UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

Aerospace Technologies Engineering *Bachelor Thesis*

Study of the development of automatic control systems for autonomous exploration robots

Report

Balado Ordax, Ramiro

Thesis Director: Nejjari Akhi-elarab, Fatiha

June 2018

Abstract

This thesis explores the necessities of autonomous exploration rovers within the context of a high latency communications link and proposes a program architecture made up of different modules which takes these necessities into account. A MATLAB simulator is designed and implemented to test the proposed software architecture and to verify the correct behaviour of the proposed algorithms within each module. Finally, these modules are implemented in C++ and tested for verification with a real rover.

Contents

1 Introduction **6**
1.1 Aim . . . 6
1.2 Scope . . . 6
1.3 Requirements . . . 6
1.4 Background . . . 7
1.4.1 GRASS Rover . . . 7
1.5 State of the Art . . . 8

2 Architecture **10**
2.1 Requirements . . . 10
2.2 System Architecture . . . 10
2.3 Module Architecture . . . 11

3 Environment Recognition **13**
3.1 Environment Data . . . 13
3.2 Data Structure . . . 13
3.3 Data Processing . . . 14
3.3.1 Linear Least Squares . . . 14
3.3.2 Weighted Least Squares . . . 15
3.3.3 Node Weight . . . 16
3.4 Data Transmission . . . 17
3.5 MATLAB Simulator . . . 17
3.5.1 Environment Data . . . 18
3.5.2 Data Processing . . . 18
3.6 Results . . . 18

4 Path Finding **22**
4.1 Dijkstra's Algorithm . . . 22
4.1.1 Algorithm Overview . . . 22
4.2 A* Search Algorithm . . . 23
4.3 D* Lite . . . 23
4.3.1 Algorithm Overview . . . 24
4.4 MATLAB Simulator . . . 24
4.5 Results . . . 24

5 Controller **26**
5.1 Control Model . . . 26
5.1.1 Forward Velocity V_r . . . 26
5.1.2 Angular Speed Ω_z . . . 27
5.2 Actuator Model . . . 27
5.2.1 Wheel Slippage . . . 27
5.3 MATLAB Simulator . . . 28
5.4 Results . . . 28

6 State Estimation **30**
6.1 System Model . . . 30
6.2 Sensor Model . . . 31
6.3 Linear Kalman Filter . . . 31
6.4 Extended Kalman Filter . . . 32
6.4.1 Matrix $\boldsymbol{F}_k$ . . . 32
6.4.2 Matrix $\boldsymbol{H}_k$ . . . 33
6.5 Results . . . 34

7 Inter-Process Communication **35**
7.1 Choice of Communication Protocol . . . 35
7.1.1 Named Pipes . . . 35
7.1.2 Unix Domain Sockets . . . 36
7.1.3 D-Bus . . . 36
7.2 Socket Protocol . . . 36
7.3 Stream/Sequence Socket Workflow . . . 37
7.3.1 Server Socket . . . 37
7.3.2 Client Socket . . . 38
7.3.3 Duplex Communication . . . 38
7.3.4 Termination of the Communication . . . 38
7.4 Packet Structure . . . 38
7.5 Module Messages . . . 39
7.6 Results . . . 40

8 External Communication **41**
8.1 Serial Protocol . . . 41
8.2 Packet Structure . . . 42
8.3 Sensor Data . . . 42
8.4 Actuator Data . . . 43
8.5 Results . . . 43

9 Supervisor **44**

10 Conclusions **45**

11 Future Work **46**

List of Figures

1.1 The GRASS Rover . . . 8
1.2 Coordinate Axes . . . 8

2.1 General System Architecture . . . 11
2.2 General Module Architecture . . . 11

3.1 Environment Data Structure . . . 14
3.2 Inclination Weight Function . . . 17
3.3 Point cloud with fitted planes in the simulator . . . 18
3.4 Weight map of surface shown in Figure 3.3 . . . 18
3.5 Reference Setup of LIDAR Scan . . . 19
3.6 Cloud Point data retrieved from LIDAR . . . 19
3.7 Bucket Weight Map of LIDAR scan . . . 20

4.1 Computed shortest path in simulator . . . 24
4.2 Computed shortest path . . . 25

5.1 Forward Velocity . . . 27
5.2 Angular Speed . . . 27
5.3 Wheel Slip . . . 28
5.4 Shortest path and route traced by the rover in the simulator . . . 28

7.1 Stream/Sequence Socket Workflow . . . 37
7.2 Packet Structure . . . 39
7.3 Data Structure . . . 39

List of Tables

7.1 Type codes . . . 39
7.2 Module codes . . . 39
7.3 Message codes . . . 40

8.1 Sensor codes . . . 42

9.1 Order Instructions . . . 44

Chapter 1

Introduction

1.1. Aim

The main objective of this project is to successfully study, design and develop automatic control systems to satisfy the needs relating to the control of an autonomous exploration robot. The purpose of these automatic control systems is to confer the robot with a reasonable amount of autonomy to be able to carry out its mission with little human control. These systems include the ability to recognise the environment, the ability to find and follow a path to its destination, and the ability to recognise its own movement through the environment.

1.2. Scope

The scope of this work consists in the following points:

- Design and verification of a kinematic and dynamic model.
- Design and verification of a simulation environment in which to test all the subsequent modules.
- Design and development of environment recognition techniques.
- Design and development of dynamic path finding algorithms.
- Design and development of self-localization techniques to acquire data on the movement of the robot.
- Design and development of the control system to follow a prescribed path.
- Integration and refactoring algorithms to fit the real robot.
- Verification of the proper functioning of the modules within the robot.

1.3. Requirements

- Accurate and verified kinematic and dynamic models of the exploration robot.

- A functioning simulator being able to test different environment recognition, path finding and control algorithms easily. This simulator must be able to accurately simulate the kinematics of a rover on a specified surface according to the provided kinematic model, as well as accurately simulate the acquisition of environment data.
- Environment recognition techniques being able to gather information on the steepness and roughness of the terrain, and organize and analyze such information to provide path information to the path-finder.
- A fast dynamic path finding algorithm implementation that takes into account the constant stream of new environment data and the physical restrictions and clearance requirements of the robot.
- Accurate algorithms of self-localization based on different sensors to accurately determine the current position and attitude of the rover.
- A control system capable of directing the rover to follow the assigned path based on the kinematic model.

1.4. Background

This project stems directly from a currently active mission within the UPC Space Program, a student projects association. In 2017, a new mission, under the name of GRASS, was proposed, which consisted in designing and building an exploration robot, with the objective of familiarizing ourselves with the workflow of a real engineering project, with the intricacies of managing a big team, and of course, with the objective of familiarizing ourselves with the technologies involved in designing and creating an exploration robot.

This project obviously consists of many specializations, which can be grouped into three departments: Structural, electronics, and software. The structural and electronics department are not of the concern of this thesis, as this project is encompassed exclusively within the software department.

Due to the fact that the design of the software for a rover is complex, very little progress has been done in this aspect, due to several factors. Firstly, we could only work on it during our free time after classes. This reduced significantly the available hours to design it. Secondly, the design of the software greatly depends on the previous designs of the structure and electronics, which hadn't been completed until recently.

Both of these facts contributed to the slow progress of the software department, and this TFG is in part proposed to fix this and produce a useful rover within a reasonable timeframe.

1.4.1. GRASS Rover

The GRASS Rover, henceforth referred to as the rover, is a six wheeled exploration rover equipped with a Raspberry Pi as its main computing unit and three Arduinos, an Arduino Uno for controlling the LIDAR and state sensors, another Arduino Uno for controlling the XBee radio communication module, and lastly an Arduino Mega for controlling the four wheel motors. The communication between the Arduinos and the Raspberry Pi is done through serial through USB cables.

The chassis of the rover is made out of aluminium and is designed as a rocker bogie system, as can be seen in Figure 1.1. This makes sure all wheels are in contact with the ground at all times.

The two front and hind wheels are powered and incorporate a tachometer, while the remaining middle two are unpowered.

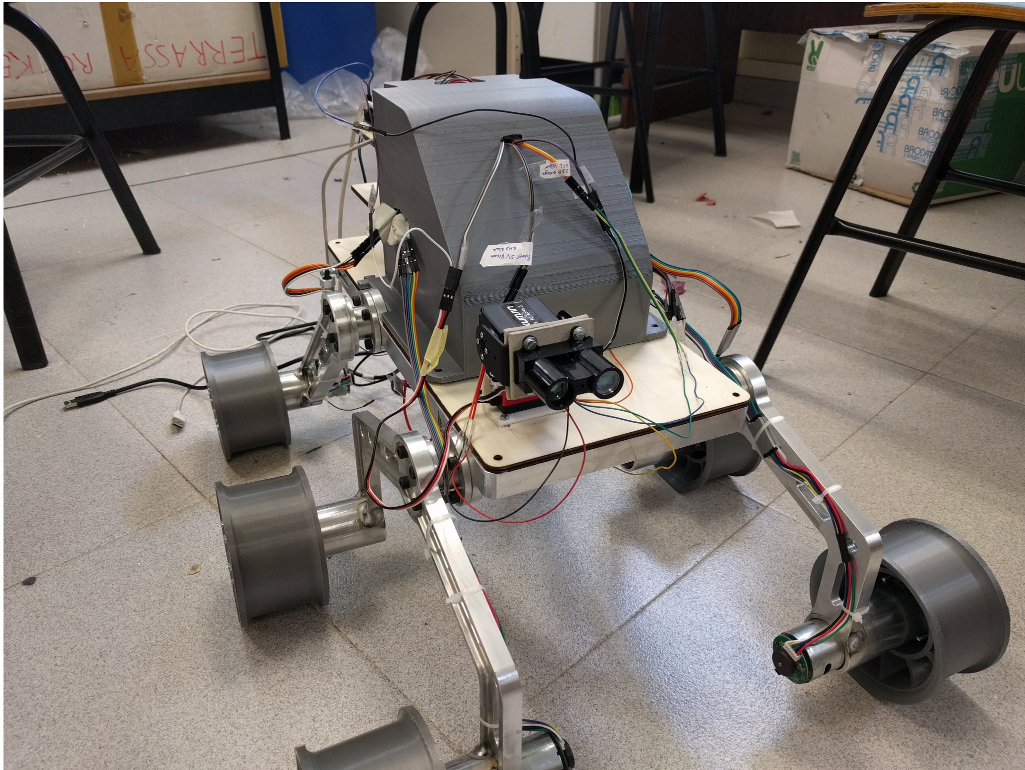


Figure 1.1: *The GRASS Rover*

The LIDAR sensor is mounted on a platform with two servos that allow the rover to scan its surroundings and gather environment geometry data to be processed.

The entire rover is powered by a cluster of 10 2200 mAh LiPo 3 cell batteries in parallel for a total of 22Ah and around an hour of autonomy.

A set of local axes are defined as in Figure 1.2 to conveniently describe its local coordinate system.

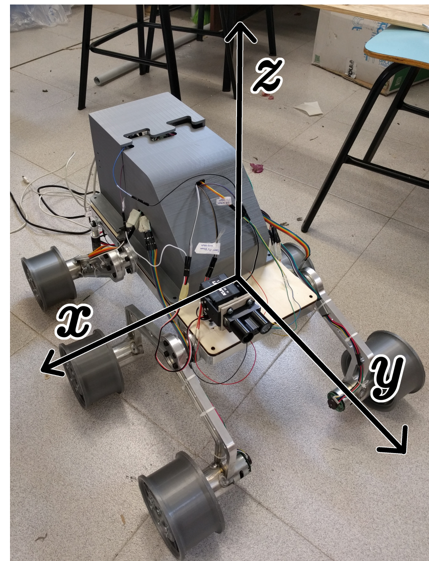


Figure 1.2: *Coordinate Axes*

1.5. State of the Art

The main source of projects with a similar objective appear in reconaissance rovers, mostly on the Moon and Mars, but also in Venus. The newer rovers, like Curiosity or Spirit/Opportunity, incorporate extensive computing capabilities, combined with an extensive array of state and environment sensors. Going back in time the computing power of the rovers quickly decreases to the point of not even having a dedicated cpu and simply being radio

controlled from Earth, as is the case with the Lunokhod program.

In the case of Curiosity, the addition of two pairs of navigation cameras (navcams) and four pairs of hazard cameras (hazcams) allows it to navigate the surface of a planet and avoid obstacles as it encounters them. A pair of cameras at the mast also give the rover a wider field of view. These cameras can be used to reconstruct a 3D topography of the surrounding terrain thanks to stereoscopic imaging. This data can be further processed into a navigational map to plan a route between two goal locations.

In the case of the Spirit/Opportunity rovers, the data is processed in the Rover Electronics Module (REM) within its body[14]. This processing unit contains 128 MB of DRAM. The REM communicates with the instruments, sensors and actuators through a standard VMEbus.

One of these sensors is the Inertial Measurement Unit (IMU), which contains sensors for the measurement of inertial parameters of the rover, such as acceleration or angular velocity. A part from relying on the IMU, these rovers make a full stop every 10 seconds to reassess the situation and recalculate the path to its objective.

The software architecture of these rovers is divided in modules, each with a very specific task, like attitude control, inertial measurement, motor control or radio subsystems[15].

Chapter 2

Architecture

As with all big projects, a good planning and good understanding of the structure of the project is absolutely necessary for the correct development of the project itself. As this project is multifaceted, a proper design of the architecture of the different components, and of the project as a whole, is necessary. This design begins by defining the purpose and objectives this project must satisfy.

2.1. Requirements

As part of the GRASS mission, the objective of this project is to confer the rover with the software tools necessary to traverse a terrain similar to those found on the Moon or Mars. As these requirements are very open-ended, it is necessary to define a new set of more restrictive requirements to facillitate the design of the project.

Most of these requirements stem from the limitations of the hardware of choice. As the number of sensors and actuators is limited, new, more restrictive requirements are imposed on the design of the programs to run. The processing power available to the rover is also limited, so the algorithms that must be chosen need to be lightweight enough to be run on a small form computer.

This project was also intended to be fairly extensible, considering that this is only a small scale version of the rover that at some point will be built, with more sensors and hardware.

2.2. System Architecture

Keeping all these requirements in mind, the final system architecture was designed to facilitate the incorporation of additional sensors or modules, while being relatively simple to describe, develop, operate and maintain.

This structure is based on a simple information loop, which begins at the Environment Recognition stage, which feeds information to the Path Finding module, which again supplies the Controller with the path to follow, which in turn produces movement which is captured by the State Estimation module, which loops back to all preceding modules. The general concept of this design can be seen in figure 2.1.

These modules are discussed more in-depth in Chapters 3, 4, 5 and 6, respectively.

Additionally, a few more modules are needed for the correct operation of the rover. These include a module to handle the communication between the Raspberry Pi and all the sensors and

actuators. This module is not part of the information loop as defined above, yet still require a proper module to be usable. This is discussed in Chapter 8.

One last module that is necessary is a Supervisor. This module is connected to all other modules and makes sure all other modules are working to accomplish the assigned task, and to direct all modules to another task when ordered to. This will be discussed in Chapter 9.

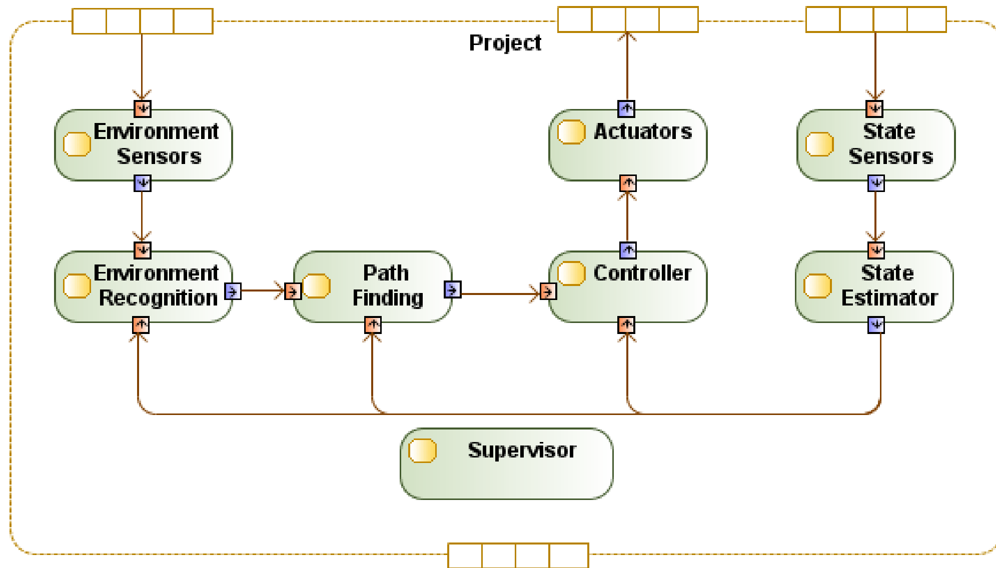


Figure 2.1: *General System Architecture*

2.3. Module Architecture

It has been considered that each module ought to run as separate processes, so as to allow each module to run at its natural speed and to get the most out of the processing power of the on board computer.

This design choice requires for each module to be able to communicate with each other, and, therefore, each of these modules will contain a communications interface, which will allow their inter-process communication.

A generalized structure of this architecture can be found in Figure 2.2. The specifics of the IPC subroutine as well as the communication protocols are discussed in Chapter 7.

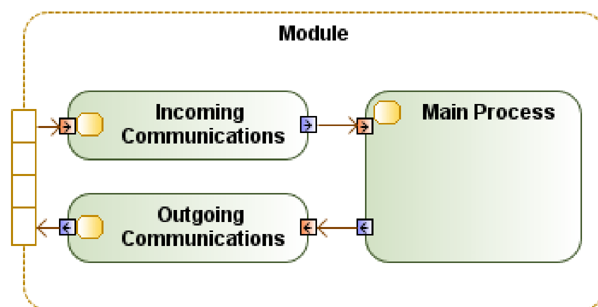


Figure 2.2: *General Module Architecture*

As each module runs on a single thread, a clear circuit must be designed to properly handle both the main processing task and the communications. Therefore, each module will be instantiated in the Incoming Communications phase, which will lead to the Main Process, and finally the Outgoing Commu-

nications, only then to loop around and start again.

Inside each of these communication sub-modules, general procedures for packing and assembling messages are provided. These include templated functions and variadic objects to aid in the encoding and decoding of the messages.

A Module abstract class is also defined, which contains the basic necessary functions to be able to interact with the rest of the modules. This includes two general communication handling procedures and a general process function. These functions can be overridden at any time taking advantage of the polymorphism that C++ provides.

Chapter 3

Environment Recognition

To effectively traverse a foreign and unknown terrain, the rover must be able to process environment data and produce a usable map of its surroundings. The objective of the Environment Recognition module is to process the data coming from the environment sensors, structure it, and produce a usable graph on which the Path Finding module can work.

3.1. Environment Data

The environment data the rover can receive comes from the Sensor module, which receives its data from a Lidar ranging sensor, and is therefore a point cloud data set. These data points represent three dimensional points on the surface of the terrain, characterized by the distance to the sensor and the orientation of the sensor itself.

Moreover, each point can be assigned a probabilistic error of measure, which will determine the likelihood that this measurement was correctly taken. This error will be taken into account when processing the points. Points with a lower error will be given more weight in the computation.

These points can be transformed to the global coordinate system knowing the orientation and position of the rover. But a point cloud gives only very little information on the actual shape, elevation, roughness, inclination or, in general, traversability, of the terrain, so it's the job of the Environment Recognition module to transform this point cloud data set into more manageable data types.

3.2. Data Structure

Keeping in mind the requirements of the Path Finding module, that is, it requires a graph structure to operate, the evident conclusion is to subdivide the cloud points into buckets that will represent the nodes of the graph. Following this design, the main procedure of the Environment Recognition module is to assign weights to these nodes based on the cloud points contained in it.

To better analyze and process these buckets each will be assigned a pair of integers which will uniquely identify it, and will be stored in a map using this pair of integers as the key. Figure 3.1 shows a schematic view of the key-bucket structure.

This pair of integers will be assigned according to its relative position with respect to the initial position of the rover at the time of the initialization of the module. However, this pair of integers does not represent the position of the node, as the distance of the nodes generally will not be 1, but will be defined in the config file for the module.

As the points are detected from the Sensor module, they will start to fill up the buckets. To reduce the load on the processor, the weights of the nodes will not be recalculated every time a point is added, but only every so often points. This number of points after which the weight is recalculated will also be determined in the config file of the module.

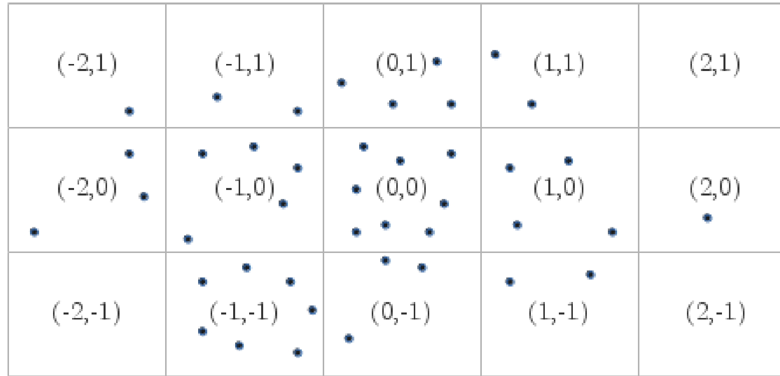


Figure 3.1: *Environment Data Structure*

A map structure allows for random access to the data it contains, and is therefore a suitable structure, since the points will need to be added in a randomized manner and do not require sequential access to them.

3.3. Data Processing

The processing of these buckets must produce data describing the traversability of the terrain. From this set of points, data like inclination, roughness or elevation must be extracted and then further processed into a single value, which will correspond to the weight of the node.

A simple, fast and efficient solution to this requirements is fitting a plane to all the points in the bucket. This fitting of the plane can be efficiently performed using a linear least squares approximation.

3.3.1. Linear Least Squares

A linear least squares problem can be stated as the solution to Equation 3.1:

$$\frac{\partial}{\partial \beta_j} \sum_{i}^{n} r_i^2 = 0 \tag{3.1}$$

$$r_i = z_i - \hat{z}_i \tag{3.2}$$

$$\hat{z}_i = \sum_{k}^{d} \beta_k \Phi_k(\bar{x}_i) \tag{3.3}$$

Where r_i are the residuals of the points z_i with respect to the estimators $\hat{z}_i$. Φ_k are the functions that serve as basis, which, in this case, are the linear functions on each axis. Finally, β_j is the solution vector to this linear system of equations.

Operating on Equation 3.1, the system to solve is the following:

$$\Phi\Phi^T\beta = \Phi z \tag{3.4}$$

$$\Phi = \begin{bmatrix} \Phi_{1,1} & \Phi_{1,2} & \dots & \Phi_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_{d,1} & \Phi_{d,2} & \dots & \Phi_{d,n} \end{bmatrix} \tag{3.5}$$

In this case, the basis functions are the constant function $\Phi_1(\bar{x}_i) = 1$ and the coordinates of the point $\Phi_2(\bar{x}_i) = \bar{x}_{ix}$ and $\Phi_3(\bar{x}_i) = \bar{x}_{iy}$.

The advantage of using a least squares approximation is the ability to quickly re-train the estimators by simply storing some processed data on the previous points. We can prove this by subdividing the matrices into two parts: The part o representing the points that have already been processed, and n, representing the new points.

$$\begin{bmatrix} \Phi_o & \Phi_n \end{bmatrix} \begin{bmatrix} \Phi_o^T \\ \Phi_n^T \end{bmatrix} \beta = \begin{bmatrix} \Phi_o & \Phi_n \end{bmatrix} \begin{bmatrix} z_o \\ z_n \end{bmatrix} \tag{3.6}$$

$$(\Phi_o\Phi_o^T + \Phi_n\Phi_n^T)\beta = \Phi_o z_o + \Phi_n z_n \tag{3.7}$$

Therefore, by simply storing the matrices $\Phi_o\Phi_o^T$ and $\Phi_o z_o$ we no longer require storing all the points.

Incorporating the measurement error in this system is fairly easy. The weighted least squares fitting takes charge of that.

3.3.2. Weighted Least Squares

Keeping in mind the procedure taken in the last section, incorporating a weighting system in the linear system is fairly straightforward. The corresponding target function becomes $S = \sum w_i r_i^2$. All in all, the system to solve is the following:

$$\Phi W \Phi^T \beta = \Phi W z \tag{3.8}$$

Where W is a diagonal matrix containing the reciprocal of the variance of the measurement $1/\sigma_m^2$.

Again, through an equivalent procedure to the one in the last section, the computation and storage of these matrices can be significantly sped up by only storing the matrices $\Phi_o W_o \Phi_o^T$ and $\Phi_o W_o z_o$.

LIDAR Variance

In order to assign a representative value to the variance of the LIDAR measurement, the variance of the actual measurements must be understood.

A LIDAR measurement is composed of two independent variables θ and ϕ, the two angles of the orientation of the LIDAR sensor, and the distance measurement r. If both angles are considered to have a constant variance, the total variance on the distance measurement is the error due to the distance itself and the error propagated by the angles, plus the uncertainty of the current position $\vec{x}$.

$$\sigma_m^2 = \sigma_{\vec{x}}^2 + \sigma_r^2 + r^2\sigma_\theta^2 + r^2\sigma_\phi^2 \tag{3.9}$$

The actual variances for each of the variables r, θ and ϕ need to be measured on the rover itself, and the uncertainty of the current position will be provided by the state estimator.

It is worth pointing out that, congruent with common sense, the farther away a measurement is done with LIDAR, the variance of the measure increases extraordinarily quickly. This is key to understanding that environment data at a very long range with imprecise angle measurements are essentially useless. Nonetheless, these data points can be useful as a very rough first estimate of the terrain at a distance.

3.3.3. Node Weight

Once the least squares is performed, the data that can be obtained from it are the height of the center of the node, which represents the average elevation within the node, and the vector normal to the plane.

This data provides information on the elevation and inclination, but not on the roughness of the terrain. To obtain data that provides an estimate on the roughness of the terrain, the Mean Squared Error with respect to the fitted plane should be calculated.

Traditionally, on a graph to be solved by a path finding algorithm, the weight is assigned on the edges of the graph. In this case, as the essence of the problem lends itself to be easier to work on when split in buckets, the weights computed by this module will be assigned to the nodes.

To be able to assign a final value to the weight of the nodes we must first discuss heuristics. A more in depth discussion can be found in Chapter 4.

For an algorithm based on A*, like D*, to be optimal, the heuristic used must also be admissible. This means the heuristic never overestimates the weight to reach a goal. The heuristic also needs to be consistent for the algorithm to be optimal. That is, the difference of estimates for two nodes must be less than the weight of the edge between them. This last condition also implies an admissible heuristic.

Formally, this last condition can be expressed as the following inequality for starting node X and neighbor node Y:

$$h(X) - h(Y) \leq w(X,Y) \tag{3.10}$$

Since the heuristic used is the distance to the target node, the left hand side of the inequality is upper bounded by the distance between the nodes: $h(X) - h(Y) \leq d(X,Y)$.

If we define the weight of an edge as the distance between the nodes multiplied by a coefficient bigger than 1, this conditition is conserved and the inequality can be rewritten as follows:

$$h(X) - h(Y) \leq d(X,Y) \leq C \cdot d(X,Y) = w(X,Y) \tag{3.11}$$

Furthermore, we can split this C coefficient into the contributions from node X and node Y. The way to split these can be done several ways. The two most obvious is averaging both values, either arithmetically or geometrically. Since the weights of the edges are additive when looking for a solution, the arithmetic mean will be used.

Therefore, each of these coefficients themselves must be bigger than 1. A weight function therefore must be designed to finally assign a weight to each node.

Weight Function

By the previous definition of the weight coefficient, the weight represents the multiplier of the distance between the nodes. For example, a weight of 2 means that, for a given total path, the rover would expend twice as much cost compared to a path with node weight 1.

A function must be found that accurately represents the relative expense of traversavility for each node, taking into account the inclination and roughness of the node. A function has been designed using the angle of inclination, and has the form shown in Figure 3.2. This function can be approximated very closely using the formula $W(\alpha) = 1 + 2\tan^{1.25}(\alpha)$. This function, using the normal vector as a parameter, is simply transformed into Equation 3.12.

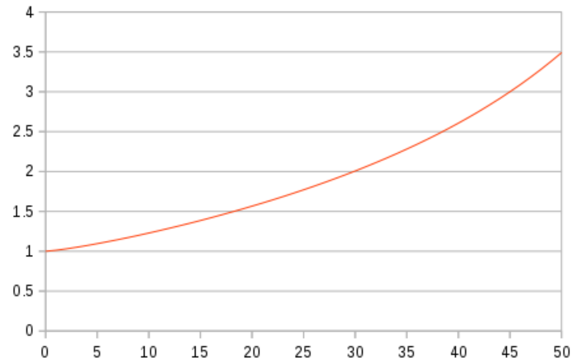


Figure 3.2: *Inclination Weight Function*

$$W_{inc}(\hat{n}) = 1 + 2\left(\sqrt{\frac{\hat{n}_x^2 + \hat{n}_y^2}{\hat{n}_z^2}}\right)^{1.25} = 1 + 2\left(\frac{\hat{n}_x^2 + \hat{n}_y^2}{\hat{n}_z^2}\right)^{0.625} \tag{3.12}$$

With n_x, n_y and n_z being each of the components of the vector normal to the plane.

The default value for this weight of the unvisited nodes has to be chosen so that the weight to travel through this node is not optimal, but so that it would be chosen when the other options are significantly worse. In other words, the default weight is the threshold at which the rover would choose to traverse unknown territory instead of going through a more perilous known node. This threshold has been chosen to be 2, as it is equivalent to a slope of 30°, which has been considered to be the limit of acceptable inclination for the rover.

Once the computing of the weight is finished, the modified nodes are pushed into a message queue and transmitted when the processing is finished.

3.4. Data Transmission

Using the protocol developed in Chapter 7, the packets to be transmitted contain the node key, that is, two integers, and the weight, a float. This data is transmitted to all the clients.

Moreover, a client may request a specific node to be sent. In this case, the data will only be sent to this specific client. This can be useful, for example, if a module has missed previous updates due to not being instantiated yet.

3.5. MATLAB Simulator

The implementation of this module into the simulator adheres fairly closely to the above description, but has some peculiarities.

The constructor of the module is fairly straightforward. It initializes the input and output maps and remaining variables. There are two maps in total: One that stores the points, which the module processes, and another one in which the module stores the output data of this processing.

3.5.1. Environment Data

Due to the fact that there's no sensor module in the MATLAB simulator, the acquisition of data is implemented directly in the Environment module, as seen in Listing A.35. This acquisition of data is produced in the function `sensePts()`. This function generates a number of random 2D points in a gaussian distribution around the position of the rover.

These 2D points are then evaluated with the terrain elevation function `f()` to obtain the final coordinate. This point can then be inserted in the point map to be further processed.

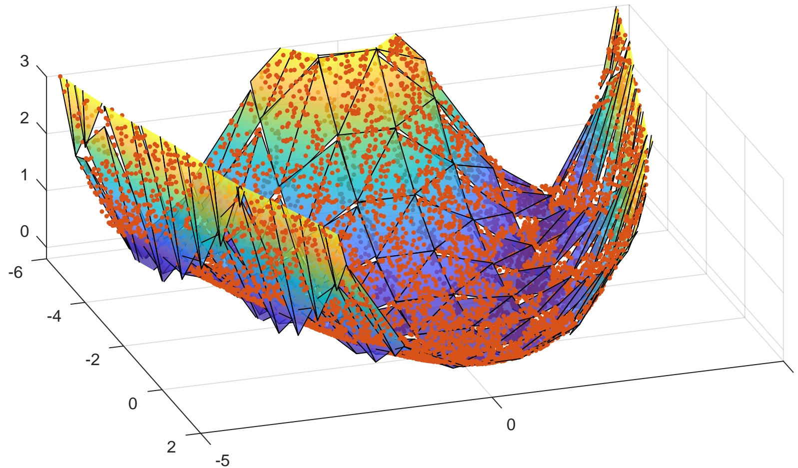


Figure 3.3: *Point cloud with fitted planes in the simulator*

3.5.2. Data Processing

In the case of the MATLAB Simulator, the data processing is only done once in the constructor of the `model` object. This does not suppose any major drawback or defficiency and produces much more readable and understandable code.

Since efficiency or speed is not a major concern of the simulator, it has not been optimized in the way described in Section 3.3. In the case of the simulator, each bucket is processed every time the function `process()` is called. This function iterates through every bucket, updating the weights of their respective nodes with the function `updateWeight()`.

The function `updateWeight()` first collects all points inside the bucket and checks if the number of points is enough to properly assign a weight to the node. In case of not being enough, the default weight is applied. Otherwise, the weight is calculated with the function `fitPlane()`. This function implements the weight function described in 3.3. Once calculated, this weight value is inserted into the node map.

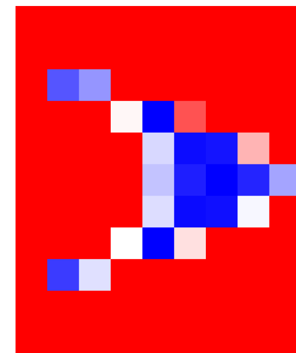


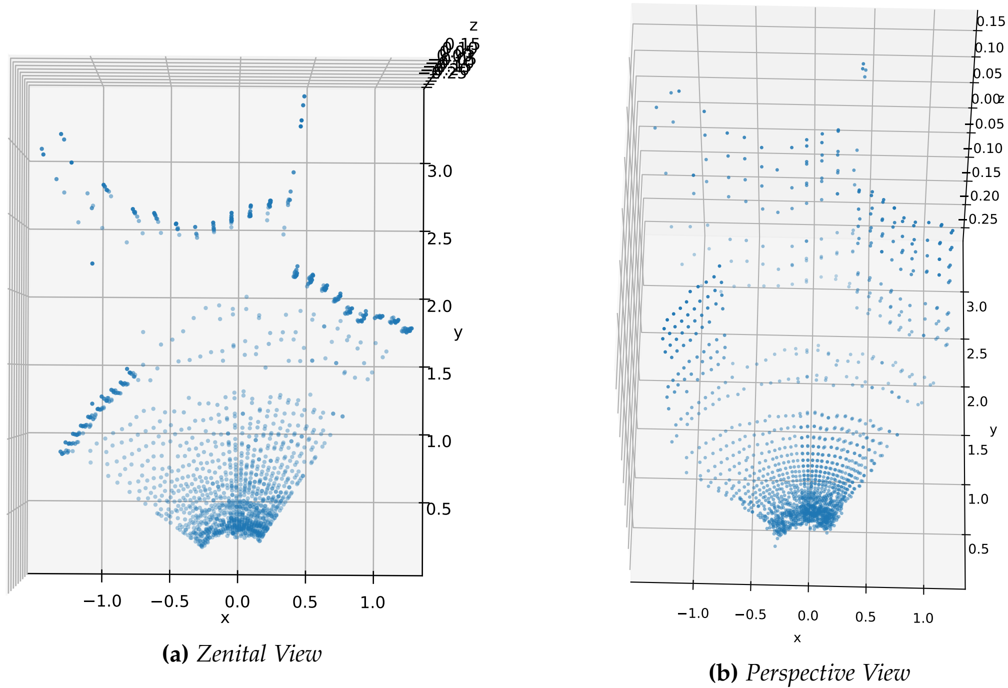
Figure 3.4: *Weight map of surface shown in Figure 3.3*

3.6. Results

The module has been successfully implemented as per Listings A.6 and A.19. A test setup was built to test the LIDAR sensor in the lab with three big cardboards at different depths with respect

to the rover. A photograph describing the test setup can be seen in Figure 3.5.

Figure 3.5: *Reference Setup of LIDAR Scan*



(a) *Zenital View*

(b) *Perspective View*

Figure 3.6: *Cloud Point data retrieved from LIDAR*

This test was threefold. First, to test that the LIDAR data came in the right format. Second, that the Arduino could relay the information from the LIDAR and sensors to the Raspberry Pi.

And finally, that the processing done by the Raspberry Pi produced a usable map for the Path Finding module.

First, the LIDAR information came in the right format and units, but the servomotors turned out to be scanning too fast. This produced warping and aliasing of points when moving the servos a long distance. Adding a simple delay until the servo is at a stable position solved the problem. This solution makes the scanning a bit slower, but produces accurate results.

The next difficulty that was encountered was that the position given to `Servo.write()` did not correspond to the actual angle moved by the servos. This is likely due to the fact that the used servos have a wider range than most, and thus must encompass their entire true range into the virtual range of $[0°, 180°)$. A simple linear transformation is enough to correct this deviation.

After all the corrections had been applied, the LIDAR point cloud results are shown in Figure 3.6. As can be seen, the cardboard sheets can be clearly identified as vertical walls of points. The distances from the LIDAR to the cardboards have been measured and accurately correlate with the distance obtained from the scan.

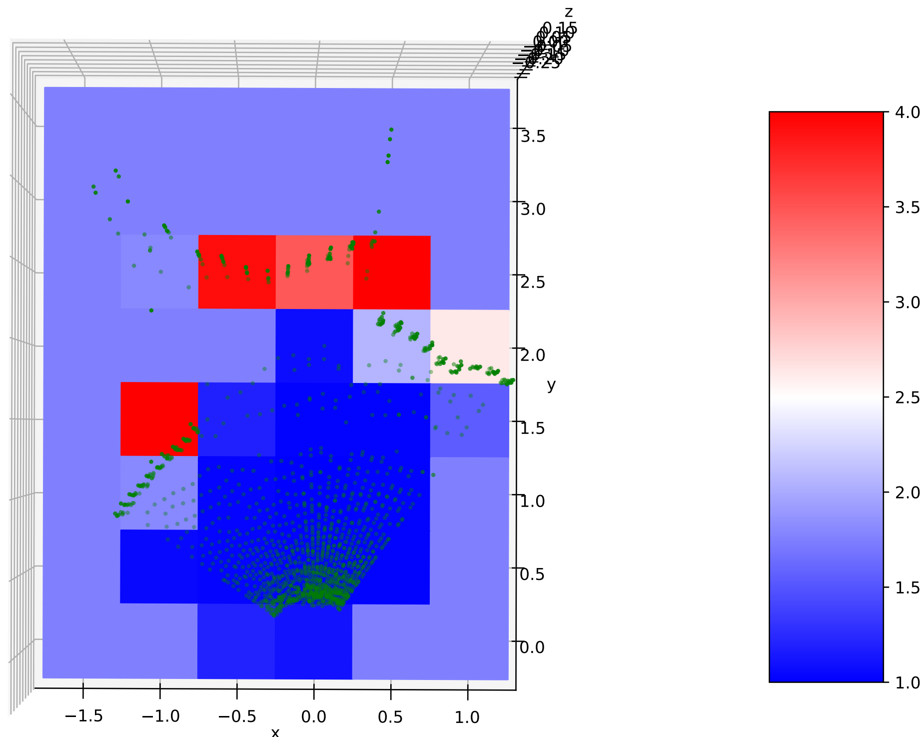


Figure 3.7: *Bucket Weight Map of LIDAR scan*

From this data, the Environment Recognition module creates the weight map as shown in Figure 3.7.

As expected, Figure 3.7 clearly shows that the algorithm is assigning worse values to the locations where the slope is steeper and values closer to 1 where the slope is level with the horizon. Some of the buckets that contain a cardboard do not present such a striking increase in weight. This can be explanied through two different reasons.

Firstly, these buckets also contain points that are level with the horizon. This reduces the slope of the fitted plane and therefore reduces the cost of that bucket.

Secondly, in the buckets that hardly contain any ground points, this effect can be explained by

the inability of the linear least squares approach to be fitted to a vertical plane. This is due to the fact that the linear least squares approach only considers the z coordinate to have measurement error. It cannot deal with measurement errors in the x and y values, and as such, fails when approximating vertical planes. This can be solved by using a different fitting approach, like orthogonal least squares.

These shortcomings are considered minor, and do not suppose a catastrophic failure of the algorithm, since quick fixes have been proposed to aleviate these inconveniences. These tests are therefore considered passed.

Chapter 4

Path Finding

The path finding algorithm is essential for the proper movement of the rover in any environment it is located. It must take into account both the environment data which is supplied by the Environment Recognition module (Chapter 3), and the current state of the rover provided by the State Estimation module (Chapter 6).

As the information from the Environment Module comes in the form of a graph, common graph solving algorithms could be used, like Dijkstra or A*. Considering the domain of this problem, these algorithms are not suitable, since they must recalculate the entire path for every change in the weights of the graph. As the Environment module will provide regular updates to the information of the environment, this would slow down immensely the processing of the graph. Therefore, an algorithm must be found or developed that takes into account previously found solutions.

This class of algorithms are tipically called Incremental Search Algorithms, and are widely used in robotics for precisely this reason. A whole family of this class of algorithms can be found under the name D*, and are the most widely used path finding algorithms in robotics.

4.1. Dijkstra's Algorithm[3][5]

The Dijkstra's Algorithm, named after Edsger Dijkstra, is one of the earliest graph traversal algorithms. Created in 1956, it has remained one of the most widely used shortest path algorithms since its creation due to its great efficiency for non directed static graphs. As all path finding algorithms, its main objective is to find the shortest path between two nodes in a graph or network.

In its most common implementation today, this algorithm assigns a value to each node representing the shortest path between each node and a specific node, generally called the origin.

Dijkstra's algorithm is proven to be the algorithm without a heuristic that expands the least amount of nodes to find its target, and as such, it has remained a very useful and widely used algorithm.

4.1.1. Algorithm Overview

The algorithm is initialized by assigning to each node a value of infinity except for the origin, which is assigned the value 0. The origin is then added to the Open Set of nodes. That is, the nodes that are contiguous to the nodes which have already been visited, but have not been analyzed themselves. The Open Set must be ordered by the assigned value of the nodes. This value represents the shortest known path to that node at that point in time.

While the Open Set is not empty, and the destination node has not been placed in the Closed Set, take the node from the Open Set with the smallest value and move it to the Closed Set. Assign to each of its neighbours a value equal to the value of the previous node plus the weight of the edge if it is lower than the current weight of the node. Then, if this new value is lower than the previous one, add this neighbor to the Open Set if is not there already.

To reconstruct the path, simply follow the neighbors with the lowest value starting from the target node.

4.2. A* Search Algorithm[11]

The A* algorithm can be seen as an extension to the Dijkstra's algorithm, by which a new way of ordering the Open Set is defined. This new way of ordering the Open Set takes into account the proximity of the nodes in the Open Set to the target node. This allows for a more focused search of the possible nodes.

The algorithm basis is essentially the same as the Dijkstra's algorithm, only that, instead of ordering the Open Set by the value of the node, it orders it by the value plus the heuristic from the current node to the target node: $f(n) = v(n) + h(n)$.

This heuristic must comply with some requirements for the algorithm to be efficient. Firstly, it must be admissible. This means that, for each node, the heuristic must never overestimate the cost of reaching the goal. In other words, the heuristic must always be less than or equal to the actual cost to reach the goal.

Secondly, the heuristic must be consistent. In general terms, it means it must satisfy the triangle inequality between two adjacent nodes and the goal node: $h(N, G) \leq h(M, G) + c(N, M)$.

This guarantees that A* will find the optimal path between two nodes if such a path exists.

The following are common heuristics widely used for A*:

$$\text{Euclidean norm: } h(N, M) = \sqrt{(N_x - M_x)^2 + (N_y - M_y)^2} \tag{4.1}$$

$$\text{Taxicab distance: } h(N, M) = |N_x - M_x| + |N_y - M_y| \tag{4.2}$$

$$\text{Diagonal distance: } h(N, M) = \sqrt{2}\min(|N_x - M_x|, |N_y - M_y|) + ||N_x - M_x| - |N_y - M_y|| \tag{4.3}$$

4.3. D* Lite[13]

The D* Lite algorithm started as a combination of heuristic search methods, specifically A*, and incremental serach methods, which allow for faster computation of graphs with similar weights. The algorithm on which D* Lite is based was called Lifelong Planning A*. D* is an addition to this last algorithm which allows for a moving origin node, much like a robot would move across a terrain.

To accomplish this task, D* assigns to each node two values: g and rhs. The rhs value works like the node value in A*. It is the best estimate of the cost to reach this node from the goal node. The g value is an additional value to keep track of the consistency of the node. If g<rhs, then the node is said to be underconsistent. This means that its g value is underestimating the cost of reaching this node. If g>rhs, then the node is said to be overconsistent, which means its g value is overestimating the cost of reaching this node.

In general, the algorithm tries to make consistent only the nodes that are required to reach the destination. To do this, the list of inconsistent nodes must be kept in a specific order defined by the key pair $key(s) = [\min(g, rhs) + h(s_{start}, s) + k_m; min(g, rhs)]$. These nodes must be ordered according to a lexicographic ordering by their respective keys. k_m is a value that keeps track of the path that the robot has already followed.

4.3.1. Algorithm Overview

To initialize the algorithm, all nodes must be set to $g = \infty, rhs = \infty$. The goal node is initialized with $rhs = 0$ and inserted into the Open Set.

While the start node is inconsistent or there are nodes in the Open Set with a lower key than the start node, the first node in the Open Set is processed. If the node is overconsistent it is made consistent by $g = rhs$, and its neighbors are updated and inserted into the Open Set if inconsistent. If it is underconsistent, the node is made overconsistent by $g = \infty$ and itself and its neighbors are updated and inserted into the Open Set if inconsistent.

Updating a node consists in recalculating its rhs value by taking the minimum of $rhs(u) = c(u, s') + g(s')$, with respect to its neighbours s'.

Once the initial route has been found, the weight updates are introduced and the affected nodes inserted into the Open Set. The previous procedure is then called and the Open Set is again depleted until no key is left lower than the current node.

4.4. MATLAB Simulator

The algorithm implemented in the Path Finding module shown in Listing A.28 in the MATLAB simulator was chosen to be Djikstra's algorithm, since the computational power of the desktop computer it runs on is more than enough to not have to worry about overworking the cpu.

Taking Figure 3.4 as the weight map to use, the simulator produced the path shown in Figure 4.1 that minimizes the cost from traveling from the origin located at $(-5, -3)$ to the destination located at $(-5, 3)$. As can be seen, the algorithm avoids nodes with a high weight like the peak on the left, which represent nodes with a higher slope that the rover would not be able to traverse.

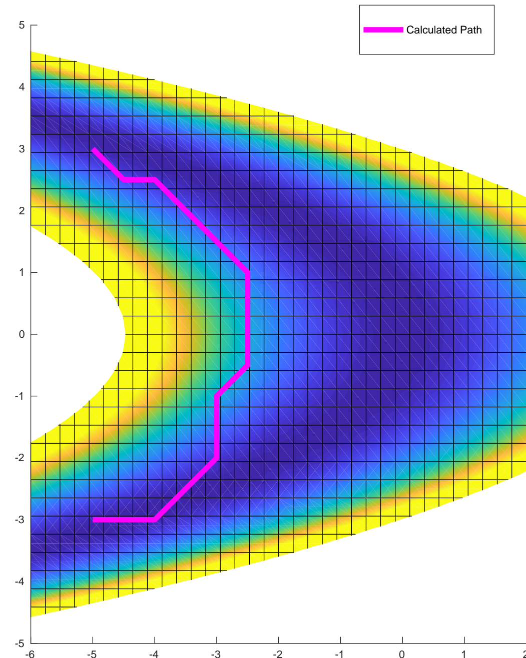


Figure 4.1: *Computed shortest path in simulator*

4.5. Results

The Path Finding module in the rover has been implemented as in Listings A.9 and A.20 and tested with the weight data obtained from the LIDAR scan in Figure 3.7. This data set is not very extensive, but it will be enough for verification of the algorithm.

The algorithm was set to find the shortest path between nodes $(0, 0)$ and $(-1, 7)$. As can be seen in Figure 4.2 in purple, the calculated optimal path avoids all the nodes with a higher cost of

traversal, depicted in red in this figure.

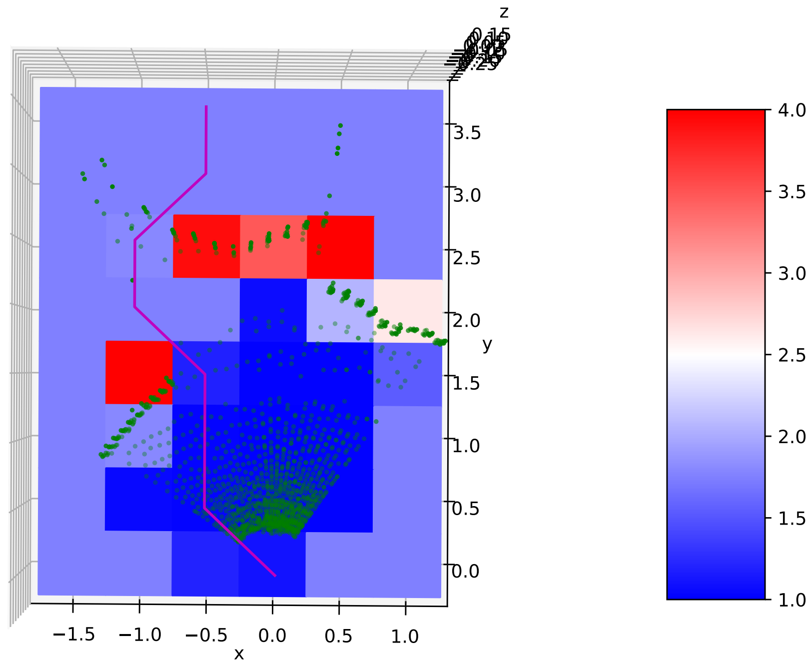


Figure 4.2: *Computed shortest path*

Chapter 5

Controller

For the rover to move to its objective it requires a module that supplies the actuators with the orders to follow. The task of calculating the signals to feed the actuators is taken up by the Controller module.

This module takes into account both its current position and orientation and the destination it needs to reach.

5.1. Control Model

To simplify the task of this module, two main driving parameters will be calculated: The linear forward speed and the rotating speed. Since this rover does not have directional control these two parameters are enough to define any possible movement by the rover. These two parameters will then be converted into specific signals for each of the rover wheels.

The simplest way of conceptualising this control system is through two vectors. One is the target vector, which spans from the rover origin and reaches the target destination. The second vector is the forward direction of the rover, the y direction in the local coordinate system.

Since the target position does not incorporate a vertical coordinate, the manipulations done to these vectors must keep a vertical symmetry. This symmetry can be easily achieved by projecting both vectors to the xy plane. Once these two vectors have been projected, it essentially turns into a 2D problem.

Two controllers must now be designed with the information from these two 2D vectors. The main information that can be extracted from these vectors is the angle between them. Since this system imposes a 2π periodicity, the functions that control this system must also keep this periodicity.

Additionally, both of these controllers are fitted with a PI controller to guarantee that there's no residual error. The values to be assigned to the K_p and K_i of these controllers will be tuned experimentally.

5.1.1. Forward Velocity V_r

The objective of the controller is to head towards the objective. Therefore, it has been concluded that when the two vectors are parallel and with the same direction the velocity must be maximum, and when the two vectors are on opposite direction the velocity must be minimum. This lends itself well to a cosine function with the angle between the two vectors as the parameter: $V_r = V_{max}\cos(\Delta\theta)$. A plot of this function on a polar graph can be seen in Figure 5.1.

As stated before, this function retains the 2π periodicity and complies with the stated requirements.

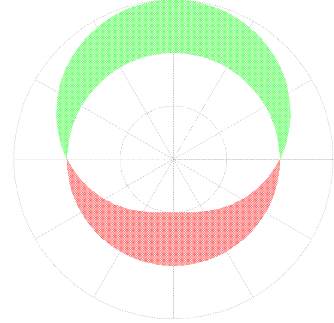


Figure 5.1: *Forward Velocity*

5.1.2. Angular Speed Ω_z

Similarly the angular velocity, which results from the differential speed between the wheels of both sides of the rover, can be controlled accordingly by either the sine of the angle or the tangent of the angle. The only criteria that must be fulfilled is that when the vectors are parallel, the angular speed must be zero and that the value in either direction from the parallel must be opposite, so as to provide a rectifying control. The final decision was to apply a control proportional to the tangent of half the angle, so as to keep the 2π periodicity: $\Omega_r = K_\omega \tan(\Delta\theta/2)$, where Ω_r is the angular velocity of the rover in the local xy plane. This function can be visualized in Figure 5.2.

Another option would be to use a function propotional to the sine of the angle, but this would imply very slow turning speeds for angles close to π. Consequently, this option was discarded in favor of the tangent.

As the tangent grows to infinity close to π, a saturation function must be applied to keep this values in a manageable range. This won't be done explicitly in this model, but on the next section, where these two values are transformed to actual inputs to the wheels.

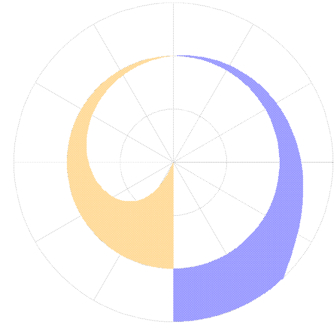


Figure 5.2: *Angular Speed*

5.2. Actuator Model

To be able to translate these calculated values into real actuator inputs, an actuator model must be developed. This model must convert the linear and angular velocities into wheel speeds.

Considering the rover as a rigid body, the standard equations of rigid body kinematics can be applied to find the velocities of the wheels in the local system of coordinates.

$$\vec{v}_w = \vec{V}_r + \vec{\Omega}_r \times \vec{r}_{r\to w} \tag{5.1}$$

5.2.1. Wheel Slippage

If the wheels were directional, the task of finding the wheel speeds would be concluded. On the case at hand, the wheels are not directional and rely on slip to move and interact with the surface.

Wheel slip is the relative motion between the contact surface of the wheel and the terrain it is moving through. In a no-slip condition, the relative velocity of the wheel material in contact with the terrain and the terrain itself is zero.

Wheel slip can be rectified by increasing the wheel speed so that the projection of the wheel speed vector v_s onto the desired velocity vector v_w is equal to the desired velocity vector. It can be written as in Equation 5.2:

$$\vec{v}_s \cdot \frac{\vec{v}_w}{|\vec{v}_w|} = |\vec{v}_w| \tag{5.2}$$

This equation has a singularity when $\vec{v}_s \perp \vec{v}_w$, when $\cos(\angle(\vec{v}_s, \vec{v}_w)) = 0$. This singularity would bring the wheel rotation speed to infinity. Since each wheel has a maximum rotation speed, the wheel will simply saturate at its maximum rotation speed.

Considering that, in local coordinates, the angular velocity vector can be considered to lie in the vertical z axis and that the forward velocity v_r and wheel velocity v_s always point in the y direction, the system is simplified into the following equation:

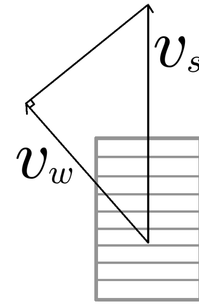


Figure 5.3: *Wheel Slip*

$$v_s = \frac{(\Omega r_y)^2}{v_r + \Omega r_x} + v_r + \Omega r_x \tag{5.3}$$

Where r_x and r_y are the distance between the wheel center and the coordinate system origin. In terms of v_w for each wheel, Equation 5.4 results:

$$v_s = \frac{v_{wx}^2}{v_{wy}} + v_{wy} \tag{5.4}$$

5.3. MATLAB Simulator

The MATLAB Controller module has been successfully implemented as in Listing A.34.

Since the MATLAB kinematic model does not incorporate the wheel slip equation and the rover is simply modelled as a point particle with linear and angular velocity, the control parameters can be directly used to modify the state of the rover.

The results of the control model have been positive and allows the rover to follow the described path, as seen in Figure 5.4. The rover closely follows the path prescribed by the Path Finding module.

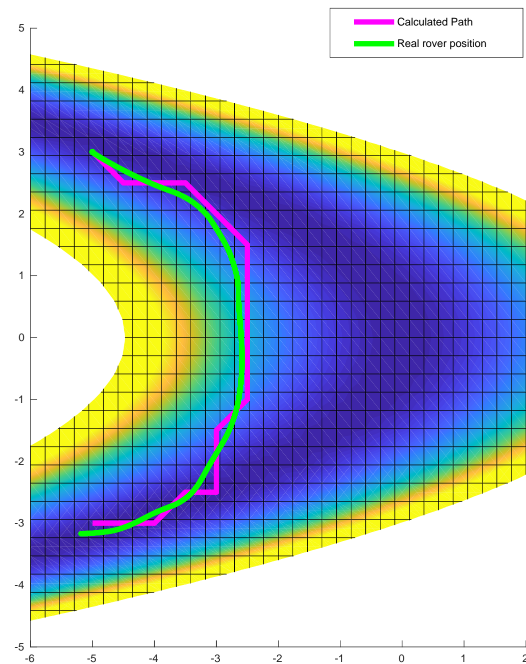


Figure 5.4: *Shortest path and route traced by the rover in the simulator*

5.4. Results

The Control Module has been implemented as in Listings A.12 and A.18 and tested on the real rover together with the External Communications module to be able to communicate with the Arduino that drives the motors.

The test was divided in two phases. Firstly, the linear velocity test. Given an objective node, the output velocity of the module correctly correlated with the proposed function pictured in Figure 5.1. The Arduino correctly assigned this velocity to each wheel and finally the linear velocity derived from this driving signal agreed with the measured distance travelled divided by the time the test was run for.

The second phase corresponded with the rotational speed part. This test was similarly set up to the last, in which the Control Module assigned a rotational speed, which was then converted to each of the wheels rotational speed and transmitted through the Communications module to the

Arduino and finally assigned to the motors. The results of this test are not as accurate as the last test. This is due to the fact that these tests were done without the tyres. This contributes to an unforeseen increase in wheel slip which contributes to the overestimation of the rotational speed of the rover.

More tests have to be carried out with the tyres properly installed to be able to reach an accurate control model. Nonetheless, these initial tests provide a solid and somewhat accurate basis to improve on.

Chapter 6

State Estimation

6.1. System Model

The first step in defining a state estimation procedure is to model the dynamical process at hand. This model can be easily obtained from the rigid body kinematic equations[1][2][7]:

$$\dot{\vec{x}} = \vec{v} \quad (6.1) \qquad \dot{\vec{v}} = \vec{a} \quad (6.2) \qquad \dot{q} = \frac{1}{2}\vec{\omega} \cdot q \quad (6.3)$$

With $\vec{x}$ being the position, $\vec{v}$ the velocity, $\vec{a}$ the acceleration, q the quaternion $q_{L \leftarrow G}$ that transforms from Global to Local coordinates, and $\vec{\omega}$ the angular velocity. All these variables are considered in the global coordinate system.

Furthermore these equations are transformed into the form $\hat{x}_k = f(\hat{x}_{k-1})$.

$$\hat{x}_k = \begin{pmatrix} \vec{x} \\ \vec{v} \\ \vec{a} \\ q \\ \vec{\omega} \end{pmatrix}_k = \begin{pmatrix} \vec{x}_{k-1} + \Delta t \vec{v} + \frac{1}{2}\Delta t^2 \vec{a} \\ \vec{v}_{k-1} + \Delta t \vec{a} \\ \vec{a}_{k-1} \\ q_{k-1}\theta_q(-\Delta t) \\ \vec{\omega}_{k-1} \end{pmatrix} \tag{6.4}$$

Where θ_q is the rotation quaternion produced by ω_{k-1} in Δt. Ideally, it takes the following form[9][17]:

$$\theta_q(\Delta t) = \begin{pmatrix} \cos(|\vec{\omega}|\Delta t/2) \\ \frac{\vec{\omega}}{|\vec{\omega}|}\sin(|\vec{\omega}|\Delta t/2) \end{pmatrix} = q_{G^+ \leftarrow G^-} \tag{6.5}$$

A singularity appears when $\omega = 0$, and therefore this form must be changed into a second order approximation, which yields the following form:

$$\theta_q(\Delta t) = \begin{pmatrix} 1 - (|\vec{\omega}|\Delta t/2)^2/2 \\ \vec{\omega}\Delta t/2 \end{pmatrix} \tag{6.6}$$

Where G^- is the previous global coordinate system and G^+ is the next global coordinate system after a time period of Δt. The argument of θ_q is negative due to the way the composition of quaternions work:

$$q_{L \leftarrow G^+} = q_{L \leftarrow G^-} \cdot q_{G^- \leftarrow G^+} = q_{L \leftarrow G^-} \cdot \theta_q(\Delta t)^{-1} = q_{L \leftarrow G^-} \cdot \theta_q(-\Delta t) \tag{6.7}$$

This model is not linear due to Equation 6.3. The kinematics of rotation are not linear with respect to the orientation and the rate of change of orientation.

6.2. Sensor Model

The rover as it is designed currently has three sensors. An accelerometer, a gyroscope, and a tachometer for each driving wheel. In order for the filter to work, the equations describing the sensor readings from the current state must be first defined. Since the sensors are conformal to the body of the rover, these readings are obtained from the local coordinate system.

The fact that these equations are nonlinear is due to the fact that the state of the rover is considered to be in the global coordinate system, and must be rotated to fit the local system. Storing the variables in a local coordinate system would remove the nonlinearity of the sensor model, but would instead move the nonlinearities to the system model.

$$\hat{z}_k = \begin{pmatrix} \vec{a}_m \\ \vec{\omega}_m \\ v_{sj} \end{pmatrix}_k = \begin{pmatrix} q_k(\vec{a}_k - \vec{g}_k)q_k^{-1} \\ q_k\vec{\omega}_k q_k^{-1} \\ v_y - \omega_z r_{xi} \end{pmatrix} \tag{6.8}$$

6.3. Linear Kalman Filter

The Linear Kalman Filter is an algorithm by which an estimate of the state variables is obtained based on a prediction of the future state in combination with sensor measurements.

This estimate is obtained based on the combination of the uncertainties of both the prediction estimate and the measurements, both represented by their respective Covariance matrices.

The Kalman Filter uses the following convention for the matrix nomenclature[6]:

$$\begin{aligned} \hat{x}_k&: \text{Current best estimate of state vector} \\ \boldsymbol{P}_k&: \text{Covariance of } \hat{x}_k \\ \boldsymbol{F}_k&: \text{Prediction model matrix. Solves the equation } \hat{x}_k = \boldsymbol{F}_k\hat{x}_{k-1} \\ \hat{z}_k&: \text{Sensor reading vector} \\ \boldsymbol{R}_k&: \text{Covariance of } \hat{z}_k \\ \boldsymbol{H}_k&: \text{Sensor model matrix. Solves the equation } \hat{z}_k = \boldsymbol{H}_k\hat{x}_k \\ \boldsymbol{K}_k&: \text{Kalman gain matrix} \end{aligned} \tag{6.9}$$

These definitions, especially $\boldsymbol{F}_k$ and $\boldsymbol{H}_k$, only work for linear models, since these matrices mustn't depend on the state vector. Nonetheless, it is useful to explain the Linear Kalman Filter in order to understand the Extended Kalman Filter.

The basic equations that must be implemented in the module can be subdivided in the two basic steps. Firstly, the Predict step:

$$\begin{aligned} \hat{x}_k &= \boldsymbol{F}_k\hat{x}_{k-1} \\ \boldsymbol{P}_k &= \boldsymbol{F}_k\boldsymbol{P}_{k-1}\boldsymbol{F}_k^T \end{aligned} \tag{6.10}$$

This step propagates the state and predicts a future state based on the previous state. The covariance of the predicted state is also propagated accordingly.

Secondly, the Update step:

$$\begin{aligned}\hat{y}_k =& \hat{z}_k - \boldsymbol{H}_k \hat{x}_{k-1} \\ \boldsymbol{S}_k =& \boldsymbol{R}_k + \boldsymbol{H}_k \boldsymbol{P}_{k-1} \boldsymbol{H}_k^T \\ \boldsymbol{K}_k =& \boldsymbol{P}_{k-1} \boldsymbol{H}_k^T \boldsymbol{S}_k^{-1} \\ \hat{x}_k =& \hat{x}_{k-1} + \boldsymbol{K}_k \hat{y}_k \\ \boldsymbol{P}_k =& (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_{k-1}\end{aligned} \tag{6.11}$$

This step uses real noisy measurements to more accurately describe the predicted state.

These two stages tipically alternate, but they don't need to. Since the speed of the processor is faster than the sampling rate of the sensors, several Predict cycles may happen before an Update cycle happens.

6.4. Extended Kalman Filter

The Extended Kalman Filter is an extension to the linear Kalman Filter by which the Kalman Filter can be applied to nonlinear systems. This is done by using the functions $f(\hat{x}_k)$ and $h(\hat{x}_k)$ and evaluating the matrices $\boldsymbol{F}_k$ and $\boldsymbol{H}_k$ using the Jacobian matrices of these functions, respectively, with respect to $\hat{x}_{k-1}$.

Instead of evaluating the prediction using equation 6.10, the state will be propagated using the nonlinear function $\hat{x}_k = f(\hat{x}_{k-1})$. Similarly, the innovation will be evaluated using the nonlinear function $\hat{y}_k = \hat{z}_k - h(\hat{x}_{k-1})$.

6.4.1. Matrix $\boldsymbol{F}_k$

The matrix $\boldsymbol{F}_k$ is the Jacobian matrix of the prediction function, which estimates the current state $\hat{x}_k$, with respect to the previous state $\hat{x}_{k-1}$.

$$\boldsymbol{F}_k = \boldsymbol{J}_{\hat{x}_k} = \frac{\partial \hat{x}_k}{\partial \hat{x}_{k-1}} \tag{6.12}$$

To calculate this Jacobian, it is easier to decompose the original system into the linear kinematics and rotational kinematics parts, since they are completely independent.

$$\boldsymbol{F}_{lin} = \begin{pmatrix} \boldsymbol{I}_3 & \Delta t \boldsymbol{I}_3 & \frac{1}{2}\Delta t^2 \boldsymbol{I}_3 \\ \boldsymbol{0}_3 & \boldsymbol{I}_3 & \Delta t \boldsymbol{I}_3 \\ \boldsymbol{0}_3 & \boldsymbol{0}_3 & \boldsymbol{I}_3 \end{pmatrix} \tag{6.13}$$

$$\boldsymbol{F}_{rot} = \begin{pmatrix} \hat{e}_q \cdot \theta_q & q \cdot \theta'_q \\ \boldsymbol{0} & \boldsymbol{I}_3 \end{pmatrix} \tag{6.14}$$

Where $\hat{e}_q$ is the vector basis of the quaternion vector field: $\hat{e}_q = \begin{bmatrix} 1 & i & j & k \end{bmatrix}$, and $\theta'_q = \partial\theta_q / \partial\omega_i$ which is a row vector of 3 quaternions and takes the following form:

$$\theta'_q = \begin{pmatrix} -\omega_i \Delta t^2/4 \\ -\Delta t/2\delta_{ij} \end{pmatrix} = \begin{pmatrix} -\omega_x \Delta t^2/4 & -\omega_y \Delta t^2/4 & -\omega_z \Delta t^2/4 \\ -\Delta t/2 & 0 & 0 \\ 0 & -\Delta t/2 & 0 \\ 0 & 0 & -\Delta t/2 \end{pmatrix} \tag{6.15}$$

Finally, F_k can be assembled from its constituent parts by:

$$\boldsymbol{F}_k = \begin{pmatrix} \boldsymbol{F}_{lin} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{F}_{rot} \end{pmatrix} \tag{6.16}$$

6.4.2. Matrix $\boldsymbol{H}_k$

As said before, the matrix H_k is defined as the Jacobian of the expected sensor vector $\hat{z}_k$ with respect to the current state $\hat{x}_k$:

$$\boldsymbol{H}_k = \boldsymbol{J}_{\hat{z}_k} = \frac{\partial \hat{z}_k}{\partial \hat{x}_k} = \begin{pmatrix} \frac{\partial \vec{a}_m}{\partial \hat{x}_k} \\ \frac{\partial \vec{\omega}_m}{\partial \hat{x}_k} \\ \frac{\partial v_{sj}}{\partial \hat{x}_k} \end{pmatrix} = \begin{pmatrix} \boldsymbol{J}_{\vec{a}_m} \\ \boldsymbol{J}_{\vec{\omega}_m} \\ \boldsymbol{J}_{v_{sj}} \end{pmatrix} \tag{6.17}$$

Once again, this Jacobian can be analyzed separately, starting with $J_{\vec{a}_m}$, the Jacobian of the accelerometer:

$$J_{\vec{a}_m} = \begin{pmatrix} \boldsymbol{0}_3 & \boldsymbol{0}_3 & \frac{\partial \vec{a}_m}{\partial \vec{a}_k} & \frac{\partial \vec{a}_m}{\partial \vec{q}_k} & \boldsymbol{0}_3 \end{pmatrix} \tag{6.18}$$

After careful manipulation, the first term results in the rotation matrix expressed by the quaternion q. This matrix can be expressed in the following manner[10]:

$$\boldsymbol{R}_q = (\vec{q}_v \otimes \vec{q}_v + (q_w \boldsymbol{I} + q_v^{\times})^2)/|q|^2 \tag{6.19}$$

Where q_w is the scalar part of q, q_v the vector part of q, the $\otimes$ operator is the outer product of its operands, and $q_v^{\times}$ is the skew symmetric representation of vector q_v.

The second term of Equation 6.18 is harder to analyze. Making use of the properties of the quaternions under differentiation, the second term can be split into two parts:

$$\frac{\partial \vec{a}_m}{\partial q_i} = \frac{\partial}{\partial q_i}(q\vec{v}q^{-1}) = \frac{\partial q}{\partial q_i} \cdot \vec{v}q^{-1} + q\vec{v} \cdot \frac{\partial q^{-1}}{\partial q_i} \tag{6.20}$$

Where $\vec{v} = \vec{a}_k - \vec{g}_k$. The first term of Equation 6.20, as deduced previously, results in $\hat{e}_q \vec{v} q^{-1}$[16]. The second differentiation takes more analysis to produce a usable form.

Recall the definition of the inverse of a quaterion: $q^{-1} = q^*/|q|^2$, with q^* being the conjugate quaternion, that is, the quaternion with the vector part negated.

The chain rule can be applied once more to transform the derivative into the following[18]:

$$\frac{\partial q^{-1}}{\partial q_i} = \frac{\partial}{\partial q_i}\left(\frac{q^*}{|q|^2}\right) = \frac{\partial q^*}{\partial q_i}\frac{1}{|q|^2} + q^* \frac{\partial}{\partial q_i}\left(\frac{1}{|q|^2}\right) \tag{6.21}$$

The derivative of the conjugate is simply the conjugate of the vector basis $\hat{e}_q^*$. The second term, after careful manipulation, turns out to be the following:

$$\frac{\partial}{\partial q_i}\left(\frac{1}{|q|^2}\right) = -2\frac{q^{-1}}{|q|^2}\vec{q}_i \tag{6.22}$$

Where $\vec{q}_i$ is the vector containing the corresponding quaternion coefficients: $\begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}$. Reassembling equations 6.20, 6.21 and 6.22, the final Jacobian is obtained:

$$\frac{\partial \vec{a}_m}{\partial q_i} = \hat{e}_q \vec{v} q^{-1} + q\vec{v}\left(\frac{\hat{e}_q^* - 2q^{-1}q_i}{|q|^2}\right) \tag{6.23}$$

The Jacobian for the gyroscope $J_{\vec{\omega}_m}$ follows a similar procedure, only replacing the vector $\vec{v}$ with the angular velocity vector $\vec{\omega}$.

$$J_{\vec{\omega}_m} = \begin{pmatrix} \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \frac{\partial \vec{\omega}_m}{\partial \vec{q}_k} & \frac{\partial \vec{\omega}_m}{\partial \vec{\omega}_k} \end{pmatrix} \tag{6.24}$$

$$\frac{\partial \vec{\omega}_m}{\partial q_i} = \hat{e}_q \vec{\omega} q^{-1} + q\vec{\omega} \left(\frac{\hat{e}_q^* - 2q^{-1}q_i}{|q|^2} \right) \tag{6.25}$$

$$\frac{\partial \vec{\omega}_m}{\partial \omega_i} = \boldsymbol{R}_q \tag{6.26}$$

Finally the Jacobian for the tachometer $J_{v_{sj}}$ takes the following form:

$$J_{v_{sj}} = \begin{pmatrix} \mathbf{0}_3 & \frac{\partial v_{sj}}{\partial \vec{v}_k} & \mathbf{0}_3 & \frac{\partial v_{sj}}{\partial q_k} & \frac{\partial v_{sj}}{\partial \vec{\omega}_k} \end{pmatrix} \tag{6.27}$$

$$\frac{\partial v_{sj}}{\partial \vec{v}_i} = \boldsymbol{R}_{2,i} \tag{6.28}$$

$$\frac{\partial v_{sj}}{\partial \vec{\omega}_i} = -(\vec{r}^{\times} \boldsymbol{R})_{2,i} \tag{6.29}$$

$$\frac{\partial v_{sj}}{\partial q_i} = \left(\hat{e}_q \vec{v} q^{-1} + q\vec{v} \left(\frac{\hat{e}_q^* - 2q^{-1}q_i}{|q|^2} \right) \right)_{2,i} - \vec{r}^{\times} \left(\hat{e}_q \vec{\omega} q^{-1} + q\vec{\omega} \left(\frac{\hat{e}_q^* - 2q^{-1}q_i}{|q|^2} \right) \right)_{2,i} \tag{6.30}$$

Where the subindex $_{2,i}$ means the second row of the matrix.

6.5. Results

The State Estimator module has been implemented in Listings A.7 and A.22, and the results have been mixed. The reasons for this are several, but the major ones that have been found are the following.

Firstly, the alignment of the accelerometer sensor with respect to the body of the rover is crucial, and a tiny perturbation causes the system to become unstable. This has led us to significantly reduce the reliance on the accelerometer to the point of simply not using it in the Kalman Filter. It can nonetheless be used to estimate the initial orientation of the rover with respect to the horizon in the setup stage.

Secondly, it has been found that the Jacobians of the sensors with respect to the transformation quaternion also make the system unstable. The Jacobians have been checked to make sure the mathematical development is correct, but it still does not produce a stable system. It has finally been decided to set these derivatives to zero.

Lastly, as the wheels have been tested without the tyres, the amount of slip that the wheels undergo is slightly bigger than anticipated. This has resulted in slight overestimates with respect to the velocity and angular speed. Increasing the measurement error of the tachometers in the $\boldsymbol{R}_k$ matrix reduces the reliance of the model on the tachometers for the angular speed, which is accurately measured by the gyroscope. Since wheel slip has only happens in rotation, the linear velocity can still be relied on to integrate the path.

Chapter 7

Inter-Process Communication

The task of providing a means of communication to all the different modules is a crucial task in the well functioning of the rover. The communication link and protocol need to allow for certain flexibility, while also providing sturdiness and predictability. Several different options allow for this communication to happen, and will be discussed and compared accordingly.

7.1. Choice of Communication Protocol

Within this open set of different methods of communication, one will need to be chosen to carry this task. Considering the operating system in which this rover will be based on, three main options arise:

1. Named Pipes
2. Unix Domain Sockets
3. D-Bus

7.1.1. Named Pipes[4]

A Named Pipe, also called FIFO pipe, is a kind of simple IPC that allows for unidirectional data flow. Anyone familiar with the workflow in a Unix based system will have come across the concept of an anonymous pipe, most often simply called pipe, which allows for the unidirectional communication between two consecutive programs using the character `|`. A named pipe is simply an extension of this concept to a domain where the pipe can persist in time, even after the connected programs have terminated.

As the pipe persists as a file in the filesystem, they can be easily found and connected to by any program that has been executed before or after the pipe creation. These pipes are simply accessed by the filename and directory in which they reside.

The main drawback of this method of communication is the fact that they are a simplex form of communication, that is, data can only flow in one direction. Additionally, these pipes can only allow for one receiver, as the pipe contents are removed once they are read.

Moreover, while these pipes do technically allow for multiple senders, there is no guarantee to the data integrity and might generate data races between the senders, which would result in the output being scrambled.

Therefore, each connection between two modules would require at least one pipe if no duplex connection is required, therefore bringing the amount of pipes necessary to be proportional to n^2. This would incur severe problems in the amount of procedures that would have to be called in order for the modules to communicate properly.

7.1.2. Unix Domain Sockets[4]

In a general computing sense, a socket is simply an endpoint of a communication channel. Network sockets, for example, allow for internet networks to operate properly and provide an interface between the operating system and the network controller.

In the case of Unix Domain Sockets, which receive their name from the fact that they operate in the Unix kernel domain, as opposed to the internet domain, these allow for the communication between processes by binding a server socket to a socket file within the filesystem. Once bound, any client socket can connect to this socket and communicate in full-duplex mode to transmit arbitrary data.

Since they are not part of the internet domain, they do not suffer from common disadvantages of sockets that stem from the necessity to rely on a physical layer to transmit the messages. They do not suffer, for example, from disconnect errors or timeouts, and do not rely on the speed of the network to operate.

Since each agent in this network would probably require a socket, the amount of sockets necessary would be proportional simply to the number of modules active. This poses a great advantage when compared to the pipes, as it would reduce the amount of sockets to a factor of n.

Additionally, these sockets rely directly on the kernel, and therefore would not need linking to additional supporting libraries to make them work. This is a great advantage in keeping the code portable.

All in all, sockets offer a lot of portability and flexibility, while also keeping the maintenance of the connection relatively simple. Therefore, sockets will be used as the means of inter-process communication in the rover.

7.1.3. D-Bus

D-Bus, short for Desktop-Bus, is a Unix library that offers another option in IPC. D-Bus consists in a centralized daemon and the clients that connect to it. Each client has a bidirectional communication channel with the bus, and the bus itself handles the delivery of the messages.

The D-Bus was especifically designed to solve two specific cases. Firstly, the communication between desktop applications to allow for better integration within the desktop manager, and secondly, to handle the communication between the desktop manager and the operating system.

None of these cases are tailored to the use case at hand, so the D-Bus won't be considered in the development of the IPC systems.

7.2. Socket Protocol

Another issue that must be resolved is the choice of protocol to use, since depending on the protocol, the process of instantiation of a socket differs radically. The three main protocols in the Unix Domain are `SOCK_STREAM`, `SOCK_DGRAM` and `SOCK_SEQPACKET`.

A Stream socket is a socket that provides sequenced, reliable, two-way, connection based byte streams. A sequenced stream guarantees that the packets will be received in order. In the context of communications, a reliable protocol is one that guarantees that the packages will be delivered.

As this protocol is oriented to constant streams of data, it does not preserve packet boundaries, and as such the messages would have to be dissected by the program itself.

A Datagram socket is a socket that provides connectionless, not necessarily reliable, datagrams. As opposed to the stream socket, this does not offer a connection based communication, which means a connection cannot be kept alive and it must be the initiative of the sender to provide data to the other socket. Additionally, this requires both sockets to provide a specific connection point in the filesystem. It is worth pointing out that, while the description implies an unreliable protocol, this only applies to internet sockets. The Linux kernel guarantees order and reliability of the datagrams.

Lastly, a Sequence Packet socket is a combination of both of the above sockets. It's a connection-aware, sequenced, reliable socket that preserves packet boundaries.

All in all, the main differences which set them appart is the fact that Stream and Sequence sockets are connected while Datagrams are connectionless, that Stream and Sequence require only one named socket while Datagrams require two, and that a Stream doesn't preserve packet boundaries while Datagram and Sequence do.

Keeping these differences in mind, the most suitable protocol for this use case is the Sequence Packet socket.

7.3. Stream/Sequence Socket Workflow

Stream and Sequence sockets can be divided in two main workflows. Firstly, the server socket, which owns the socket file, and secondly the client, which connects to the socket. Each workflow will be described in more detail in the following sections.

All sockets in Linux are created using the kernel call `socket()`, which returns a file descriptor of the socket. This file descriptor uniquely identifies this socket to the process, and will be used as the identifier to operate on the state of the socket.

The call to the `socket()` function requires both the domain in which the socket will operate, and the type and specific protocol to use. For the intended use, the domain will be `UF_UNIX`, while the type and protocol implementation are `SOCK_STREAM` and `0` respectively.

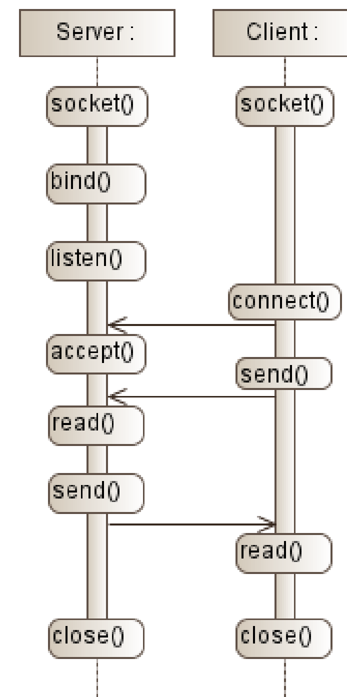


Figure 7.1: *Stream/Sequence Socket Workflow*

7.3.1. Server Socket

As stated before, the server socket owns the socket file through the kernel call `bind()`, and therefore only one server may own any socket file at any time. This call to `bind()` requires the file descriptor of the already created socket and the address of the socket file to bind to.

To make sure this `bind()` call is successful, the socket must be unbound from any other possible servers. This is necessary, since a socket will remain bound even after the server process has died. This is done with a call to `unlink()` with the address of the socket. This is not strictly necessary if there is a guarantee that the previous server has properly unlinked it before.

Once the server has been propely bound, the next step is to allow it to listen for connections. Binding simply takes ownership of the socket file, but does not necessarily put the socket in a position to handle connections. This is done through the `listen()` call: "`listen()` marks the socket as a passive socket, that is, as a socket that will be used to accept incoming connection requests using `accept()`."[8]

Once the server is properly listening to a socket file, the process of accepting connections and handling messages can proceed.

To accept an attempt of a conection from a client, the server must call `accept()`. The call to this function returns a file descriptor identifying the client socket. This file descriptor can be stored and further used to send messages to the clients.

7.3.2. Client Socket

The workflow of a client socket is simpler than that of the server, since a client doesn't have to either bind or listen to a socket. The only procedure the client needs to undergo is to send a connection request to the server using the kernel function `connect()` and the address of the server socket.

7.3.3. Duplex Communication

Once the previous steps have been taken, the calls to send and receive data from the socket are very similar for both the client and the server. To send data, the `send()` function must be called using a file descriptor. The same is true for receiving data. The function to call in this case is `read()` with a file descriptor and a buffer.

The only difference between server and client in both procedures is that the client must use its own file descriptor, while the server must use the accepted socket file descriptor.

The structure and the actual contents of each module are to be designed for each module. Nonetheless, to simplify such design, a basic generalized structure will be used for all modules, and each module will be able to send their own data formatted within this proposed structure. This structure will be discussed in the following section.

7.3.4. Termination of the Communication

In order to properly terminate the communication, both sockets require to call the kernel function `close()`. This will close all active connections in case of multiple being open and will make the socket inoperable. The server might also want to call `unlink()` to delete the socket file.

7.4. Packet Structure

In network design, a packet is a structure that consists of a header with control information and metadata, and a finite amount of data to be sent. This structure can be applied to our case, with a header that identifies the sender, the nature and structure of the package and the size of the data, and the data itself subsequently. This structure can be seen in Figure 7.2.

The first field, the Source Identifier, identifies which module is the sender of the package. This allows the socket to identify which connection corresponds to which module, and to corroborate that the sender matches the connection. This field will be represented as a pair of two characters occupying two bytes in total, and each module will be assigned a unique pair, as per Table 7.2.

The second field is specific to each module, and typifies a general reason or purpose to the message. This will be particular to each module, but a few general purpose ones are described in Table 7.3. This field will also be represented as a pair of characters taking up two bytes.

Finally, the Data Size field corresponds to an `int32_t` which represents the size of the Data Payload. This is not necessary for the IPC sockets, since Sequence Packets conserve packet boundaries, but will be useful in Chapter 8 for transmitting serial data.

Field	Source Identifier	Message Type	Data Size	Data Payload
Size (bytes)	2	2	4	<2040

Figure 7.2: *Packet Structure*

Following these fields, the data itself will be attached. The data has an artificial limit imposed by the size of the buffer which reads the packet. It has been considered that 2048 bytes is enough for all packages in this network.

The structure of this data will also be defined in a way to simplify the processing of it. A diagram of the data layout can be seen in Figure 7.3. This data structure is similar to BER encoding of ASN.1[12], but further simplified for this use case.

The "type" fields will be represented by a single character representing one of the built-in types. This will allow not only for dimensioning the following fields, but also to allow in decoding the data. Table 7.1 contains the implemented type codes that can be processed.

Type	Code
`void*`	`'a'`
`bool`	`'b'`
`char`	`'c'`
`int32_t`	`'d'`
`int64_t`	`'e'`
`float`	`'f'`
`double`	`'g'`

Table 7.1: *Type codes*

The "value" fields will contain the actual data, and will be sized accordingly to the size requirements of the data it contains.

This data structure allows for great flexibility when sending and receiving messages, as it doesn't impose any strict limitations on the type, size, order, or representation of the data.

Field	Type s1	Value v1	Type s2	Value v2	...	Type sN	Value vN
Size (bytes)	1	sizeof(s1)	1	sizeof(s2)	...	1	sizeof(sN)

Figure 7.3: *Data Structure*

Although a simpler system could have been designed, this structure allows for further modules and message types to be incorporated at a future date and allows for more flexibility in the type of data to be sent.

Module	Identifier
Ext. Comms	`"ex"`
Env. Recognition	`"ev"`
Path Finder	`"pf"`
Controller	`"cr"`
Actuator	`"at"`
State Estimator	`"st"`
Supervisor	`"su"`

Table 7.2: *Module codes*

7.5. Module Messages

As each module requires to transmit its information and to receive information from the modules it is connected to, Table 7.2 with the module codes, as well as Table 7.3 with the message purpose codes, are presented to serve as reference.

Module	Purpose	Type Code	Description
Env. Recognition	"nw"	ddgg	New computed weight and height
Path Finding	"np"	gg	New Next Position
Controller	"ws"	gggg	Linear velocity for each wheel
State Estimator	"ps"	ggg	Calculated current position
	"or"	gggg	Calculated current orientation
Ext. Comms	"ac"	ggg	Accelerometer reading
	"gy"	ggg	Gyroscope reading
	"li"	ggg	LIDAR range reading
	"ta"	gggg	Tachometer reading
	"gp"	ggg	GPS reading

Table 7.3: *Message codes*

These tables are extensible if, in the future, more modules and message types are added to extend the functionality of the rover.

7.6. Results

The IPC implementation is as shown in Listings A.11 A.14, A.4, A.16, A.2, A.15 and A.13

The implementation of this IPC configuration has been arduous, since dealing with kernel functions is not user-friendly at all, but the end result is an essentially flawless data transmission layer and protocol. This setup allows for up to 1 GB of bandwidth. It is unlikely and in fact not recommended to reach such levels as it would slow down all other essential tasks, but the possibilities are endless.

Chapter 8

External Communication

The Raspberry Pi must communicate with and control the external microcontrollers that allow for the rover to interact with the environment. This is done through the External Communication module. This module is in charge of acting as an interface between the kernel messaging link and the external serial link.

This module can be thought of as simply a proxy between the arduino device file and the rest of the system. This module standardizes the connection between external agents with incoming data and the rest of the modules that request this information. This information, in the case at hand, will be collected and transmitted by an Arduino through a USB cable. To establish this connection, the Arduino establishes a serial connection with the Linux kernel, which instantiates a character device file which a program can connect to.

8.1. Serial Protocol

To communicate with the controlling computer, the Arduino uses a serial protocol through a UART, a Universal Asynchronous Receiver-Transmitter. This allows for slow data transmission between the two systems.

The connection between the Arduino and the main processing unit is not trivial. A serial communication must first define whether the conection will be synchronous or asynchronous. Moreover, the connection must also define the baud rate, the number of data bits, the parity bit and the number of stop bits.

This configuration is usually given by a string similar to "9600/8N1". The first number defines the baud rate, that is, the number of bits transmitted per second including the parity and stop bits. The following three characters define the number of data bits per packet, the parity bit which can be N for none, O for odd and E for even, and the number of stop bits, which is usually 1 or 2.

It has been decided that the protocol characteristics to use would be **115200/8N1**. 115200 is the fastest transmission speed that Linux supports, and since the data is transmitted through a short USB cable, the parity check is set to none since the chances of bit corruption are minimal.

Setting this parameters in the Arduino environment is trivial. The function `Serial.begin()` accepts two parameters, firstly and integer which represents the baud rate, and secondly an enum representing the configuration of the packets.

Setting these parameters in the Linux environment requires a bit more work. Once a file descriptor has been obtained for the tty port using `open()`, the function `tcgetattr()` must be called with the file descriptor to obtain the current settings for this port. To change the baud rate, the

functions `cfsetispeed()` and `cfsetospeed()` are called with the corresponding enum, in this case `B115200`.

Next, the flags `CLOCAL` and `CREAD` must be set in `termios.c_cflag`. This allows the stream to listen to the incoming data. Next, the data bit number must be set. First, the flag must be cleared with the `CSIZE` flag and the correct number of bits assigned with `CS8`.

The parity bit is set by clearing or setting the flag `PARENB`, and the number of stop bits is modified with the flag `CSTOPB`.

The last flag to unset is the flag that disables Canonical mode, `ICANON`. Canonical mode, among other features, does not release the data stored in the kernel until a line delimiter character is placed in the message buffer. This is useful for human-computer interactions, but it's completely unnecessary and even counterproductive for real time applications, since it slows down the time dependent data.

Finally, to load this configuration into the kernel, the function `tcsetattr()` must be called with the file descriptor, and the buffer must be cleared of garbage data with `tcflush()`.

8.2. Packet Structure

The packet structure that will be used in the External Communication module is identical to the one defined in Chapter 7. The true benefit of adding a size field can be seen in this module, since the serial protocol does not preserve packet boundaries.

Keeping in mind that the Arduino cpu is an ATmega328p, with an Atmel AVR32 architecture, it does not allow for all the types that the x86-64 or the newer ARM architectures allow. In particular, floating point numbers and integers are restricted to a maximum of 32 bits. This restricts the types of data that can be transmitted and received shown in Table 7.1, which removes the types `int64_t` and `double`.

8.3. Sensor Data

The data the Arduinos transmit is the data collected from an accelerometer and gyroscope and the tachometers for each wheel. This data is collected in each of the three local axes for the accelerometer and gyroscope and subsequently sent through the serial link. This data is then requested from the External Communication module by the State Estimator module described in Chapter 6.

There is also data coming from the LIDAR sensor, critical for the proper behavior of the Environment Module, described in Chapter 3.

Additional data that could potentially be used is GPS and magnetometer. These provide information directly on the position and orientation of the rover, which would help keep the covariance of the measurements from increasing.

Each of these sensor readings will be firstly transmitted from the corresponding Arduino to the Raspberry Pi with the module tag `"ar"` and the purpose field corresponding to Table 8.1. Once this is received by the Raspberry Pi, the module tag is changed, the data is preprocessed and then relayed to all the connected sockets.

Sensor	Purpose
Accelerometer	`"ac"`
Gyroscope	`"gy"`
LIDAR	`"li"`
Tachometer	`"ta"`
GPS	`"gp"`

Table 8.1: *Sensor codes*

8.4. Actuator Data

Apart from the sensor data, the Arduino that controls the wheels must also receive information from the Raspberry Pi in order to properly control the speed of the wheels. This data is transmitted as well through the External Communications module. This data is then processed by the Arduino and used as a signal for the PID controllers of each wheel.

8.5. Results

In general, the module worked as designed in Listings A.5 and A.17. The reception of messages from the Arduinos worked with minor flaws, with some bytes not being transmitted, in which case the entire message is simply dropped. As the protocol has been designed to mitigate this kind of error, this did not suppose a serious malfunction of the module.

The transmission of messages generally worked properly. The one big roadblock that was encountered was that the Arduino libraries do not work as expected. This has caused serious reworking of the structure of the protocol to adapt it to the libraries, to the point of simplifying the message protocol to the bare minimum just for this task.

Most embedded systems and microcontrollers, like Mbed, provide partial support for the C++ standard library, while Arduino does not. The creators of Arduino have decided that implementing their own libraries with different mechanics from the ones in the C++ standard library would be a good idea. This causes serious incompatibilities between code designed and developed for most microcontrollers and code built for Arduinos.

Once overcome this difficulty, one minor flaw that was found was that, if the data flow coming from the Raspberry Pi towards the Arduino had a higher bandwidth than the baud rate allowed, the serial link would seize due to a UART overrun error. To solve this issue, only the most recent data is transmitted and a cooldown timer has been implemented to slow down the bandwidth through the serial link.

Chapter 9

Supervisor

The purpose of the supervisor module is to accept orders from the long range communications system and convert them to concrete instructions for each of the modules.

These instructions, for example, may take the form of stopping the motors, rescanning the environment and recalculating the path for a new objective when a "reset" order has been received. Table 9.1 proposes a few of these instructions to be implemented.

Order	Environment Recognition	Path Finding	Controller	State Estimation	External Communication
Calibrate Sensors	–	–	Stop movement	Calibration	–
First Scan	Scan Environment	–	Stop movement	–	–
Full Reset	Reset map	Reset Weights	Reset PID	Reset State Estimate	Reboot Comms
Change Objective	–	Change Objective Node	Reset PID	–	–
Hibernate	Deactivate	Deactivate	Deactivate	Low Power Mode	Low Power Mode

Table 9.1: *Order Instructions*

This module would also incorporate a single or multiple stage watchdog timer. A watchdog timer is a system by which a module can be identified as unresponsive or malfunctioning, which would trigger a reset event for that module. This makes sure that all modules in the system remains responsive and working properly at all times.

It has been considered that the current state of the implementation does not necessitate of this module, and therefore it has been decided that this module would not be implemented at the present time, since it is dependent on all of the systems being completely developed and more than thoroughly tested.

Chapter 10

Conclusions

The end result of this project is a functional control system that can be incorporated into a working rover to confer autonomous capabilities to it.

A clear and well defined architecture has been designed to allow for the interaction of all the modules with each other, all the while allowing for multiprocess computation of each of the main tasks.

The rover has been given the ability to obtain data from its environment, process it and interpret it. This data comes in the form of a point cloud from the LIDAR sensor and is then processed and interpreted to produce a traversability map. This data acquisition technique has been tested both in the simulator and on the rover and has been shown to be accurate and useful.

A few path finding algorithms have been discussed and one has been implemented to find a path from the traversability map produced by the environment sensors. This algorithm has been tested both on the simulator data and on real data and has proven useful for providing a path free of obstacles.

A controller module has been implemented with control and actuator models to properly drive each of the wheels to follow the path given by the path finding module. This control model has been implemented and tested in a simulator and has proven to be effective. This control model has also been implemented in the rover and is fully functional.

An Extended Kalman Filter has been implemented to obtain reliable data on the current position and orientation of the rover. This filter combines data from an accelerometer, a gyroscope and the wheel tachometers to produce an accurate estimate of the state variables of the rover. This filter has been implemented in the rover with mixed but mostly positive results.

Methods for the communication between modules have been designed, implemented and tested successfully. These allow for the structured transmission of data between modules. A simple serialization protocol has been designed according to the necessities of the rover. Moreover, the ability to communicate between them allows for each to run on a different process, therefore efficiently using the avaliable computing resources.

A module has been implemented to interact with the attached Arduino microcontrollers. This module allows for bidirectional communication to receive sensor data and to send actuator signals.

Lastly, a module has been described that would control the behaviour of each of the modules and make them work in unison during the different stages of the rover life cycle.

The project has an estimated cost of 7,800 €.

Chapter 11

Future Work

Based on the results obtained from the various tests of each module, the following goals could be undertaken to improve the performance of the rover.

A structure that allows for multiple buckets maps could be implemented in the environment recognition module. This structure would give information on all spatial frequencies, as opposed to the single frequency it analyzes currently. The low frequency information could be, for example, obtained from satellite altimetry, while the high spatial frequencies could be performed by stereoscopic imaging or LIDAR data. Additionally, better plane fitting algorithms could be implemented, like orthogonal least squares, and algorithms that detect outliers caused by measurement errors. New weight functions could be studied as well to find a function that suits this data better.

As for the path finding module, a new algorithm could be developed that allows for any angle traversal of the weight map, like Field D*. This would remove the limitation of travelling between nodes of the grid and would allow for the rover to take the ideal path through the map.

Different control models could be studied that would make the rover take into account forward wheel slip and loose soil terrains.

Additional sensors could be designed and installed to improve on the state estimation. These could include visual odometry, magnetometers, GPS or directional tachometers. A sensor model would have to be developed for each and properly integrated into the module.

Bibliography

[1] J. Balaram. "Kinematic state estimation for a Mars rover". In: *Robotica* 18.3 (2000), pp. 251–262. ISSN: 02635747. DOI: 10.1017/S0263574799002234. URL: http://www.journals.cambridge.org/abstract%7B%5C_%7DS0263574799002234.

[2] Pedro Batista et al. "Accelerometer Calibration and Dynamic Bias and Gravity Estimation: Analysis, Design, and Experimental Evaluation". In: *IEEE Transactions on Control Systems Technology* 19.5 (2011), pp. 1128–1137. ISSN: 1063-6536. DOI: 10.1109/TCST.2010.2076321. URL: http://ieeexplore.ieee.org/document/5594974/.

[3] Ashish C. Bhatia and Robin L. Travers. "Introduction". In: *Seminars in Cutaneous Medicine and Surgery* 31.3 (2012), pp. 151–152. ISSN: 10855629. DOI: 10.1016/j.sder.2012.07.001. arXiv: 2010(ret.29.4.2010). URL: https://www.jstor.org/stable/2583667?origin=crossref%20http://www.ncbi.nlm.nih.gov/pubmed/22929350.

[4] Ieee Computer et al. "IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004) Standard for Information Technology–Portable Operating System Interface (POSIX®)–Base Specifications, Issue 7". In: 2008.7 (2008).

[5] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://link.springer.com/10.1007/BF01386390.

[6] Ramsey Faragher. "Understanding the basis of the kalman filter via a simple and intuitive derivation [lecture notes]". In: *IEEE Signal Processing Magazine* 29.5 (2012), pp. 128–132. ISSN: 10535888. DOI: 10.1109/MSP.2012.2203621. arXiv: 1105-. URL: http://ieeexplore.ieee.org/document/6279585/.

[7] Kaiqiang Feng et al. "A New Quaternion-Based Kalman Filter for Real-Time Attitude Estimation Using the Two-Step Geometrically-Intuitive Correction Algorithm". In: *Sensors* 17.9 (2017), p. 2146. ISSN: 1424-8220. DOI: 10.3390/s17092146. URL: http://www.mdpi.com/1424-8220/17/9/2146.

[8] Linux Foundation. *Linux Manual*. 2015. URL: http://man7.org/linux/man-pages/.

[9] Basile Graf. *Quaternions and dynamics*. 2008. arXiv: 0811.2889. URL: http://arxiv.org/abs/0811.2889.

[10] K Großekatthöfer and Z Yoon. *Introduction into quaternions for spacecraft attitude representation*. 2012, pp. 1–16.

[11] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/.

[12] International Telecommunication Union. "X.690 (07/2002)". 2002. URL: http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf.

[13] Sven Koenig and Maxim Likhachev. "Fast replanning for navigation in unknown terrain". In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363. ISSN: 1552-3098. DOI: 10.1109/TRO.2004.838026. URL: http://ieeexplore.ieee.org/document/1435479/.

[14] NASA JPL. "Mars Exploration Rover". In: *NASA Facts* (2004). URL: https://mars.nasa.gov/mer/mission/.

[15] G.E. Reeves and J.F. Snyder. "An overview of the Mars exploration rovers' flight software". In: *2005 IEEE International Conference on Systems, Man and Cybernetics* 1 (2005). ISSN: 1062922X. DOI: 10.1109/ICSMC.2005.1571113. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.666.5354%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[16] Joan Solà. *Quaternion kinematics for the error-state Kalman filter*. 2017. arXiv: 1711.02508. URL: http://arxiv.org/abs/1711.02508.

[17] Nikolas Trawny and Stergios I. Roumeliotis. *Indirect Kalman filter for 3D Attitude Estimation*. Tech. rep. 2005-002. 2005, pp. 1–25. DOI: 10.2514/6.2005-6052. URL: http://www-users.cs.umn.edu/%7B~%7Dtrawny/Publications/Quaternions%7B%5C_%7D3D.pdf.

[18] Dongpo Xu et al. "Enabling quaternion derivatives: the generalized HR calculus: Table 1." In: *Royal Society Open Science* 2.8 (2015), p. 150255. ISSN: 2054-5703. DOI: 10.1098/rsos.150255. URL: http://rsos.royalsocietypublishing.org/lookup/doi/10.1098/rsos.150255.