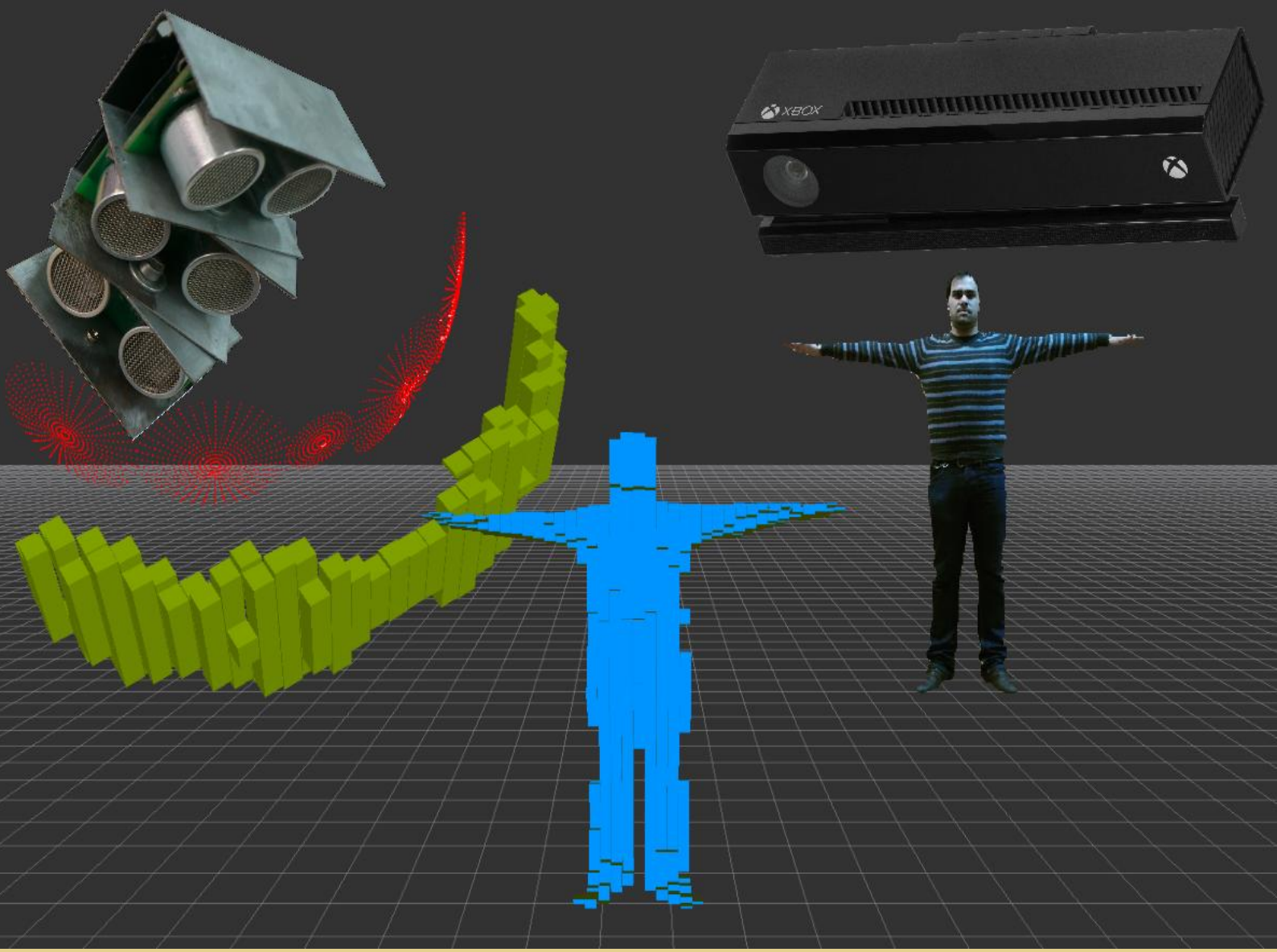


David Marques da Silva

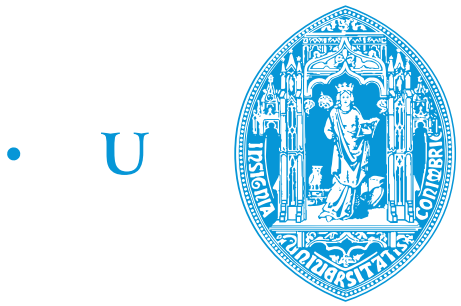
Grid-Based Representations in Mobile Robotics Using Multisensory Data

Dissertação de Mestrado em Engenharia Eletrotécnica e de Computadores (Automação e Robótica)

Fevereiro de 2016

· U C ·

UNIVERSIDADE DE COIMBRA



FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Grid-Based Representations in Mobile Robotics Using Multisensory Data

David Marques da Silva

Dissertação para obtenção do Grau de Mestre em
Engenharia Electrotécnica e de Computadores

Orientadores

Orientador: Professor Doutor Urbano José Carreira Nunes
Co-Orientador: Doutor Cristiano Premebida

Júri

Presidente: Professor Doutor Paulo José Monteiro Peixoto
Vogais: Professor Doutor João Filipe de Castro Cardoso Ferreira
Professor Doutor Urbano José Carreira Nunes

Fevereiro de 2016

NOTICE

Agradecimentos

Gostaria de começar por agradecer aos meus pais, pois sem eles não estaria aqui neste momento, disponibilizaram tudo o que tinham para a minha formação e educação. Agradeço ao meu orientador, o Professor Doutor Urbano Nunes e coorientador, Doutor Cristiano Premebida que disponibilizaram todo o material, tempo e precioso conhecimento essencial à realização deste projeto. Este trabalho foi financiado pela FCT (Fundação para a Ciência e Tecnologia) e pelo programa COMPETE (co-financiado pela FEDER), sob o projeto "AMS- HMI12:RECI/EEI-AUT/0181/2012". Agradecimentos também ao ISR (Instituto de Sistemas e Robótica), onde foram desenvolvidos os trabalhos que conduziram a esta dissertação. Um abraço aos meus amigos e camaradas de laboratório que estiveram diariamente ao meu lado fornecendo opiniões e soluções aos problemas por vezes encontrados. Um carinhoso abraço à Margarida que sempre me incentivou e apoiou mesmo quando não lhe dedicava o tempo e companhia que ela merecia.

“La perfection est atteinte, non pas lorsqu’il n’y a plus rien à ajouter, mais lorsqu’il n’y a plus rien à retirer.”

— Antoine de Saint-Exupéry

Abstract

A new approach is described for building 3D grid-based representations of the surrounding environment of a mobile robot, having the input data in the form of 3D point cloud. Such point clouds may be generated by 3D sensors like LIDAR, moving laserscanner and RGB-D sensors (e.g. Kinect). The proposed approach results in a fast method for 3D space representation. The proposed approach is based in a 2D grid. Around the 2D grid, variable sized rectangular structures are generated to map the occupancy of a given area in 3D. This method can also provide the most usual environment representations used in mobile robotics, which are: 2D occupancy grids, 2.5D grids (elevation or height maps), and the 3D voxel representation. The implementations presented here have been developed on the Robot Operating System (ROS) software thus, it can be used in the large majority of robotic platforms.

Keywords

Grid Representation, Occupancy Map, Kinect, Point Cloud, ROS.

Resumo

Nesta dissertação é apresentada uma nova abordagem para a construção de modelos de representação do espaço 3D baseados em grelhas, tendo por entrada nuvens de pontos 3D. As nuvens de pontos podem ser obtidas por meio de sensores 3D como por exemplo LIDAR, laserscanner ou sensores RGB-D (e.g. Kinect). A abordagem descrita nesta dissertação de Mestrado consiste num método rápido para a representação do espaço 3D. A flexibilidade da abordagem permite também obter as diferentes tipologias de representação baseadas em grelhas utilizadas na robótica móvel: Grelhas de ocupação 2D , grelhas 2.5D (mapas de elevação) e a representação voxel 3D. Os métodos aqui descritos foram implementados em ambiente ROS e serão disponibilizados à comunidade científica.

Palavras Chave

Representação em Grelha, Mapa de Ocupação, Kinect, Nuvem de Pontos, Robotics Operating System (ROS).

Contents

List of Figures

List of Tables

List of Acronyms

SLAM	simultaneous localization and mapping
2D/3D	Two-Dimensional/Three-dimensional
2.5D	Elevation Map
I2C	Inter Integrated Circuit
RGB-D	Red, Green, Blue and Depth
MLS	Multi Level Surface
NDT	Normal Distributions Transform
NDT-OM	Normal Distributions Transform Occupancy Map
RC	Rectangular Cuboid
MVOG	Multi Volume Occupancy Grids
SDA	Serial Data Line
SCL	Serial Clock Line
ROS	Robotics Operating System
RVIZ	Ruby Visualization Tool

1

Introduction

Contents

1.1 Motivation

The main requirement for a ground-based mobile robot is to move in its surrounding environment without collision with static or moving obstacles. In other words, a mobile robot should be able to travel and reach its destination safely, without damaging itself, other robots, harming humans or any other kind of life form that can lay in its path or moving around it. Over the years, many solutions have been proposed for environment representation in mobile robotics applications. Firstly, two dimensions maps were used, which can be categorized into topological maps that use graph structures to represent the environment. On those maps, only vital information remains and unnecessary detail has been removed. These maps lack scale, hence distance and direction are subject to change and variation, but the relationship between points is maintained. More widely used are the metric maps referenced in [Elfes, 1989, Thrun, 2003, Thrun, 2002] and [Ferreira and Dias, 2014] which capture the area (2D) or the volume (3D) of the surrounding space with metric coordinates. Merging the benefits from both maps have been studied by [Thrun, 1998]. The more common and widely used way to represent maps are the discrete Cartesian-grids which can be easily quantified and can divide the space in a regular and tractable form. Other kind of grids have been considered, such as the polar grids [Premebida et al., 2015] that are based on polar coordinates of the space to create the grid cells. This approach makes sense when using rotary sensors like laser range finders that returns the observed points in polar coordinates or in "spherical-like" coordinates. With the need of better space perception, the 2D representation have been extended to a method that also accept the height of the tallest object laying in that cell - the elevation maps or also known as 2.5D maps [Herbert et al., 1989]. However, the information provided from 2D and 2.5D maps lead to limitations from vertically overlapping features such as tunnels and bridges in outdoor environment, to tables, stairs or inclined walls indoor. Even if the height of the terrain as well as the robot were taken into account, those problems would continue to happen. A more detailed explanation of these and others representation methods is provided in chapter 2.

1.2 Objectives

The work in this thesis explores an accurate and flexible 3D representation of surrounding space for mobile robotics applications. A grid based representation seems to be the best approach since they are easy to build, represent and maintain. Such representation should also be able to provide 2D, 2.5D and 3D occupancy maps because not all applica-

tions require a 3D map and the processing of that additional data might be impossible in real-time in some robotic platforms. That information can then be used for path planning, obstacle avoidance and possible simultaneous localization and mapping (SLAM).

1.3 Main Contributions

The main goal of this work is to improve environment representation and evaluation of occupied space. The main contributions of the proposed method are:

- Ability to work with point clouds from various ROS formats such as: PointCloud and PointCloud2. Those point clouds can also be loaded from ROS bag files or standard PCD file format;
- Height definition is preserved from the source point cloud;
- The resulting data structure is scalable and flexible in the sense that 3D, 2.5D, and 2D representations can be easily obtained for future applications if needed;
- Not only height (Z) can be preserved. This method and its algorithms can also be used in depth (X) and in width (Y) if needed but the main focus of this work will rely on Z-axis.

1.4 Dissertation Outline

Diagram 1.1 demonstrates the main processing modules necessary to achieve the 3D representation developed in this thesis. This Master Thesis has eight chapters structured as follows:

- **Chapter 2:** Introduction of the existing methods of space representation and occupancy, describing them with some examples and pictures, pointing their benefits and liabilities;
- **Chapter 3:** Provides an overview of some essential algorithms to be the development of this work. It focuses on the *Sort & Unique* module as shown in figure 1.1;
- **Chapter 4:** Presents a formulation of the problem addressed in this dissertation, and details the proposed solution. The *Data Indexing* as well as the *Clustering* from figure 1.1 are detailed.

- **Chapter 5:** An overview of the hardware and software used and developed to achieve the proposed objectives are described. Moreover, experimental results are presented and discussed.
- **Chapter 6:** This chapter contains the conclusions of this work, and provides a path for future work in this field.

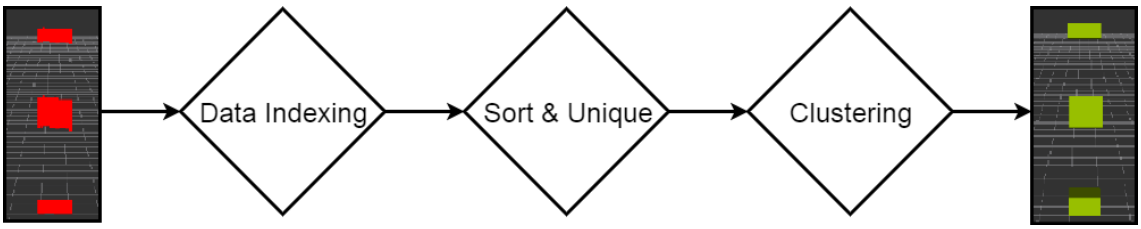


Figure 1.1: Data flow diagram from point cloud to the representation method.

2

Related Work

Contents

In this chapter some of the most used representation methods will be reviewed, pointing out their qualities and weaknesses for future usage. In particular, the section 2.3 of this chapter will give emphasis to seven different grid-based representations using 3D point cloud as input data.

2.1 Direct Representation

The direct representation of the environment can be used when depth sensors are present, such as laser scanners or stereo-cameras. This method considers the measures returned directly by the sensor to represent the environment, without extracting any characteristics about it. This type of map can be constructed simply by aggregating the measured points to lead to a representation of the environment similar to a cloud of points. The direct usage of that point cloud have been considered by [Cole and Newman, 2006] and [Nüchter et al., 2007], but this method does not model the free and unknown space. Instead, those points are directly taken from the sensor and require very high accuracy. Moreover, the size of the map grows linearly and without an upper bound with the number of sensor readings.

2.2 Topological Representation

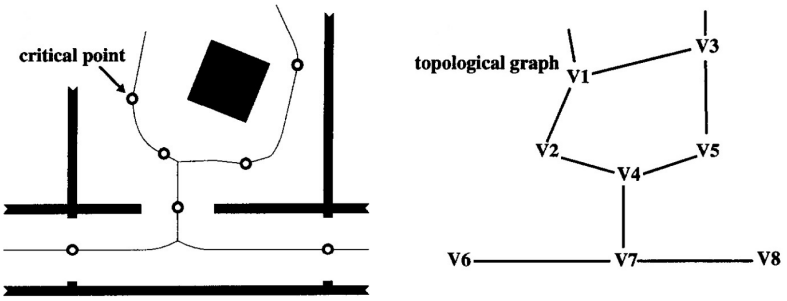


Figure 2.1: Left: Critical points, points where the clearance of all points in its neighbourhood is not smaller than the point itself. Right: Each value corresponds to a region between nodes, and each critical line to an arc. From [Thrun, 1998].

Topological approaches, represent robot environments by graphs as illustrated in Figure 2.1. Nodes in such graphs correspond to distinct situations, places, or landmarks. They are connected by arcs if there exists a direct path between them. Topological approaches determine the position of the robot relative to the model primarily based on landmarks or distinct, momentary sensor features. The key advantage of topological representation is their compactness. The resolution of topological maps corresponds directly to the complexity of the environment. That compactness gives them three key advantages over grid-based approaches,they permit fast planning, facilitate interfacing to sym-

bolic planners and problem-solvers, and, they provide more natural interfaces for human instructions. Since topological approaches usually do not require the exact determination of the robot's position, they often recover better from drift and slippage-phenomena which is a problem that should constantly be monitored and compensated in grid-based approaches. Finally, both paradigms have pro ans cons, as summarized in table 2.1.

Grid-based (metric) approaches	**Topological approaches**
+ easy to build, represent, and maintain + recognition of places (based on geometry) is non-ambiguous and view point-independent + facilitates computation of shortest paths - inefficient planning, space-consuming (resolution does not depend on the complexity of the environment) - requires accurate determination of the robot's position - poor interface for most symbolic problem solvers	+ permits efficient planning, low space complexity (resolution depends on the complexity of the environment) + does not require accurate determination of the robot's position + convenient representation for symbolic planner/problem solver, natural language - difficult to construct and maintain in large-scale environments if sensor information is ambiguous - recognition of places often difficult, sensitive to the point of view - may yield suboptimal paths

Table 2.1: Advantages and disadvantages of grid-based and topological approaches to map building. From [Thrun, 1998].

2.3 Grid Based Representations

In this type of representation, the environment is subdivided into a set of cells that form the metric grid, where the resolution depends on the cell size. The grid is usually square in shape and, when extended to three-dimensional space, it takes cubic form as we will see in subsection 2.3.4. Other shapes of grid can be considered for specific usages as presented in subsection 2.3.2. The positive and negative aspects of this representation are presented in table 2.1.

2.3.1 2D Occupancy Map

Most used 2D occupancy maps are probabilistic, this means that each cell contains a probability of occupancy, where values near to zero corresponds a free cell and values

near to one an occupied cell (with high certainty). The probability in between could result from multiple sensor readings, increasing or decreasing that probability if an object or a person was entering or leaving a cell. This method was first introduced by Alberto Elfes and Hans Moravec in [Moravec and Elfes, 1985], further development and testing were then presented in [Elfes, 1989]. Years later, Sebastian Thrun, at first with [Thrun, 2002] and then in his article [Thrun, 2003], brought contribution to this method, introducing an algorithm where the occupancy of each grid cell is dependent of its neighbours in both two dimensions. This improvement reduced erroneous readings from sensors due to noise or superposed readings, typical for example with sonars, resulting in maps more accurate than those generated using traditional techniques.

2.3.2 2.5D Elevation Map

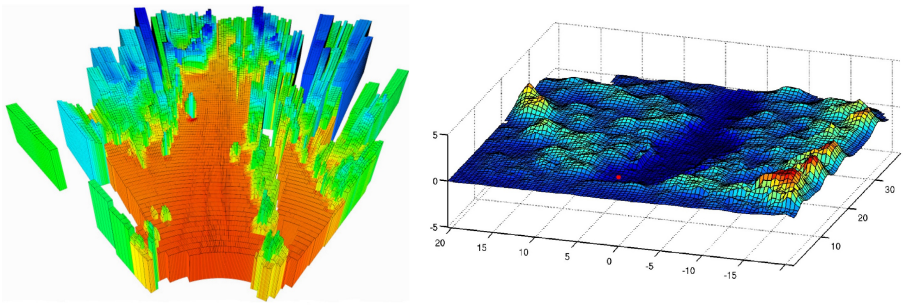


Figure 2.2: Left: 2.5D in polar grid of urban environment from [Premebida et al., 2015]. Right: Elevation map of rough terrain with space-carving kernels from [Hadsell et al., 2009].

The 2.5D or also known as elevation map was introduced to map non-flat surfaces, giving the height of obstacles or irregular terrain. This method is in all similar to the traditional 2D map but with additional information of height on each grid cell. One of the first applications of this method was intended to be for a planetary rover to Mars [Herbert et al., 1989]. Another interesting implementation of this method is presented in [Premebida et al., 2015] where polar grids were used due to the rotary laser range finder that returns points in polar or "spherical-like" coordinates. The decrease in density of points with the increase of distance, resulting from that kind of sensors is discussed and improved by [Hadsell et al., 2009] for posterior mapping of rough terrain in elevation maps. Two images of this representation are provided in figure 2.2.

2.3.3 Log-Spherical Representation

This representation, presented by [Ferreira et al., 2008], consists in a spherical representation of space centred in the sensor or group of sensors. The generated maps are

called by the author: *"Bayesian Volumetric Maps"* or (BVM). As demonstrated in figure 2.3, each BVM cell is defined by two limiting log-distances, $log_b\ \rho min$ and $log_b\ \rho max$, two limiting azimuth angles, θ_{min} and θ_{max}, and two limiting elevation angles, ϕ_{min} and ϕ_{max}. The value b is calculated and it value depend on the number of cells needed for a given application.

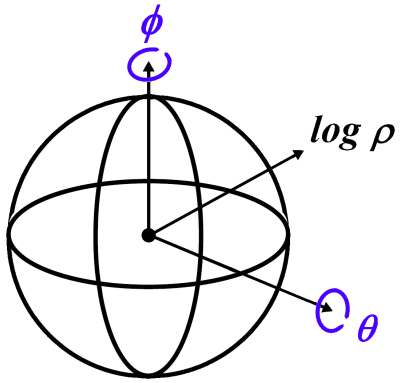


Figure 2.3: Log-Spherical space representation.

The logarithmic progression allows more precision for objects closer to the observer. A great advantage of the BVM over Cartesian implementations of occupancy maps is the fact that the log-spherical configuration avoids the need for time consuming ray-casting techniques when computing a gaze direction for active exploration, since the log-spherical space is already defined based on directions (θ, ϕ). The BVM have a strong probabilistic background for processing both empty and occupied space, map update and sensor data fusion. This approach is interesting in applications where the sensors may only rotate around the egocentric origin and the whole perceptual system is not allowed to perform any translation. Map matching may prove very difficult or even impossible if any translation of the sensor is performed. The processing of this representation and mapping method have also been applied by the authors in a Graphics Processing Unit (GPU) in [Ferreira et al., 2011], greatly improving the the time to process the data. Further development of this work can also be consulted in [Ferreira et al., 2013].

2.3.4 3D Regular Cubic

More recently, there has been some research from 2D and 2.5D maps to a 3D representation of the surrounding space, mostly using a Cartesian-grid as the reference plane. The technology based on the Velodyne and the Kinect sensors, for example, provides wider point clouds and more dense ones, respectively. Dense point clouds can also be provided by stereoscopic vision as used by [Moravec, 1996]. This factor shifted the focus of the robotics research community from 2D and 2.5D towards 3D representation and oc-

cupancy maps. One of the first to develop a partition of space into a 3D matrix of cubic voxels were [Roth-Tabak and Jain, 1989], allowing each cell with one of three different states: *0* for an empty voxel, *0.5* for unknown state and *1* for occupied. With the use of those dense point clouds a solid algorithm was needed for the update of the probabilistic spatial occupancy grid map and a good solution is provided by [Pathak et al., 2007].

In order to store multi-resolution maps, a specific data structure have been developed by [Guttman, 1984], called the *R-Tree*. The key idea of this data structure is to group nearby objects and represent them with their minimum bounding rectangle in the next higher level of the tree. The *"R"* in *R-Tree* is for rectangle. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also can not intersect with any of the contained objects. At the leaf level, each rectangle describes a single object. At higher levels the aggregation of an increasing number of objects. R-tree is a balanced search tree so all leaf nodes are at the same height. Another type of tree structure is presented by [Einhorn et al., 2011], it consists in a mapping technique that chooses the resolution of each cell adaptively by merging and splitting cells depending on the measurements. It is called N^d- *tree*, for example if *N=2* and *d=3*, means that each node or voxel in this case, is divided by *2* in its *3* dimensions, what leads to $2^3 = 8$ smaller voxels, this specific type of division is called an octree.

Octrees have been widely used over the past two decades and was firstly introduced by [Meagher, 1982]. The storage and access of this data structure can prevent useless traversal of regions of little interest, but can also add consumption both in time and space, a study of these aspects is found in [Wilhelms and Van Gelder, 1992]. An effective method to avoid numerical computation of probabilities for a range sensor with Gaussian error distribution is presented in [Payeur et al., 1997], improving that way performance and compactness. Implementations of this data structure are vast such as mapping urban environments with a stereo camera combined with a laser scanner rotating in a screw-like pattern is found in [Fournier et al., 2007] or real-time SLAM with octree for exploration of underwater tunnels with submersible sonars. A demonstration of the division of a voxel and its matching tree is provided in figure 2.4.

The octomap developed by [Wurm et al., 2010] and updated in [Hornung et al., 2013] provides an explanation of existing octree approaches, and how issues such as map update, map overconfidence, and compression are addressed. Octomap achieves probabilistic and compact 3D maps. Its tree structure allows multi-resolution and also process free space mapping. The framework is available as an open-source *C++* library and has already been applied in several robotics projects. The representation of this and other methods are illustrated in figure 2.5.

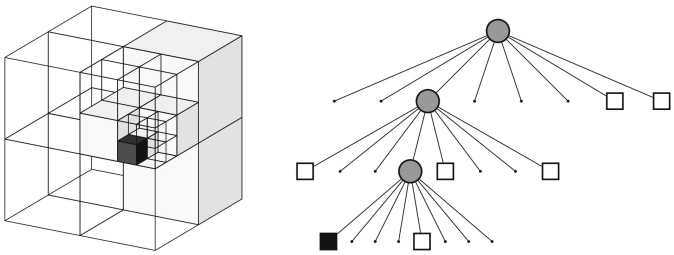


Figure 2.4: Example of an octree storing, free (shaded white) and occupied (black) cells. The volumetric model is shown on the left and the corresponding tree representation on the right. From [Hornung et al., 2013].

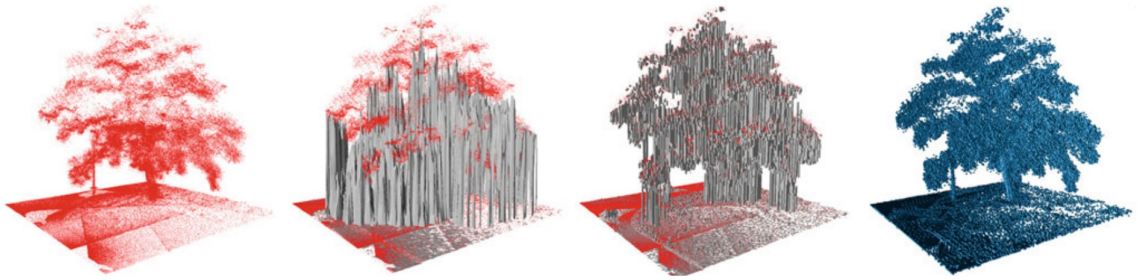


Figure 2.5: 3D representations of a tree scanned with a laser range sensor (*from left to right*): Point cloud, elevation map, multi-level surface map, and volumetric (voxel) representation of occupied space only. Image from [Hornung et al., 2013].

2.3.5 Multi Level Surface

[Triebel et al., 2006] and [Rivadeneyra et al., 2009] made an effort to create a compact data structure with the introduction of multi-level surface maps *(Multi Level Surface (MLS))*. Similarly to elevation maps discussed earlier, *MLS* maps represent 3D structures as height values over a horizontal grid, but allow for the storage of vertically overlapping objects. The vertical space on top of each grid cell is divided in intervals and those intervals are only merged if they are closer than the size of the robot, this greatly limits the application of this representation for indoor navigation. This method greatly reduces the memory requirement but it only records occupancy. It does not provide a mechanism for decreasing the occupancy value of objects located on the map nor any information about free space. Thus, any erroneous readings such as false sensor positives are never removed from the map. The third image from left to right of figure 2.5 depicts MLS representation method.

2.3.6 Normal Distributions Transform Occupancy Map

Normal Distributions Transform Occupancy Map from [Saarinen et al., 2013a] is an extension of Normal Distributions Transform (NDT), which is a grid based representa-

tion, much like the well known 2D occupancy grid map, but capable of obtaining similar accuracy while using a larger cell size. The basic idea is to first accumulate sensor measurements into grid cells and then use them to compute a sample mean and a covariance for each cell, in other words, a set of Gaussian probability distributions. NDT Occupancy Map (Normal Distributions Transform Occupancy Map (NDT-OM)) is a 3D spatial model which concurrently estimates both the occupancy and the shape distribution in each cell. This process is divided into two stages. At first it is performed an estimation of the mean and covariance over time and then an estimation of consistency of the distribution, formulated as probabilistic occupancy estimation. Further explanation and results of this method have been updated in [Saarinen et al., 2013b]. A representation of the results returned by this method are presented in figure 2.6.

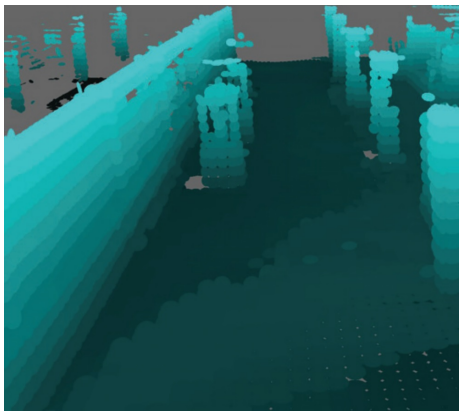


Figure 2.6: Sample from [Saarinen et al., 2013b] occupancy map of an industrial environment

.

2.3.7 Adaptive Rectangular Cuboids or RMAP

The RMAP by [Khan et al., 2014] uses axis aligned rectangular cuboids (Rectangular Cuboid (RC)). It relies on occupancy grid and on RC obtained based on point cloud density. The RMAP occupancy grid is based on the *R-tree* data structure which is composed of a hierarchy of RC in a tree structure with root, leaves and inner nodes. Initially all grid cells i.e. leaf branches of the *R-tree* occupancy grid are of cubic volume based on the chosen resolution of the grid as explained in subsection 2.3.4. The insertion of new rectangles into the *R-tree* is made through the process of least expansion. This process requires a search for inner branches in the hierarchy that lead to the minimum expansion of the minimum bounding axis aligned rectangles. The merging process of voxels only happen if they have the same probability of occupation within a threshold. When that condition is met, the merging can occur first in x direction and only in y direction if x is

not possible. Only then, z is taken into consideration if the two others directions were not possible. When merging a cuboid already merged in a previous step, it always try to assemble them increasing its bigger side if possible. More information and results about the merging process can be found in the updated version [Khan et al.,]. An image of the resulting representation is provided in figure 2.7.

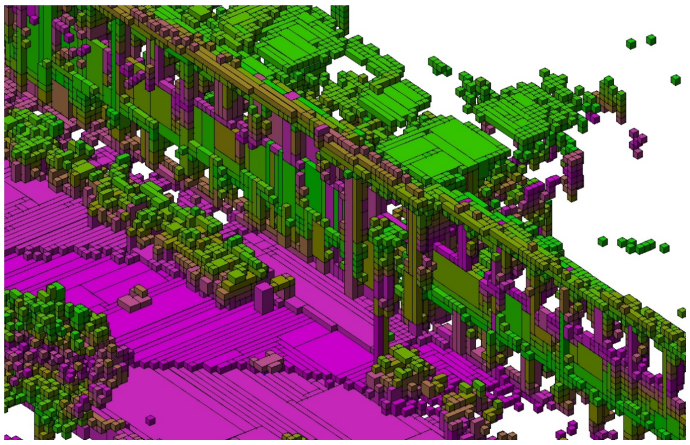


Figure 2.7: Visualization of fused occupied grid cells on the Freiburg campus dataset (colors have been assigned for the ease of visualization) From: [Khan et al.,]

.

2.3.8 Multi Volume Occupancy Grids

The model developed by [Dryanovski et al., 2010] named multi-volume occupancy grids (Multi Volume Occupancy Grids (MVOG)) is the closest work to this thesis. But being a model for micro aerial vehicles, it holds some serious limitations. It is strictly related to multi-level surface maps. Likewise MLS, MVOG uses continuous vertical volumes placed over a horizontal grid of fixed resolution. Though, unlike MLS, MVOG record both positive and negative readings, combining them into distinct positive (obstacle) and negative volumes (free space). This method uses a rasterization algorithm to obtain which grid cells a line or in this case a beam of a laser range-finder crosses. Knowing that the beam passed through that space without finding any obstacle, that volume is then considered free. Strangely a positive and a negative volume can be superposed and when that occurs, a calculation of occupancy probability is performed. Since each volume has its own "weight" of probability depending on its height.

2.4 Summary

In table 2.2 the strengths and weaknesses of the previously explained methods are summarized.

Methods	Advantages	Disadvantages
2D Occupancy Map	- Fast updability and access - Simplicity	- only applicable in planar environments
2.5D Elevation Map	- Same as 2D occupancy map	- Problems from vertically overlapping features - Only one structure per cell
Log-Spherical Representation	- Log distance allows more precision for close objects -Strong probabilistic background -Ray-tracing not needed due to its (θ, ϕ) nature	- Map matching after a translation seem difficult or even impossible
3D Regular Cubic	- Support all the algorithms developed for 2D - Tree datastructures allow multi-resolution and fast access	- Large memory requirements - Dense amount of structures depending on grid size
Multi Level Surface	- Allow fine multi-resolution on height	- Impossible to decrease occupancy - Do not map free space - Merging ability depend on aplication
NDT-OM	- Efficient mapping of large-scale dynamic environment	- Gaussian shaped structures are hard to perceive. - High number of variables per structure (11), makes it slower than octomap with the same grid size. (Information from [Saarinen et al., 2013a])
RMAP	- Might offer less consumption of resources than standard 3D due to its merging ability	- Merging ability is more favorable for x and then y and lastly z. - R-tree is a balanced tree, so most of nodes will tend to be empty, increasing research time
MVOGs	- Optimal definition of height	- Minimum height greater than or equal to 1 - The gap between any two volumes in the same list is greater than 1 - Volumes from different lists can overlap, this means that different structures can map the same space.

Table 2.2: Advantages and disadvantages of mapping methods.

3

Methods and Algorithms

Contents

In this chapter different sorting algorithms will be discussed, motivated by their usefulness in simplifying our data structure and keeping it organized in sorted order. Various methods are presented since, the efficiency of a sorting algorithm greatly depend on the order of the input data. Some of the most efficient algorithms presented here will be implemented as stated in chapter 4. On the last section, a small explanation of the unique algorithm is made, due to its ability to reduce the number of points with equal value in a given set.

3.1 Sorting Algorithms

Introduction

In this section, an overview of different sorting algorithms is provided,, pointing the benefits with regard to the approximate number of operations (O) of each method. The methods presented in the sequel are commonly used to work with lists and most of them can be extended to use with vectors.

Vectors from C++ (std::vector) are sequence containers representing arrays that can change in size. Vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container. On the other hand, linked lists (std::list) can store each of the elements they contain in different and unrelated storage locations, being that way easier to insert or delete any point on the list. That advantage also brigs the disadvantage of higher access time since the elements are not contiguous in memory.

3.1.1 Bubble Sort

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item "*bubbles*" up to the location where it belongs.

If there are *n* items in the list, then there are *n-1* pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete. At the start of the second pass, the largest value is now in place. There are *n-1* items left to sort, meaning that there will be *n-2* pairs. Since each pass places the next largest value in place, the total number of passes necessary will be *n-1*. After completing the *n-1* passes, the smallest item must be in the correct position with no further processing required.

To analyse the bubble sort, we should note that regardless of how the items are arranged in the initial list, *n-1* passes will be made to sort a list of size n. The total number of comparisons is the sum of the first *n-1* integers. The sum of the comparisons made to order the list is $\sum_{x=1}^{n-1}(n-x)$ which results in the following partial sums formula $\frac{1}{2}n^2 - \frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These "wasted" exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop.

3.1.2 Selection Sort

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n-1 passes to sort n items, since the final item must be in place after the $(n-1)^{th}$ pass.

The selection sort makes the same number of comparisons as the bubble sort and is therefore also $O(n^2)$. However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies.

3.1.3 Insertion Sort

The insertion sort, although still $O(n^2)$, works in a slightly different way. It always maintains a sorted sub-list in the lower positions of the list. Each new item is then "*inserted*" back into the previous sub-list such that the sorted sub-list is one item larger.

With Insertion Sort there are again *n-1* passes to sort n items. The iteration starts at the first position and moves through position *n-1*, as these are the items that need to be inserted back into the sorted sub-list. It performs shift operations that moves a value up one position in the list, making room behind it for the insertion. Remember that this is not a complete exchange as performed in the previous algorithms.

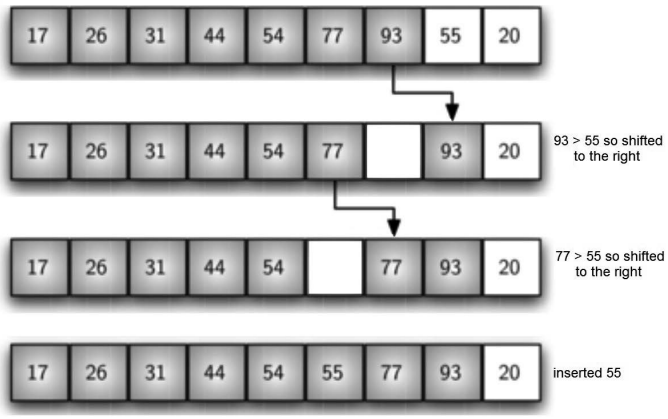


Figure 3.1: Example of insertion.

The maximum number of comparisons for an insertion sort is the sum of the first *n-1* integers. Again, this is $O(n^2)$. However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list.

One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.

3.1.4 Shell Sort

The shell sort, sometimes called the "*diminishing increment sort*", improves on the insertion sort by breaking the original list into a number of smaller sub-lists, each of which is sorted using an insertion sort. The unique way that these sub-lists are chosen is the key to the shell sort. Instead of breaking the list into sub-lists of contiguous items, the shell sort uses an increment *i*, sometimes called the *gap*, to create a sub-list by choosing all items that are *i* items apart. The way in which the increments are chosen is the unique feature of the shell sort.

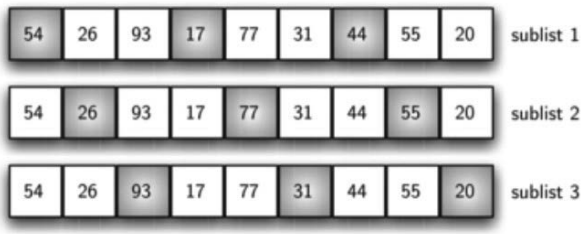


Figure 3.2: Data selection with increments of three.

One may think that a shell sort cannot be better than an insertion sort, since it does a complete insertion sort as the last step. It turns out, however, that this final insertion sort

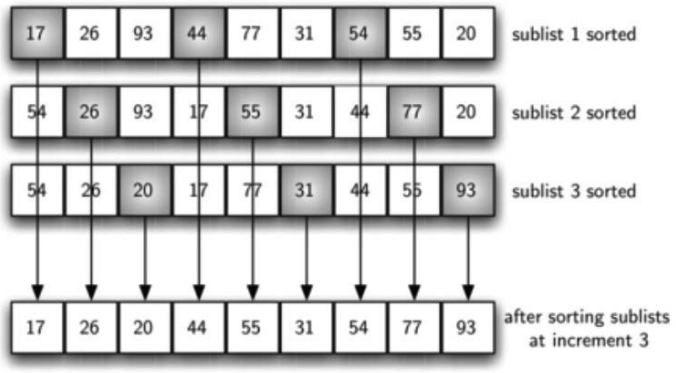


Figure 3.3: A Shell Sort after sorting each sublist.

does not need to do very many comparisons nor shifts since the list has been pre-sorted by earlier incremental insertion sorts, as described above. In other words, each pass produces a list that is "more sorted" than the previous one. This makes the final pass very efficient.

Although a general analysis of the shell sort is well beyond the scope of this text, we can say that it tends to fall somewhere between $O(n)$ and $O(n^2)$, based on the behaviour described above.

Several "gap" methods are presented in literature and we will highlight three of them for a sample with N equal to 100:

- **Shell Sequence:** $floor(\frac{N}{2^k}) < N, k \in \mathbb{N}^0$; **gaps:** [100, 50, 25, 12, 6, 3, 1] [Shell, 1959]
- **Pratt Sequence:** $(2^k < N) \cup (3^k < N), k \in \mathbb{N}^0$; **gaps:** [96, 81, 72, 64, 54, 48, 36, 32, 27, 24, 18, 16, 12, 9, 8, 6, 4, 3, 2, 1] [Pratt, 1972]
- **Knuth Sequence:** $(\frac{3^k-1}{2}) < N$, $k \in \mathbb{N}$; **gaps:** [40, 13, 4, 1] [Knuth, 1968]

3.1.5 Merge Sort

This method adopted a divide and conquer strategy as a way to improve the performance of sorting algorithms. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition. If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

First, the list is split into halves. We already computed that we can divide a list in half $log_2(n)$ times where n is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge

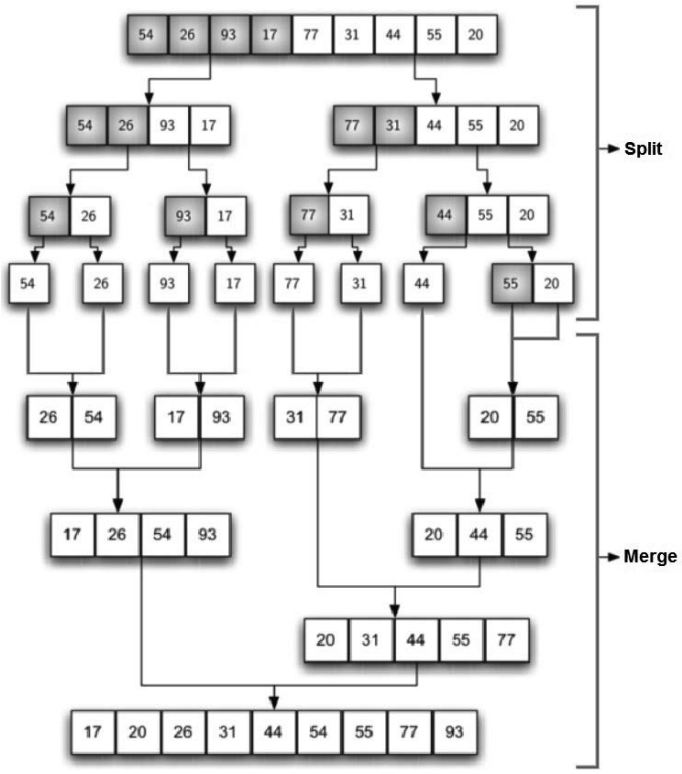


Figure 3.4: The two essential parts of merge sort.

operation which results in a list of size n requires n operations. The result of this analysis is that $log_2(n)$ splits, each of which costs n for a total of $nlog_2(n)$ operations. A merge sort is an $O(nlog_2(n))$ algorithm.

It is important to mention that an additional number of operations $O(k)$ can be added by the slicing operator k which is the size of the slice. This is possible to avoid if we simply pass the starting and ending indices of each slice along with the list when we make the recursive call.

It is important to notice that the merge sort function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the list is large and can make this sort problematic when working on large data sets.

3.1.6 Quick Sort

The quick sort also uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the pivot value. There are many

different ways to choose the pivot value, we will simply use the first item in the list as an example. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort. While finding the split point it will at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

Partitioning begins by locating two position markers, let's call them left-mark and right-mark at the beginning and end of the remaining items in the list. The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point.

We begin by incrementing left-mark until we locate a value that is greater than the pivot value. We then decrement right-mark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. We can exchange these two items and then repeat the process again.

At the point where right-mark becomes less than left-mark, we stop. The position of right-mark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

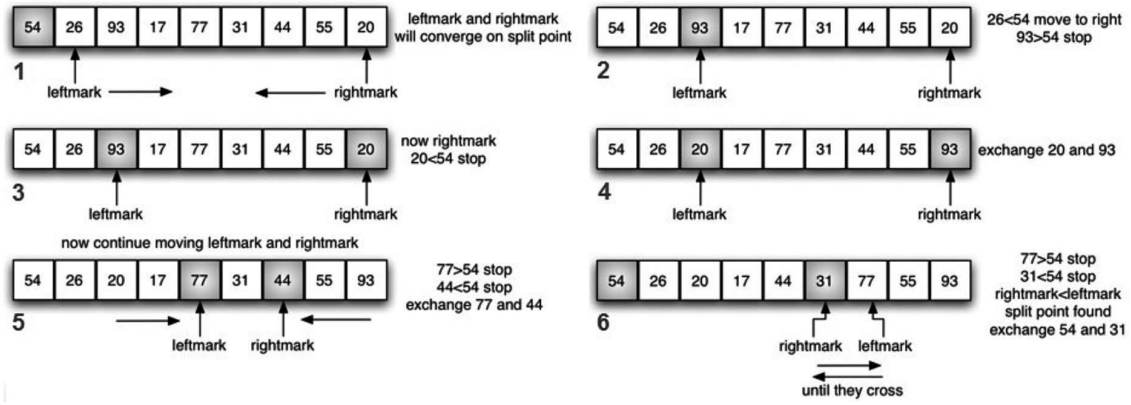


Figure 3.5: Quick sort first iteration, finding split point.

The quick sort function invokes a recursive function which begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted.

To analyse the quick sort function, note that for a list of length n, if the partition always occurs in the middle of the list, there will again be $log_2(n)$ divisions. In order to find the split point, each of the n items needs to be checked against the pivot value. The result is

$nlog_2(n)$. In addition, there is no need for additional memory as in the merge sort process.

Unfortunately, in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of *n* items divides into sorting a list of *0* items and a list of *n-1* items. Then sorting a list of *n-1* divides into a list of size *0* and a list of size *n-2*, and so on. The result is an $O(n^2)$ sort with all of the overhead that recursion requires.

As mentioned earlier, there are different ways to choose the pivot value. In particular, we can attempt to alleviate some of the potential for an uneven division by using a technique called median of three. To choose the pivot value, we will consider the first, the middle, and the last element in the list. Then we pick the median value and use it for the pivot value. The idea is that in the case where the the first item in the list does not belong toward the middle of the list, the median of three will choose a better *"middle"* value. This will be particularly useful when the original list is somewhat sorted to begin with.

3.2 Unique Algorithm

After the acquisition of a sorted vector or array, it is of great interest removing all equal values. Reducing the number of repeated values will reduce the amount of data to process. Those equal values, after the chosen sort method applied, are positioned next to each other. That way, it is easy to compare a value to its next and evaluate if they are equal. If they are, only that value is kept. If they are not, the next value is compared to its next one. Those comparisons proceed until the end of the array or list is reached. The implementations and coding, of the *Std::unique* from *C++* library has been used to perform this task since, from its definition, it removes all but the first element from every consecutive group of equivalent elements in a given vector or array.

4

Proposed Representation Method

Contents

4.1 Overview

In this chapter, a detailed description of a method for 3D environment representation is provided. The increasing density of points obtained by recent sensors turned the real time processing of those points into a very challenging task. The approach introduced here aims to reduce the input point cloud in terms of "x-y" coordinates (i.e., down-sampling with regard to the grid map), while maintaining the complete definition of the input point cloud regarding the *"z"* (vertical) elements. In order to keep the process fast, the number of operations and calculations had to be as small as possible. All implementations were carried out in ROS software and *C++* programming language. The functional block diagram, showing the information flow from the input to the final representation is given in figure 4.1. An explanation of the *"Data Indexing"* and *"Clustering"* modules (*"Sort & Unique"* was detailed in Chapter 3) will be provided in this chapter as well as it capability to generalize to other representations is demonstrated in figure 4.3.

Figure 4.1: Data flow diagram of the proposed method.

4.2 Data Indexing and Clustering

The proposed method is based on a 2D grid G of square cells C_{ij}, $i, j \in \mathbb{N}^0$ lying in the xy-plane. Any point $p = [px, py, pz]^T$, $p \in \mathbb{R}^3$ from the point cloud is projected onto a grid cell C_{ij}. Two situations may happen: cells without points, or cells with one or more points. In order to store more than one point on a cell, each C_{ij} has a corresponding array $A_{ij} = \{A^0, A^1, \cdots, A^n\}_{ij}$, where n is the number of points on that given C_{ij} cell. In order to index the point cloud coordinate system (world reference system) values to our grid and vice versa, the functions expressed by 4.2.1, 4.2.2, and 4.2.3 are considered.

$$Val = \left(floor\left(RawData \times \frac{1}{cellSize} \right) \times cellSize \right) + \frac{cellSize}{2} \tag{4.2.1}$$

The equation 4.2.1 collects a point coordinate X or Y in the point cloud (referenced as RawData), one by one, and maps them into their respective cell centre *Val* in the world coordinate system.

$$Coord = floor\left(\frac{Val + Centre}{cellSize} - 1 \right) \tag{4.2.2}$$

Equation 4.2.2 calculates the cell address (i.e., the integer and unique location in the grid) from the real value obtained in 4.2.1. This address is used for storage in the "3D vector", which is in this implementation the combination of three C++ "std::vector" as follows, $V \in vector<vector<vector<float>>>$, so a point A^m from cell C_{ij} can be accessed with V[i][j][A^m]. Since the first address of a vector is 0 *(zero)*, one unit (-1 in equation 4.2.2) is subtracted in order to use that space in memory. *"Centre"* is the coordinate of the sensor with respect to the world reference system. So, *Coord* can either be i or j of C_{ij} if *Val* is X or Y respectively. All Z values are stored in their respective A_{ij} vector without any modification.

$$Val = ((Coord + 1.5) \times cellSize) - Centre \tag{4.2.3}$$

Finally, equation 4.2.3 is the opposite of equation 4.2.2 and it recovers the cell centre value in the world coordinates from its indexes i or j in the array A_{ij}. The added 1.5 value is needed because of two operations performed in equation 4.2.2: The floor operation lead to the addition of 0.5 plus 1 unit since it was previously subtracted for reasons already explained.

It is important to note that the above explanation focuses on the accuracy of height, that is, the Z coordinate. Furthermore, these equations and associated functions are also prepared to maintain the accuracy of X or Y coordinates if needed. This will be demonstrated further in this thesis.

At this stage, it is possible to store, access and represent the points centred in their respective grid cell without losing any resolution with regard to the height, i.e. the points in Z-axis. Even so, the number of stored points is the same as the input. By looking at the points within the same cell (in an array A_{ij}), it was found that a considerable number of them had the same height value, due to the high density point cloud. Although originally they had different X and Y values, they were different points but when centred in a cell, they are equal. Those equal points have to be removed to preclude that the same points are processed as many times as they are present in the structure. For that purpose the *Std*::*unique* (see section 3.2) is used but, by its definition, it only removes elements that are equal and next to each other in a given vector. In other words, the array of values need to be sorted before to be processed by *"Unique"* function.

As stated in section 3.1 the selection of the sorting algorithm, greatly depends on the data to sort and how it is arranged in first hand. It is known, for instance, that the points from the kinect sensor are returned from top to bottom and left to right of the image. If the points on the top are the first into the A_{ij} structure, it is expected that those points have

higher Z values (or height). But this information cannot be relied upon because it can be flawed by empty spaces (due to: obstacles, objects, missing points) but it can be assumed that the points are mostly in descending order. The chosen algorithm was the *Shell Sort* with *Pratt* sequence because it is simple to implement and can achieve in the worst case ($n \times log_2(n)$) operations, being n the number of points to sort. Other sorting algorithms have also been implemented during this work, namely:

- Shell Sort with Knuth, Pratt and Shell sequence.
- Merge Sort.
- Std::sort implementation of a relatively standard quick-sort. Available directly from *C++* libraries.
- Timsort: A hybrid stable sorting algorithm, derived from merge sort and insertion sort. It is standard for python language and also used by *Java*.

It is important to note that even if the total number of points is considerably large, the number of points in each array is considerably smaller, and depend on the cell size chosen. This means that n is relatively small, hence the effectiveness of the sorting algorithm will bring small differences in time versus another algorithm. Anyway even if an improvement is small it might sum with others to achieve a greater improvement. After an array containing the Z values is sorted, the *unique* algorithm removes all duplicate values in that given array.

This sorting wasn't just needed to remove the duplicate values, in fact it will make further processing easier and more efficient. At this moment it is important to remember that each cell $\in G$ has its "vertical" (Z-axis) elements given by the set A, where in case A is not empty we have $\{A_0, A_1, \cdots, A_{n-1}\}$, with $A_i = z$. In short, we have a set of sorted z-values elements per cell. In order to obtain the proposed representation, a clustering method is used in order to extract a set of clusters from A. For that to happen, we will cluster points next to each other on each array. Since the points are sorted in the array, while they satisfy the condition given by 4.2.4 the first given point A_{ij}^0 will expand point by point until the next are further than the tolerance given by the user. At that moment the structure expand from A_{ij}^0 to A_{ij}^m. The next structure will begin at $A_{ij}^m + 1$ and the process is repeated.

$$if\ (|A_{i+1}| - |A_i|) \leq tolerance \tag{4.2.4}$$

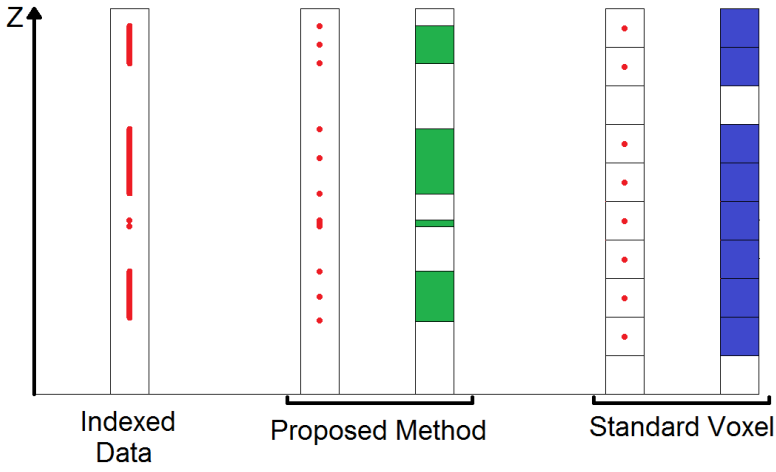


Figure 4.2: Illustration of the method compared to standard voxel representation.

Only three points are stored back to the array from each resulting structure, the bottom, middle and top Z values. This is shown in figure 4.2. Those values are needed to form the visualization polygon. The tolerance referred in condition 4.2.4 is chosen by the user and should take into consideration the density of points provided by the sensor.

For example, for the first structure of the array, the points are stored back to it in the following order:

- A_{ij}^{0} is the first point of the sorted array.
- $\frac{A_{ij}^{m}+A_{ij}^{0}}{2}$ corresponds to the middle point.
- A_{ij}^{m} is the last and higher point of the structure.

Figure 4.3 demonstrates the functions developed to achieve various representations of the environment. Images of points are also provided in that figure, those points are key structural features to achieve those representations.

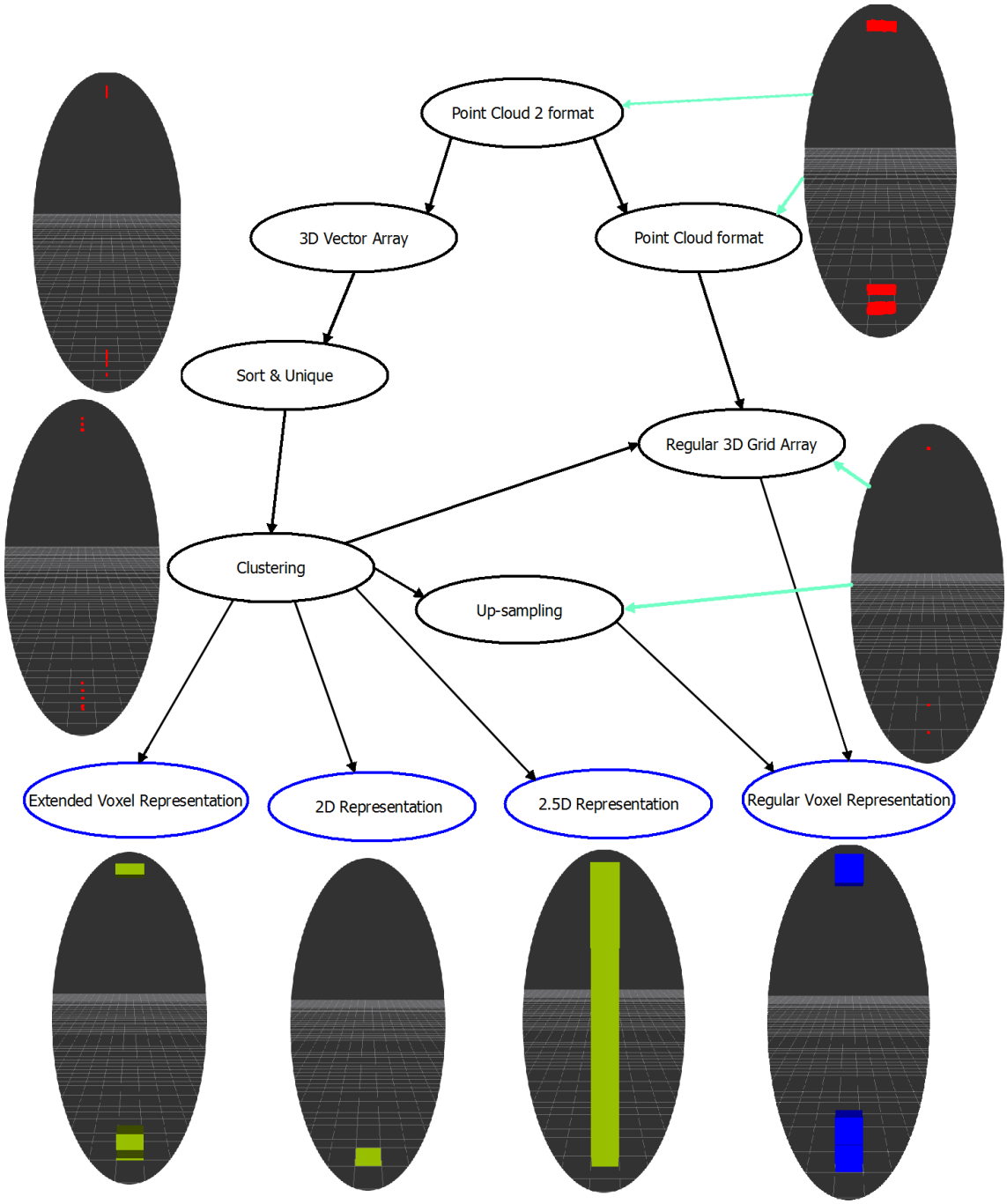


Figure 4.3: Diagram of the developed and implemented functions (in ROS), and their representation.

5

Experiments, Tests and Results

Contents

The following tests have been performed on a laptop with an Intel I7 processor that has 4 cores, 8 threads at a maximum of 3.5 GHz per core. It is important to state that this thesis work is intended to be mounted on a robotic wheelchair as referred in appendix chapter 9.

5.1 Sensors for Experiments

One of the main requirements to any mobile robotic platform is to travel through space safely and, in order to ensure this, a complementary sensor based solution was adopted. Specifically, two complementary sensory technologies were chosen, an infra-red depth camera and six ultrasonic range finders. A discussion about their main characteristics is found in this chapter as well as an explanation about their assembly set-up.

Sensor	**Sonar SRF04**	**Sonar SRF08**	**Kinect v1**	**Kinect v2**
Refresh Rate	$\sim$22[Hz]	$\sim$15[Hz]	30[Hz]	30[Hz]
Field of View	$\sim$60°	$\sim$60°	57°H. / 43°V.	70°H. / 60°V.
Depth Resolution	$\pm 1cm$	$\pm 1cm$	640 × 480	512 × 424
Image Resolution	-	-	640 × 480	1920 × 1080
Range *(approx.)*	3cm to 3m	3cm to 6m	60cm to 4m	3cm to 4.5m

Table 5.1: Main sensors specifications

5.1.1 UltraSonic Ranger

An ultrasonic ranger works by transmitting a pulse of sound outside the range of human hearing, at *40khz*. This pulse travels at the speed of sound (roughly *340 m/s*) away from the ranger in a cone shape and the wave reflects back to the ranger from any object in the path of the sonic wave. The ranger pauses for a brief interval after the pulse is transmitted and then awaits the reflected wave in the form of an echo. The controller driving the ranger then requests a ping, the ranger creates the sound pulse, and waits for the return echo. If received, the ranger reports this echo to the controller and the controller can then compute the distance to the object based on the elapsed time.

Both *SRF04* and *SRF08* are from *Devantech* and are presented in 5.1, they have the same size and transducers. They rely on the same working principle as described above, but that is where the similarities end. The newer version have a built-in light sensor on the front. It communicates through an I2C bus, which allows sixteen addresses and therefore sixteen different devices. Since it has a *PIC16F872*, the user do not need to use a timer on

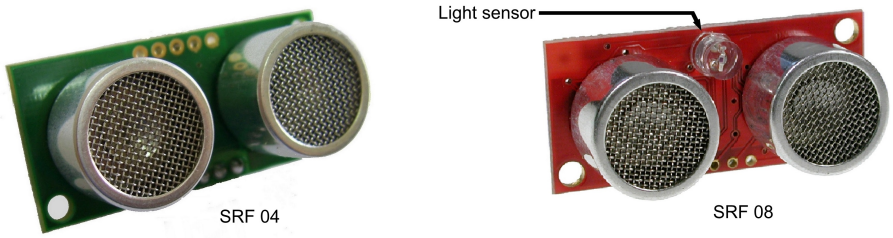


Figure 5.1: The two models of ultra sonic range finders used.

his own controller to wait for ranging to finish as in the *SRF04* model. The range value is provided in inches, centimetres or flight time in *uS* after each ranging that value is stored in a specific register. Some registers are available for the user to write into, allowing him to increase the range up to eleven meters and also decrease it. The main reason to reduce the range is to get the range information quicker and by doing so, improve the refresh rate stated in the table 5.1. After that, we must not forget to also reduce the gain of the transducer because it is possible to receive an echo from the previously emitted pulse, resulting in erroneous values. Another way to avoid that from happening is to reduce the time the *SRF08* listens for an echo.

This type of sensor is interesting because it is small, cheap and, have low power consumption (*250mW* for the *SRF04* and *75mW* for the*SRF08*). Due to the conical shape of the detection area, it is difficult to locate with precision an object or obstacle far away from the sensor. In section 5.2, a method to minimize that effect will be presented.

5.1.2 Kinect Depth Sensors

This type of sensor referred as RGB-D device, since they provide coloured image and depth information within their field of view. Those two sensors are presented in figure 5.2. The main characteristics of both sensors were previously stated in table 5.1 but there are some relevant aspects yet to be discussed. Both sensors emit infra-red light for acquiring depth. The first version relies on a method called structured light while the second version uses a time-of-flight principle. The structured light method project a known pattern onto the scene and infer depth from the deformation of that pattern, while time-of-flight provides distance based on the known speed of light, measuring the time of an infra-red light signal between the camera and the subject for each point of the image. Detailed information about both techniques, comparison of sensors, testing and results are provided in *"Kinect Range Sensing: Structured-Light versus Time-of-Flight Kinect"* by [Sarbolandi et al., 2015]. Both devices were meant to be used directly with the gaming platform *"Xbox"* and, for that reason, little open- source solutions are available for researchers. A software called *"libfreenect"* from *"OpenKinect"* were used. It is an open

source driver package, used to test the first version of the sensor. In order to calibrate and test the second version of the Kinect device, a package of tools called *iai_kinect2* from [Wiedemeyer, 2015] were used. That completely open- source package allowed the modification of the frame rate and other parameters such as resolution:

- **High definition (HD):** *1920×1080* points
- **Quarter High definition (qHD):** *960×540* points
- **Standard Definition (SD):** *512×424* points

The higher definitions were obtained by up-sampling and interpolation as informed by [Wiedemeyer, 2015] and the results are really satisfying. As the mapped area increase with distance, with the native resolution, objects placed further away the sensor had low definition. This huge increase of resolution lead to choose this second version of the sensor for further experiments.

Figure 5.2: The two models of *Microsoft Kinect* devices available.

5.2 Sonar Array Assembly

The tests provided in appendix chapter 8 lead to the conclusion that the area covered by one sonar was too large and the associated uncertainty was too high. On the figure 5.3 five different zones are shown. They are formed by three sonars on top of each other and rotated by 30^{o} relatively to the one in the middle. The 30^{o} were chosen to minimize the area of each zone.

To simplify the next explanation of this method the red area will be referred as *sonar 1*, the blue area as *sonar 2* and the green area as *sonar 3*. The chosen way to merge the readings, in order to minimize the detection areas from the three sonars to these five detection zones is performed in the following manner:

- **Zone 1:** Sonar 1, reading from sonar 2 & 3 are not within sonar 1 reading $\pm$ threshold.

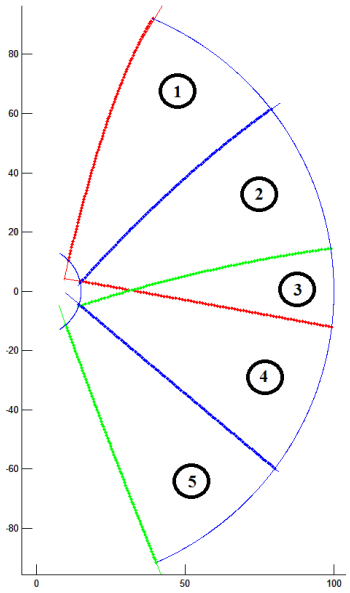


Figure 5.3: Depiction of the five zones obtained with three sonars.

- **Zone 2:** Sonar 1 & 2, reading from sonar 1 & 2 are equal ± threshold.
- **Zone 3:** Sonar 1, 2 & 3, reading from sonar 1, 2, 3 are equal ± threshold.
- **Zone 4:** Sonar 2 & 3, reading from sonar 2 & 3 are equal ± threshold.
- **Zone 5:** Sonar 3, reading from sonar 1 & 2 are not within sonar 3 reading ± threshold.

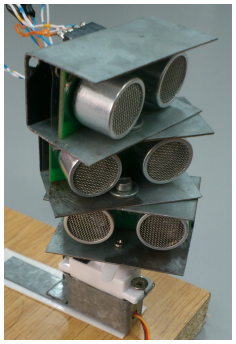


Figure 5.4: Image of one of the two arrays of three sonars on top of a servo.

It is important to state that the zone number three is the smaller one and it is the area where the three sensors record the same range within a threshold. To maximize the usage of that zone, the above mentioned set of three sonars are put on top of a servo that allows a rotation of 180 degrees, that way, it will always be moving clockwise or counter-clockwise to the nearest measurement taken, to maximize the usage of that zone.

The array of sensors and the servo used can be seen in figure 5.4. It is also important to mention a case that can lead to wrong readings with this approach. For example if the sonar array is located inside a cylindrical structure, all sensors will return the same reading within a threshold which will lead to the assumption that the obstacle is located in region number 3. That assumption is wrong and all 5 zones should be considered in that particular case. Due to the lack of more precise information provided by this type of sensors, there is no way to correct that assumption.

The testing layout as well as the communication protocols for the experiments performed is depicted in figure 5.5.

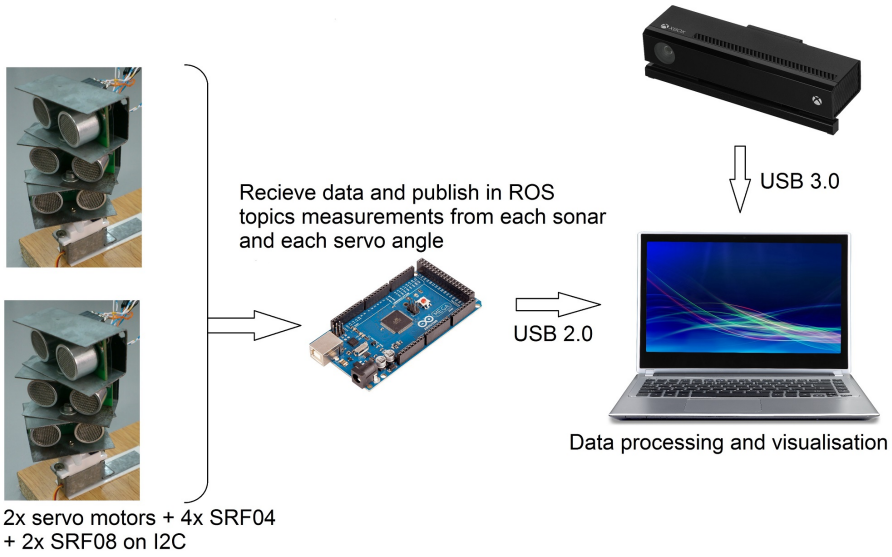


Figure 5.5: The testing layout for software testing and implementation.

Each devantech SRF04 need four wires in order to operate, power, ground, Serial Data Line (SDA) and, Serial Clock Line (SCL) while the SRF08 can share the SDA and SCL with other devices thank to his Inter Integrated Circuit (I2C) communication protocol.

5.3 Software Overview

This work has been developed under *Ubuntu* operating system, based on *C++* programming language and using ROS messages and data structures to record, store and process datasets. The key ROS modules used for experimental implementations and results reported in this Thesis are:

- **Publisher/Subscriber:** A publisher can send all types of messages available to ROS to the core node, then, any other node can request to subscribe to that topic to see and use those messages.

- **sensor_msgs**::**PointCloud:** This message holds a collection of 3D points $p = [px, py, pz]^T$,$p \in \mathbb{R}^3$, plus optional additional information about each point such as colour. That information is easy to access and modify as needed.
- **sensor_msgs**::**PointCloud2:** This message holds a collection of N-dimensional points, which may contain additional information as the *sensor_msgs*::*PointCloud*. The point data is stored as a binary blob, which makes it more efficient but may become more difficult to modify and access information.
- **visualization_msgs**::**Marker:** This message is used to represent geometric shapes such as arrows, cubes, cylinders, spheres and even text. Other fields such as color, position and lifetime are also available. The *visualization_msgs*::*MarkerArray* allows the storage and representation of numerous messages of this type.
- **tf**::**Transform:** All represented object or group of objects need a position and orientation, this structure allow movement and rotation in 3D space.

Let us now see and explain the interaction between *nodes* (i.e., the programs and the messages) that are published by those *nodes*. It is required that all ROS applications have a *"ROSCORE"*, that is, a collection of nodes and programs that are pre-requisites for the ROS-based system to oversees the remaining nodes and make communications possible between them. This interaction is seen in figure 5.6 by the black bi-directional arrows. The blue arrows point from the origin of the message to the node receiving it for further processing or actual usage.

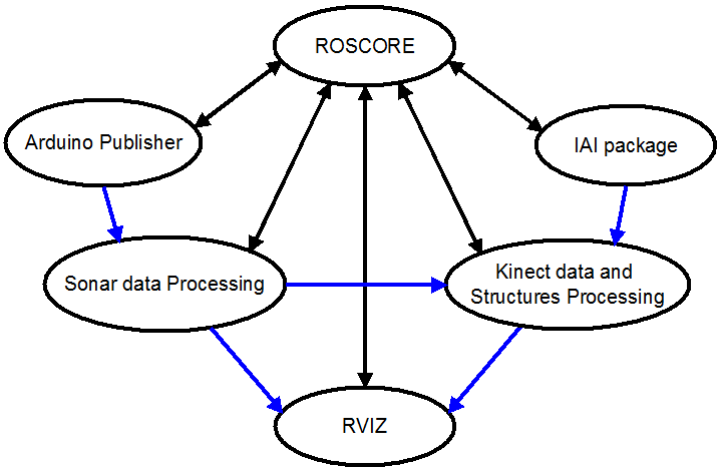


Figure 5.6: Diagram showing interaction between nodes

The *IAI package* node returned the point-cloud from the kinect sensor in **sensor_msgs**::**PointCloud2** format. Due to the difficulty to access and keep this data orga-

nized, an implementation of various functions were essential to transform it to **sensor_msgs**::**PointCloud** and three dimensional vector of *float* numbers in order to work with it more easily. To represent those vectors in Ruby Visualization Tool (RVIZ) or to store into file, the vectors can be transformed to *Point Cloud* and then to *Point Cloud 2* if needed.

Other functions to load or store *".pcd"* files containing datasets in Point Cloud format were implemented. Another function to load *".bag"* files previously recorded are also available. The called *".bag"* files have an important role in ROS, and a variety of tools have been written to allow users to, process, analyse, and visualize them in RVIZ for example. With all the data available, a way to process that amount of information in real-time were needed. The *Kinect2* sensor returns through the *IAI node* a maximum of $1920 \times 1080 = 2073600$ points. Each one of these points has three values associated { X, Y, Z } and that number of points occur 30 times per second (30Hz). It is easy to conclude that a simple laptop computer are unable to process such number of points at that rate. Not willing to lose information about precision, resolution were maintained, but the number of frames per second had to be shortened. Because of that large amount of information, the method presented here needed to be simple, with as low number of operations as possible and yet without losing definition in at least one of the three dimensions. It is this and only this reason that lead to the development of the method proposed.

Variables such as grid size, cell size, robot and sensors positions in that grid are inputs provided by the user to the program. With those relative positions and dimensions it is possible to map the points to their respective cell centre maintaining their height untouched.

5.4 Sonar Point Cloud Calculation

One of the first implementations made was a way to minimize the area of detection of the sonars or array of sonars to settle an hardware configuration as described above. Since those sensors detect obstacles in a conical shape and in 3D coordinates, a way to represent that area was needed for further processing of this representation method. With the help of equations 9.0.4 and 8.0.2 it was possible to define with precision each area or zone shown in figure 5.3. Various look-up tables are present in the code of the node called "Sonar data Processing", those tables have information about the start angle of each zone and its aperture, both under and over 15 centimetres and, always under 1 metre from the sensor. After the reading of the three sensors, the conditions given above are used to process in which zone is the obstacle. From the readings, the smaller distance d is stored. Knowing the distance d in centimetres and the zone, an access to the lookup tables is made to retrieve the starting angle of that zone α and the aperture of it, given by the angle θ. The value *ppc* means points per circle and is chosen by the user.

Input: d, θ, α, ppc
Output: Point Cloud
1: i=0
2: Do while i < θ/2
3: j=0
4: Do while j < ppc
5: auxX = d × $\sin\left(90 - \frac{\theta}{2} - i\right)$
6: auxY = d × $\cos\left(90 - \frac{\theta}{2} - \mathrm{i}\right)$ × $\sin\left(\frac{360}{\mathrm{ppc}} \times \mathrm{j}\right)$
7: Z = d × $\cos\left(90 - \frac{\theta}{2} - i\right)$ × $\cos\left(\frac{360}{\mathrm{ppc}} \times \mathrm{j}\right)$
8: X = $\cos\left(\alpha + \frac{\theta}{2} - i\right)$ × auxX - $\sin\left(\alpha + \frac{\theta}{2} - i\right)$ × auxY
9: Y = $\sin\left(\alpha + \frac{\theta}{2} - i\right)$ × auxX + $\cos\left(\alpha + \frac{\theta}{2} - i\right)$ × auxY
10: j=j+1
11: i=i+1

Figure 5.7: Calculation of points coordinates $\{X, Y, Z\}$ to create a Point Cloud.

The lines 5 to 7 of figure 5.7 demonstrate the actual creation of the points around the X axis while lines 8 an 9 are a rotation of those points around the Z axis for them to be placed in their belonging zone. This process results in a point-cloud of the zones where an obstacle is detected, the five zones are presented in figure 5.8.

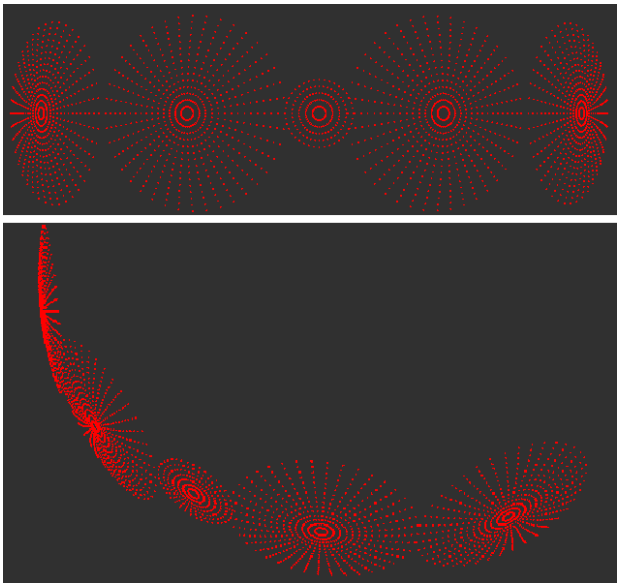


Figure 5.8: Point Cloud calculated from the area of detection of each zone. With ppr=40.

5.5 Results

This section reports results from experiments carried out on various real sensory data used to validate and to illustrate the implementation discussed in chapter 4. The computer

used in the experiments allowed to process around six frames per second, the images from the Kinect sensor are 1920 $\times$ 1080 pixels and each pixel has three associate coordinates X, Y and Z. The *IAI node* also process the color of each point even if it is not used in this proposed representation method. The images in this chapter were obtained using a "5 $\times$ 5" centimetres cell size unless stated otherwise.

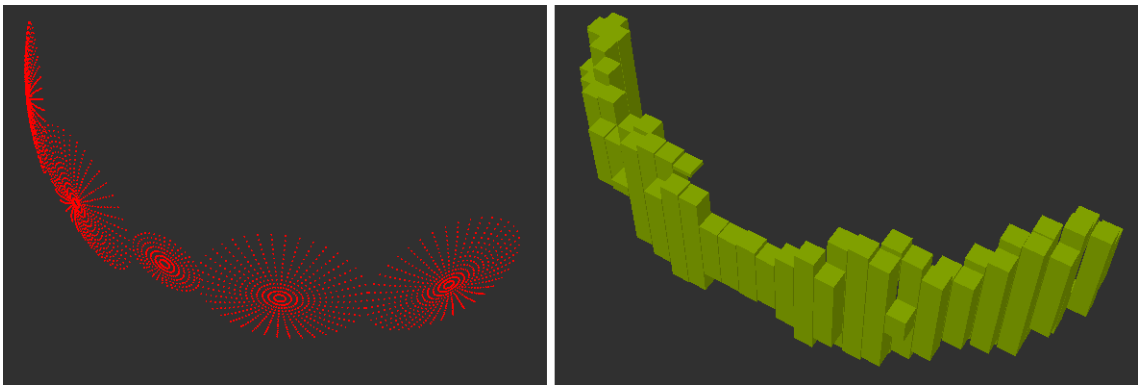


Figure 5.9: Representation of the *Point Cloud* provided by one of the sonar arrays

Figure 5.10: A simple wall: Original point cloud from Kinect 2 sensor on the left, variable height voxels on the middle and standard voxels on the right.

The original point cloud from figure 5.10 has 1.603.171 points. For the variable adaptive voxels, only 111 points are needed, 3 points per vertical structure. For the regular voxel representation, 1.000 points were needed, one per voxel and note that more voxels were needed to map residual points (on the bottom of right image).

Figure 5.11 demonstrate the ability of this method to represent different surfaces over the same grid cell while keeping the definition given by the sensor. All structures should have the same height but due to noise characteristic of Kinect sensors, it is not the case.

The figure presented in 5.12 demonstrate that the method explained in this thesis can also be exported to outdoor spaces and with *Velodyne* sensors. The point cloud is from a *"KITTI"* dataset provided by [Geiger et al., 2013].

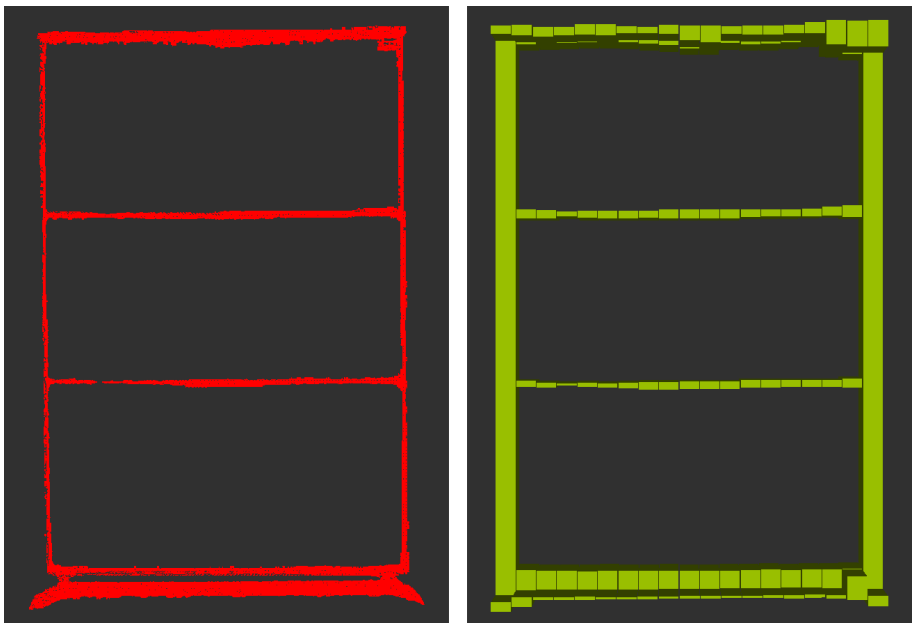


Figure 5.11: Front part of a bookcase.

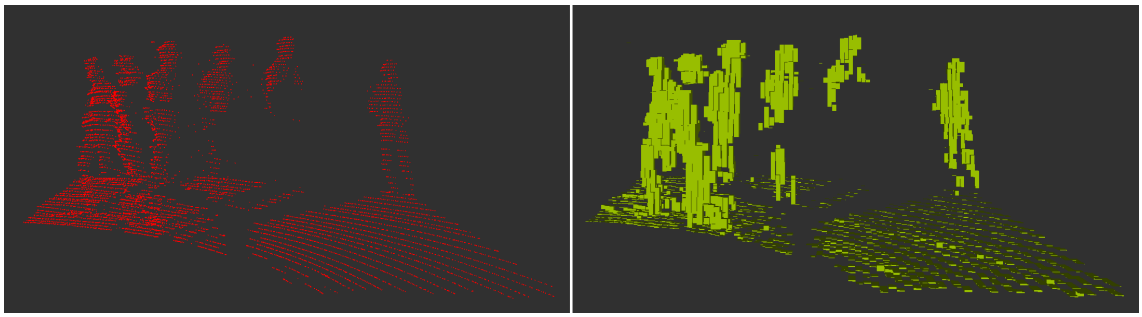


Figure 5.12: Six persons in a outdoor environment.

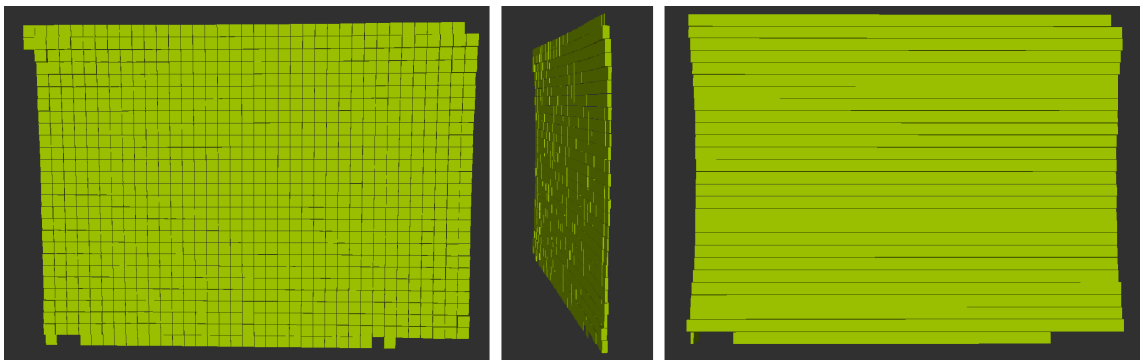


Figure 5.13: Keeping sensor definition in X coordinate in left and middle image and Y definition in right image.

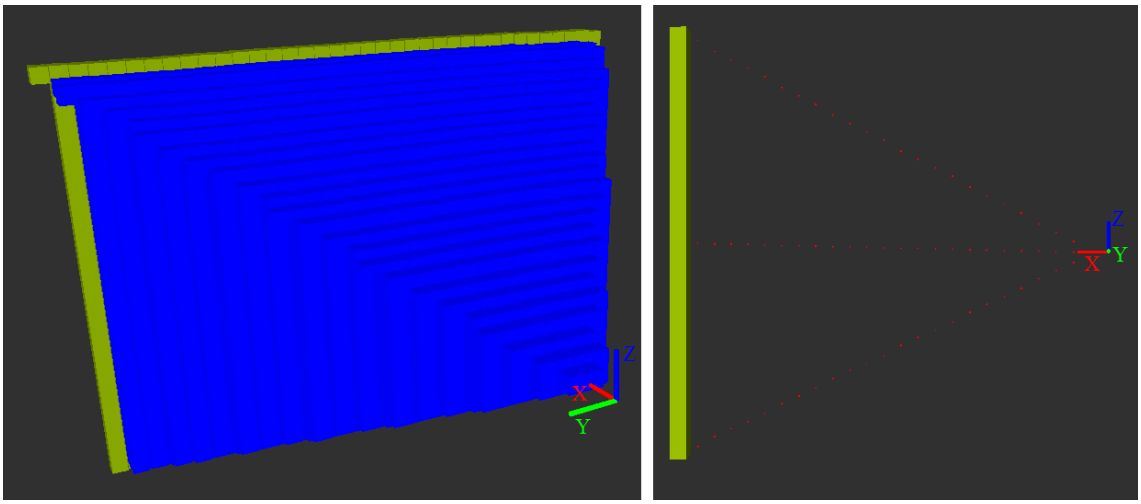


Figure 5.14: **Left:** Free space in blue. **Right:** Points representing the structure of the free space.

An attempt to map the free area between the sensor and the obstacles is demonstrated in the left part of figure 5.14. Two lines are projected from the center of the sensor to the top and and bottom of each occupied volume. From those lines, only one point per line and per cell is kept as shown in the right part of figure 5.14. The middle point is calculated from the top and bottom ones, that point represent the centre of each structure that depicts a free volume (in blue). This algorithm present some overlapping between free structures when multiple obstacles lie in depth (X direction). This issue will be referred in section 6.1 of the next chapter.

The table provided in 5.2 quantify the points and structures presented in the figures from 5.15 to 5.17. The mean of *Variable Height Voxels* needed to represent the nine samples is 183, for the *Standard Voxels* an average of 659 voxels are needed to represent the same volumes but with less definition. Since a *Variable Height Voxel* need 3 points per structure, an average of "183 $\times$ 3 = 549" points are stored for the method proposed here versus 659 for the *Standard Voxels*. So in average, this method uses less memory since less points are stored and always need less structures to represent the same volume and with higher precision.

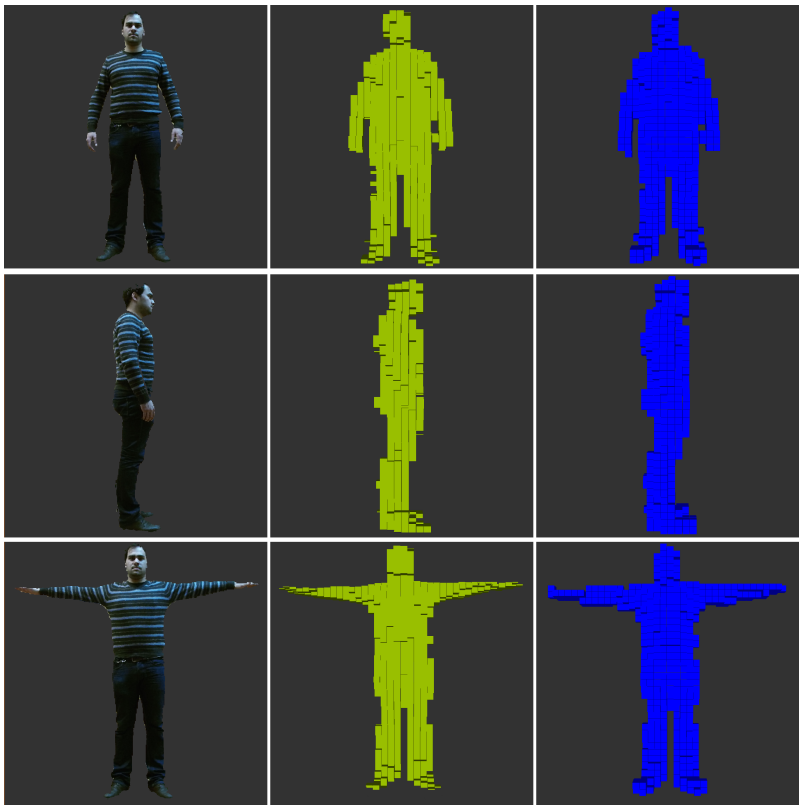


Figure 5.15: Representations of a standing person.

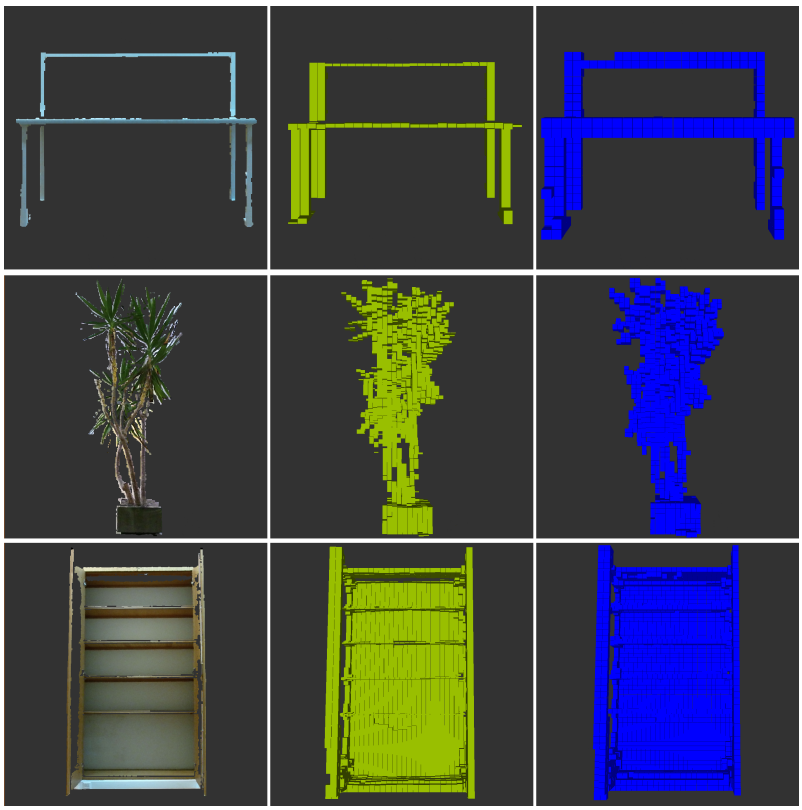


Figure 5.16: First line: A bench; Second line: A plant; Third line: A closet .

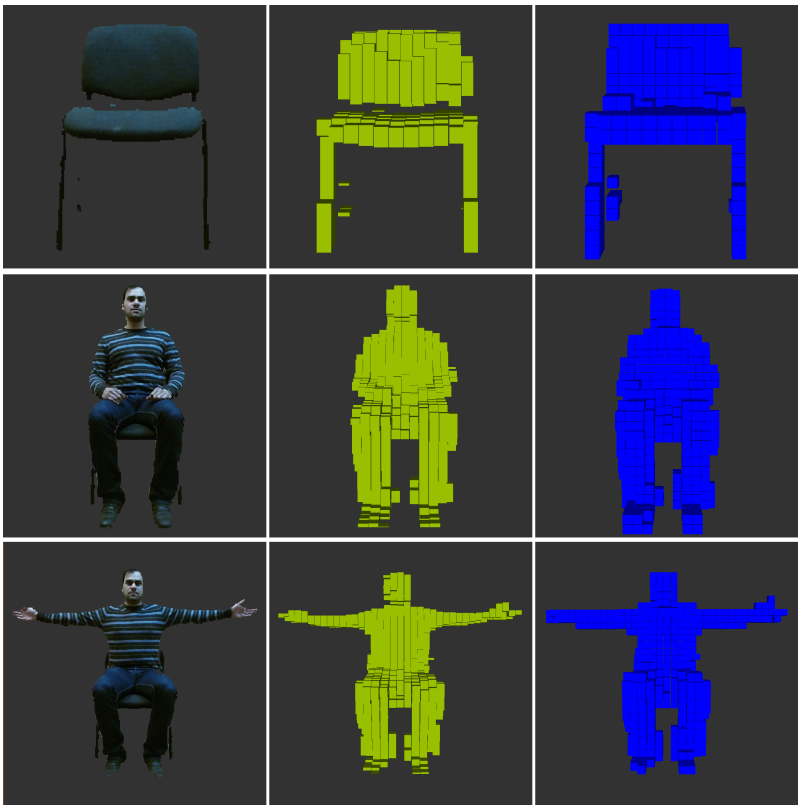


Figure 5.17: A chair and a seated person in different positions.

Figure	Row	Point Cloud	Variable Height Voxels	Standard Voxels	Time (ms)
5.15	1^{st}	143.154 points	140	659	31,0507
	2^{nd}	99.651 points	102	418	27,9779
	3^{rd}	142.975 points	183	674	31,2319
5.16	1^{st}	22.972 points	83	224	23,7153
	2^{nd}	96.993 points	703	1791	24,6782
	3^{rd}	313.366 points	666	2261	40,4457
5.17	1^{st}	36.896 points	97	210	23,4143
	2^{nd}	121.481 points	203	622	29,9869
	3^{rd}	138.905 points	232	680	30,0971

Table 5.2: Quantification and comparison of structures presented.

6

Conclusions and Future Work

Contents

6.1 Conclusions

The proposed approach is robust and the algorithm is fast due to its simplicity and low number of operations required from the original point cloud to the final representation. The work achieved with the sonars proved to be useful, reducing the area of detection and that way obtaining a more precise localization of potential obstacles.

Unfortunately there are improvements and developments still to be made that will be discussed on the next section 6.1.

6.2 Future Work

An interesting test would be to use linked lists instead of vectors but only in the sorting step of this approach. Lists allow constant time insert and erase operations anywhere within the sequence, these features would require less memory to sort. But, lists also bring some disadvantages such as higher access times. The test and time measurement with both structures would be useful.

The processing method of free space need improvement to avoid multiple empty structures overlapping in the same area. A merging ability would solve that issue. With both free and occupied information, a probability of occupation could be obtained for each cell from the comparison of volumes ratios over that cell.

Mounting the sonar arrays, Kinect and laser range finder to the wheel-chair for semi-autonomous or even fully autonomous navigation would be really interesting. For that, a map matching from local to global map have to be implemented. Some odometry issues causing the wheels to spin have to be corrected, perhaps improvement in the PID controller, reducing the natural frequency could solve this problem. Navigation algorithms should also be implemented.

Exploiting the ability shown in figure 5.13, achieving multi-resolution in a given required area would be of high usefulness for decision making and path planing. A zoom like feature in a given area with fine grid resolution for example.

7

References

Bibliography

[Cole and Newman, 2006] Cole, D. M. and Newman, P. M. (2006). Using laser range data for 3d slam in outdoor environments. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 1556–1563. IEEE.

[Dryanovski et al., 2010] Dryanovski, I., Morris, W., and Xiao, J. (2010). Multi-volume occupancy grids: An efficient probabilistic 3d mapping model for micro aerial vehicles. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1553–1559. IEEE.

[Einhorn et al., 2011] Einhorn, E., Schröter, C., and Gross, H.-M. (2011). Finding the adequate resolution for grid mapping-cell sizes locally adapting on-the-fly. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1843–1848. IEEE.

[Elfes, 1989] Elfes, A. (1989). Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57.

[Ferreira et al., 2008] Ferreira, J. F., Bessière, P., Mekhnacha, K., Lobo, J., Dias, J., and Laugier, C. (2008). Bayesian models for multimodal perception of 3d structure and motion. In *International Conference on Cognitive Systems (CogSys 2008)*.

[Ferreira and Dias, 2014] Ferreira, J. F. and Dias, J. (2014). *Probabilistic approaches to robotic perception*. Springer.

[Ferreira et al., 2013] Ferreira, J. F., Lobo, J., Bessiere, P., Castelo-Branco, M., and Dias, J. (2013). A bayesian framework for active artificial perception. *Cybernetics, IEEE Transactions on*, 43(2):699–711.

[Ferreira et al., 2011] Ferreira, J. F., Lobo, J., and Dias, J. (2011). Bayesian real-time perception algorithms on gpu. *Journal of Real-Time Image Processing*, 6(3):171–186.

[Fournier et al., 2007] Fournier, J., Ricard, B., and Laurendeau, D. (2007). Mapping and exploration of complex environments using persistent 3d model. In *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, pages 403–410. IEEE.

[Geiger et al., 2013] Geiger, A., Lenz, P., Stiller, C., and Urtasun, R. (2013). Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*.

[Guttman, 1984] Guttman, A. (1984). *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM.

[Hadsell et al., 2009] Hadsell, R., Bagnell, J. A., Huber, D. F., and Hebert, M. (2009). Accurate rough terrain estimation with space-carving kernels. In *Robotics: Science and Systems*, volume 2009.

[Herbert et al., 1989] Herbert, M., Caillas, C., Krotkov, E., Kweon, I. S., and Kanade, T. (1989). Terrain mapping for a roving planetary explorer. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 997–1002. IEEE.

[Hornung et al., 2013] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206.

[Khan et al., 2014] Khan, S., Dometios, A., Verginis, C., Tzafestas, C., Wollherr, D., and Buss, M. (2014). Rmap: a rectangular cuboid approximation framework for 3d environment mapping. *Autonomous Robots*, 37(3):261–277.

[Khan et al.,] Khan, S., Wollherr, D., and Buss, M. Adaptive rectangular cuboids for 3d mapping.

[Knuth, 1968] Knuth, D. (1968). The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching.

[Meagher, 1982] Meagher, D. (1982). Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147.

[Moravec, 1996] Moravec, H. (1996). Robot spatial perceptionby stereoscopic vision and 3d evidence grids. *Perception,(September)*.

[Moravec and Elfes, 1985] Moravec, H. P. and Elfes, A. (1985). High resolution maps from wide angle sonar. In *Robotics and Automation. Proceedings. 1985 IEEE International Conference on*, volume 2, pages 116–121. IEEE.

[Nüchter et al., 2007] Nüchter, A., Lingemann, K., Hertzberg, J., and Surmann, H. (2007). 6d slam—3d mapping outdoor environments. *Journal of Field Robotics*, 24(8-9):699–722.

[Ogata, 2001] Ogata, K. (2001). *Modern control engineering*. Prentice Hall PTR.

[Paiva, 2015] Paiva, H. (2015). Human-machine interface modules to support indoor navigation of a robotic wheelchair. Master thesis, University of Coimbra.

[Pathak et al., 2007] Pathak, K., Birk, A., Poppinga, J., and Schwertfeger, S. (2007). 3d forward sensor modeling and application to occupancy grid based sensor fusion. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, pages 2059–2064. IEEE.

[Payeur et al., 1997] Payeur, P., Hébert, P., Laurendeau, D., and Gosselin, C. M. (1997). Probabilistic octree modeling of a 3d dynamic environment. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 2, pages 1289–1296. IEEE.

[Pratt, 1972] Pratt, V. R. (1972). Shellsort and sorting networks. Technical report, DTIC Document.

[Premebida et al., 2015] Premebida, C., Sousa, J., Garrote, L., and Nunes, U. (2015). Polar-grid representation and kriging-based 2.5 d interpolation for urban environment modelling. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 1234–1239. IEEE.

[Rivadeneyra et al., 2009] Rivadeneyra, C., Miller, I., Schoenberg, J. R., and Campbell, M. (2009). Probabilistic estimation of multi-level terrain maps. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 1643–1648. IEEE.

[Roth-Tabak and Jain, 1989] Roth-Tabak, Y. and Jain, R. (1989). Building an environment model using depth information. *Computer*, 22(6):85–90.

[Saarinen et al., 2013a] Saarinen, J., Andreasson, H., Stoyanov, T., Ala-Luhtala, J., and Lilienthal, A. J. (2013a). Normal distributions transform occupancy maps: Application to large-scale online 3d mapping. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2233–2238. IEEE.

[Saarinen et al., 2013b] Saarinen, J. P., Andreasson, H., Stoyanov, T., and Lilienthal, A. J. (2013b). 3d normal distributions transform occupancy maps: an efficient representation for mapping in dynamic environments. *The International Journal of Robotics Research*, 32(14):1627–1644.

[Sarbolandi et al., 2015] Sarbolandi, H., Lefloch, D., and Kolb, A. (2015). Kinect range sensing: Structured-light versus time-of-flight kinect. *Computer Vision and Image Understanding*.

[Shell, 1959] Shell, D. L. (1959). A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32.

[Thrun, 1998] Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71.

[Thrun, 2002] Thrun, S. (2002). Probabilistic robotics. *Communications of the ACM*, 45(3):52–57.

[Thrun, 2003] Thrun, S. (2003). Learning occupancy grid maps with forward sensor models. *Autonomous robots*, 15(2):111–127.

[Triebel et al., 2006] Triebel, R., Pfaff, P., and Burgard, W. (2006). Multi-level surface maps for outdoor terrain mapping and loop closing. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2276–2282. IEEE.

[Wiedemeyer, 2015] Wiedemeyer, T. (2014 – 2015). IAI Kinect2. `https://github.com/code-iai/iai_kinect2`. Accessed June 12, 2015.

[Wilhelms and Van Gelder, 1992] Wilhelms, J. and Van Gelder, A. (1992). Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227.

[Wurm et al., 2010] Wurm, K. M., Hornung, A., Bennewitz, M., Stachniss, C., and Burgard, W. (2010). Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2.

8

Appendix A: Sonar Evaluation

Sonar Evaluation

A study has been performed with one sonar to evaluate its area of detection. Firstly, a flat obstacle was placed parallel to the sonar and moved from the left and then from the right side of the sensor to the center until the sonar detected it, those two points were then marked in a paper sheet. The obstacle was then moved away one centimetre and the process repeated. This was done because this sensor has two transducers, one for the emission of the wave and the other one for the reception of the echo. Because of that, the angle of detection is not symmetric to the axis between both transducers. Measurements have been made from 2 centimetres to 100 centimetres with steps of 1 centimetre. Because of the increase of the detection area with the distance, only measurements under one meter will be considered. This process is shown in figure 8.1.

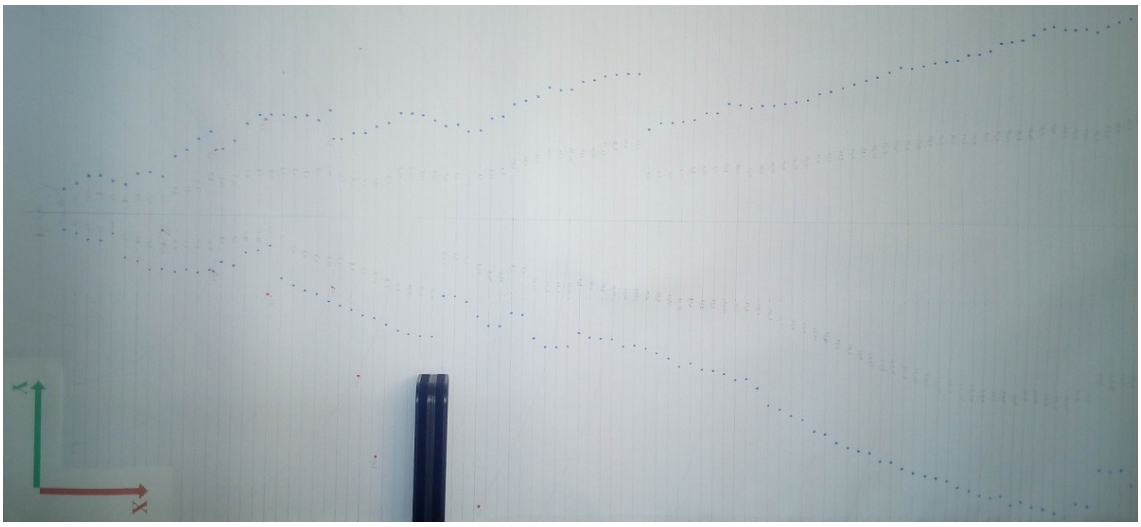


Figure 8.1: Obstacle parallel to the sensor and steps of one centimetre.

The second evaluation was made with the obstacle on top of a ruler. The ruler was retained in the centre point of detection (the point between the two transducers) and was then rotated until the sensor found the obstacle on top of it. This way, the ranging area with non parallel obstacles could be evaluated. Once again, the movement was done from left to right and then from right to left. Measurements were taken from 10 centimetres to 100 centimetres with steps of 5 centimetres this time. This procedure is shown on figure 8.2.

The left part of the figure 8.3 shows the results of the previously explained experiment. The solid line is the junction of the points obtained with the obstacle parallel to the sensor, while the dots represent the measurements taken by the second evaluation method. It is clear that this second method lead to a wider area. Which means that the angle of the obstacle can spread the detection area. This larger area will have to be considered since it is the worst case scenario and a solution that will lead to errors in mapping future

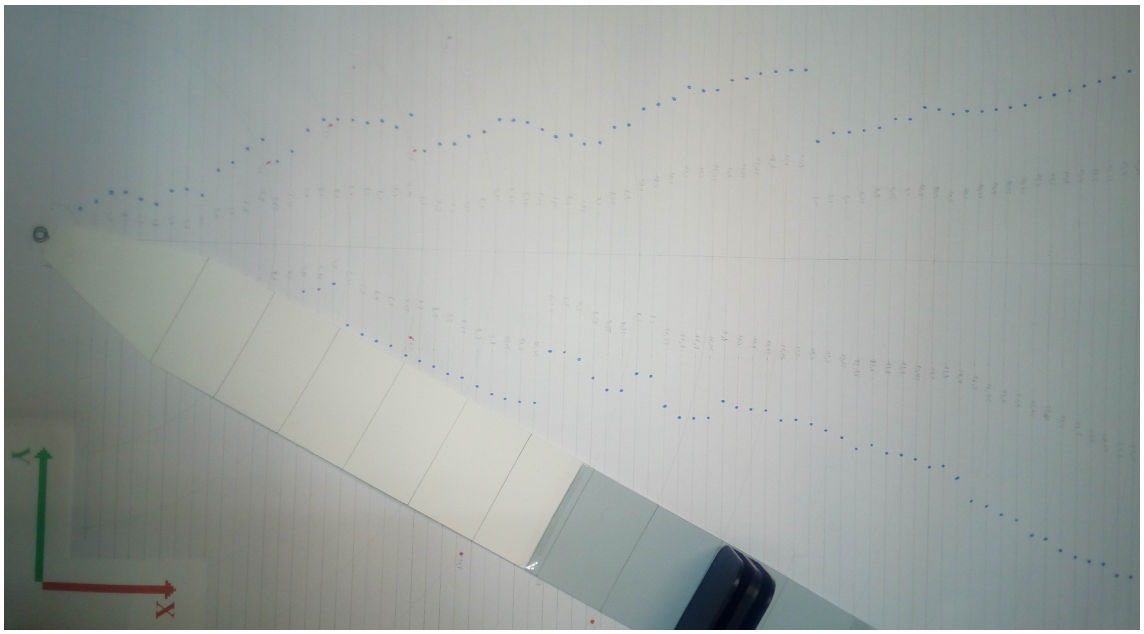


Figure 8.2: Measurement of range area with variable obstacle angle.

obstacles is unacceptable. By doing so we can conclude that the detection cone is quite large (around 60 degrees) and that is one reason that lead to only accept readings under one metre. The lines of the right part of the figure were obtained by the least squares method over the points previously obtained. Providing the equations needed to proceed with the hardware configuration of the final assembly in chapter 5.

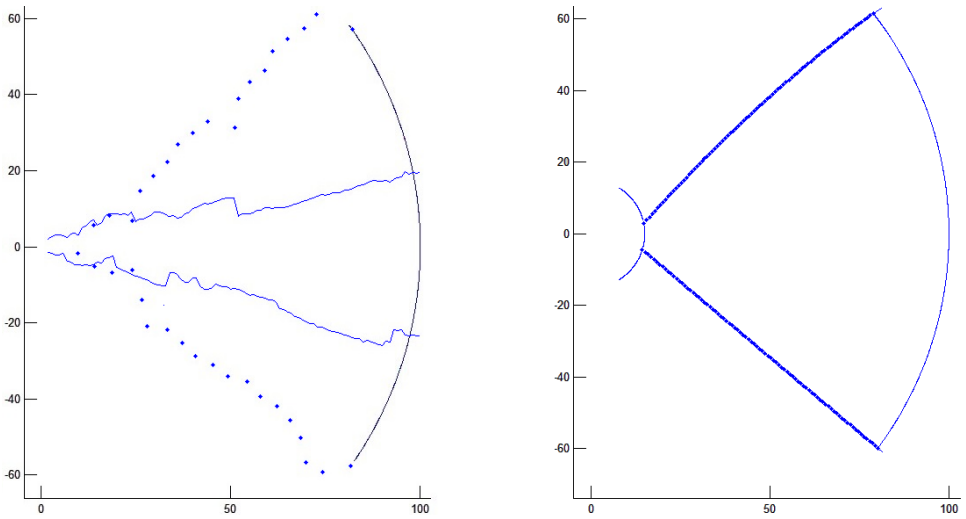


Figure 8.3: **Left:** Resulting zones for one sonar. **Right:** Quadratic regression of points.

The obtained equation for the top curve of the right figure is:

$$y = -0.0031x^2 + 1.2082x - 14.2762 \tag{8.0.1}$$

And for the bottom curve of that same figure is as follows:

$$y = 0.0001x^2 - 0.8483x + 7.7197 \tag{8.0.2}$$

It is important to note that under 15 centimetres this method is not applied since the method obtained from the obstacle parallel to the sonar provide a wider detection area and that is the one used for such short distances.

9

Appendix B: Robchair Storm Platform

Robchair Storm Platform

The implementation of the sensors and algorithms developed throughout this thesis were intended to be applied to a robotic wheelchair that will be described in this appendix. This work is an addition to [Paiva, 2015] work to the platform. Hugo Paiva implemented a joystick connected to an *Arduino* board. A Raspberry Pi have also been added to the wheelchair in order to navigate and communicate with the power driver. Of course this was not the only purpose of his thesis but his work is gladly thanked since it allowed the wheelchair to be operated. Even if most of the work presented in this chapter is of technical nature and is more practical, it took many hours to perform and was essential to the development of the future work on the wheelchair.

High	163.2cm
Width	64cm
Depth	84cm
Wheels radius	17.2cm
Maximum motor speed	1000RPM
Gear Box reduction factor	1:29
Encoders pulses per revolution	1000
Distance between Wheels	58cm

Table 9.1: *RobChair Storm*[3] main specifications

The first requirement in order to acquire the odometry was to add rotary encoders to the platform.

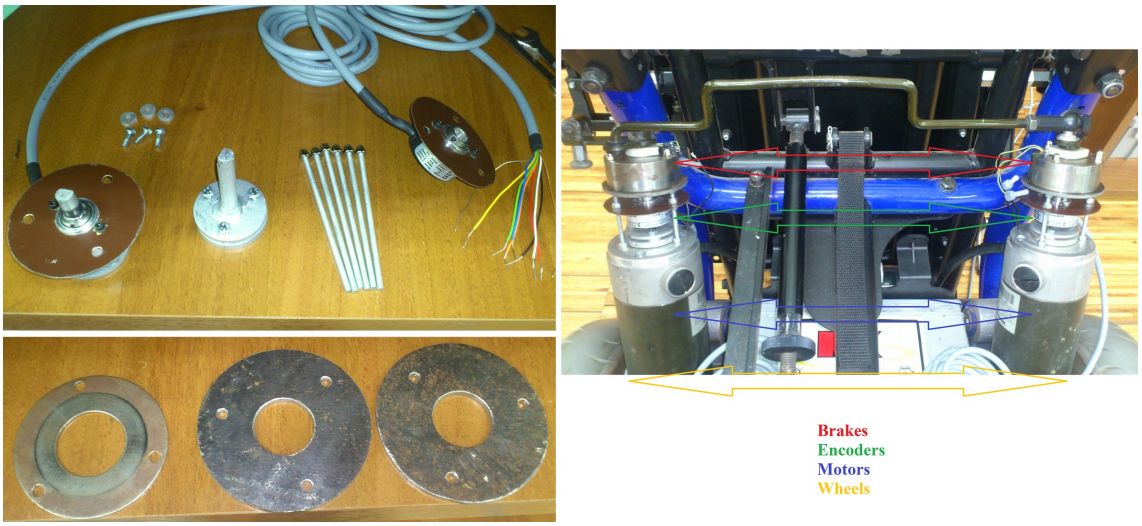


Figure 9.1: On the left, it is presented all the hand-made material to hold the encoders in place while keeping the brakes. On the right, there is a picture taken from below the wheelchair of the final assembly.

In order to connect the motors and encoders, a controller was needed. The chosen controller was the MDC2230 motor controller from Roboteq, it features a dual channel setup, which makes it possible to control the left and right motor of RobChair Storm individually. The controller features a high-performance 32-bit microcomputer and quadrature encoder inputs to perform advanced motion control algorithms in open loop or closed loop (speed or position) modes. The communication with MDC2230 motor controller can be done trough RS232, CAN or USB where USB is the one used to exchange data between the motor controller and the Raspberry Pi.

The MDC2230 also allows PID tuning to each motor, the method used to design the PID consisted in applying a step input in open loop mode with the RobChair Storm on the floor and retrieving the motors response to that stimulus. During the study of the courses of *Control* and *Digital Control* and with the precious help of the book *"Modern Control Engineering"* by [Ogata, 2001], I acquired the knowledge and mathematical equations provided in this explanation. Three tests were done to each motor, retrieving 3 peak times t_p and 3 rise times t_r for each motor. A mean of those values was calculated, providing the following results:

- **Motor 1:** t_r = 0.1365s; t_p = 0.3975s
- **Motor 2:** t_r = 0.1524s; t_p = 0.3590s

$$\omega_d = \frac{\pi}{t_p} \tag{9.0.1}$$

The maximum overshoot for motor 1 was M_p = 0.1 and for motor 2 M_p = 0.12 for a 10 % maximum speed input command.

$$\xi = -\frac{ln(M_p)}{\sqrt{\pi^2 + ln(M_p)^2}} \tag{9.0.2}$$

$$\omega_n = \xi \times \sigma \tag{9.0.3}$$

$$\sigma = -\frac{\omega_d}{\pi} \times ln(M_p) \tag{9.0.4}$$

The above obtained values were needed to input in a typical 2^{nd} order transfer function 9.0.5that represent the behaviour of each motor.

$$G(s) = \frac{{\omega_n}^2}{s^2 + 2\xi\,\omega_n s + {\omega_n}^2} \rightarrow \frac{b_0}{s^2 + a_1 s + a_0} \tag{9.0.5}$$

The controller is a standard PID with proportional K_p, integrative K_i and derivative K_d gains, it is shown in equation 9.0.6.

$$G_c(s) = K_p + \frac{K_i}{s} + K_d s \tag{9.0.6}$$

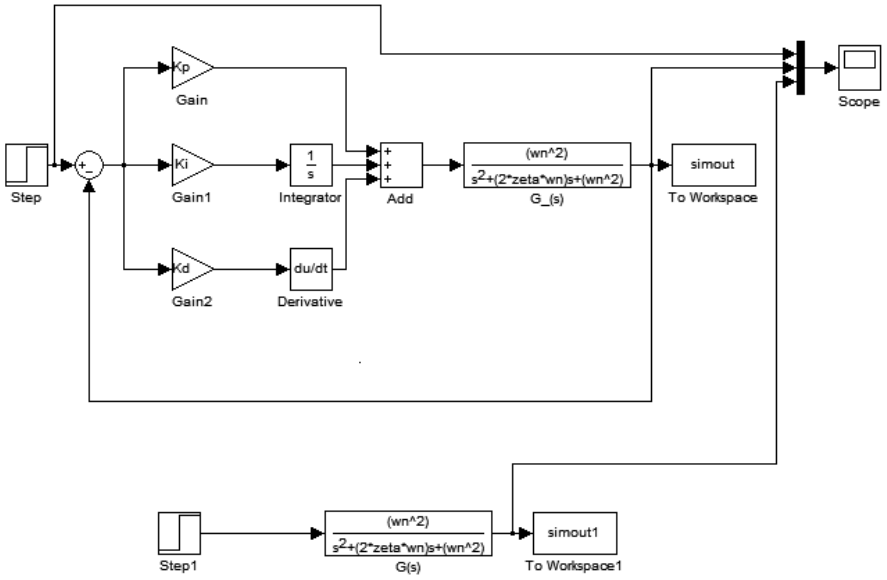


Figure 9.2: Block diagram of the closed loop system with PID and open loop.

Solving the characteristic equation given by $1 + G_c(s).G(s) = 0$ returned a 3^{rd} order equation as presented in 9.0.7 (left part). It was needed then to add a non dominant pole p to the right part of the equation that part also contains the parameters that we want the final closed loop system to follow. ξ^w=1 and ω_n^w=3.3 were chosen. The value for p is equal to $10 \times \xi^w \times \omega_n^w$.

$$s^3 + (K_d b_0 + a_1)s^2 + (K_p b_0 + a_0)s + K_i b_0 = (s+p)(s^2 + 2\xi^w \omega_n^w s + \omega_n^{w2}) \tag{9.0.7}$$

Solving equation 9.0.7 in order to the gains of the PID controller, returned the solution to obtain the gains as presented in 9.0.8.

$$K_p = \frac{2p\xi^w\omega_n^w + \omega_n^{w2} - a_0}{b_0}; K_d = \frac{2\xi^w\omega_n^w + p - a_1}{b_0}; K_d = \frac{p\omega_n^{w2}}{b_0} \tag{9.0.8}$$

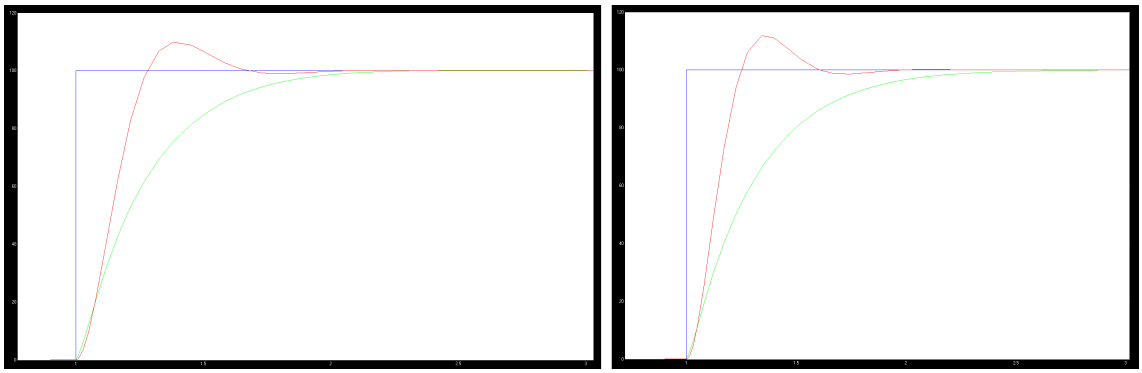


Figure 9.3: Motors behaviour. Motor 1 on the left and motor 2 on the right.

In blue colour is the reference input, in red the open loop response without PID control and the green line is the closed loop with PID controller. It is important to chose carefully the value for the non dominant pole that had to be added because it can lead to the system instability. After each change, it is advised to draw a root locus to prevent this from happening.

Metallic parts have been handmade by me to hold all the needed sensors in place as depicted in figure 9.4 and figure 9.5.

Figure 9.4: Hokuyo laser range finder to assemble in front of the *WheelChair Storm*[3].

Figure 9.5: Final layout missing the right sonar array.