - distance(position_{t+1}, \ goal)^{\beta}$$

$$0 < \beta < 1$$

This type of reward function gives the available rewards a certain slope instead of a simple win-loss binarity. Figure 58 shows a comparison of the rewards obtained by the agent around the goal and hazard positions. In the case of continuous rewards, following the slope leads to the desired outcome. On the other hand, when a simple binary reward is given, the agent must find the goal by pure random exploration in order to achieve the desired behavior.

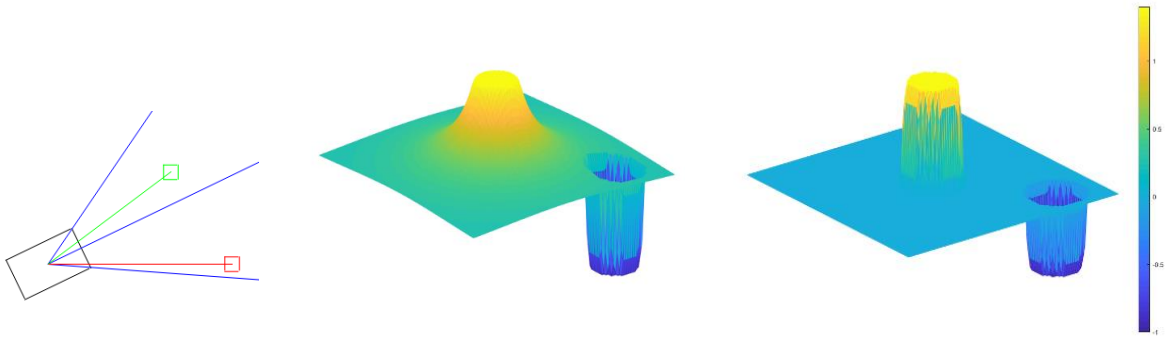


Figure 58: Rewards obtained by agent in continuous (middle) and binary (right) reward schemes

Exploration and Exploitation

To correctly learn how to satisfactory interact with its environment, the agent needs to explore it. If only predictions from the agent are used, the agent does not have a chance to explore other options. To help the agent learn where the reward maximums are, we force it to take actions at random. However, at some point, the agent's predictions need to be considered and exploited. Otherwise, the agent would only move at random.

The balance between exploration and exploitation is controlled by the ratio ε starting at 1. At the beginning of an episode, a random number is generated and compared to ε. If this number is inferior to ε, a random action is taken, otherwise the agent's choice is used. At the end of the episode, ε is multiplied by ε_decay, a number inferior but close to 1.

$$\boldsymbol{r = rand()}$$

$$\boldsymbol{action = randint()} \qquad if\ r \leq\ \varepsilon_t$$

$$\boldsymbol{action = Q(s_t)} \qquad if\ r >\ \varepsilon_t$$

$$\boldsymbol{\varepsilon_{t+1} =\ \varepsilon \cdot\ \varepsilon\_decay}$$

If ε_decay is set very close to 1, the agent's actions will take a long time to be executed but will have a long time to explore. On the other hand, if it is set lower, the agent will not have as much time to explore before having to solve the problem itself.

6.10.3 Training process

When training a new neural net, directly applying the previous process, even with a low learning rate leads to very unpredictable results. Since the new Q-values are calculated from the current best prediction, there is a certain feedback loop in the reward algorithm. This leads to small error getting amplified and causing dips in the learning process. To mitigate this, the actual training process uses two neural nets: the agent neural net, and the target neural net.

Every episode, the agent interacts with its environment but does not learn from the rewards. Instead, this state and reward are stored into a memory. When the actual training takes place, once every 5 episodes, the agent is trained over a batch of 64 state/action/reward sets. However, instead of using its own prediction as input for the training equation, it uses the target net's prediction. The target net is initially set to have the same weights as the agent net, and therefore, the same predictions. After many episodes have elapse (around 50), the weights of the agent net are copied onto the target net. This reduces the possibility of strong variations in the agent net.

Graphics Processing Units contain many simple floating-point arithmetic compute units. These are used normally to work in parallel visual data quickly. However, they can be repurposed for other applications as long as they are compatible with this limited logic. TensorFlow can in fact use CUDA cores in Nvidia's GPU dramatically improving training time. Experimentally, training time was reduced in average by 35% over regular CPU time.

The following figure shows the average reward earned by the agent for every 10-episode period. It also displays the value of $\boldsymbol{\varepsilon}$ at the time. This is a particularly useful metric to evaluate the fitness of the current agent for the environment. Since the two neural nets converge at certain episode counts, there are some visible variations in the shape of the curve when this happens.

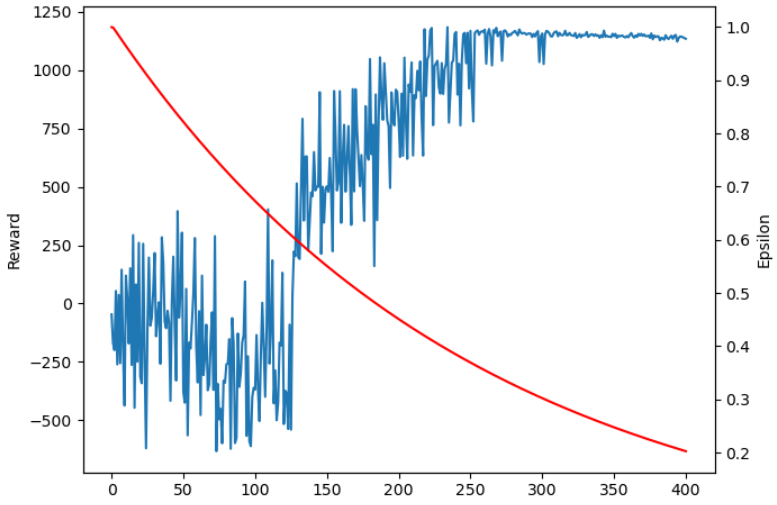


Figure 59: Average reward obtained by agent every 10 episodes and ε

6.10.4 Cloud training and hyperparameter tuning

The training process has multiple tunable settings. With particular importance for the training process are learning rate (α), discount factor (γ), and exploration factor decay (ε_decay). These are called hyperparameters; in contrast with trainable parameters such as node weight. Choosing the right hyperparameters is necessary to have an efficient and effective training. And although an experienced guess could be made, they require tuning by experimentation. [32]

This process is called hyperparameter tuning and it is extremely time consuming. Using the setup described above, simulating 4000 episodes of an easy difficulty problem took approximatively 40 minutes. This is well within feasible range. However, running the following code would take over 20 days of computing time in the previous configuration.

```
# Train for each of the following hyperparameters and save the score
for α in [0.8, 0.5, 0.2, 0.1, 0.08, 0.05, 0.02, 0.01, 0.008, 0.005, 0.001]:
      for γ in [1, 0.99, 0.96, 0.93, 0.9, 0.85, 0.8, 0.7, 0.5, 0.3]:
            for ε_decay in [0.9999, 0.9998, 0.9996, 0.9994, 0.9992, 0.999, 0.995]:

                  score = train_agent(α, γ, ε_decay, episodes = 4000)
                  save(score)
```

This a common problem to all machine learning projects. There are some techniques to limit the number of training sequences required. Instead of the exhaustive sweep described above, using gradient-base optimization can significantly reduce the training time. Even random search is typically faster than grid search.

Since all the training sequences are independent, this process can easily be parallelized. Running each sequence in a different processing unit would reduce the search time to just a single training sequence period. However, machines with this type of computing power are extremely costly to operate. For this reason, in recent years cloud based hyperparameter tuning has seen an increase in popularity. Google Cloud Platform offers a hyperparameter tuning service with proprietary optimization algorithms [15]. In this project however an exhaustive search with a temporary cluster is utilized.

Amazon Web Services (AWS) offers instances priced according to current datacenter load. This service is called Spot Instances and normally carries a cost reduction of around 80%. These instances can be programmatically requested and destroyed using BOTO, a Python interface for AWS.

To train the final neural net, a master script was created that requests 10 ‘t2.large’ AWS spot instances. The instances are launched with a previously created disk image. The image runs in Ubuntu Server 16.04 and automatically starts a Python slave script. The master can then send a TCP message to the slave machines containing information regarding the hyperparameter values to use. When the training process is done, the slave sends a report containing information regarding the final score of the neural net. The master will continuously assign work to slaves until no more hyperparameter combinations are left. At this point, the master kills all the slaves that are finished and cancels the spot request [Figure 60].

[15] More information about GCP’s HPP: https://cloud.google.com/blog/products/gcp/hyperparameter-tuning-on-google-cloud-platform-is-now-faster-and-smarter

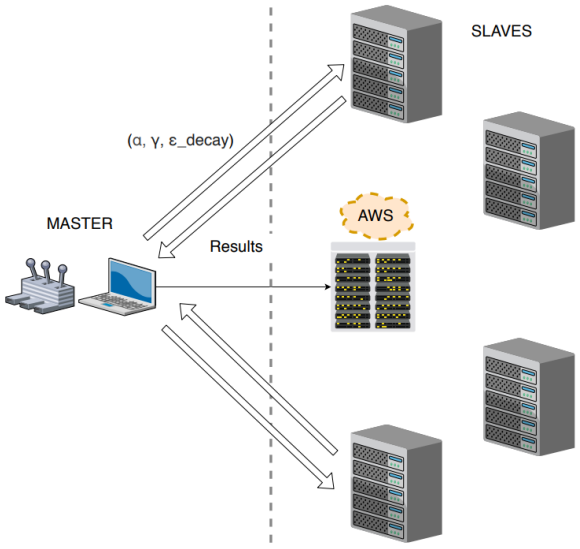


Figure 60: Network structure for cloud hyperparameter optimization

T2 instances, as most other types of virtual machines offered by cloud providers, are purely CPU based, therefore, the process will be slower as the CUDA processors used for the single training sequence are no longer available. This entire process took slightly over a day to run for a set of 336 hyperparameter combinations. The metric used to measure the fitness of each combination was the average reward over 50 episodes without random actions ($\varepsilon = 0$). This optimum was found at for the combination:

$$\varepsilon\_decay = 0.9985 \qquad \gamma = 0.98 \qquad \alpha = 0.02$$

7. Results and Analysis

7.1 Hardware Results

7.1.1 Main Board

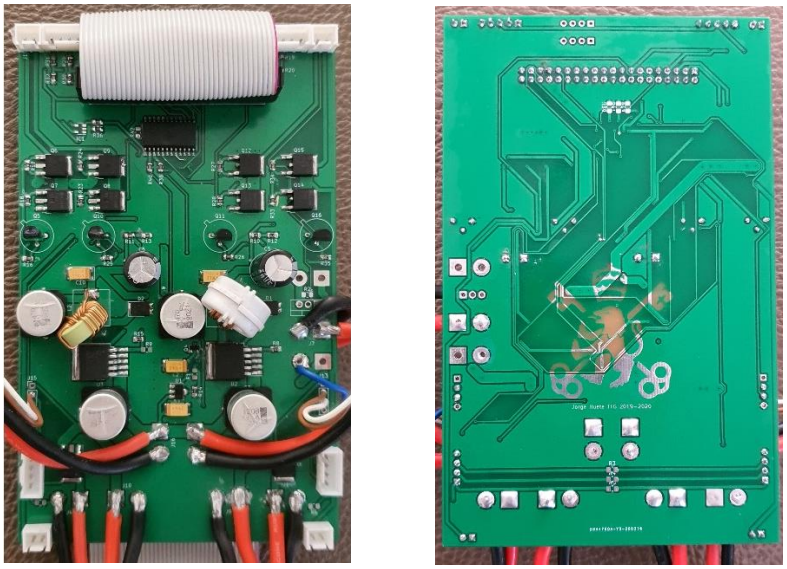


Figure 61: Main PCB. Front (left) and back (right)

7.1.2 Step-down converters

This section describes the capabilities and characteristics of the board's power supplies. The metrics used to benchmark and characterize these subsystems will be voltage output, ripple voltage, efficiency, and thermal performance. These are common metrics used to compare available commercial power supplies and therefore, useful to evaluate the final product.

Output voltage

The nominal output voltage is set by a resistor divider feedback loop. Due to noise and relatively high impedance of the cables connecting the 5V power supply, this voltage is set to 5.6V. This value is still well within the Raspberry Pi's input voltage range and helps prevent brownouts from ripples and drops.

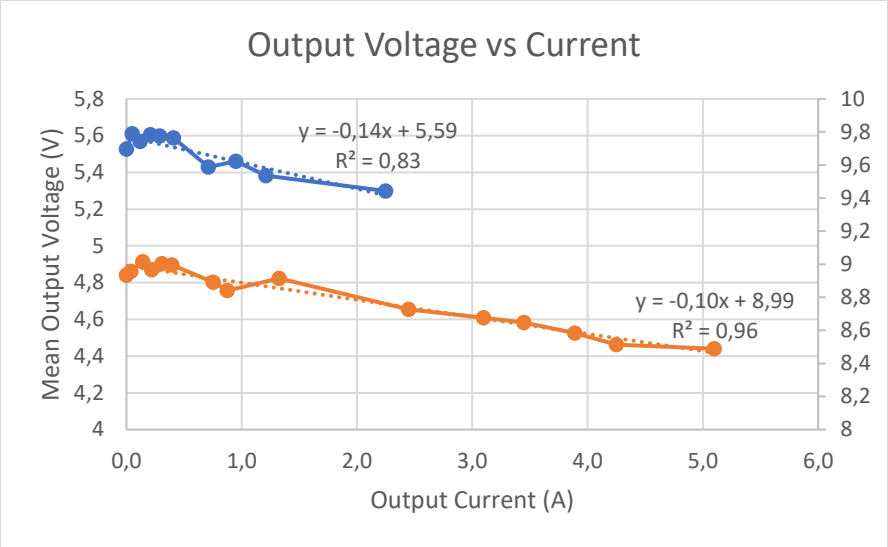


Figure 62: Output voltage vs current of 5V and 9V power supplies

Ripple voltage

As previously mentioned, switch mode power supplies are notoriously noisy, creating large ripples in the output. Even with significant capacitance at the output, both power rails have large AC components that increase with output current.

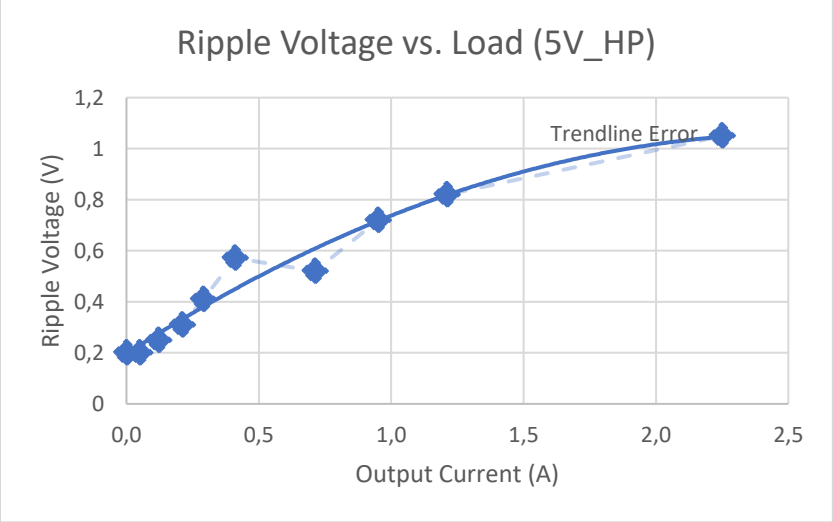


Figure 63: Ripple voltage on bus 5V_HP vs load current

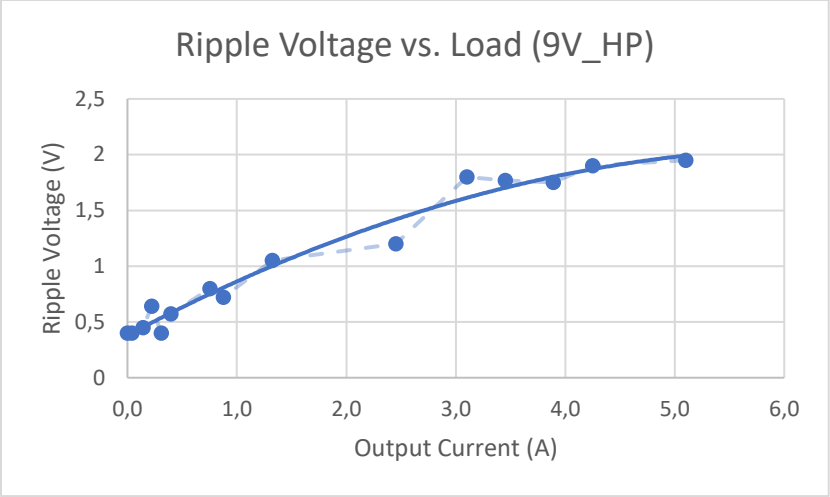


Figure 64:. Ripple voltage on bus 9V_HP vs load current

Ripple voltage seems to be a product of the nominal output voltage as well. The 9V power supply has a remarkably high ripple voltage showing oscillations from 7 to 10 volts at high currents. However, this ripple seems to be noticeably attenuated with the motor load compared to a similar resistive load.

The ripple voltage at high loads on the 5V bus might seem daunting or dangerous at first. It is important to note however, that the nominal load on this bus during operation is only 400mA, corresponding to about 500mV of ripple. Furthermore, the delicate electronics, that is the CPU, is fed through a linear regulator that significantly filters out this component.

Efficiency and thermals

Inefficiencies in the step-down converter are mostly located in the controller IC and inductor. Thermal data not only locates these inefficiencies but also gives information regarding bottlenecks. To test general performance, the board is used to generate a total of 5, 15, and 35W for periods of 10m. The temperature after the test was then measured. Between each test there was a significant cooldown period to allow component to reach ambient temperature. The following table shows the results.

Total Power	5V power	9V power	5V inductor temperature	5V converter IC temperature	9V inductor temperature	9V converter IC temperature
5W	1W	4W	24.2ºC	23.5ºC	24.5ºC	23.9ºC
15W	3.5W	11.5W	39.1ºC	26.1ºC	42.0ºC	29.5ºCº
35W	8W	27W	75.1ºC	34.0ºC	88.2ºC	35.2C

7.1.3 Battery Life

The following table shows the battery current draw and battery life at the respective power states. Battery life is determined by the time it takes the system to enter undervoltage protection (i.e. battery backup) from complete charge.

Power State	Battery Current (@12.0V)	Battery Life
Quiescent: all power supplies disabled (no CPU, no sensors, no motors)	55mA	213h*
Low-power state: 9V power supply (motors) and sensor disabled	250mA	47h
Stand-by: all subsystem enabled, motor halted	450mA	26.5h
In motion: motors powered forward direction	1.9A**	4.5h 4km*

* Estimated

** Fluctuates

The battery life times are dependent on the battery capacity. Higher capacity batteries could be used to extend these times if necessary. However, the system has enough capacity for long periods of mapping and training.

7.2 Software Results

7.2.1 Maneuver Replicability

Since the algorithms defined in 6.3, 6.4, and 6.5 are all responsible for the precision of movement, determining the replicability of movement is a valid benchmark for all of them. The simplest way to do this is to execute the same maneuver multiple times in one direction and again in the opposite. If the maneuvering process were perfect, the final position would be the same as the starting one. The following tests use as a comparison the relative position in space as well and measurements.

Linear maneuvers

The linear replicability test consists of 10 iterations of 2m movements. The robot will attempt to move and correct 2m forwards and 2m backwards for a total of 5 cycles.

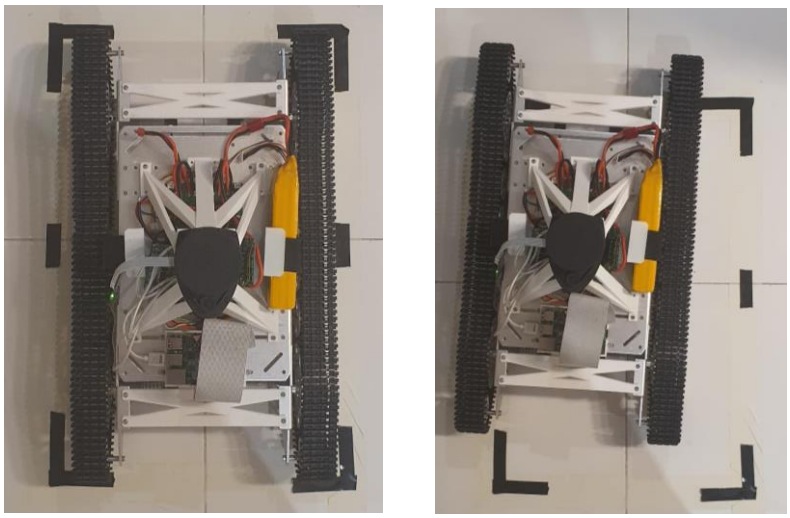


Figure 65: Linear maneuver replicability test. Starting condition (left) and final position (right)

Figure 65 shows the results of this test using a 10mm resolution division for the correction algorithm. The measured angular error at the final position was 3º and the total translation was 15cm. This implies an average error of 0.3º and 1.5cm per maneuver. However, the direction of this error is likely to be dependent of the direction of travel. Since the expected resting position is the same as the starting position, the error is likely higher if the maneuvers are in a similar direction.

Rotation maneuvers

The rotation replicability test consists of 10 iteration of 180º turns. The robot will attempt to turn and correct 180º clockwise and 180º counterclockwise for a total of 5 cycles.

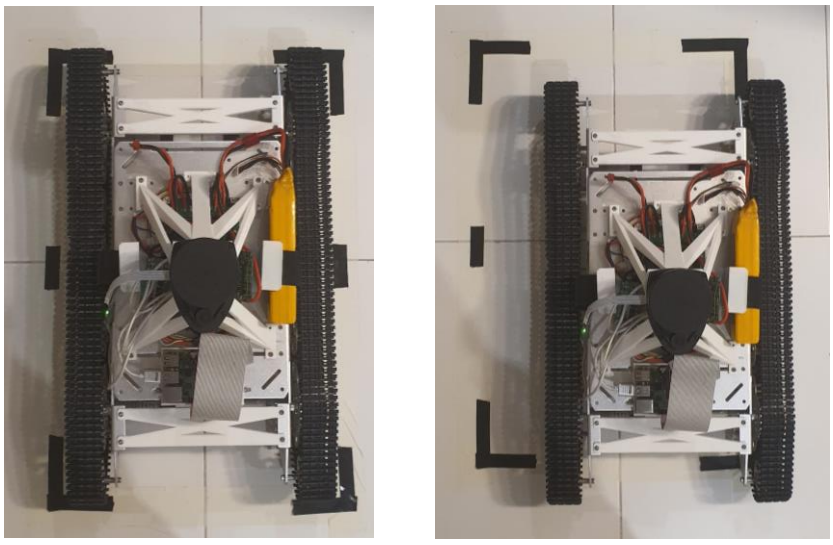


Figure 66: Rotation maneuver replicability test. Starting condition (left) and final position (right)

Figure 66 shows the results of this test using the same 10mm resolution for the correction algorithm. The angular error was negligible, and the total translation was 12cm. The lack of angular error is reproducible, nearly no error of this type is committed during a rotation. The translation error seems to be independent from direction and originate in small differences between the distribution of the tracks' movement.

These tests show where most of the error in the data is likely generated. The mapping and navigation algorithms heavily rely on the accuracy of these maneuvers. In this critical step, all the inaccuracies caused by missed steps, sliding, and unexpected movements are inherited by the final map.

7.2.2 Semiautonomous Navigation

To test the semiautonomous navigation, the robot will be placed in a relatively clutter free environment, and a total of two waypoints will be assigned. These two waypoints were chosen in such a way that the robot will have to backtrack significantly from the first one to the second one. The objective of forcing the robot to revisit the same rooms again is to test the reputation system to showcase the updates.

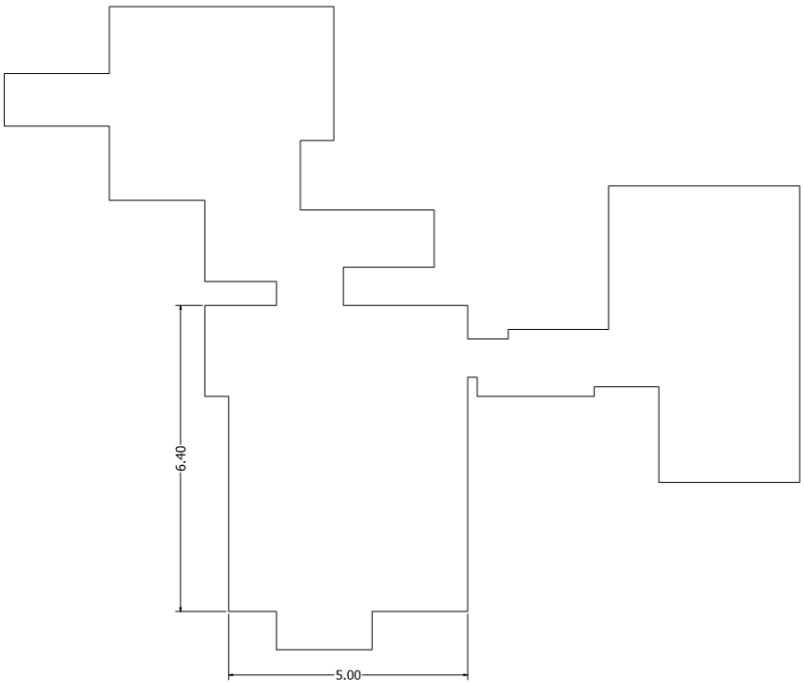


Figure 67: Reference floor map of the environment

Figure 67 shows the floor plan of the environment the robot is placed in for this test. The system is then given two waypoints. First the upper left corridor and then the center of right room. This will make the robot traverse the main room twice, forcing it to update significant part of its internal map.

Figure 68 shows the result until the first waypoint in chronological order. The darkness of the walls is proportional to the wall's current reputation. The transitions between **a** and **b,** and **c** and **d** illustrate how the path is recalculated when a new wall is found interfering with the current plan.

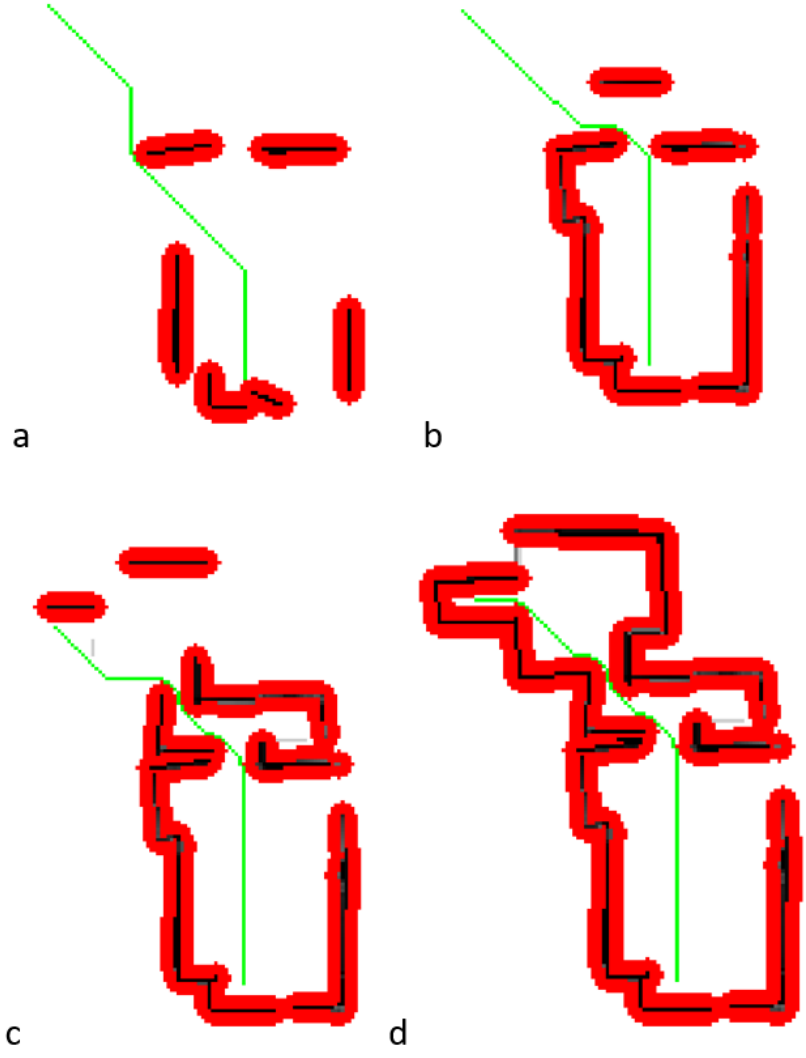


Figure 68: Semiautonomous discovery until first waypoint. Chronological order

Subfigure **d** seems to show a slight rotation of the identified walls compared to the starting room. This is likely caused by accumulated error in the position tracking. For the next portion of the test, a second waypoint is given to the robot.

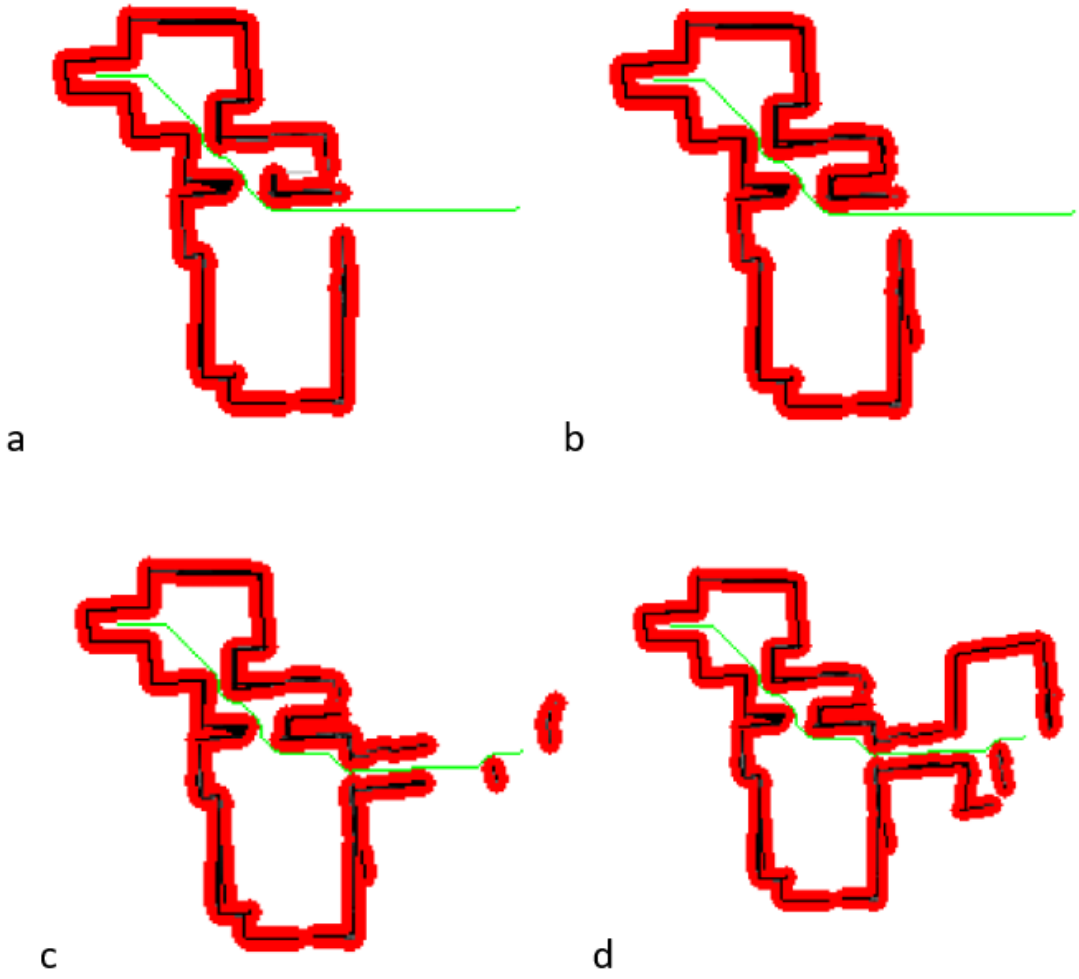


Figure 69: Semiautonomous discovery until second waypoint. Chronological order

The map updates towards the second waypoint show increasing rotational error [Figure 69]. Subfigures **b** and **c** show oblique lines over previously discovered walls. Some of these walls are correctly updated while others get obscured behind the new ones.

By the time the robot reaches the last room (subfigure d), the error is significant. The last corridor and room are noticeably tilted when compared to the rest of the map. This was likely caused by a short section of neural net navigation around the corridor door.

7.2.3 Neural Network Navigation Scenarios

This last section of benchmarking attempts to measure in greater detail the capabilities of the neural net obstacle avoidance. As previously mentioned, this system was already utilized in certain parts if the previous test. However, these were noticeably short and unchallenging sections of the environment. The following scenarios places obstacles directly in the way of the robot creating a challenging situation.

Scenario 1: Simple obstacle

For this first scenario, a single obstacle is placed in front of the robot and a waypoint is placed on the opposite side. The system should enter obstacle avoidance mode when approaching the obstacle and exit it after it has been cleared. The following figure shows the test environment.

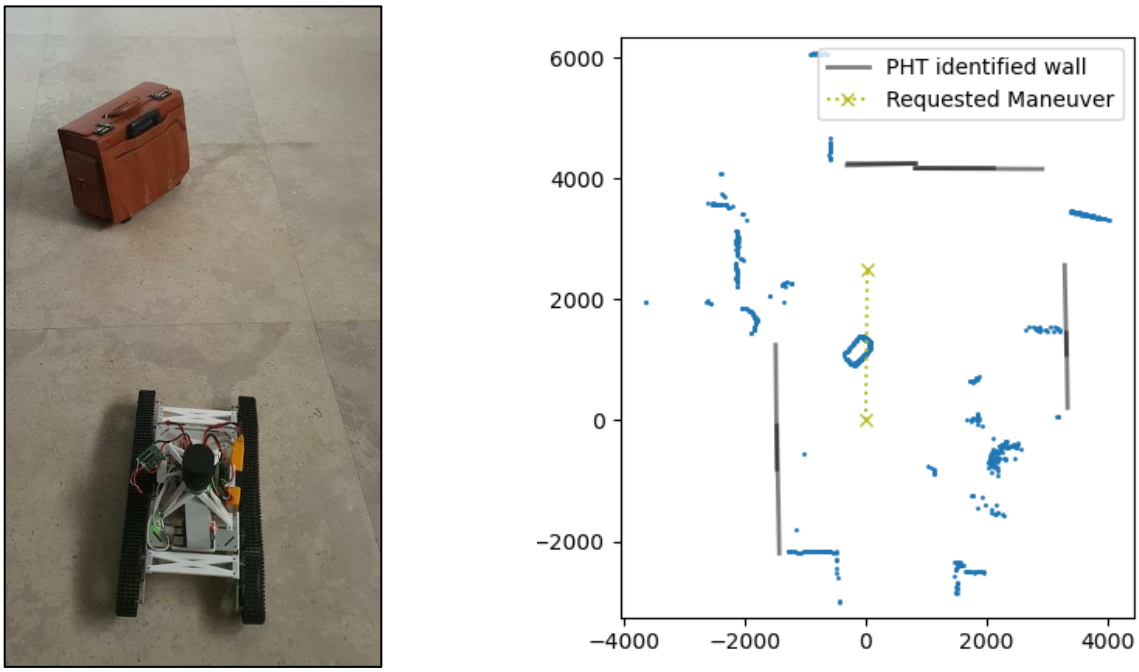


Figure 70: Simple neural net test environment. Picture (left) and scanned (right)

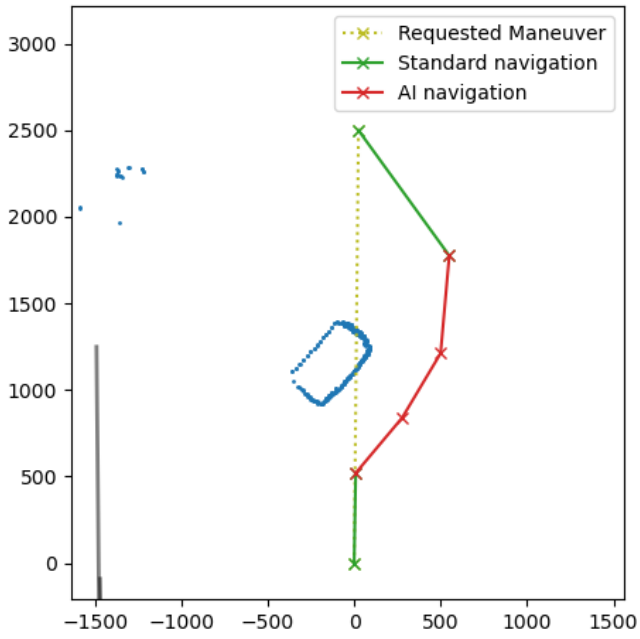


Figure 71: Path to waypoint as a combination of standard A and neural net navigations (detail)*

Figure 71 shows the result of this scenario. The robot initially attempts to reach the waypoint through the obstacle. Once the obstacle is close enough, the neural net navigation starts and rotates towards the right to avoid the obstacle. After a total of 7 predictions, the robot is far enough from the obstacle and neural net navigation is discontinued. The robot recalculates a maneuver to reach the waypoint.

Over the entire scenario, the system accumulated a positioning error of 4cm and 1º. This error is much higher than the expected error accumulated by standard navigation. This leads to the conclusion that neural net obstacle avoidance generates more error towards the position tracking algorithm than precise maneuvering.

Scenario 2: Obstacle and wall interactions

For this scenario, the robot is placed near a wall. A waypoint is placed in front of the robot in such a way that the calculated route will lead it along the wall. An obstacle is placed near the intended route but far enough to the wall to create a gap large enough to allow passage without risk. However, the obstacle is close enough as to surely engage the obstacle avoidance system. For this scenario, the wall safety boundaries have been drastically reduced.

There are two realistic solution to this problem, taking a short route between wall and obstacle and taking a longer route around the obstacle. This test is meant to determine how well the results of medium difficulty training translate to real world navigation.

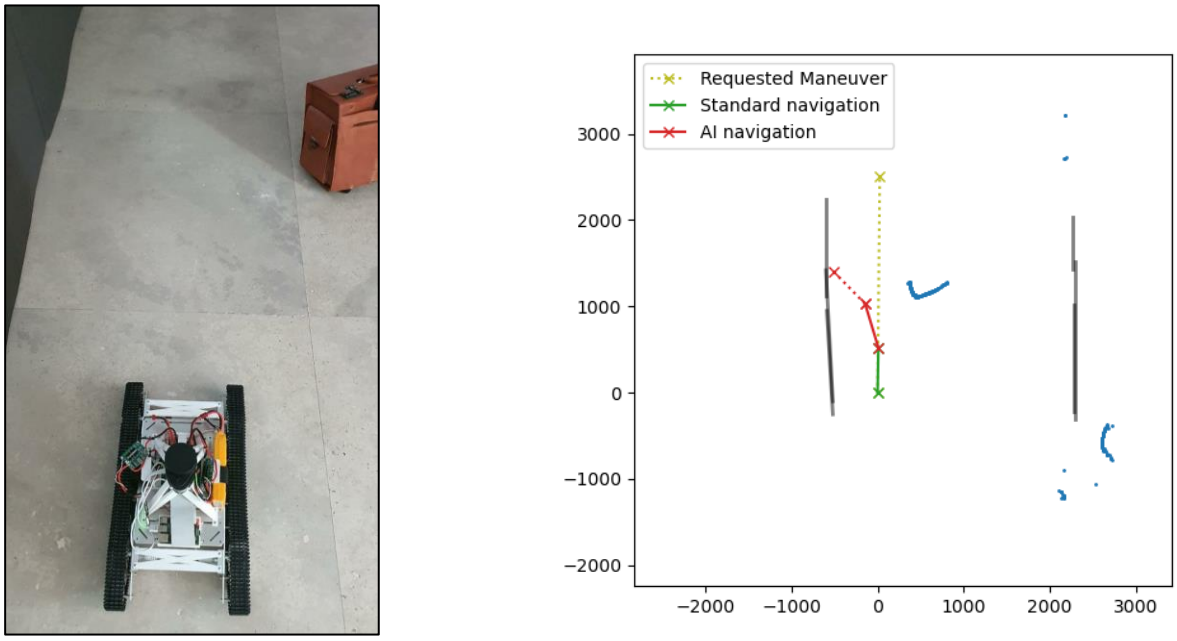


Figure 72: Wall interaction test. Picture (left) and result (right)

Figure 72 shows one of the two obtained results for this test. In this example, the neural net attempts to distance itself from the obstacle. In doing so, it turns towards the walls and advances. Another common result was a rotation loop were the neural net predicted turns towards the wall and obstacle without moving forwards.

These results are similar to experiences during autonomous navigation testing. Simple obstacles or even two obstacles without nearby walls result in a correct navigation towards the waypoint. However, more complex problems, like those involving walls lead to unsatisfactory results.

8. Conclusion and Areas of Improvement

Regarding the hardware requirements of the system, the result is mostly, although not completely, satisfactory. The main board accomplishes all the specified requirements with a high efficiency and low part count and cost. The principal requirement for the main board was power and motor management. Both switch-mode power supplies operate at suitable efficiencies (80%-85%) from batteries to load. These subsystems also have ample safety margins, operating at their high-performance middle range but still capable of delivering double the nominal power output. While difficult to measure the efficiency of the H-bridges, the fact that they can deliver around 30W without notable increase in temperature indicates that they are also highly efficient.

The internal voltage monitoring system is also satisfactory. Although each individual channel required manual calibration, it can operate at exceedingly high rates and precision. The system can effectively recognize undervoltages and take actions accordingly. However, battery and bus currents are not measured. This information would be extremely useful not only for fault detection, but also to predict the remaining battery life. It might be theoretically possible to extrapolate this from cell battery voltage drop, however, a shunt plus instrumental amplifier would be more desirable and accurate.

While in a low power state, the system has an impressive battery life. However, this could still be significantly improved. Pull resistors for logic signals and inverters draw a significant amount of current, collaborating to an important part of the board's quiescent power. Many improvements could be made to increase low power efficiency. An external microcontroller on board could be used to shut down and wake the main computer board on a timer. This would enable the board to stay asleep over a week on a single charge; likely two with little improvement to the aforementioned logic circuits.

The largest area for improvement on the main board is movement reliability. The H-bridge and motor position tracking are both handled by the CPU based on software interrupts. Moreover, the H-bridges are incapable of shorting both terminals on the motors together. This would give the robot the option to create a braking torque and reduce maneuver error. Adding a dedicated microcontroller on-board would allow it to both precisely track and brake the motors. This microcontroller could also be used to create the aforementioned ultra-low power state.

The platform assembly is most adequate. After some modifications, the tracks run true and very stable. The single discernible issue is the lack of shock absorption or damper. There is no possible motion between

ground and body of the robot, therefore, all ground irregularities are transmitted to the joints and structure. Although this is not an issue for indoor floors, it would greatly improve the flexibility of the system to encompass irregular terrain such as outdoors.

As with all hardware development, multiple versions and repairs of some components were necessary at some point. The modularity and simple manufacturing of parts greatly aided this. In particular, being able to separate the main LIDAR assembly from the rest of the robot without dismantling any other component showcased the importance of modularity.

The LIDAR driver is likely the most accomplished and polished piece of software in the system. It is almost able of utilizing the full set of sensor capabilities. As such, it will be continued as a standalone open source project to enable makers to use RPLIDAR's sensors with Python projects requiring the high sample rates they offer.

Navigation is significantly held back by issues of movement reliability. Although the system is mostly aware of error committed when moving, it is incapable of correcting it completely. This means that every time the robot moves, it accumulates more and more position error. This makes results blurry and tilted over time. This the most evident area where improvement can be achieved.

The neural net navigation used to avoid obstacles in the way is significantly more capable in the simulated environment than in real life scenarios. This might be mitigated by a closer approximation of the behavior of the robot in the simulation or simply a more complex net structure.

9. References

[1] J. Bongard, V. Zykov and H. Lipson, "Resilient Machines Through Continuous Self-Modeling," *Science,* vol. 314, no. 5802, 2006.

[2] G. Khan, A. Villaflor, B. Ding, P. Abbeel and S. Levine, "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation," 2017.

[3] H. Surmann, C. Jestel, R. Marchel, F. Mesberg, H. Elhadj and M. Ardani, "Deep Reinforcement learning for real autonomous mobile robot navigation in indoor environments," *Cornell University,* 2020.

[4] S. Bhattacharjee, K. Kabara and R. Jain, "Autonomous drifting RC car with reinforcement learning," *University of Hong Kong,* 2018.

[5] N. Kohl and P. Stone, "Machine Learning for Fast Quadrupedal Locomotion," *Univerisy of Texas,* 2004.

[6] F. Larsson and B.-E. Mellander, "Abuse by External Heating, Overcharge and Short Circuiting of Commercial Lithium-Ion Battery Cells," *IOPScience,* 2014.

[7] R. William, T. Toombs, K. Owyang and H. Yilmax, "Reverse battery protection device containing power MOSFET". US Patent US5517379A, 1993.

[8] H. Maleki and J. Howard, "Effects of overdischarge on performance and thermal stability of a Li-ion cell," *Journal of Power Sources ,* vol. 160, no. 2, 2006.

[9] C. Cooper, "Fundamentals of Buck Converter Efficiency," *Electronic Design,* 2013.

[10] G. Lakkas, "MOSFET power losses and how they," *Analog Applications Journal,* 2016.

[11] K. Kundert, "Power Supply Noise Reduction," 2019.

[12] D. Lancaster, TTL Cookbook, ISBN-13: 9780672210358, 1975.

[13] SLAMTECH, "Slamtec RPLIDAR Public SDK for C++," 2019. [Online]. Available: https://github.com/slamtec/rplidar_sdk. [Accessed 06 2020].

[14] SLAMTECH, "Slamware C++ SDK," 2019. [Online]. Available: https://developer.slamtec.com/. [Accessed 06 2020].

[15] SkoltechRobotics, "RPLidar Python Module," *https://github.com/SkoltechRobotics/rplidar,* 2017.

[16] SLAMTECH, "RPLIDAR - 360 Degree Laser Range Scanner - Interface Protocol and Application Notes," 2019. [Online]. Available: http://bucket.download.slamtec.com/ccb3c2fc1e66bb00bd4370e208b670217c8b55fa/LR001_SLAMTEC_rplidar_protocol_v2.1_en.pdf. [Accessed 06 2020].

[17] 瞰 , "A kind of method and apparatus of data processing for distance measuring sensor". China Patent CN108306649A, 20 07 2018.

[18] T. Wouter, "Python's Wiki Documentation: Global Interpreter Lock," Python, 02 08 2017. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock. [Accessed 06 2020].

[19] K. Craig, "Optical Encoders," New York University - Engineering, 2018. [Online]. Available: http://engineering.nyu.edu/mechatronics/Control_Lab/Criag/Craig_RPI/SenActinMecha/S&A_Optical_Encoders.pdf. [Accessed 06 2020].

[20] B. Croston, "Raspberry Pi GPIO Documentation: A Python module to control the GPIO on a Raspberry Pi," 02 2015. [Online]. Available: https://sourceforge.net/p/raspberry-gpio-python/code/ci/default/tree/source/. [Accessed 06 2020].

[21] P.-E. Danielsson and M. Hammerin, "High-accuracy rotation of images," *CVGIP: Graphical Models and Image Processing,* vol. 54, no. 4, pp. 340-344, 1992.

[22] R. Webderlich, "Introduction to A* Pathfinding," 09 2011. [Online]. Available: https://www.raywenderlich.com/3016-introduction-to-a-pathfinding. [Accessed 06 2020].

[23] P. Hough, "Method and means for recognizing complex patterns". US Patent US3069654A, 1960.

[24] R. Duda and P. Hart, "Use of the Hough," *Stanford Research Institute - Communications of the ACM,* vol. 15, no. 1, 1972.

[25] H. Kälviäinen, P. Hirvonen, X. Lei and E. Oja, "Probabilistic and non-probabilistic Hough transforms: overview and comparisons," *Image and Vision Computing,* vol. 13, no. 4, pp. 239-252, 1995.

[26] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, "A Density-Based Algorithm for Discovering Clusters," *Institute for Computer Science, University of Munich,* 1996.

[27] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal,* vol. 4, no. 1, 1965.

[28] C. A. Rosen, "Application of Intelligent Automata to Reconnaissance," *SRI's Artificial Intelligence Center ,* 1967.

[29] F. Duchon, A. Babinec, M. Kajan, P. Beno, M. Florek, T. Fico and L. Jurisica, "Path Planning with Modified a Star Algorithm for a Mobile Robot," *Procedia Engineering,* vol. 96, pp. 59-69, 2014.

[30] T. Matiisen, "Demystifying deep reinforcement learning," *University of Tartu - Computational Neuroscience Lab,* 2019.

[31] M. J. Mataric, "Reward Functions for Accelerated Learning," *Machine Learning Proceedings,* pp. 181-189, 1994.

[32] S. Agrawal, "Hyperparameters in Deep Learning," 02 2019. [Online]. Available: https://towardsdatascience.com/hyperparameters-in-deep-learning-927f7b2084dd. [Accessed 06 2020].

[33] "Sustainable Development Goals: Goal 12: Ensure sustainable consumption and production patterns," United Nations, [Online]. Available: https://www.un.org/sustainabledevelopment/sustainable-consumption-production/. [Accessed 06 2020].

[34] G. Khan, A. Villaflor, B. Ding, P. Abbeel and S. Levine, "Self-supervised Deep Reinforcement Learning with Generalized Computation Graphs for Robot Navigation," *Cornell University,* 2017.

10. Alignment with the UN's Sustainable Development Goals (SDGs)

10.1 [Primary] Goal 12: Ensure sustainable consumption and production patterns

Goal: 12.2 By 2030, achieve the sustainable management and efficient use of natural resources

Goal 12 of the UN's SGDs attempts to merge Earth's limited resources and human lifestyle. According to the UN, a projected population of 9.6 billion would require three earthlike planets to sustain current lifestyle. [33] Given that populations are still on the rise an effort for more efficient manufacturing and a less wasteful consumption is necessary.

While sustainable consumption can be achieved by a more responsible lifestyle, sustainable production requires new techniques and materials to ensure products' lifecycles, from manufacturing to dismantling, are as efficient as possible. This project illustrates one of those techniques: modularity. Broadly speaking, a system is considered modular when its components can be separated and recombined.

Modularity has many benefits for the product, for example it increases flexibility and upgradability. Whenever a component has surpassed its usefulness, it can simply be replaced by another, more suitable one. This component is still functional and can be reused for other purposes. However, the most relevant situation for modularity is perhaps failure. In engineering, component failure is an inescapable reality. Unfortunately, it is often the case that a single component failure leads to an entire system being considered unsalvageable. This is due to the high cost or required knowledge to replace the faulty component. A good example of this issue would be laptop computer repair. An electrical failure on the motherboard of a modern laptop requires extensive knowledge to diagnose and repair, often leading to a complete replacement of the unit. Modular design, on the other hand, only requires replacement of the damaged component. With functional diagnosis system, a user could potentially fix the issue without discarding all the other functional components.

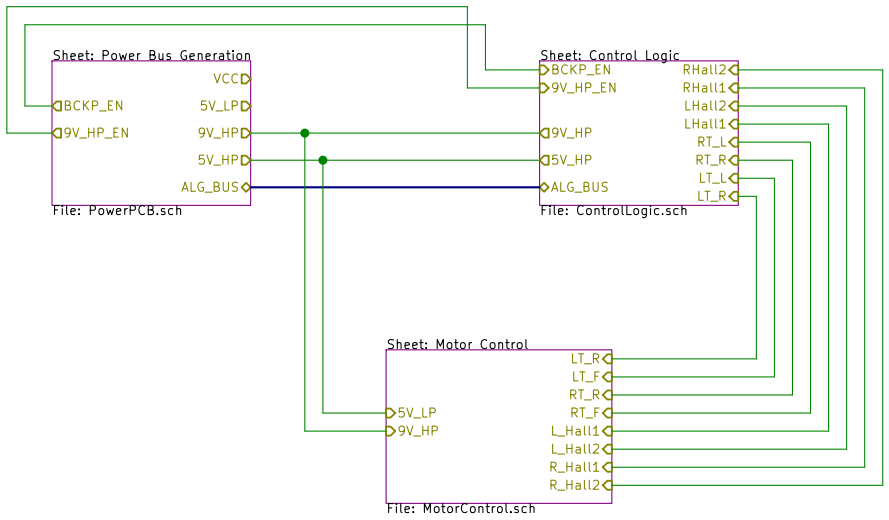
Modularity has been one of the priorities during the hardware development cycle. In total, 12 different parts were designed and fabricated to assemble the final robot (not including the metal base and tracks). Most of these parts are essential for the system and if any of them were to fail, it would become inoperative. They are however easily exchangeable. PCBs are connected by standard cabling and can be swapped in a matter of minutes at most. All the 3D printed components can easily be disassembled

without needing to dismantle the entire assembly. In fact, the biggest part of the system, the LIDAR support, was damaged at one point. By reprinting only that component and replacing it, they entire system was only down for a day.

This part weights approximatively $85g$ and is made from PLA plastic. If all adjacent components were fabricated as a single part, it would weight $215g$. By separating them, this repair saved $130g$ or 60% of the entire weight in plastics.

11. Annex

Project repository: https://github.com/jordi2224/DQNavigator



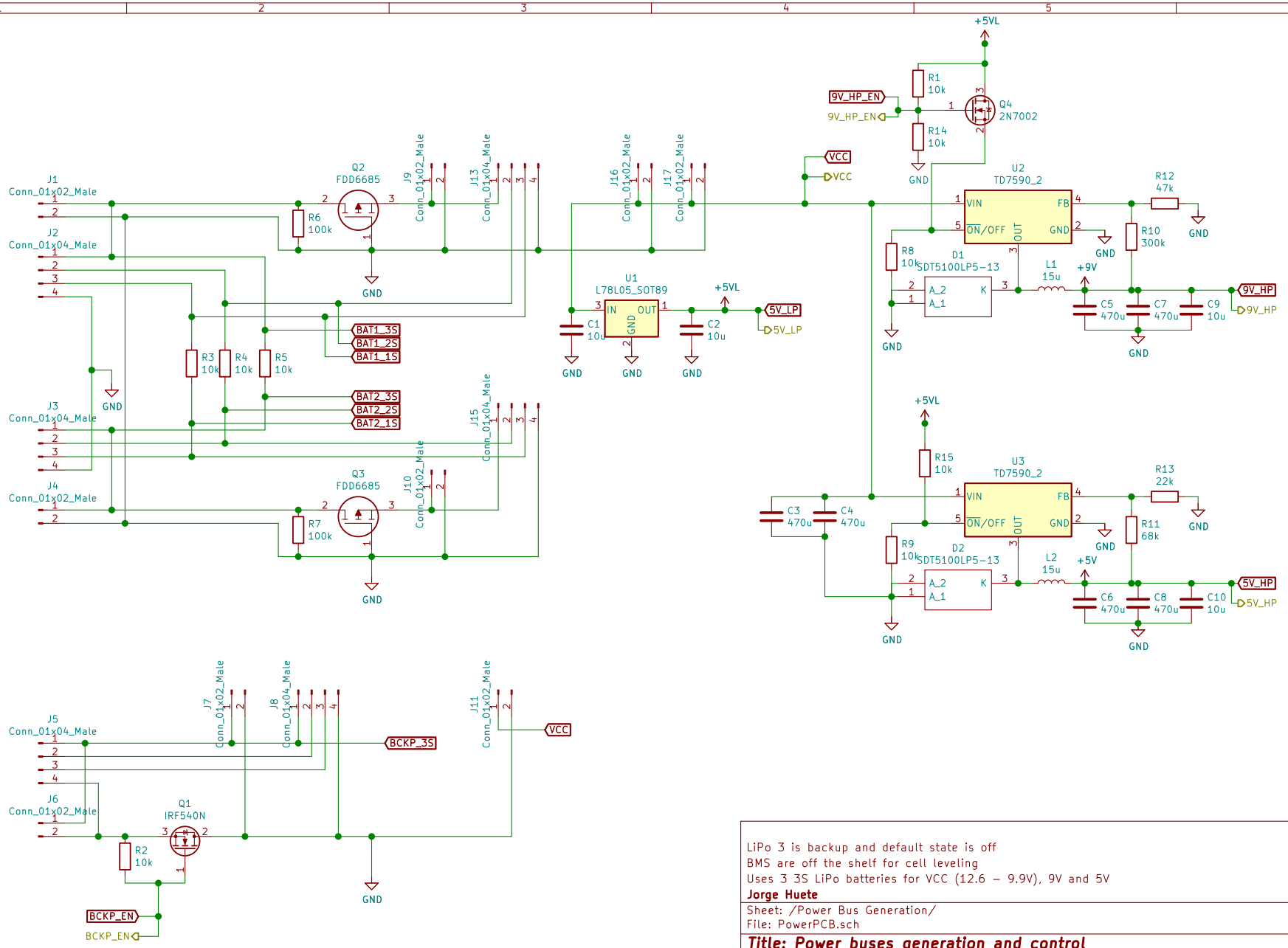
Top level sheet and block diagram

Jorge Huete

Sheet: /
File: TFG_PCB.sch

Title: ROVER Main PCB

Size: A4	Date: 2020-01-24	**Rev: 0.1**
KiCad E.D.A. kicad (5.1.5)-3		Id: 1/4



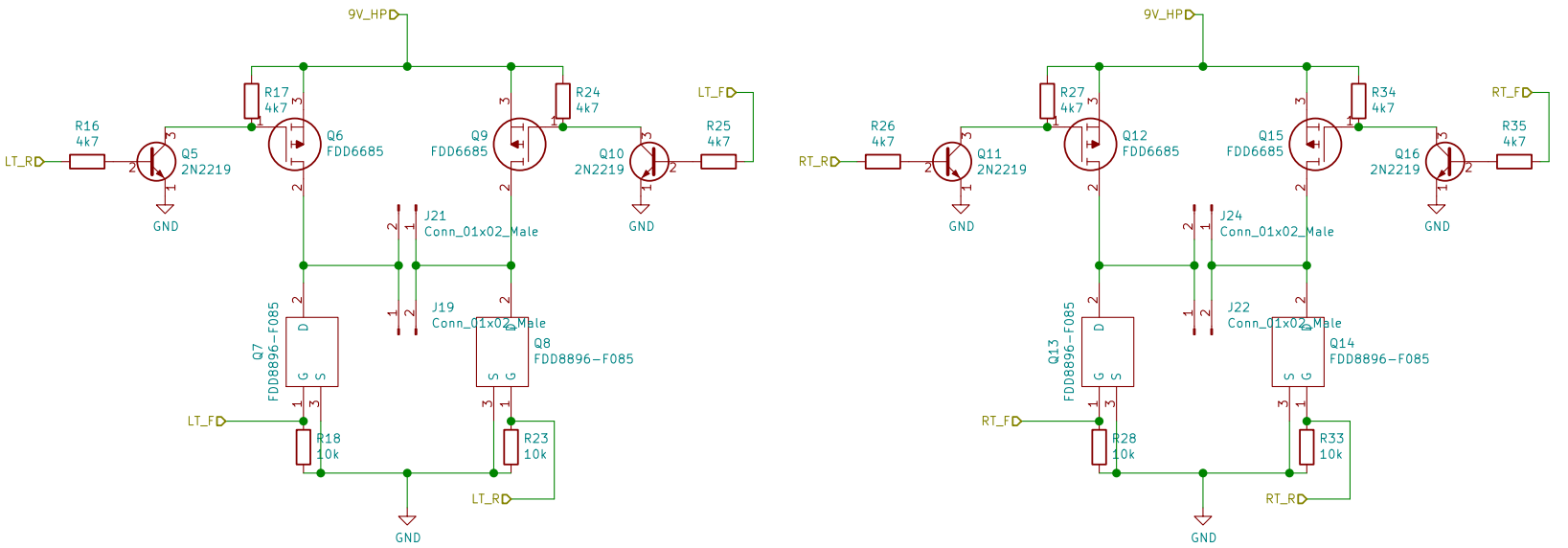
LiPo 3 is backup and default state is off
BMS are off the shelf for cell leveling
Uses 3 3S LiPo batteries for VCC (12.6 – 9.9V), 9V and 5V

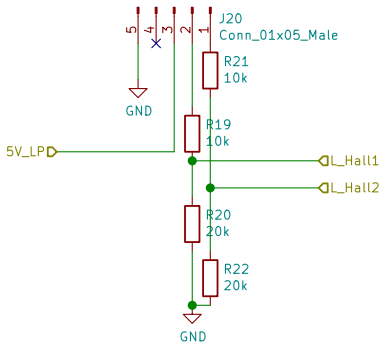
Jorge Huete

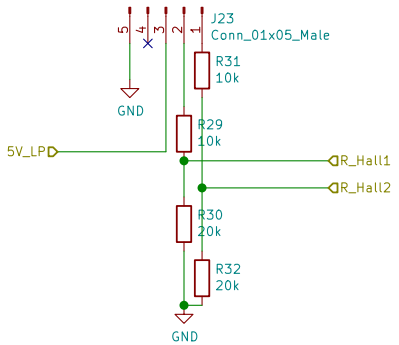
Sheet: /Power Bus Generation/
File: PowerPCB.sch

Title: Power buses generation and control

Size: A4	Date: 2020-01-24	**Rev: 0.1**
KiCad E.D.A. kicad (5.1.5)-3		Id: 2/4







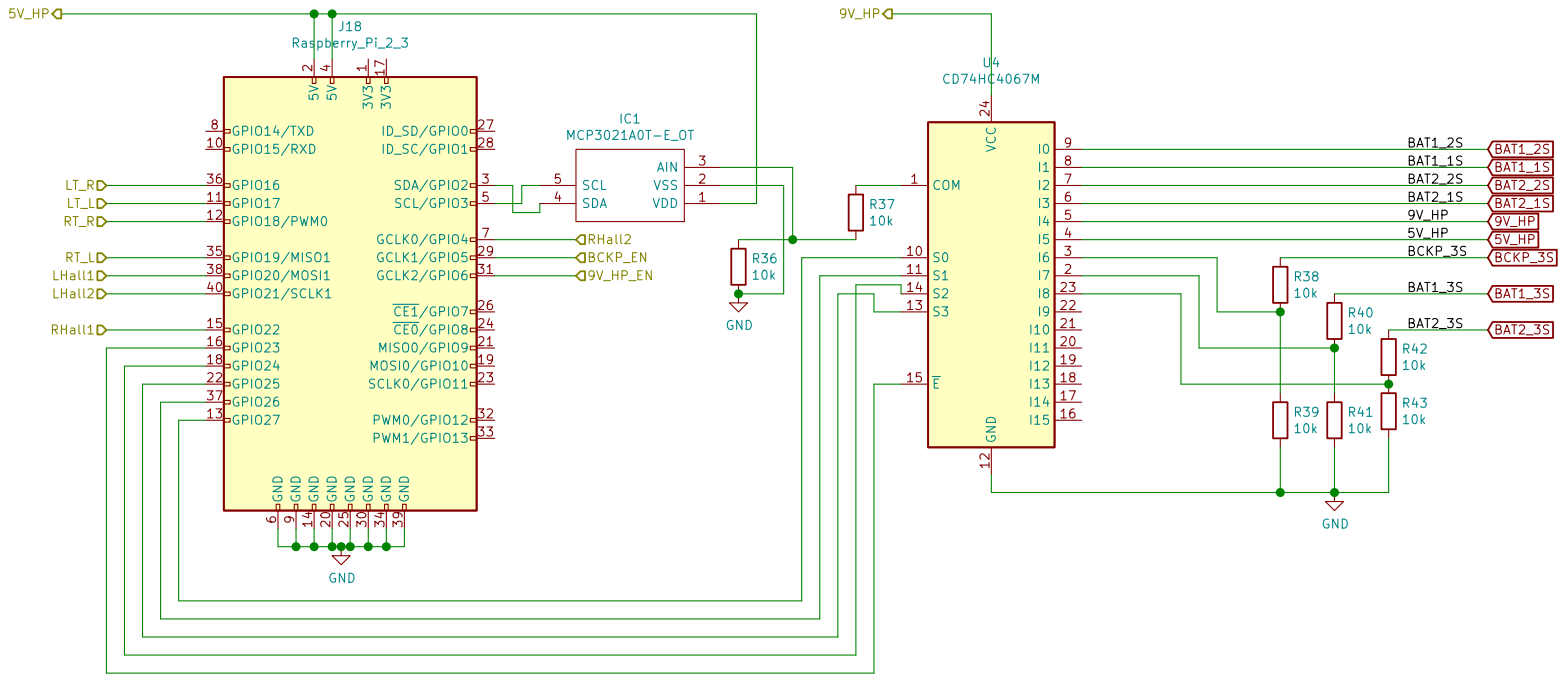
Hall effect sensors are used for rotation encoding
Includes resistor divider for Hall effect sensors
MOSFET H-Bridges for bidirectional motor power

Jorge Huete

Sheet: /Motor Control/
File: MotorControl.sch

Title: Motor Control and Feedback

Size: A4	Date:	**Rev: 1**
KiCad E.D.A. kicad (5.1.5)-3		Id: 3/4



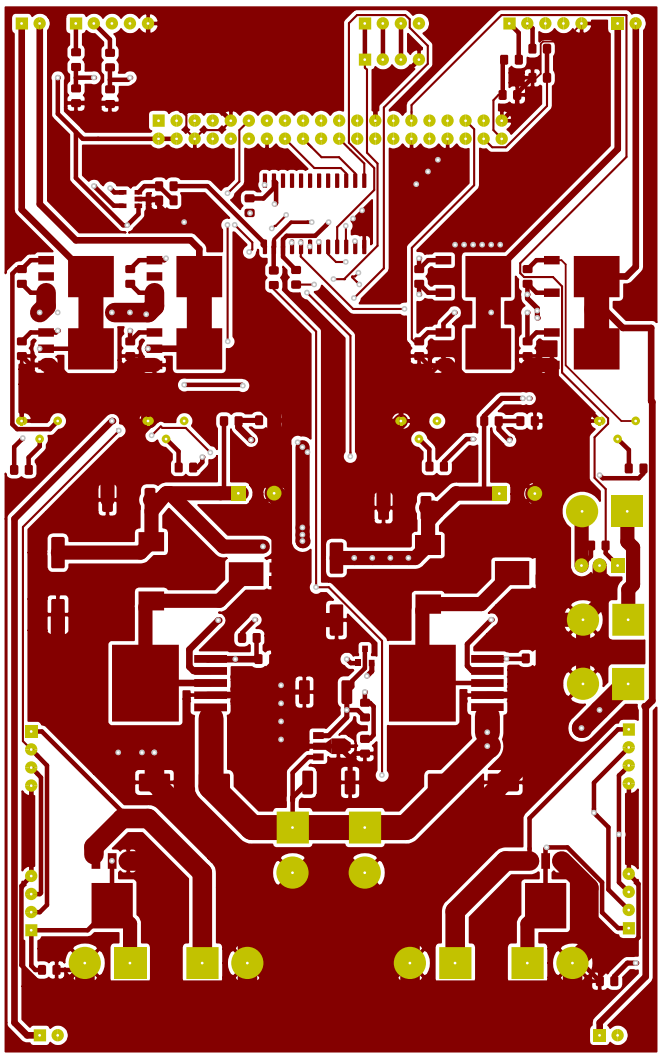
Also includes a 20x2 connector for standard Rapsberry Pi
Analog multiplexer and ADC for monitoring buses

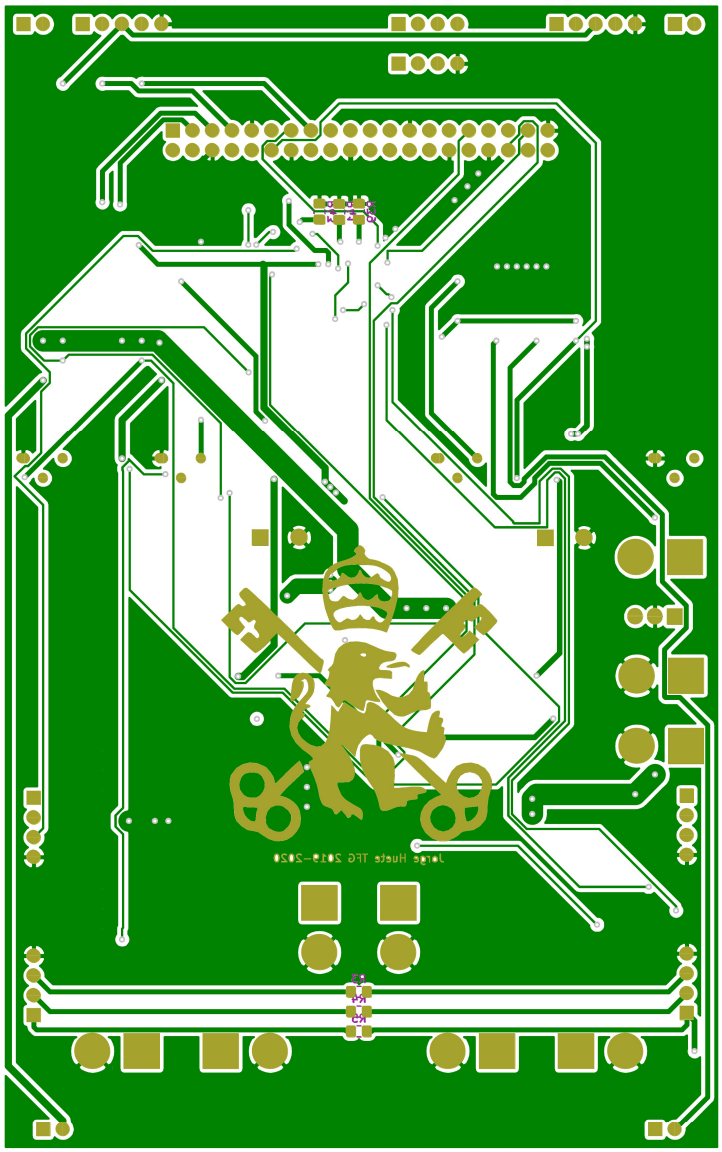
Jorge Huete

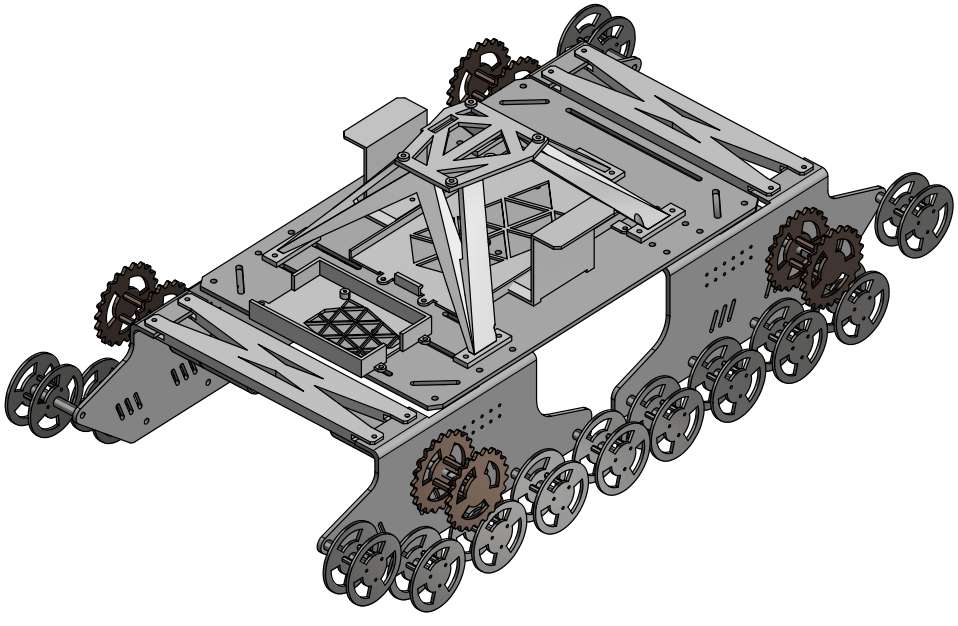
Sheet: /Control Logic/
File: ControlLogic.sch

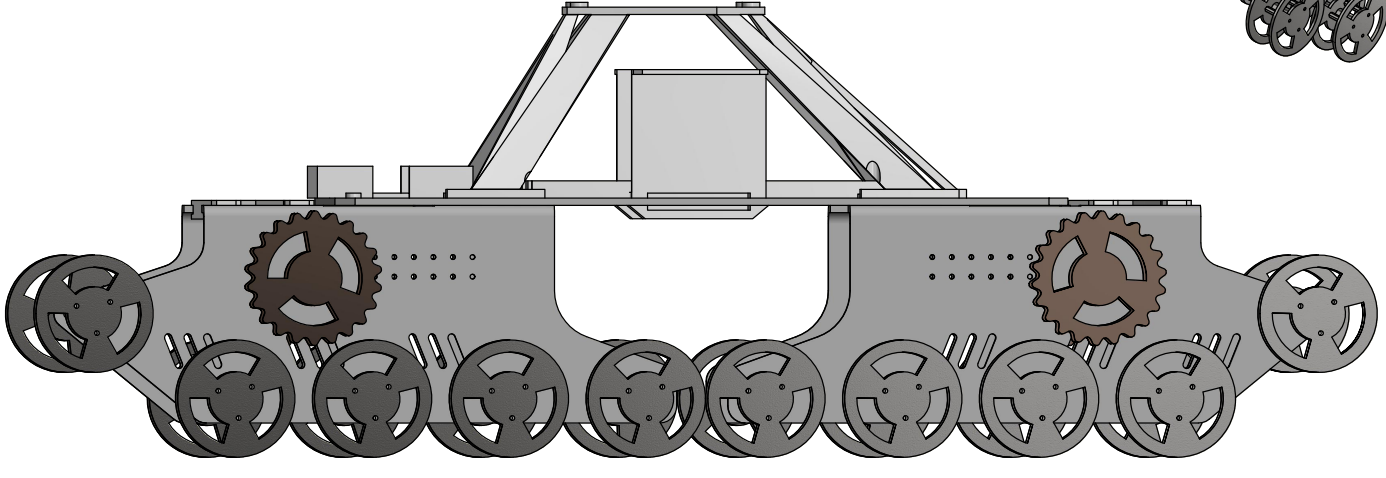
Title: Analog Monitoring and Logic

Size: A4	Date:	**Rev: 1**
KiCad E.D.A. kicad (5.1.5)-3		Id: 4/4

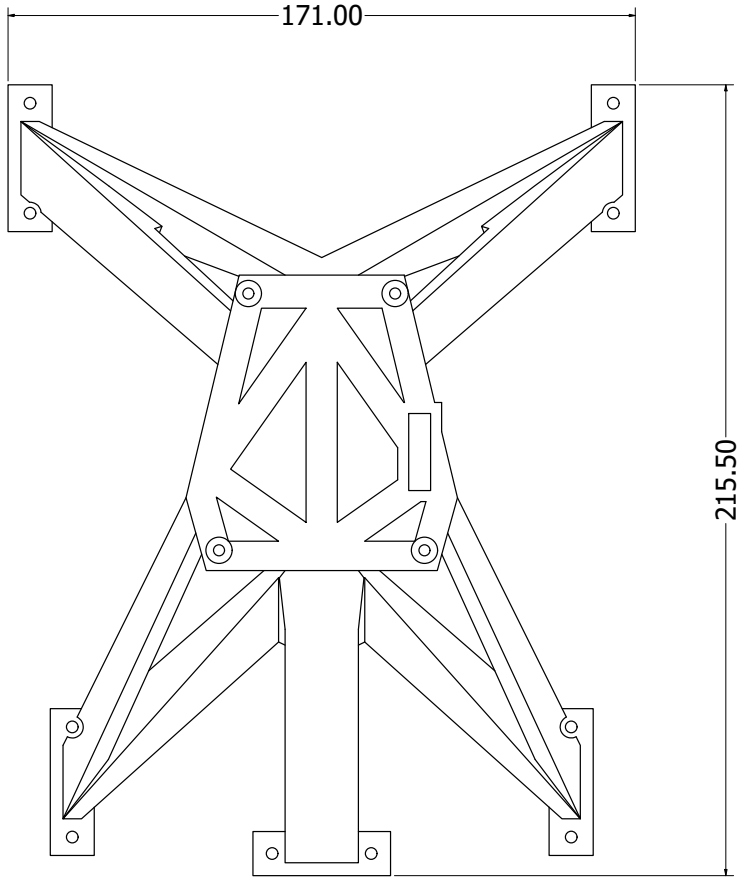


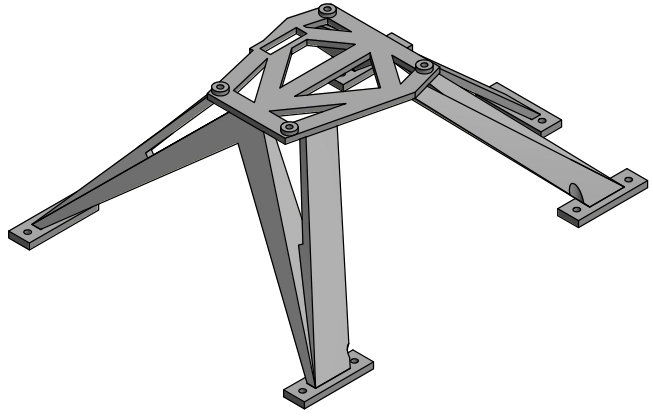


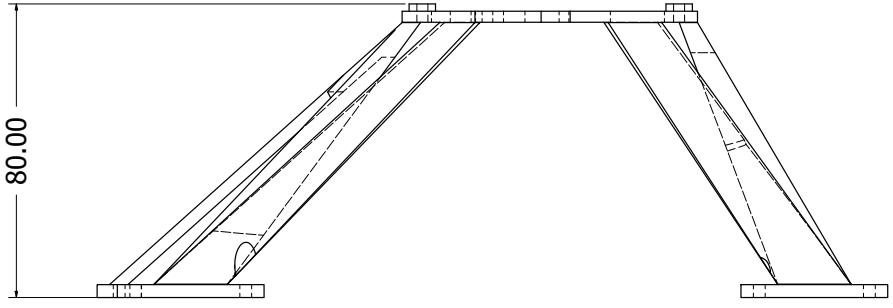




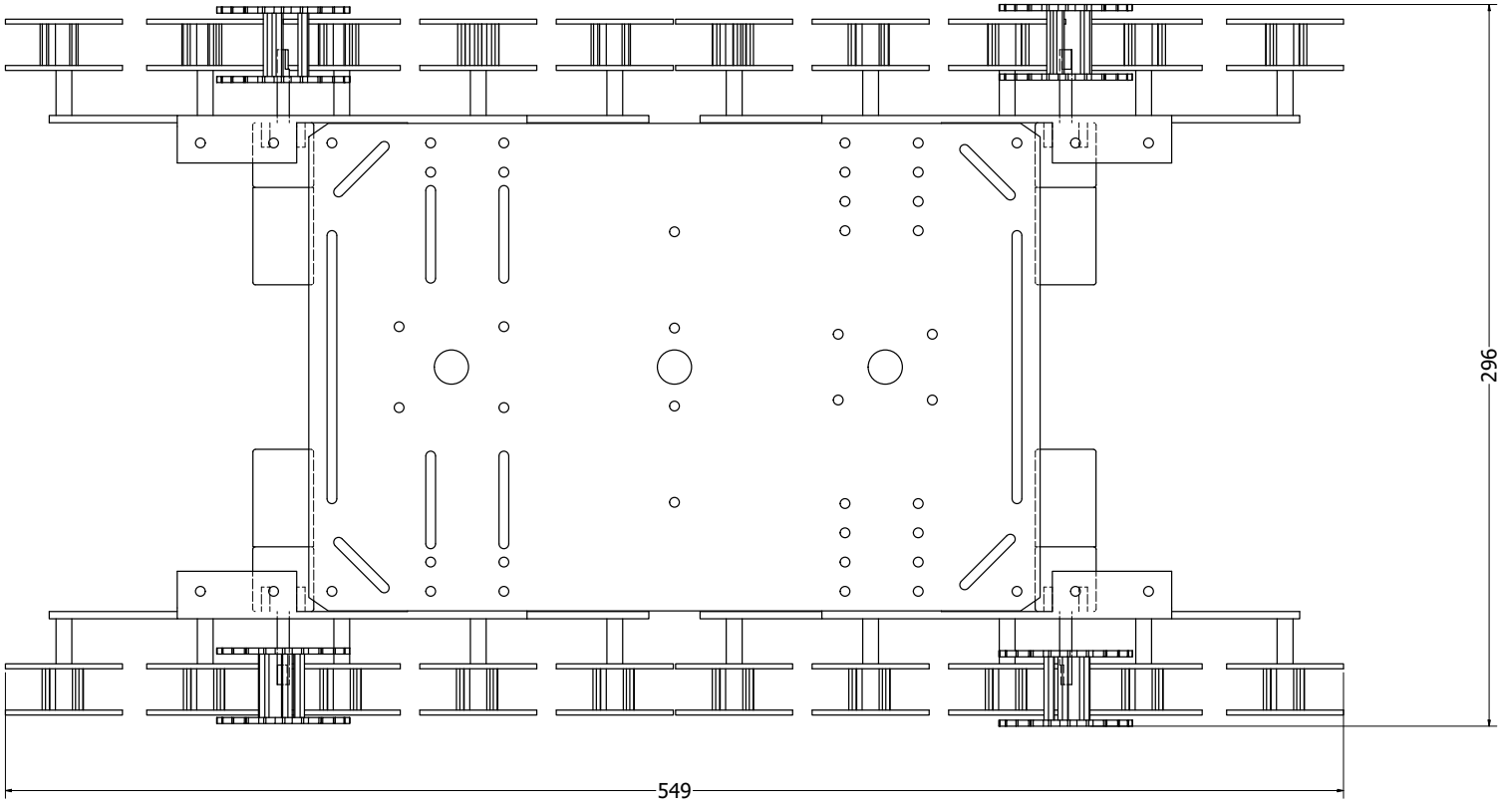
DRAWN Jorge	29/06/2020	
CHECKED		TITLE
QA		
MFG		
APPROVED		
		SIZE D / DWG NO full_assembly / REV
		SCALE 1 : 1 / SHEET 1 OF 1

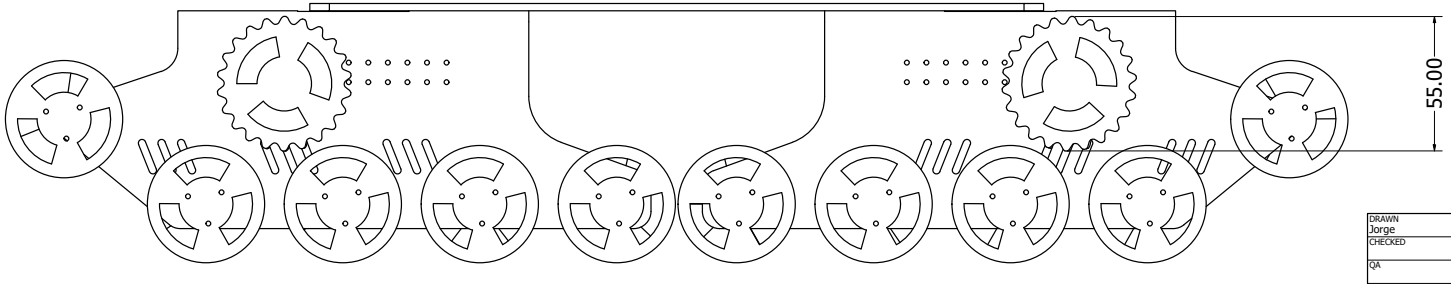




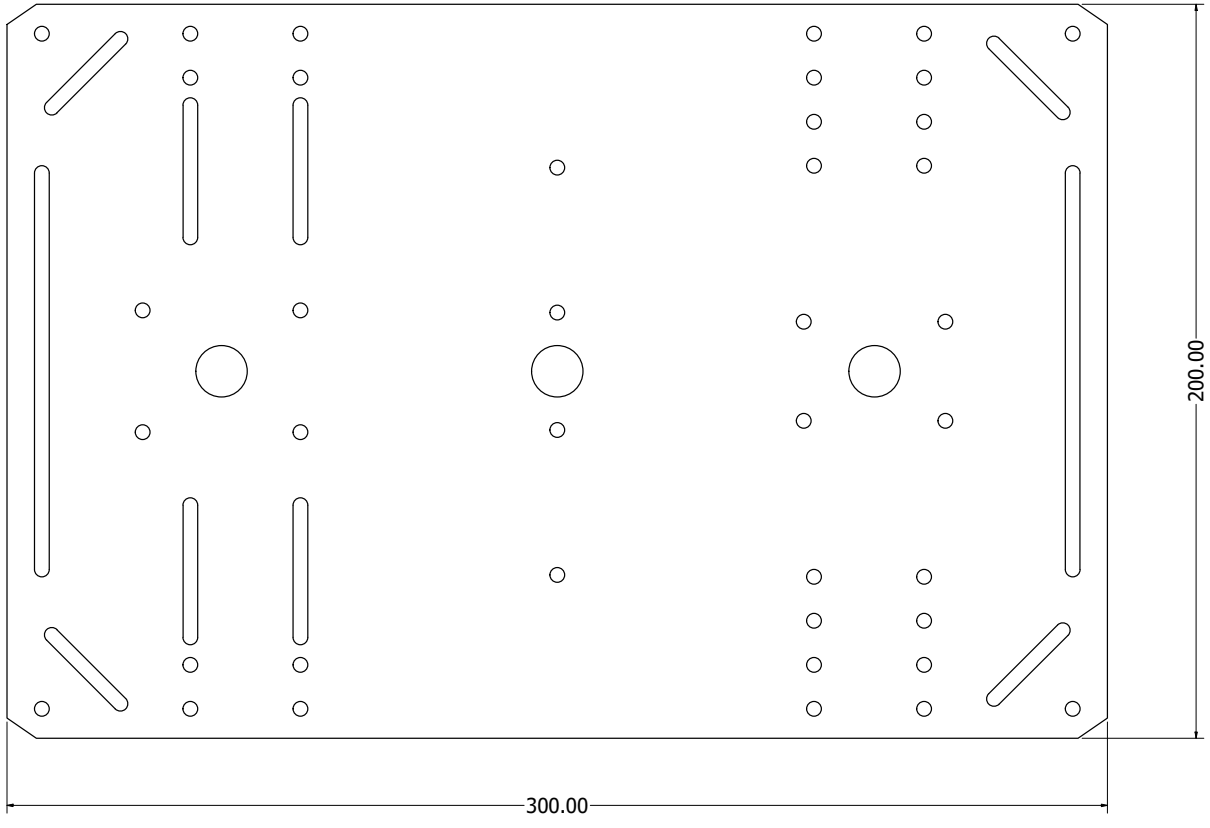


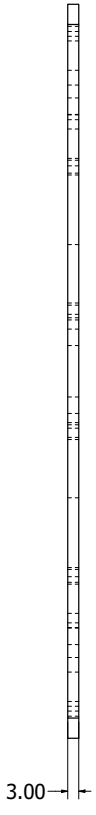
DRAWN Jorge	21/06/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO LIDAR_support
		SCALE 3:2	SHEET 1 OF 1



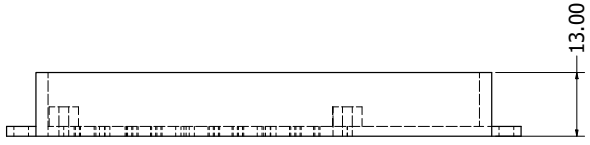


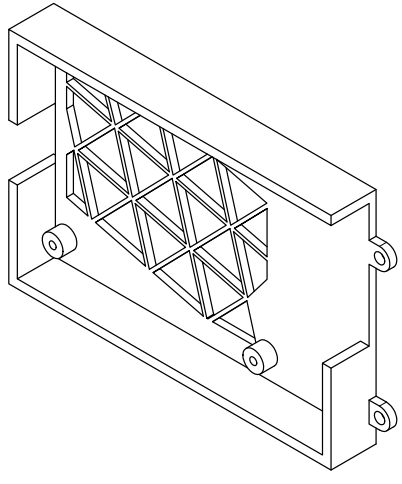
DRAWN Jorge	10/06/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO platform
		SCALE 1 : 1	SHEET 1 OF 1

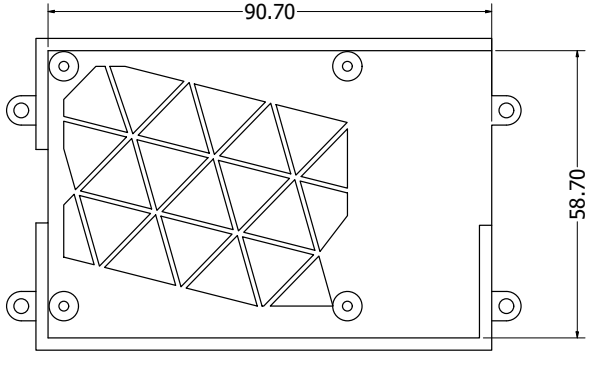




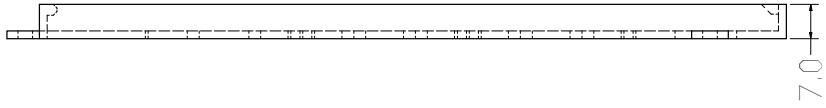
DRAWN Jorge	06/07/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO platform_board REV
		SCALE 3 : 2	SHEET 1 OF 1

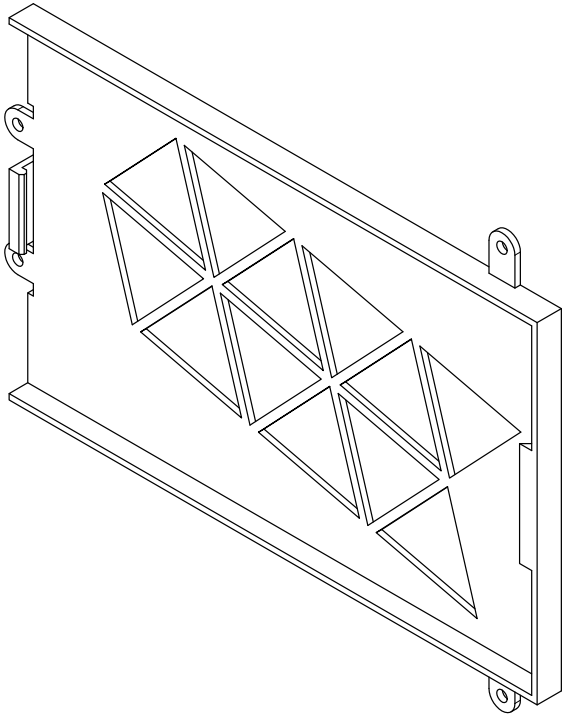


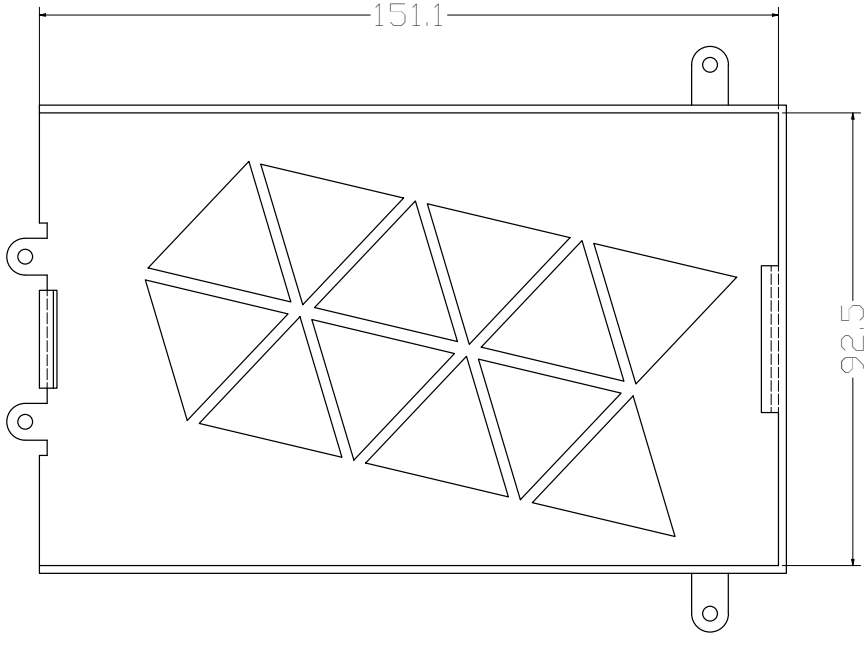




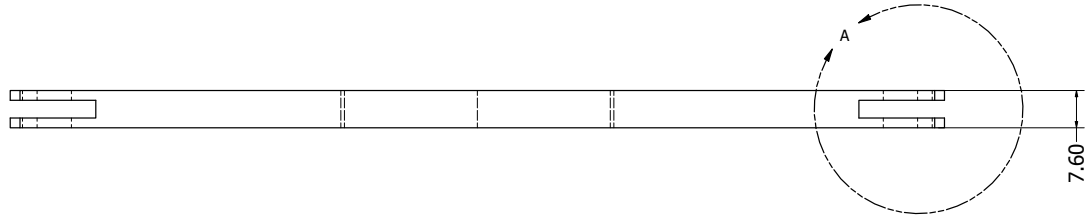
DRAWN Jorge	21/06/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO decks REV
		SCALE 2 : 1	SHEET 1 OF 1

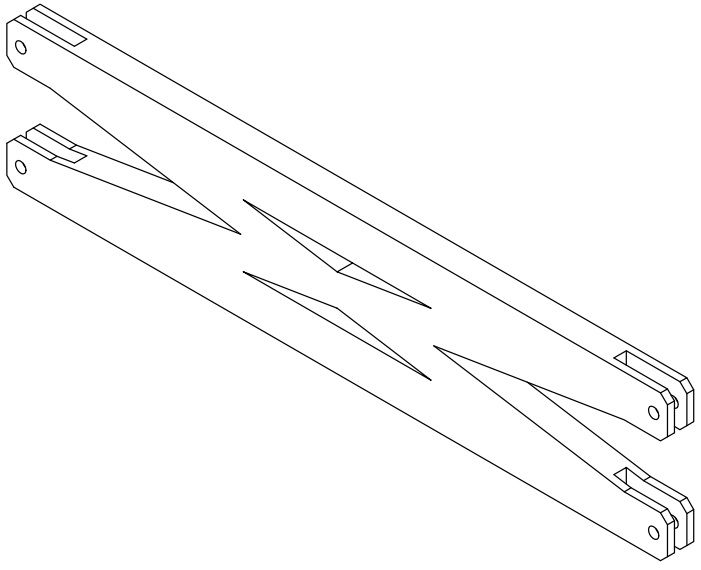


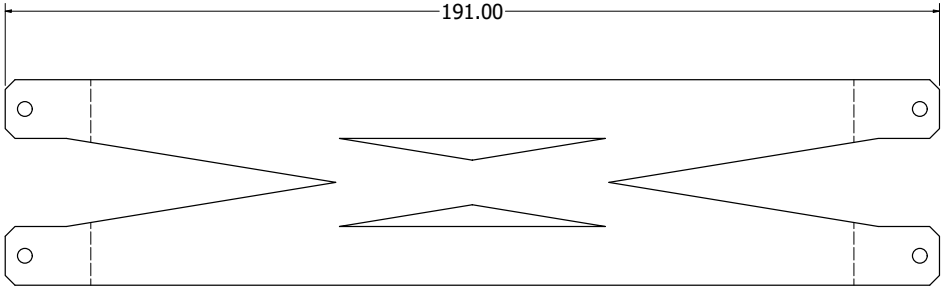


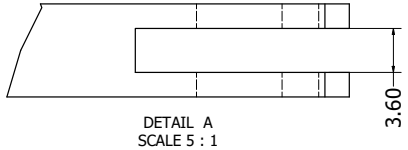


DRAWN Jorge	07/06/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO PCB_deck REV
		SCALE 2 : 1	SHEET 1 OF 1



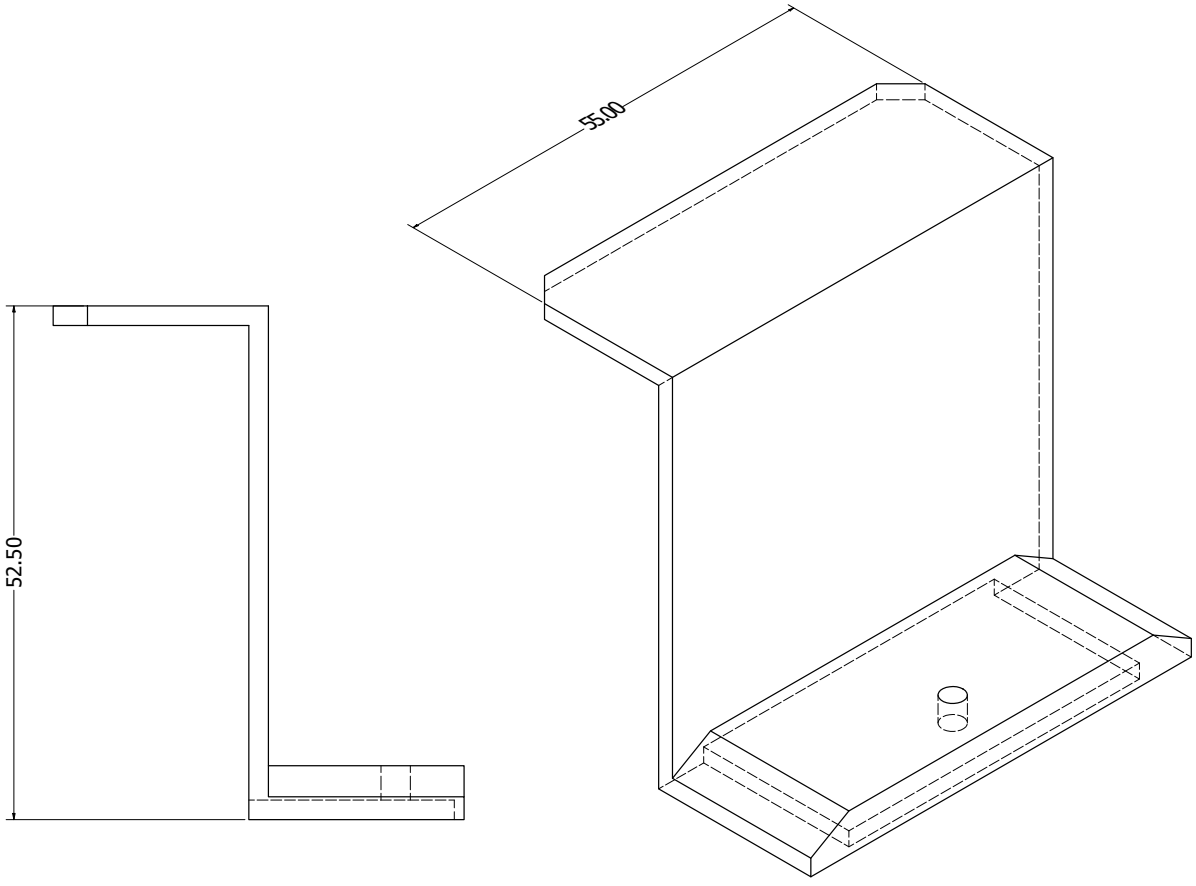


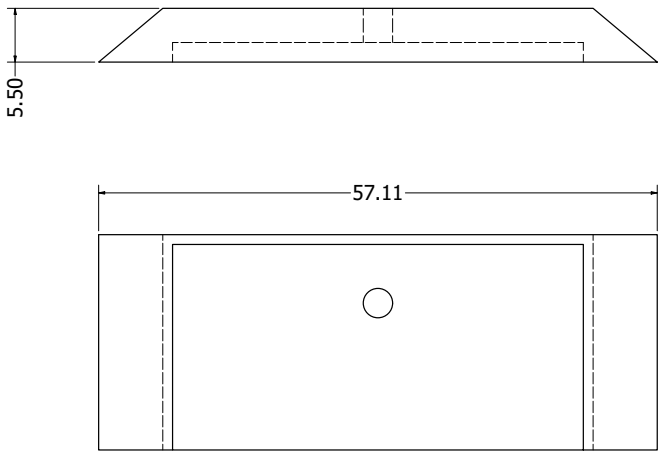




DETAIL A
SCALE 5 : 1

DRAWN Jorge	06/07/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO stabilizer
		SCALE 2 : 1	SHEET 1 OF 1





DRAWN Jorge	06/07/2020		
CHECKED		TITLE	
QA			
MFG			
APPROVED			
		SIZE D	DWG NO battery-strap
		SCALE 4 : 1	SHEET 1 OF 1