Underwater mapping using sonar

Author

Aridane Jesús Sarrionandia De León

Tutor

Jorge Cabrera Gámez

Proyecto de Fin de Carrera
Ingeniería en Informática

Escuela de Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

14 September 2015

Acknowledgements

I grateful to thank my tutor Jorge Cabrera for giving me the opportunity of working on this project and supporting me through it.

Thanks to the AVORA project for helping me through this journey and teaching me about ROS and the world of autonomous vechiles.

Thanks to my family and friends for supporting me and for knowing when to stop asking "When are you going to finishing it?". Also thanks to Eva, for dealing with whatever made up crisis I could come up with.
Aitari, amari, Aitorreri, Evari, denoi, eskerrik asko.

Contents

1 Introduction **1**

1.1 Objectives 1

1.2 Direct objectives 2

1.3 Indirect objectives 2

2 State of the art **3**

2.1 Mapping technologies 3

2.2 Sonar technologies 4

2.2.1 Echo sounder 5

2.2.2 Mechanically scanning profiler 5

2.2.3 Multibeam echo sounder 6

3 Project Requirements **7**

3.1 Hardware requirements 7

3.2 Software requirements 8

3.2.1 Libraries 9

3.3 ROS 10

3.3.1 What is ROS? 10

3.3.2 Organization 11

3.3.3 Computation Graph 12

3.3.4 Tools 13

3.3.5 Community and collaboration 17

3.3.6 TF 17

4 Project analysis **19**
4.1 Sonar characterisation . . . 19
4.1.1 Imaging . . . 19
4.1.2 Interpretation of sonar images . . . 20
4.1.3 Sonar configuration . . . 21
4.1.4 Communication protocol . . . 22
4.2 Gazebo . . . 23
4.2.1 SDF Modelling . . . 23
4.2.2 Sonar sensor . . . 24
4.2.3 Laser sensor . . . 24
4.3 Platform architecture . . . 25
4.3.1 Structural architecture . . . 26
4.3.2 Electronic architecture . . . 28
4.3.3 Software . . . 30
4.4 Analisys . . . 33
4.4.1 The competition . . . 33
4.4.2 Considered scenarios . . . 34
4.4.3 Methodology . . . 35
4.5 Design . . . 35
4.5.1 Architecture from ROS point of view . . . 35
4.5.2 Class diagram . . . 38

5 Economical and Legal Aspects **43**
5.1 License . . . 43
5.2 Expenses . . . 45

6 Development **46**
6.1 Introduction . . . 46
6.2 Multi Level Surface Maps . . . 46
6.2.1 Structure and manipulation . . . 46
6.3 Iterative Closest Points (ICP) . . . 47
6.3.1 ICP Implementation . . . 48
6.4 Gazebo simulation . . . 48
6.4.1 Collision based simulation . . . 49
6.4.2 XML parameters . . . 50
6.4.3 Laser cone based simulation . . . 54
6.5 ROS . . . 55
6.5.1 Nodes and nodelets . . . 55
6.5.2 Sonar to cloud conversion . . . 55
6.5.3 Thresholding . . . 57
6.5.4 Outlier removal . . . 59
6.5.5 Line and circle detection . . . 61
6.5.6 Mapping node . . . 63
6.6 Processing distribution . . . 73

7 Results **75**
7.1 Sonar simulation . . . 75
7.2 Sonar processing . . . 76
7.3 Mapping . . . 76
7.4 ICP . . . 78

8 Conclusions and future work **82**
8.1 Conclusion . . . 82
8.2 Future work . . . 83

Bibliography **84**

1 | Introduction

This project aims to obtain a practical and usable module to generate underwater maps by means of an imaging sonar.

Nowadays we can think of several technologies that make it possible for us to generate a map of our surroundings with different sensors. Here we have to consider three key elements, the map, the sensor and the environment. Before considering anything else we must think about the *where* that will condition everything else. What will this map represent? Under what conditions will the data be gathered? What are the main characteristics of this environment? Due the characteristics of a regular underwater environment and the conditions we want to be able to work with both the range of mapping technologies and the sensors that are suitable for the task are not the same as if we considered mapping using an aerial robot or a land robot.

Conditioned by the characteristics of an underwater environment we have considered the Multi Level Surface Maps as the way of representing the map. In the following section it will be shown why.

The other part we have to consider is the sensor we will use for the mapping conditioned to the environment. In other conditions we could consider having laser based sensors for acquiring data, they work just great for land robots but they are not good enough for underwater mapping. Some tests on laser response in an underwater environment [7] prove that, while we could use laser based sensors for close interaction and precisely mapping objects, they lack the range that a sonar provides. Yet, laser technology should not be discarded without giving it a thought, there are some developments that aim to provide better laser based sensors for underwater by means of pulsing light ([8]) and we must also balance between range and resolution, if we just need to build a model of an object that's close by a laser will do the job better than a sonar due to its superior accuracy.

Sonar is of great importance in underwater environments. Even though laser can be more resilient to environmental changes like temperature, salinity and pressure, sonar provide a key feature in the underwater world, which is visibility. With a sonar we will not get the level of detail of a laser sensor, but it is good enough for building a general map of an underwater terrain under a wider set of water visibility conditions, which is why sonar is selected as the sensor that will provide the information needed for building our map.

1.1 Objectives

Among the objectives of this project we can find direct objectives, which can be identified as what it aims to accomplish, and the indirect objectives, which are derived from the direct objectives,and also the technologies and platforms used. In general,the conditions under which the project is developed.

1.2 Direct objectives

The main objective of this project is clear: To develop a hardware-software module for building obstacle maps using sonar readings. It also aims to develop a module that will provide needed capabilities in order to overcome the missions in SAUC-E.

Also, this project is intended as the basis for future project that should expand the mapping and localisation capabilities of the platform and perform further real world testing.

1.3 Indirect objectives

This objectives do not have a direct relation with this projects mission.

- Getting to know ROS framework and obtain experience using it.
- Gain experience with software design, since the project needs communication with other software and also comprises a number of different modules.
- Acquire experience with data filtering methods.
- Acquire experience with detection methods.
- Acquire experience in 3D data handling.
- Solve problems on the current software.
- Acquire experience using LaTeX.
- Generate a proper documentation for future AVORA generations.

2 | State of the art

In this section different mapping and sonar technologies are discussed in order to get a proper image of the state of the art in this area.

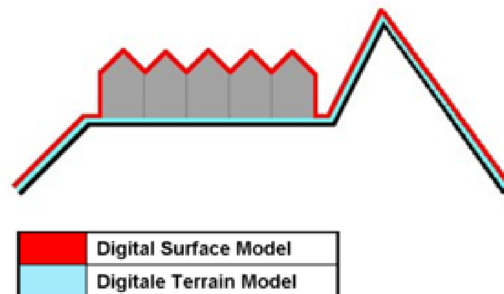
2.1 Mapping technologies

As it is widely known, the Simultaneous Localization and Mapping or SLAM problem requires both the agent to generate a map and localise himself within that map. Along the years multiple ways of representing those maps have been suggested. Commonly, to analyse also traversability, these maps discretise the world in a grid in which values of elevation or information regarding the existence of elements are attached to each measured position. The use of the terms for these models differ greatly among the community but I will go for what I think is the most sensible use of them.

Digital Elevation Model (DEM)
A DEM is usually grid based model in which every cell of the grid covers the same area and contains the elevation information for that position.

Digital Terrain Model (DTM)
A DTM can be seen as an upgraded DEM, a DTM contains XYZ information rather than only elevation. It has the DEM information combined with geographical elements and natural features such as rivers, ridges etc. This extended information generally comes from other resources that complement the bare sensor data that a DEM provides.

Digital Surface Model (DSM)
A DSM represents the different elevations found in the reflective surfaces of trees, buildings and other elements rather than just the "Bare Earth".



Point Clouds
Point clouds are very close to the raw representation of data acquired from range sensors. They store each measurement value as a 3D point, and also some other fields can be added such as colour, intensity, etc. Due to this, they are not quite efficient when it comes to map size, since there is no compression between the raw data acquired and the point cloud.

Multi Level Surface Map (MLSM)
These maps are grid based maps. The approach used in these maps is biased towards memory efficiency. Each map has a grid in which each grid cell contains a list of occupied blocks along with several parameters that detail how many cells in that vertical are ocuppied. The parameters stored allow to fuse several blocks that contain adjacent similar information. I will go deeper about this structure in the following chapters, but more information can also be found at [13] and [11].

Octomap
Octomap is a type of probabilistic occupancy grid. While the other grid types can easily become probabilistic grids, octomap takes it a little bit further to provide some other features that the rest of implementations lack. In [9], besides being a probabilistic approach, the authors propose an efficient way of mappping by storing the data blocks in octrees, and also model unmapped areas, thus claiming an improvement over the rest of mapping technologies most used in robotics like elevation maps, point clouds and MLSM.

Even though I define the terminology used in mapping, [10] defines DEM as a subset of DTM, being the most fundamental component of DTM:

> In practice, these terms (DTM, DEM, DHM, and DTEM) are often assumed to be synonymous and indeed this is often the case. But sometimes they actually refer to different products. That is, there may be slight differences between these terms. Li (1990) has made a comparative analysis of these differences as follows:
>
> - *Ground*: ”the solid surface of the earth”; ”a solid base or foundation”; ”a surface of the earth”; ”bottom of the sea”; etc.
> - *Height*: ”measurement from base to top”; ”elevation above the ground or recognized level, especially that of the sea”; ”distance upwards”; etc.
> - *Elevation*: ”height above a given level, especially that of sea”; ”height above the horizon”; etc.
> - *Terrain*: ”tract of country considered with regarded to its natural features, etc.”; ”an extent of ground, region, territory”; etc.

2.2 Sonar technologies

There is a wide range in sonar technologies but the most adequate for mapping is the active sonar, which can both emit and receive acoustic signals. Opposed to passive sonar which can only receive echoes. Several active sonar devices are described next.

2.2.1 Echo sounder

This is one of the simplest range sensor. The echo sounder emits a pulse from its transducer. When this pulse is reflected by a surface it returns to the sensor head and the time of flight can be measured and so, the distance can be estimated. This kind of device is suitable for measuring the distance of the vehicle to the seabed or structures like walls. It is normally while mounted on static positions looking down or to the sides.

Most large vessels carry with them an echo sounder for acquiring bathymetry. They are also commonly used for fishing, since the echo sounder will register a difference in depth where a school of fish is located.

Most echo sounders are not too precise, and water conditions such as temperature must be considered when intending to obtain accurate measurements with any acoustic sensor. When detailed measurements are required, one must turn to specific echo sounders for hydrography and also evaluate an array of different factors in those, like resolution, acoustic beamwidth and so on.

2.2.2 Mechanically scanning profiler

This sensor is composed of a mechanically actuated transducer which can be oriented at different angles and produce a series of measurements. Usually the size of the scan sector can be set up from a few degrees to a 360 deg scan. The scanning profiler can be oriented at different angles with respect to the vertical direction, to produce different types of maps.

The mechanically scanning profilers are capable of returning echo intensity values from the insonified area. These measurements can be used to build what is called an acoustic image, which is an acoustic representation of the environment in terms of intensity and position. The most common types of imaging sonar are the following:

Mechanically scanned imaging sonar

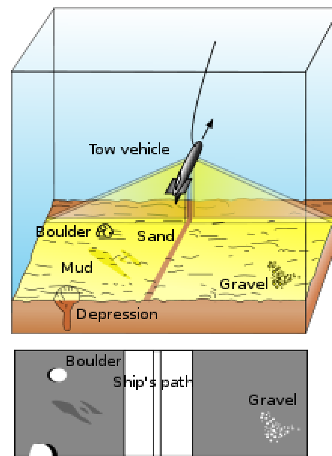
Similar to the mechanically scanning profiler, this device also has an actuated rotatory transducer which can emit fan-shaped beams at different orientations. It is usually placed in a vertical position so it can perform the scanning in the horizontal plane. These devices generally have a configurable scan sector and it is not unusual to find models which can perform full 360 deg scans, making them perfect for detecting objects around the vehicle. Its main draw-back is the slow refresh rate.

Electronically scanned imaging sonar

Also know as multibeam imaging sonar and forward-looking imaging sonar, this sonar is equipped with an array of hydrophones which produce a complete acoustic image of the insonified area in a single pulse. This area is usually limited to a small sector, but can be scanned at very high rates. The main drawback is the cost which can be around ten times the price of a mechanically scanned sonar.

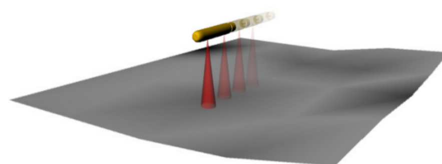
Sidescan sonar

This sonar is designed for imaging large seabed areas. It works in an analogous way to the one of the multibeam echo sounders, but oriented to imaging tasks. This sonar emits fan shaped pulses, which makes it capable of producing a strip of echo intensity measurements which, mounted looking down, can produce data to build a large acoustic image of the seabed.

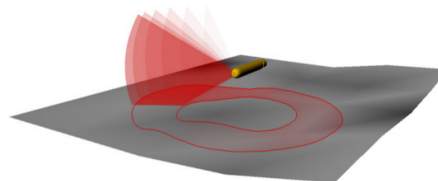


2.2.3 Multibeam echo sounder

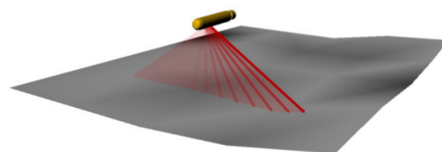
This sensor is able to produce bathymetric maps of large areas of seabed. It is composed of an array of hydrophones which emit fan shaped beams, producing a whole strip of points in the direction of the pulses emitted. This type of sonar can produce readings at high rate and resolution.



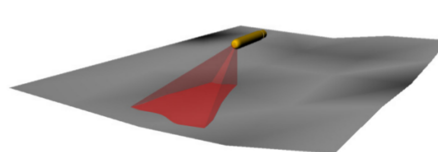
(a) Echo sounder.



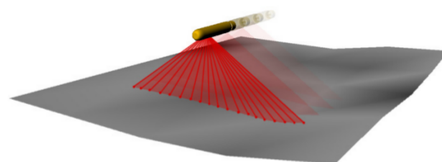
(a) Mechanically scanned imaging sonar.



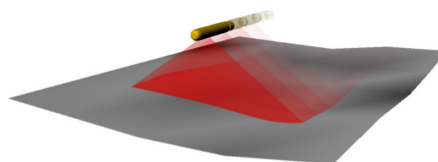
(b) Mechanically scanned profiler.



(b) Electronically scanned imaging sonar.



(c) Multibeam echo sounder.



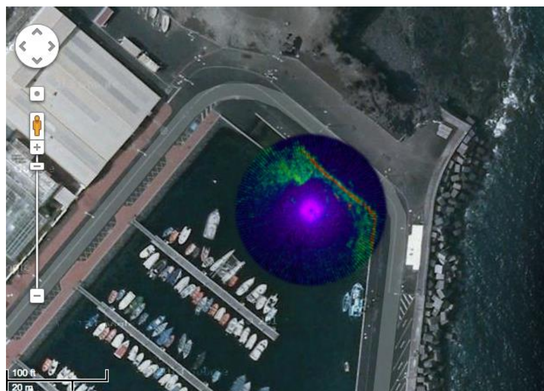
(c) Sidescan sonar.

3 | Project Requirements

In this chapter we will discuss the requirements that are needed in order to build this project. These requirements are separated into hardware and software requirements.

3.1 Hardware requirements

- *Work station*: First of all we will need a platform in which most of the work will be developed. A PC with the according software requirements would be enough since there are no special requirements for the PC unless we consider being able to connect to Ethernet or Wifi such a special requirement. A laptop will suffice for this task.
- *Sensors*: In the matter of building underwater maps using sonar it is obvious that we will need such a device, or a way of simulating it. The sonar is already provided by the AUV developed by the AVORA team. This sonar is an Imaginex 852 mechanically scanned imaging sonar.
- *Platform*: In order to manipulate the sonar we will need a platform to which it will be connected. For this we have a very complete one, the AUV Charl-e, which allows us to much more than just move the sonar and obtain readings, as is explained in the next section of Project analysis.
- *Test environment*: For the purpose of gathering sonar readings using the actual sensor, we will need an environment suitable to do so. We mainly have two test areas and what we would consider a bonus one. First of all we have a pool donated to the AVORA project by PLOCAN (Plataforma Oceánica de Canarias). For a more realistic test environment PLOCAN has also given us access to a port located in Taliarte. Finally, since part of the target of the project is to provide a module for SAUC-E it is logical to gather data from the competitions environment.



3.2 Software requirements

- *Operative system*: The PC will require a Linux based operative system. The selected one has been Ubuntu, due to being already known to the student but also because is the only officially supported operative system for ROS.
- *ROS*: In order to develop this project the ROS framework was used, both due to the fact that Charl-e was already built around this framework and the stated benefits of such framework as explained in section 3.3.
- *Editing environment*: Many editing environments have been used for this project. For editing directly inside Charl-e we have used vim, since it allows us to edit through a ssh connection. We have also used gedit, when editing from an external PC, due to its simplicity and syntax highlight.
- *Programming IDE*: In order to work more efficiently it is recommended to use an IDE for programming. After searching in the ROS web page, I found that several IDEs could be configured to be integrated with ROS. First I started using Eclipse, which needed some configuration in order to work and also tended to have some problems. Then QtCreator was selected as main programming environment, since it supports CMake projects directly and ROS catkin projects are of such kind. In the beginning of this project and the students collaboration on the AVORA project all the development in an exterior workstation to Charl-e was mas through regular text editors like gedit. After checking the ros.org web page for IDEs we found that eclipse could be used, given some configuration was done. This modification did not prove hard, and the benefits from it were as many as having such an IDE for programming. Such as autocompletion, user friendly debbuging environment etc. Also another environment was checked. The BRIDE IDE developed by the BRICS project. In this IDE you can create ROS packages using a graphic tool that allows you to create the graph architecture before you start programming, setting connections between topics and different nodes, all using a graphic environment. After the design is done you can also use it for programming. This IDE was just tested since it seemed to need more intensive familiarisation and QTCreator was already good enough to work on this project.
- *Simulator*: It is a good thing to have a simulator at hand whenever we are developing any software, more so if we are developing software that requires interaction with an environment, and even more so if that environment is underwater. Testing the system with real conditions can be hard and expensive so we turn to a simulator for basic testing.

  The simulator selected for the development of this project is Gazebo due to its easy integration with ROS and the fact that some AVORA software already considered the use of Gazebo. We will analyse Gazebo deeper in the following section.
- *LaTeXEditor* Even though an editing environment is not needed for creating documents with LaTeXthey ease the process of building a document a lot, and also offer a better user experience since they provide a lot of useful features for writing documents. In the beginning the editor used was *Texmaker* but due to some

problems it was replaced by *TexStudio* which, in my opinion offers a better user experience.

3.2.1 Libraries

This project has made use of several libraries and tools provided by different entities, but the ones considered of most importance, without talking about ROS, are the following.

OpenCV

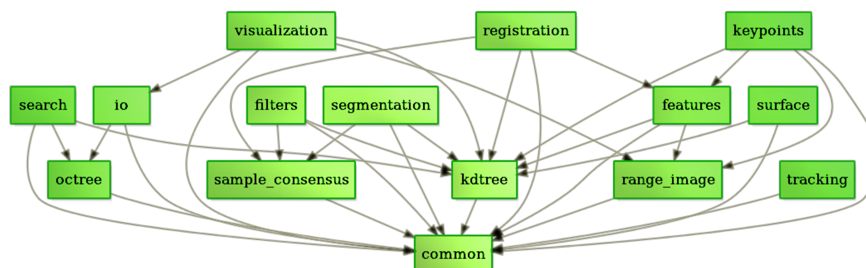
OpenCV (Open source Computer Vision) is a library of programming functions for realtime computer vision originally developed by Intel. It is released under BSD license so its free to use for both academic and comercial use. It is multi-platform, supporting Windows, Linux, Android, iOS and Mac OS. The library aims to provide an evironment in which the users can develop applications easily and efficiently, so the functionalities provided are very optimised. The library has C, C++, Python and Java (Android) interfaces so it could be easily integrated in our C++ code.

Point Cloud Library (PCL)

The Point Cloud Library (or PCL) is a large scale, open project [15] for 2D/3D image and point cloud processing. The PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. These algorithms can be used, for example, to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects in the world based on their geometric appearance, and create surfaces from point clouds and visualize them – to name a few.

PCL is released under the terms of the 3-clause BSD license and is open source software. It is free for commercial and research use.

PCL is cross-platform, and has been successfully compiled and deployed on Linux, MacOS, Windows, and Android/iOS. To simplify development, PCL is split into a series of smaller code libraries, that can be compiled separately. This modularity is important for distributing PCL on platforms with reduced computational or size constraints. Another way to think about PCL is as a graph of code libraries, similar to the Boost set of C++ libraries. Here's an example:



Boost

Even though is not the most important library used in the system I believe is worth mentioning. The Boost library is an Open Source library released under its own licence, which is pretty much the same as a Gnu General Public License, but allows commercial and non-commercial derivative works and several other features that can be checked at [1]. In general the library aims to extend the features of C++, and from that features I have taken the smart pointers and the multi_array. The first is a pointer that manages itself, destroys itself when no object is using it, and are less messy to use than plain C pointers. The latter is used for the map grid structure, since this multi_array can grow dynamically when we need it to, so the map can start growing without any inconvenience when the data that comes is out of the bounds set on creation.

3.3 ROS

3.3.1 What is ROS?

The Robot Operating System is an open-source framework for developing robot software. It is as flexible middleware that offers a great amount of tools and libraries that makes it simple to create complex and robust robot behaviour in a wide range of platforms. ROS offers common operative system services like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between process and package management. ROS is similar in some aspects to other robot frameworks such as Player, YARP, Orocos, CARMEN, Orca, MOOS and Microsoft Robotics Studio.

ROS used a graph-oriented architecture. Every system developed in ROS can be seen as a graph, where each node is a process that is connected to other nodes by its inputs and outputs, which is accomplished by using ROS communication infrastructure. These inputs and outputs are the inter process communication channels provided by ROS, such as topics and services. This also allows to easily distribute our architecture among machines and still maintain the original architecture.

The ROS description on their web page [3] explains that its main goal is not to be a framework with the most features, but to support code reuse in robotics research and development. Apart from that, and in support of the defined primary goal, there are some other goals for the ROS framework:

- *Thin*:ROS aims to be as thin as possible, making it easy to use code written for ROS with other robot software frameworks. ROS is also easy to integrate with other frameworks and has been successfully integrated with OpenRave, Orocos and Player.
- *ROS-agnostic libraries*: It is preferred to write clean interface libraries that won't depend on ROS resources so they can actually be used in other non-ROS projects.

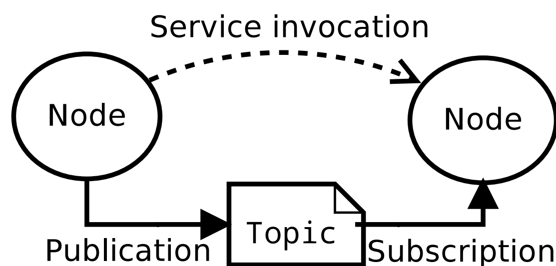
- *Language independence*: ROS is easy to implement in any modern programming language. It is already implemented in Python, C++ and Lisp, and as experimental libraries in Java and Lua.
- *Easy testing*: ROS provides a unit/integration test framework called rostest.
- *Scaling*: ROS is appropriate for large runtime systems and large development processes.

Currently, ROS is primarily tested in Ubuntu and MAC OS X systems, being Ubuntu the only operative system listed as supported in the installation web page.

3.3.2 Organization

ROS organizes the file system in a layered fashion. In our ROS workspace we can find the following elements:

- *Packages*: Packages are the most granular unit of a ROS system. They can contain nodes, a ROS-dependent library, datasets, configuration files or anything else that should be together. Usually each package contains nodes with similar functionalities, or some semantic relation between them.
- *Metapackages*: Metapackages are packages that contain packages, so mainly is a way of organising your packages. They come handy when your system is made up of several repositories and you have each one being a metapackage in your system.
- *Package Manifests*: Package manifests are xml files that contain metadata regarding the package they are in, such as name, version, description, license, dependencies, and other meta information like exported packages.
- *Message types*: Message description files, they are stored in the msg folder of the package and define the structure of the messages in a way similar to C structs.
- *Service types*: They are similar to message files, but are stored under the srv folder and contain a request message and an answer message.



3.3.3 Computation Graph

The ROS Computation Graph is the peer-to-peer network of ROS processes that are being executed at a time. We will go through the elements that comprise this graph:

- *Nodes*: Nodes are the processes performing computation. A ROS system will contain many nodes executing e.g. a node controlling a laser range sensor, other controlling the wheel motors, other performing localisation, other path planning etc.
- *Master*: The ROS Master is the process that provides name registration and lookup to the rest of the elements of the Computation Graph. The Master is the one that makes possible for nodes to find each other, communicate and invoke services.
- *Parameter Server*: Currently is part of the master. Allows to store data by key in a central location such as node parameters.
- *Messages*: Nodes communicate among each other by means of messages, which are structures provided by the system or defined by the user. There are several types of messages supported e.g. int, boolean, string, etc.
- *Topics*: Messages between nodes are sent using publish/subscribe semantics. A topic is the name that identifies the channel to which nodes publish and subscribe.
- *Services*: Instead of a many-to-many one-way communications, services allow us to communicate using request/reply interactions as often required in distributed systems.
- *Bags*: Bag files are the files used by ROS to save all the communications produced during the recording, so we can play back the data from topics for further developing and testing.

Topics can be seen as a conversation between nodes. One or several nodes will be writing data using a topics name, and other nodes are able to listen to what's coming through that channel.

Nodes vs nodelets

Generally each node corresponds to a process in our system. Meaning that each communication channel takes a TCP/IP connection, producing one of the criticised aspects of ROS: high system overhead. In order solve the issue of several connections with large amounts of data, ROS introduce the nodelts. Nodelets are basically nodes that instead of running on a process, several nodes can be run as threads of the same process. This way, instead of using the TCP/IP connection we have allow zero copy passing of data between nodelets. Besides from that, nodelets pursue the following design goals:

- Use the existing C++ ROS interfaces.

- Dynamically load as plugins to break build time dependencies.
- Location transparent except for performance improvements.
- Writing code in a node or a nodelet will be minimally different.

The main difference when using nodelets is that they have some implementation differences to nodes, and also that they require a nodelet manager to be running in order to launch and stop the nodes.

Since this project deals with the transmission of large amounts of data, and also this data in transmitted through several nodes, at the point of going for a more distributed system I went for separating the processing stage into nodelets, since we have several tasks transmitting point clouds and communication between each other. This way the performance of the system would be enhanced.

3.3.4 Tools

ROS offers a set of tools that are very useful for visualising data, configuring node parameters, testing our system, etc.

roslaunch

roslaunch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on. This makes it very easy for us to create launch files for a whole set of nodes. In the case of this project we have a launch folder for each package where we can find a launch file for the nodes of this package. For example, the *sonar_processing* package has a launch file for each nodelet, but also a launch file that allows us to run all the nodelets together, making it much easier to run all the processing stages. In the case of the *sonar_processing* package it is even more important, since nodeletes depend on a different configuration that nodes to be run, so we need to run a *nodelet manager* to be able to run the rest of nodelets. Another interesting feature is the possibility of loading *YAML* files where the parameters of the nodes are contained. These can be found in the *yaml* folder of some packages.

rostopic

The rostopic command lets us acquire information regarding the topics that are being used in our system. It has several options:

- rostopic bw: display bandwidth used by topic
- rostopic echo: print messages to screen

- rostopic find: find topics by type
- rostopic hz: display publishing rate of topic
- rostopic info: print information about active topic
- rostopic list: list active topics
- rostopic pub : publish data to topic
- rostopic type: print topic type

rqt_reconfigure

This rqt plugin provides a way to dinamically view and edit the parameters of a node. This way we can change the behaviour of our nodes on execution time.

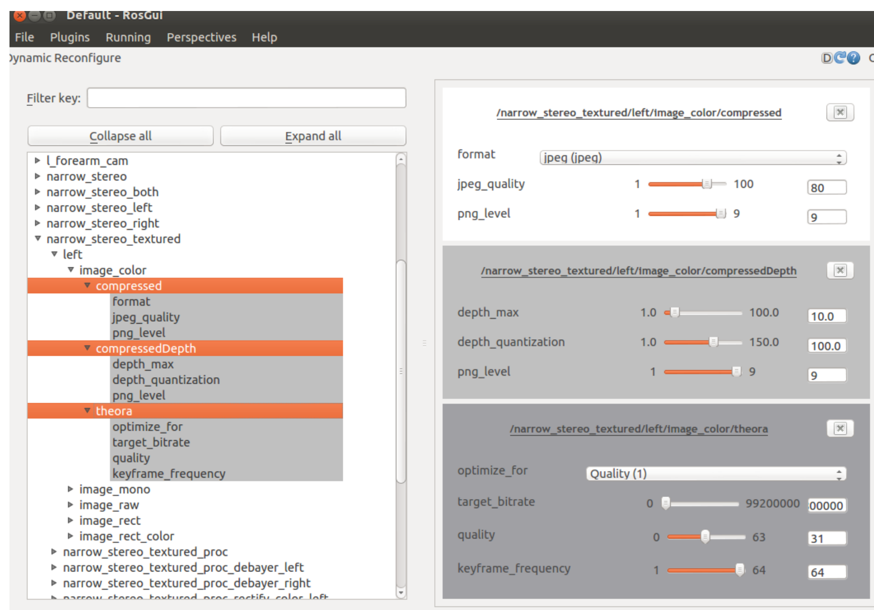


Figure 3.1: rqt_reconfigure

rqt_graph

The rqt_graph is a great tool for visualising the general structure of our system which also helps us check how the nodes are wired up and diagnose possible errors in the communications scheme.

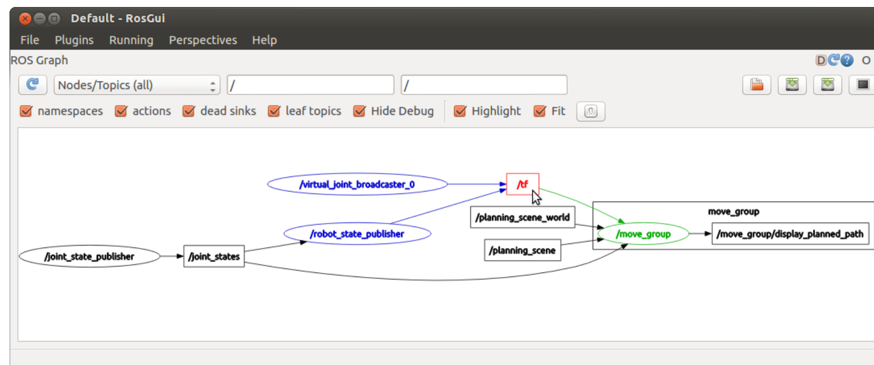


Figure 3.2: rqt_graph output

rqt_publisher

The rqt_publisher plugin kind of works in the same way as rostpic publish. The difference is that it offers a graphical user interface that makes it very easy to publish to the existing topics, select the content, the rate of publishing, etc. Making it an useful tool for testing our system among other uses.

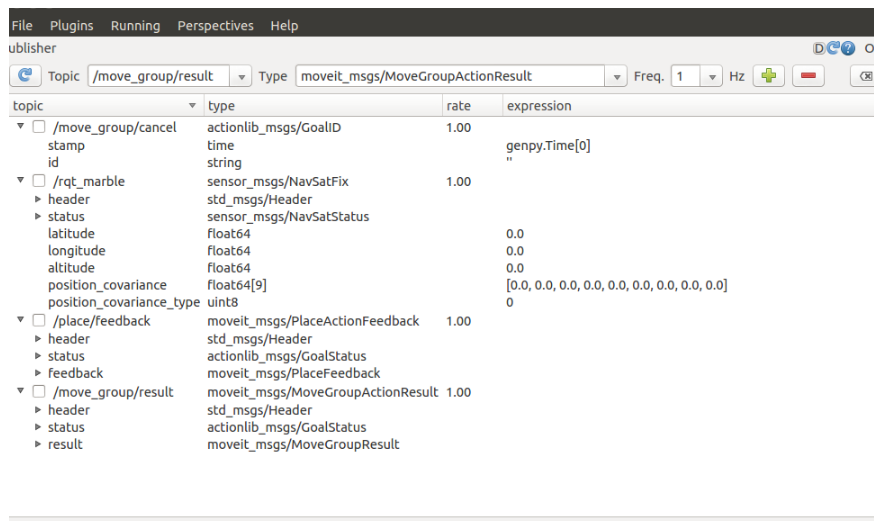


Figure 3.3: rqt_publisher

rqt_console

Rqt_console is a viewer plugin that displays messages published to rosout, which is the output of the ros output message system, the same way as stdout is the standard output for text output in regular programs. It collects messages over time, and lets you view them in more detail, as well as allowing you to filter messages by various means.

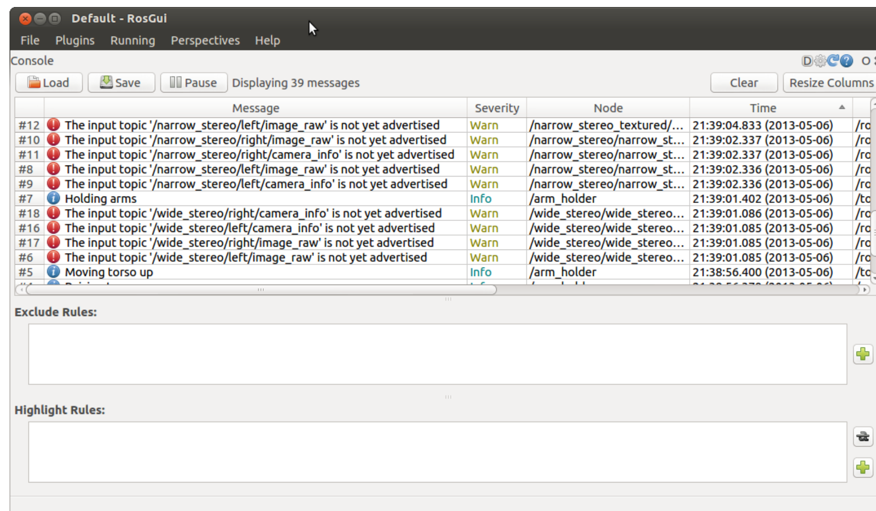


Figure 3.4: rqt_console

rqt_bag

In order to work with bag files with a graphical interface, rqt provides us with rqt_bag. Among its features it has the capability of playing and recording bag files, showing images as thumbnails in the timeline of the bag file reproduction, plot configurable time-series of message values, export messages in a time range to a new bag, etc.

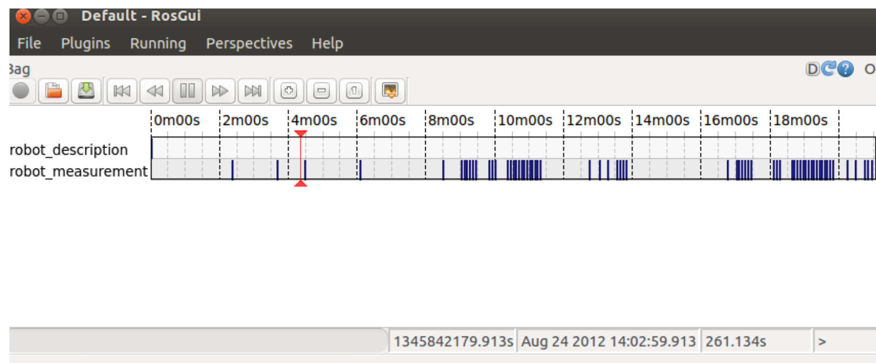


Figure 3.5: rqt_bag

rviz

Rviz is one of the greatest tools provided by ROS. This package visualises a wide range of elements like robot models, point clouds, images, markers etc. With the correct set up you can actually see a model of your robot and visualise the data it is producing in the same 3D environment.

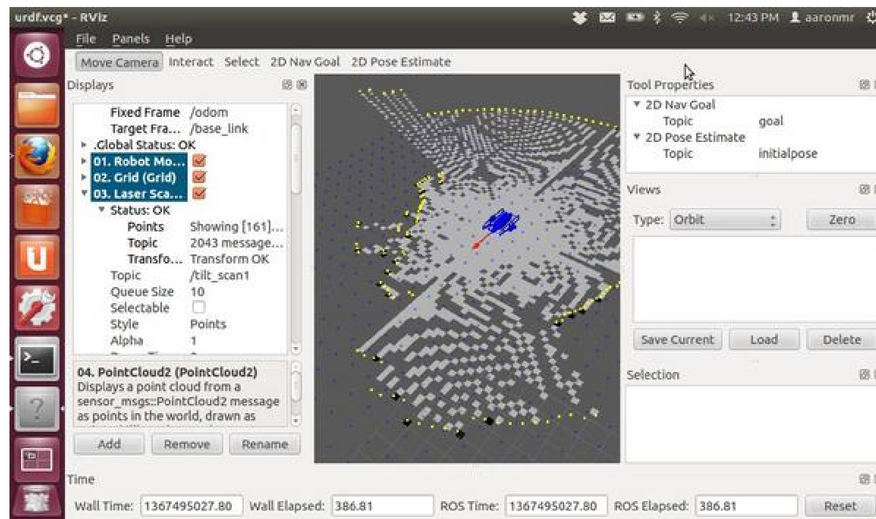


Figure 3.6: Rviz visualization

3.3.5 Community and collaboration

There is another level of great importance among the ROS environment, which is the community. The contribution of the community enhances several aspects of ROS, which include the following:

- *Distributions*: Ros Distributions are collections of stacks that the user can install. They work in a similar manner to the Linux distributions, they can be seen as a collection of software that maintains consistent versions across a set of software. This can be easily seen in the name of the packages installed, which follow the pattern "ros-distribution-package_name".
- *Repositories*: ROS relies on a federated network of code repositories in which we can find code developed by different institutions that release their own software components.
- *The ROS Wiki*: The ROS wiki provides a web portal and forum where you can find the documentation about ROS. Any singed up user can contribute to the wiki by writing documentation, providing corrections or updates, writing tutorials, etc.
- *Mailing lists*: There is a ros-users mailing list which works as the primary communication channel about ROS updates but also as a forum to ask questions among users.
- *ROS Answers*: This is the Q&A site for ROS related questions, is a great place to start searching when something goes awry.

3.3.6 TF

TF is how ROS manages coordinate systems relative to the environment or as part of the system itself. A robotic system typically has many 3D coordinate frames that

change over time, such as a world frame, base frame, gripper frame, head frame, etc. tf keeps track of all these frames over time, and allows you to perform several operations regarding these coordinate frames, such as:

- Calculation of transformation from one frame to another
- Transformation of data from one frame to another.
- Calculation of transformations between frames at a given moment e.g. five seconds ago, now, etc.

We will find three basic frames that tend to be part of every mobile robot platform:

- *map* The map frame contains the odometry data corrected by means of sensor that work at discrete jumps over time, like the product of a SLAM package, this frame usually provides corrections of the drift found in odom at discrete jumps.
- *odom* The transformation form base_link to odom is the one provided by the odometry in the vehicle (continuous source of data) that tends to drift over time.
- *base_link* Usually a frame set up as the origin in the robot. Generally it is set in the center of mass of the robot.

4 | Project analysis

In this chapter several topics are covered. First we will describe and analyse the behaviour of the underwater imaging sonar and how do we model it in the simulator. We will also go through the architecture of the AUV and the architecture of the developed software, first from a ROS point of view, and then from a class and use case diagram.

4.1 Sonar characterisation

Before talking about any further development that comes on top of the sensor, it is important to have a look at how our sonar behaves and what kind of data it produces.

4.1.1 Imaging

Our sonar, the imagenex 852, is a fan-shaped mechanically-scanned imaging sonar. This type of sonar will emit a fan shaped sound wave at each head angular position, wich are reached by mechanically rotating the internal sensor head. The beam's movement through the water will generate different points that form the sonar image of the area insonified. The different points represent the time (or slant range) that takes each echo to return, and the intensity of each point represent the echo return strength. Several characteristics are needed in order to produce an image of some quality from the sonar image.

- The angle through which the beam is moved is small.
- The fan-shaped beam has a narrow angle.
- The time to transmit a pulse is short compared to the time required to receive the echo.
- The echo return information is accurately treated.

The produced image provides the viewer with enough data to draw conclusions about the environment being scanned, thus, we aim for the autonomous robot to do the same, being able to recognise sizes, shapes and the surface reflecting characteristics of the different targets being focused.

The imaging sonar is mainly purposed as a viewing tool, since the data is better suited for a human to interpret, but we hope that with proper processing we can produce data with enough quality to be used in mapping and navigation.

4.1.2 Interpretation of sonar images

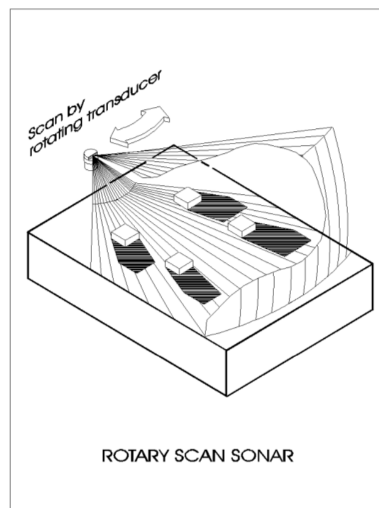
In many cases the sonar image will closely resemble an optical image of the same surroundings. But, in other cases the sonar image can be difficult to interpret due to its greater disparities with the expected image. This is why there are a lot of factors to be considered when processing this data for autonomous navigation and mapping. A sonar image will always have less resolution than an optical image or laser range data, and will also present more noise and strange effects.

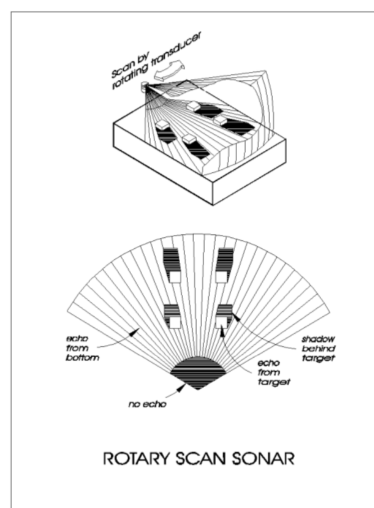
Generally, rough objects reflect sound well in many directions and are therefore good sonar targets. Smooth angular surfaces may give very strong reflections in one particular direction, but almost none at all in other directions. Some objects, such as smooth plane surfaces, may be difficult to see with a sonar. They can act as a perfect mirror (specular reflectors) reflecting the sonar pulse off in unexpected directions, never to return, or making it bounce affecting posterior readings and adding some sort of phantom effect.

Mainly we could sum up in two the factors that will affect the echo return and the strength: the material of the surface, and the normal of the surface to the pulse origin.

These two factors could be used to estimate the material the surface is made of and its normal, for example, since mud absorbs the pulse emitted it will provide low strength echoes, and might even not bounce the echo back, but we could use this to segment the materials the sea floor is made of.

Another factor to take into account is that the ranges shown on the sonar image are "slant" ranges, this means that we won't usually know the relative elevations of the targets, only the range from the transducer. This means that two targets, which are displayed in the same location of the beam, may be at different elevations. This is due to the fact that both have reflected an echo that produces the same return time. In order to estimate with the height of the objects we can analyze the shadow they produce, as shown in the figure.





4.1.3 Sonar configuration

The sonar can be configured through several parameters that can greatly modify its behaviour and the quality of the readings. Due to the importance of these parameters they have to be tuned correctly, but also understood properly. Those parameters are the following:

- *Range*: It defines the radius of the circumference that the readings will cover. It can be set to 5, 10, 20, 30, 40 and 50 meters. We have to take into account that, the higher the range, the larger the time required awaiting for the echo to come back to the sensor, meaning that a bigger delay is introduced.
- *Direction*: This parameter allows us to change the direction in which the sonar head rotates so we can have clockwise or anticlockwise behaviour.
- *Gain*: The gain defines the multiplier of the intensity of the acoustic pulse in dB. High gains produce higher intensity results but also more noise. It varies from 0 to 40 dB in 1dB increments.
- *Absorption*: The absorption coefficient, along the frequency and the distance, take part in the definition of the transmission loss. 20 = 0.2 dB/m 675, 850 KHz.
- *Train angle*: 0 to 140 (-210 to +210 degrees) in 3 degree steps. Experimentally, we have discovered that this parameters works along with the sector width angle to set the centre of the fan shape.
- *Sector width*:It is the width in degrees of the area that the sonar head will cover. This, along with the train angle, help us define sections that will be scanned, helping us reduce the refresh time among other features. It can take values between 0 and 120 (0 to 360 degrees) in 3 degree steps.
- *Step size*: Defines the size of the sonar moving transducer head, which can be 0 (no step), 3 or 6 degrees per step.
- *Pulse length*: Length of acoustic transmit pulse, ranges from 1 to 255 μsec in 1 μsec increments.
- *Data points*: Can select between 250 and 500 points in each beam.
- *Switch delay*:This delay can make the head pause before sending its return data to allow the commanding program enough time to setup for serial reception of the return data. It can take values from 0 to 510 miliseconds in 2 miliseconds increments.
- *Frequency*: It can take be set to 675 KHz or 850KHz, this parameter affects the transmission loss.

Transmission loss and tunning of absorption and frequency

The definition of transmission loss(TL) is *"The accumulated decrease in acoustic intensity as an acoustic pressure wave propagates outwards from a source."* As the acoustic wave propagates through its medium the signal intensity is reduced with increasing range due to spreading and attenuation/absorption. Stoke's law of sound attenuation is a formula where the attenuation of sound in a Newtonian fluid due to its viscosity is calculated. It states that the amplitude of a plane wave decreases exponentially with the distance travelled at a rate α, given by

$$\alpha = \frac{2\eta\omega}{3\rho V^3}$$

where η is the *dynamic viscosity coefficient* of the fluid, ω is the sound's frequency, ρ is the fluid density, and V is the speed of sound in the medium. Taking into account this equation we can see that in order to have a lower rate we should set a lower frequency.

The absorption, measured in dB/m is represented by the attenuation coefficient a which has two primary causes: viscous friction and ionic relaxation. The attenuation caused by viscous friction comes from the conversion of sound energy into heat due to internal friction at a molecular level within the fluid. There are several factors can be analysed deeper, but in general we can see that the terms depend proportionally on frequency, so a lower frequency should produce less absorption.

4.1.4 Communication protocol

Along other data the sonar provides the current sonar transducer head and an array of slant ranges with the intensity of each echo return. In the following images we can see the command used for changing the configuration and the format of the data received.

Byte #	Description							
0 – 7	**0xFE**	**0x44**	Head ID	Range	Reserved 0	Rev	Reserved 0	Reserved 0
8 – 15	Start Gain	Reserved 0	Absorp-tion	Train Angle	Sector Width	Step Size	Pulse Length	Reserved 0
16 – 23	Reserved 0	Reserved 0	Reserved 0	Data Points	Reserved 0	Reserved 0	Reserved 0	Reserved 0
24 – 26	Switch Delay	Freq-uency	Term. **0xFD**					

Byte #	Description					
0 to 5	ASCII 'I'	ASCII 'M',or 'G' '	ASCII 'X'	Head ID	Serial Status	Head Pos (LO)
6 to 11	Head Pos (HI)	Range	Reserved 0	Reserved 0	Data Bytes (LO)	Data Bytes (HI)
12 to (N-2)	Echo Data 252, 500 Data Bytes					
N-1	Term. **0xFC**					

4.2 Gazebo

Even though ROS supports building models of our robots and articulating them, we need a simulation environment that allows us to interact with a scene. For this purpose Gazebo is very adequate.

Gazebo (http://gazebosim.org/) is a multirobot simulator for outdoor environments. It is capable of simulating a population of robots, sensors, and objects in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects.

In earlier ROS versions Gazebo was integrated with ROS, being a ROS package (the same can be said for PCL). Gazebo is now independent from ROS and its installed as a standalone package.

Among its features we can find:

- Dynamics Simulation (Using ODE, Bullet, Simbody, and DART as physics engine)
- Advanced 3D graphics using OGRE
- It provides various sensors such as laser range finders, 2D/3D cameras, Kinect style sensors, contact sensors, force-torque, and more.
- It provide simple interfaces for plugin development, letting use the tools provided for our own modified elements and behaviours
- Gazebo provides a variety of robot models including PR2, Pioneer2 DX, iRobot Create, and TurtleBot. And it's very simple to add our own using SDF and Xacro.
- Simulations can be run on remote servers and interfaced with using socket-based message passing, the same way as we could do with ROS.

The newest version of Gazebo includes water dynamics simulation among its features, letting us model a submarine with its own propellers and interact with them. Sadly this feature is very recent and the Gazebo versions that can be integrated with ROS Indigo do not contain it.

For our sonar simulation gazebo provides two types of base sensors: a collision based sensor which is already included as part of Gazebo, and a Laser Ray based sensor that works through modification on a plugin.

4.2.1 SDF Modelling

SDF is an XML format that describes objects and environments for robot simulators, visualization, and control. Originally developed as part of the Gazebo robot simulator, SDF was designed with scientific robot applications in mind. Over the years, SDF has become a stable, robust, and extensible format capable of describing all aspects of robots, static and dynamic objects, lighting, terrain, and even physics.

There are a lot of tutorials where one can learn about how to model elements with SDF, but here we will focus on how do we model our sensor so we can understand the parameters and the modifications made on the SDF format associated to the sonar sensor in the development section.

Sensor definition

There are two types of sensors relevant to this project, sonar based and ray based. From the <sensor> XML element we use the following tags:

- *<always_on>* With the value of true/false or 1/0, this parameter lets us decide if the sensor should start working the moment the model is spawned on the simulator or if it should wait for some process to subscribe to its topic to start working. For testing it is a good thing to set this to false, but having it set to true mimics what would happen in the real world (unless we have a sensor that awaits for an event of some sort to start working).
- *<update_rate>* This is the rate at which the sensor update event happens. Usually this will directly correspond to the rate at which the data is publish, but not necessarily.
- *<plugin>* This tag is for the plugin associated to the sensor. This allows us to define the Gazebo-ROS plugin linked to the sensor. This tag also contains its own XML child elements, that work as parameters for the plugin itself and will be addressed specifically for each plugin.

4.2.2 Sonar sensor

This sensor works as an ultrasound range sensor, it simulates the sound pulse emitted using a cone, and gives us the closest collision between the come and the environment, published as a message with information about the 3D position of the collision.

XML parameters

For sonar based sensor we have the *<sonar>* element, which is a child of the *<sensor>* element, with the following child elements:

- *<min>* Minimum distance at which we detect collisions.
- *<max>* Maximum rage of the sonar.
- *<radius>* Radius of the cone at maximum range.

4.2.3 Laser sensor

The laser sensor is a ray based sensor, and as such, it receives the collision of a number of preconfigured rays that span a parametrized area. This type of sensor provides the collision of each ray with the environment. The main difference with the sonar sensor is that the sonar will hand out the collisions using a cone, while the laser only obtains the collision using a line, which is not very realistic if what we want to simulate is a sonar that emits an ultrasound pulse.

XML parameters

For ray sensors we have the *<ray>* element as child of the the *<sensor>* element, and can be configured with the following childs:

- *<scan>*
  - *<horizontal>* Horizontal setup of the rays
    - *<samples>* Number of simulated rays to generate per complete laser sweep cycle.
    - *<resolution>* This number is multiplied by samples to determine the number of range data points returned. If resolution is less than one, range data is interpolated. If resolution is greater than one, range data is averaged.
    - *<min_angle>* Horizontal angle where the swipe begins.
    - *<max_angle>* Horizontal angle where the swipe ends.
  - *<vertical>* Has the same childs as the *<horizontal>* element.
- *<range>*
  - *<min>* Minimum distance for each ray.
  - *<max>* Maximum distance for each ray
  - *<resolution>* The resolution of each ray range.

It is important to note that the ROS plugins required for these sensors are located under the *charle_description* package.

4.3 Platform architecture

Here we discuss about the platform that has supported the development of the software module developed for this project which name is Charl-e.

Charl-e's structure is based on a modular and easy to assemble architecture. In this architecture we can differentiate between structural architecture, electronic architecture and software architecture.

4.3.1 Structural architecture

General structure

The vehicle is composed of an external frame made of high-density polyethilene (PE-500) due to its low water absorption, low density, strength and excellent properties to be mechanized. In this structure we can find attached different kinds of devices. Such as wet sensors, thrusters, some other actuators, the housings where the electronics and the batteries are disposed, and the tilting imaging sonar. The housings and sonar are fixed to the frame in a fashion that allows fast access and easy external manipulation in case a removal has to be performed. All this components are attached to the frame using marine steel spoke resistant to marine environments. The AUV is propelled by 4 sets of SeaBotix BTD150 thrusters attached to the external frame in the following fashion: two at each side pointing forward along the surge axis, and other two along the vertical axis, one on bow and another on stern. Thus providing the vehicle with the ability to move forward and backwards, up and down, turn board and starboard, and also manipulate its pitch and yaw. Due to the character of the competition for which this robot was developed, there was no actual need of pitch control, so just an adequate trimming was enough to provide stability.

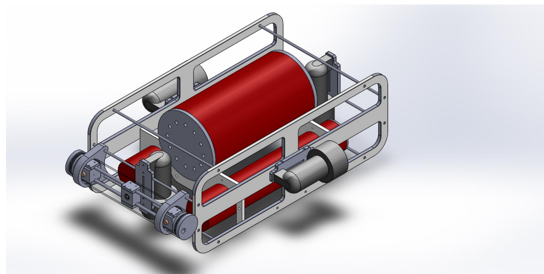


Figure 4.1: Main housing

Main housing

For the pressure vessel (main housing), that contains the electronics, we have a 200mm diameter PVC pipe. The cover cap used for this enclosure box, has been made out of a solid bar of 200 mm diameter PVC16 tube The caps come with 2 o-rings of 4 mm section, in order to provide a perfectly watertight closure. These caps were made with a CNC (Computer Numerical Control) machine, one of which features waterproof connectors leading inside to the electronics. The box contains most of the AUV systems: embedded computer, power and control boards for the SeaBotix thrusters, Mti IMU, pressure/depth sensor, servo controllers and more.

Electronics rack

Inside the main enclosure the electronics are organized on a newly designed interior rack and ring system. 3D printed rings slide into the tube and are joined together machined plastic boards which also serve as the differing floors of the AUV electronics system (see fig 2). This new design presented unique challenges for the design team due to the need to fit the same amount of electronics as the year before, in an even smaller space. The four floors of the electronic rack are as follows:

- Floor 1: Battery shelf - supplies power to internal electronics
- Floor 2: Computation shelf- where the computer and its power supply are located
- Floor 3/ 4: Electronics shelf- Containing the control systems and electronics boards

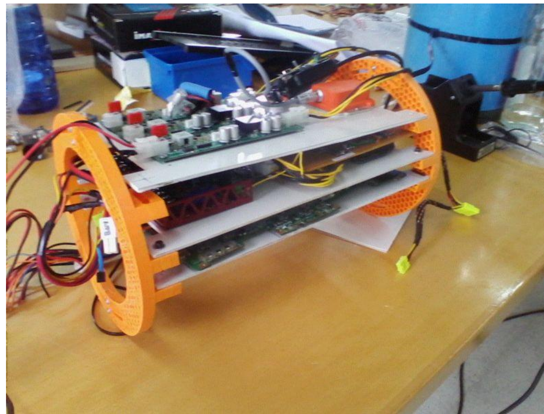


Figure 4.2: Electronics rack

Camera and laser

The new design for the camera housing switched from a pan-tilt system to an encapsulated tilt system. With this change, the camera is no longer outside of the body profile, creating less drag and buoyancy issues. The camera features a servo control system that rotates three individual housings connected by rods. A bearing support system was used to reduce the torque load on the servo. The addition of a second laser helps increase the computing accuracy of the obstacle avoidance sensing. The design uses two green laser pointers of 20mW and an analog CCD camera with a resolution of 640x480.

Battery packs

They are made of 90mm PVC tubing and house in total 6 H-38120S Headway LiFePO4 cells. The tubes are sealed with custom designed PVC caps with waterproof connectors to allow the batteries to power the motors. A dual feature of the battery packs is their ability to move on their supports. This makes them a key component of the buoyancy system. Also, this design provides an easy way of replacing batteries.

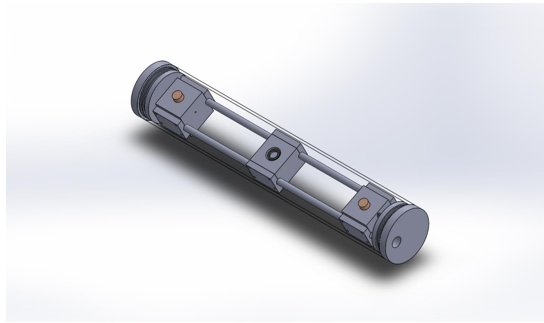


Figure 4.3: Camera housing

Thrusters

The AUV features 4 SeaBotix BTD150 thrusters. Two are placed horizontally on both the starboard and port sides of the vehicle. They are responsible for both forward and reverse surge as well as yaw rotation in either direction. The other two are placed at the bow and stern of the AUV and are mainly responsible for propelling it down and up. These two thrusters can also be used to adjust pitch; however, proper trimming of the vehicle normally renders this an unnecessary movement.

Sensors

The sensors are directly attached to the frame and can be easily moved to suit the needs of the mission. Another innovation is the positionable sonar which can rotate 90 degrees; allowing the vehicle to make maps of objects on the bottom.

4.3.2 Electronic architecture

Main computer

For the main computing unit we have an Acer one ultra book laptop. But we have removed the keyboard, screen, and the whole outer structure, leaving only the main board. This allows us to reduce the space occupied, heating and power consumption.

Data communication

The devices communicate with the computer using serial communication. For the Sonar, Xsens Mti, and camera capturer we use a direct connection to the computer, and for interfacing between sensors/actuator and the main computer we use Arduino Mega I/O Boards.

Power supply

We use three packs of LiFePo4 batteries as power supply for the whole AUV. Two packs of six cells are used for the four thrusters and the other pack of four cells is used for the electronic of the main housing. The electronic design makes a separation between power electronic and sensor electronics. To protect the batteries, BMSs monitor every cell of each pack of batteries. The battery state is controlled during the discharge and charge states. The electronic battery pack is connected to three mini-Box DC-DC converters. They provide three different voltage levels (5, 12 and 24V) to the electronics devices except the thrusters.

Proprioceptive sensors

Monitoring the state if the vehicle is critical for safety reasons. There are several sensors equipped in the vehicle that allow us to monitor this state. We measure both internal variables like temperature, pressure, humidity inside the housing, and external variables like pressure or speed. One of the most important sensors inside the main housing are the humidity and leak sensors that will allow us to emerge in case of leaks inside of it. Also the temperature sensors are important since we have a lot of electronic devices inside a closed housing, which generates a considerable raise in temperature.

Inertial Measurement Unit

An inertial measurement unit it is an indispensable device to know the orientation and the pose of the vehicle in each moment. We incorporate an MTi Xsens sensor to solve this task. This sensor is able to provide an accurate $360\,\deg$ pose and the orientation referred to Earth's magnetic field and gravity. Sadly, this is not entirely the case when it is located inside the main housing, surrounded by electronics. We have yet to characterize its behaviour inside the main housing but we are also considering locating it in its own housing.

Sonar

Avora vehicle is equipped with an Imagenex 852 miniature sonar. Due to its small size, weight and easiness of interfacing, this sensor is well suited for usage in small underwater vehicles. This sonar can work in a range of distances between 15 cm and 50 m; the beam width is $2,5\,\deg$. To make a precise $360\,\deg$ scan, the sonar needs 16 seconds. The sonar signal frequency can be adjusted, we set it to 850 KHz to reduce the contributions of noise. The sonar is fixed to the AUV by means of a servomotor. This allows changing the angle (tilt) of the scanning plane of the sonar from horizontal to vertical. The vertical orientation is employed to estimate the altitude of the vehicle working similarly as an altimeter.

Sonar data are processed to register environment features that are integrated by the mapping system

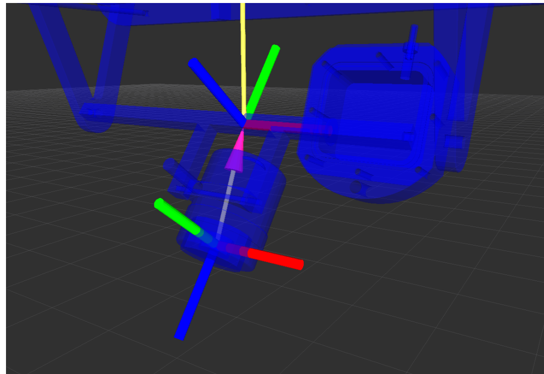


Figure 4.4: Sonar rotation mechanism

Propulsion system

For propulsion we have 4 SeaBotix BTD150 thrusters two for depth control, and two for linear and angular movement. They are controlled by two dual-channel Roboclaws 2x15A boards. and powered by two packs of six cells of LiFePo4 batteries connected to a Y-connector. The Y-connector combines the power of the two packs of batteries and can provide power to the four thrusters with only one pack of batteries connected. This way, the AUV can continue its mission if one pack of batteries becomes discharged or the BMS has shut down that pack due to any other problem (excessive current, severe unbalance...). For security a power cut-off circuit is used to stop the thrusters and can be activated using pushing the emergency button or by software.

Vision system

We use an analog color camera equipped with a Sony HAD II CCD sensor (vertical resolution 700 lines). EasyCap USB camera framegrabber digitizes the analog camera signal before being processed by the computer. Two 20mW green laser pointers are situated at both sides of the camera to calculate distance and an approximate size of the objects. An RC servo motor makes possible to orient the camera over 170 deg, this camera tilting can obtain frontal images, sea bottom or surface images.

4.3.3 Software

Most of our software is integrated using ROS as middleware. This integration allows us, following our architectural design, to develop modules quite fast and easily. On the other hand we rely on an asynchronous architecture which depends on different modules communicating among them and performing different tasks at the same time, for which ROS is very suitable.

Architecture

Through the lifespan of the project, a layered modular architecture has been followed as a design standard for the development of the different packages that build our system in order to give it the different capabilities we will need in the variety of tackled scenarios. This architecture aims to create a considerably simple system, which should be easy to understand by both developers and the rest of the team. Also it tries to provide an architecture in which adding a new module can be done quickly, thanks to the independence between modules and the intra system communication scheme used, which is basically supported by ROS. We will go through the different layers of the architecture and will comment on the most important modules of each layer.

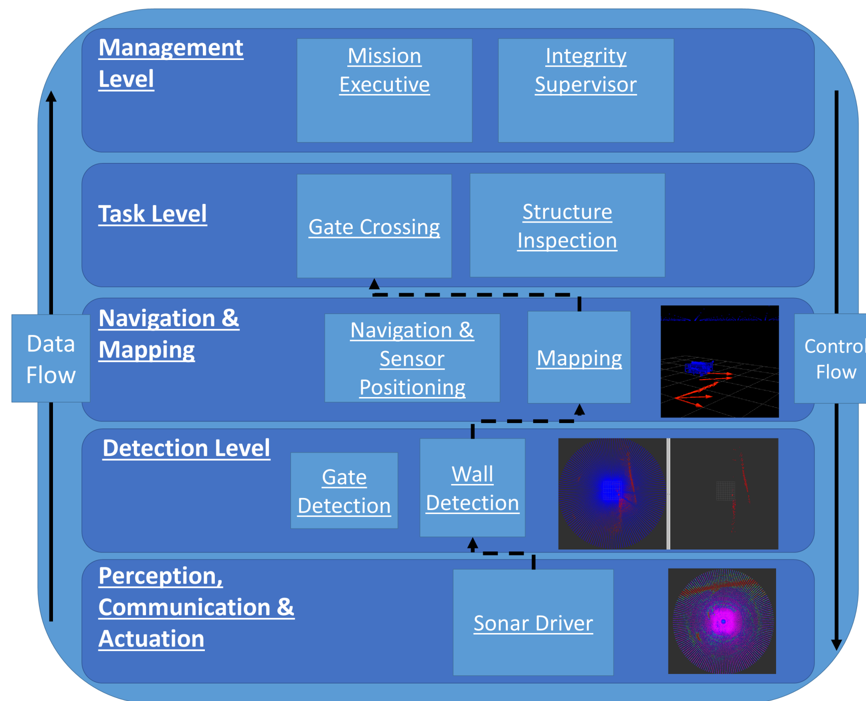


Figure 4.5: Software architecture

World modelling

Our imaging sonar is the main tool used for modelling our environment, since it is mounted with a servomotor it can provide us with data while tilting to obtain a proper map. Due to the accumulation of errors and the accuracy of the sensor measurements, we use a probabilistic world model, we basically accumulate evidence of the different elements found in our environment, as walls and buoys, and push them to the upper layers of the architecture with an uncertainty value, which will be managed by the mapper module in order to accumulate the evidence of certain elements.

Sensor and actuator level

In this level we can find the different drivers needed to provide the rest of the system with the data coming from the sensors and also the communication protocols needed for the hardware. This level will also receive control data from higher levels so the different actuators can be controlled, and also some of the sensor dynamically reconfigured, e.g. changing the sonar range, or sector size.

Detection level

Here we can find the modules that are in charge of interpreting the data coming from the sensor and produce something useful for upper layers. These modules will process the sensor data and will transmit to other modules the characteristics of the elements detected, and also a level of uncertainty associated to the measurements, which value will depend on the accuracy and drift of the sensor, but also the level of confidence that the INS generates while is not corrected by sonar odometry.

Navigation and mapping

We aim to navigate using an INS and fusing it with odometry estimated from the sonar images. The INS would be fed by our Xsens IMU, and then would be reinforced by the occasional data coming from the sonar. The odometry produced by the sonar is a product of the wall detection algorithm, since we can use the position of the walls of the arena to localise ourselves and correct the drift that the INS produces. We trust that periodical sparse scans will be enough so we can keep up with the rest of the demands of the system that require different uses of the sonar. Basically the combined odometry will be produced by an Extended Kalman Filter (EKF) that will fuse IMU data with occasional odometry from the sonar. Since the data from the INS will be much more frequent the system will behave mostly as an INS, and the sonar odometry will be in charge of palliating the drift of the INS. Also, the system is prepared for accepting visual odometry from the algorithm proposed in [6] which can be used as another input for the EKF node. For mapping we have developed a node that builds a Multi Surface Level Map using point clouds that are a product of the sonar readings. This module can be fed with both the processed sonar data, and the product of the detection nodes.

Task level

In this level we find the modules that perform different tasks, e.g. gate crossing, wall following. These tasks are basically modelled using a relatively simple state machine. These automatons will flow through one state or to another depending on the current state of the platform and the environment, e.g. when a buoy is detected.

Management level

This level is the one in charge of the general control of the robot, it encompasses the highest level behaviours. Here we can find the mission executive and the integrity supervisor modules. The first one will be the one launching the different tasks needed for accomplishing a mission. Also it will perform the tasks simple enough not to require a state machine. The second one will be in charge of taking over the system in an emergency. It will monitor the different sensors to detect dangerous situations and take actions to avoid them or sustain the lowest possible damage

4.4 Analisys

4.4.1 The competition

In order to further comprehend the motivation of this work and its environment it is interesting to understand how the competition for which the robot and the software were designed takes place. The SAUC-E competition takes place at La Spezia, Italy, in a salt water basin of 120 meter long and 50 meter wide, with a constant depth of 5,5 msw and negligible currents. The competition it's formed by a series of missions that the AUV will have to accomplish which are explained next:

Gate Crossing
The first mission consists on crossing a gate made with two buoys situated at least at 8 meters from the starting point and 2 meters apart from each other. This distance makes the localization of the gate quite difficult by means of visual detection due to the poor visibility given in the environment and the fact that there's no prior knowledge of where the buoys are situated from the sonars initial position. The task can be accomplished by traversing at the controlled depth towards the centre of the Arena, make a 90 degree turn, and pass through the validation gate.

Underwater structure inspection
In this mission the vehicle must perform the inspection of an "underwater structure" made of cylinders in a pipe-like arrangement. The structure consists in concentric cylinders and a set of circles placed on top of another set to increase the object's height. The objective is to inspect this structure an imaging sensor while maintaining a required stand-off distance from it.

Wall Inspection
This mission consists of four parts. First a wall must be followed with a position between 2 and 4 meters from it and an "anomaly" (buoy) must be detected. Second, after detecting the anomaly, the AUV must signal the ASV to go to the location where the anomaly is. Third the ASV must be capable of acquiring and detecting the anomaly based on the information given by the AUV and giving the location of the anomaly.

Black box detection and area mapping
In this mission three tasks have to be performed. First we have to build a map of the

environment. Second, a stationary black box must be found, and third, we have to surface in the zone within 3 meters of the black box.

As we can see from the missions, this competition requires the robot to be able to make a map of its surrounding environment in order to know its location and identify some elements like walls, buoys, boxes etc. But not only considering SAUC-E's environment we find the need for mapping. From the beginning if we think of autonomous navigation, we think of mapping and localisation. With this sort of missions and needs it is logical to develop a good and reliable component in order to map an underwater environment, and also it has great value outside the AVORA project itself since... In an underwater environment we could think of mapping using feature detection over camera, laser, or fusing laser and camera data which would lead us in the direction of Visual Simultaneous Localisation And Mapping (V-SLAM). The main problem with this approach is that laser beams are attenuated and dispersed in water while cameras limit most of their applications to navigation in clear water and close to the seafloor. In the case of SAUC-E competition we find an environment with such conditions of poor visibility, making the laser and camera not very useful unless we are working within distances of around 2 meters. For all of the above, we approach the underwater map construction by means of a sonar.

4.4.2 Considered scenarios

In order to approach the development of this project we need to define the different situations we might face. In this section I will discuss about this scenario and the capabilities developed in order to perform correctly in any them.

Scenario 1: Static observer and static environment

In this scenario the platform remains still while the sonar head gathers readings. The movement considered is the one of the sonar head, since this one can be easily measured by the movement of the servomotor that rotates the sonar for the 3D scan. The environment is also considered static, so no elements are supposed to move, appear or disappear.

In order to perform the mapping under the conditions explained we only need the basic features of the software module, which are the data filtering and its posterior conversion to map blocks. We also won't need any actuators or different sensors from the vehicle rather than the sonar and the servomotor in it.

Scenario 2: Dynamic observer and static environment (Step movement)

This scenario does not consider mapping while moving. Instead we assume that we will have different static positions from which we will map. In this scenario use each of these static poses to provide ourselves some odometry so we can know where we are, and also add to the map the data we have just gathered. In order to accomplish this, a custom Iterative Closest Points (ICP) that works with the developed structures for Multi Surface Level Maps has been developed. ICP will be discussed further in this section.

Scenario 3: Dynamic observer and static environment (Continuous movement)

In this scenario we aim to provide odometry data while we move, so we can transform the data received while moving. This will also depend on how fast the sensor can provide us with reliable data so that the algorithm can find the matched needed to produce an accurate (and correct) result. For this purpose, some modifications have been added to the original ICP algorithm, which will be explained in the development section.

4.4.3 Methodology

The methodology followed in general has been in spiral with hints of prototyping. First of all there was an objective identification, but then each module has been more or less developed sequentially. The functionalities of the packages where developed and tested separately, but, in order to test smaller versions without the complete functionalities, small prototypes of each module were developed. For example, in the beginning the system was tested with thresholding and detection but not a proper structure for map generation.

4.5 Design

4.5.1 Architecture from ROS point of view

Here we will talk about how the system is configured in terms of the ROS Computation Graph, going through the packages and nodes that build the system. There are some structures we have to talk about before jumping to the description of the nodes and the data they produce. We can also see the computation graph extracted from *rqt_graph* for an overview of nodes and topics in figure 4.6.

Messages and data types

In order to communicate the nodes we need different message types that will hold the data produces the nodes of our system. There are several messages that have are of importance for the communication of these nodes and the processing that takes place inside them.

> **PCL points and point clouds**
> PCL provides templated clouds so that the user can create clouds with any point data type provided by PCL. The user can even define its own point type and integrate it to create point clouds of custom point types. The point type used for this project is the PCL::PointXYZI type. This point contains the regular XYZ position data, but it also adds an intensity value, which we can use to hold the echo return strength for our sonar. The point cloud that holds this data has been renamed to "IntensityCloud" so it can be more user friendly when developing.

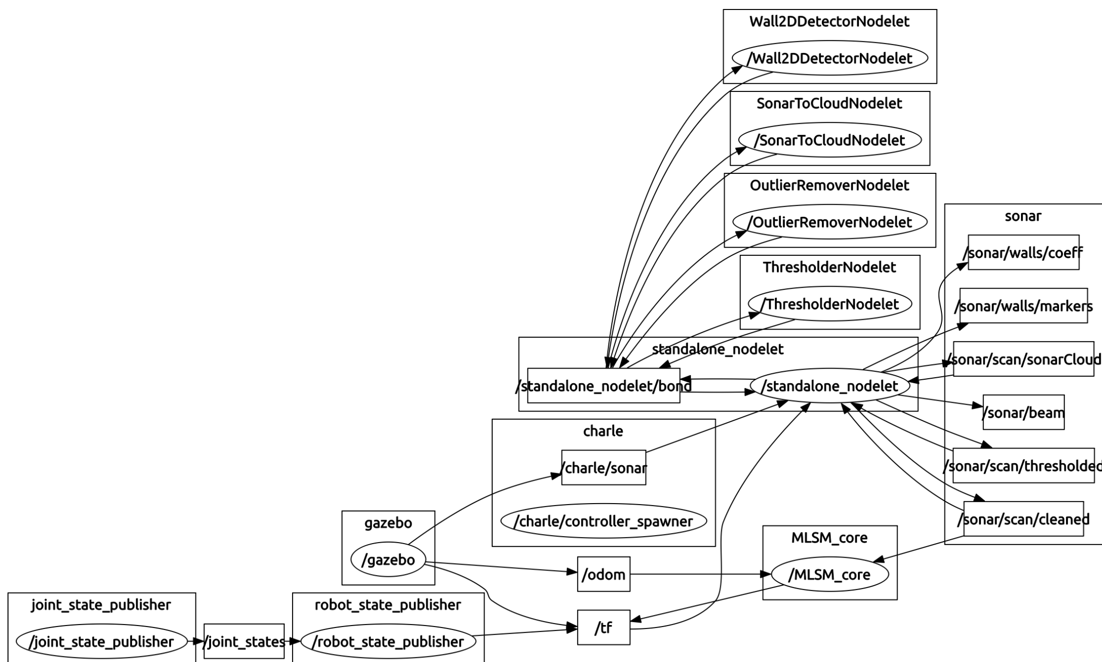


Figure 4.6: Computational graph of project nodes

ROS point clouds

ROS provides its own pointclouds to use as messages which are of *sensor_msgs::PointCloud2* type. This clouds do not hold the attributes in the same way as PCL clouds, instead, they are designed to hold pretty much whatever point cloud type we want and then easily conver it back to a PCL cloud, which are easier to manage. This is the structure of a *sensor_msgs::PointCloud2*.

```
std_msgs/Header header
uint32 height
uint32 width
sensor_msgs/PointField[] fields
bool is_bigendian
uint32 point_step
uint32 row_step
uint8[] data
bool is_dense
```

As we can see there is not much to get directly from it. The other problem we would face is that we would not be able to use all the tools provided by PCL, so the usual way to proceed is to convert these clouds to PCL clouds and them work with them.

Custom cloud messages

Even though ROS and PCL provide quite complete data types, we needed some data that was not covered in their point cloud types and messages, which was the timestamps of each recorded beam. Thus, we created a new message con-

taining the original *sensor_msgs::PointCloud2* and an array of timestamps containing the timestamp of each point. This was required for the modified ICP which takes into account when each point is measured. This message type is *avora_msgs::StampedIntensityCloud*.

Driver package

imagenex_852_driver
This node is the sonar driver. It basically uses serial communication for implementing the protocol specified in the manual of the Imagenex 852 sonar. It is one of the components that did not change through the evolution since the implementation was solid enough. Publications:

- /sonar/raw (avora_msgs::SonarScanLine): ROS topic where the raw sonar data is published, along some configuration options and the ROS message header. The struct is as follows:

```
Header header
# scan
float32 angle
uint8[] intensities

# sonar parameters
float32 maxrange_meters
float32 range_resolution
uint8 gain
float32 sensorAngle
```

Sonar_processing package

This package holds the nodes responsible of processing the raw data produce by the sonar driver. It contains the implementation of the filtering steps and also nodelets for detection in point clouds. *SonarToCloud*
This nodelet takes the *avora_msgs::SonarScanLine* and turns it into a point cloud in the robots reference system, using the data from the sonar servomotor position. It allows for different configurations. It can store the beams until a certain amount are stored or the servomotor position is changed. It can also convert the readings to laser-like point clouds, where each beam produces only one valid measure or just turn them into point clouds that will be filtered later.

Subscriptions:

- avora_msgs::SonarScanLine

Publications:

- /sonar/scan/sonarCloud (avora_msgs::StampedIntensityCloud)

- /sonar/scan/laserCloud (sensor_msgs::PointCloud2)

Thresholder
This nodelet is the first step to produce clean data from the raw readings. It performs an intensity based thresholding operation on the point cloud, being able to select different methods that will be explained in the development section.
Subscriptions:

- sesor_msgs::PointCloud

Publications:

- /sonar/scan/thresholded (avora_msgs::StampedIntensityCloud)

OutlierRemover
This removes outlier points from the point cloud, it mostly helps eliminate the high intensity points that appear alone or in very small clusters. Subscriptions:

- sensor_msgs::PointCloud2

Publications:

- /sonar/scan/cleaned (avora_msgs::StampedIntensityCloud)

LineDetector
This nodelet takes the incoming clouds and detects lines in them producing the points associated to the line. Subscriptions:

- avora_msgs::StampedIntensityCloud

Publications:

- /sonar/scan/line (sensor_msgs::PointCloud2)
- /sonar/scan/line/coefficients

mlsm_manager

Here we can find the node developed for mapping and scan matching, which makes use of the classes and tools developed for handling MLS maps. *MLSMCore*
This node is in charge of mapping. It receives *avora_msgs::StampedIntensityCloudPtr* which contains the cloud of generated points and the timestamps for each point. This node will produce a point cloud with the means of each block, or the blocks using a ROS marker array. It also produces the frame transformation from "odom" to "map" which helps us localise ourselves.
Subscriptions:

- avora_msgs::StampedIntensityCloud

Publications:

- /MLSM/cloud (visualization_msgs::MarkerArray)
- /MLSM/Markers (sensor_msgs::PointCloud2)

4.5.2 Class diagram

Generally, object wise, each ROS node object inherits from the parent *ros::Node* class, and the functions and procedures are implemented as part of the child node class. This is the case of the processing nodelets. Due to their simple nature, they can easily be implemented as nodes that contain some processing functions. In the case of the mapping node we can find a more elaborate construction, but still, its quite simple thanks to the segmentation of processing units into nodes and nodelets.

```
gazebo::sensors::ImagingSonarSensor
-----------------------------------
private:
  physiscs::MeshShapePtr sonarShape
  transport::SubscribetPtr contactSub
  msgs::SonarScanLineStamped scanLineMsg
  ContactMsgs_L   incomingContacts
  math:Pose sonarMidPose
  double rangeMin
  double rangeMax
  double radius
  double angle
  double step
  double openingAngle
  int binCount
-----------------------------------
public:
  ImagingSonarSensor()
  ~ImagingSonarSensor()
  Load(const std::string& worldName)
  Init()
  bool UpdateImpl( bool _force)
  Fini()
  [Setter and getters]
  OnContacts(ConstContactsPtr& msg)
```

Figure 4.7: Collision based gazebo sonar

```
gazebo::AvoraLaserSonar
-----------------------------------
private:
  physiscs::WorldPtr
  sensors::RaySensorPtr
  ros::NodeHandle* node_handle
  ros:Publisher publisher
  ros:Publisher scanLinePublisher
  avora_msgs::SonarScanLine scanLine
  string topic
  string frame_id
  int binCount
  double step
  double angle
  SensorModel sensor_model
-----------------------------------
public:
  AvoraLaserSonar()
  ~AvoraLaserSonar()
  Load(sensors:SensorPtr sensor, sdf::ElementPtr sdf)
  Reset()
  Update()
```

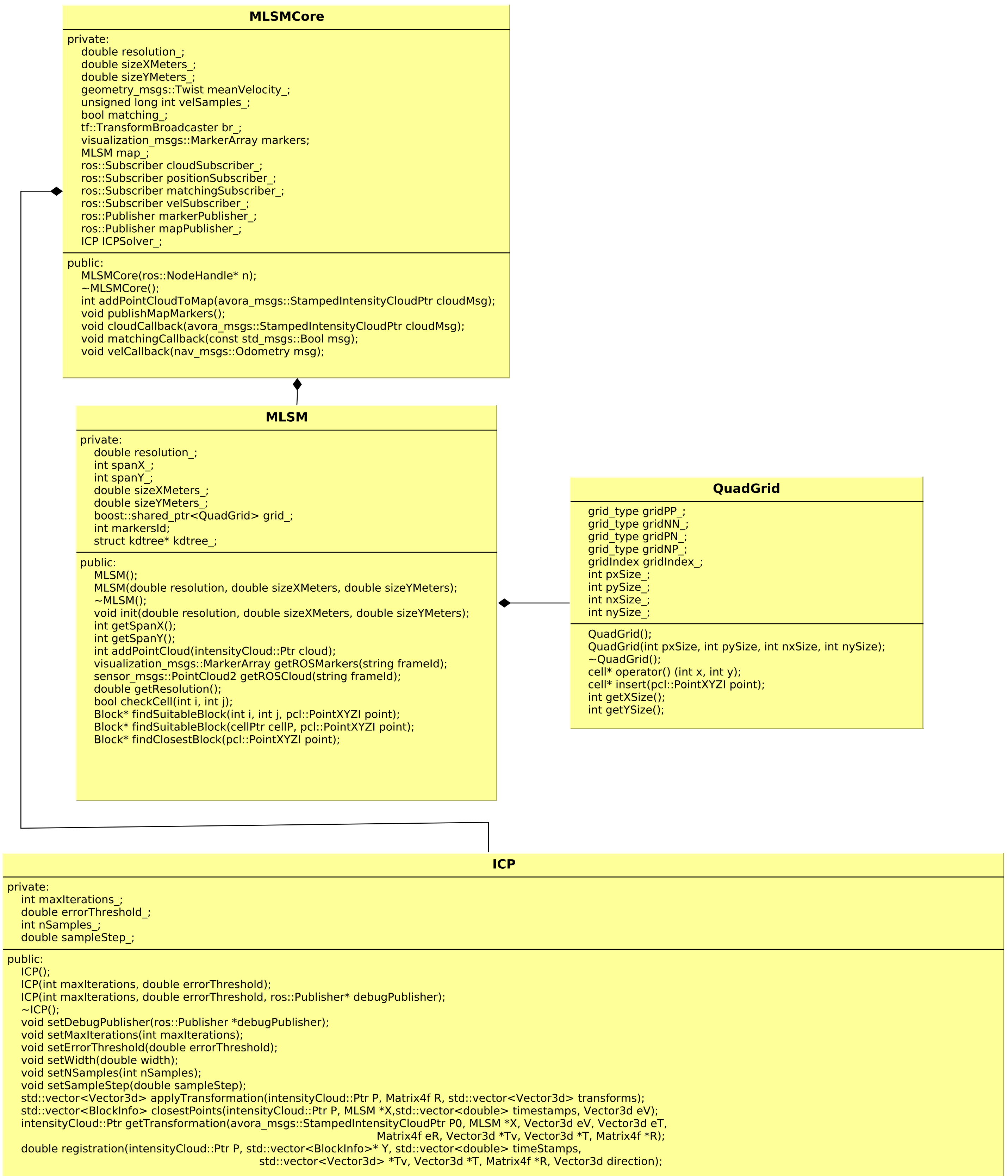
Figure 4.8: Laser based gazebo sonar

```
sonar_processing::SonarToCloud
---------------------------------------------
private:
  string mode_
  string targetFrame_
  int scanSize_
  ros::Subscriber beamSubscriber
  ros::Publisher laserCloudPublihser_
  ros::Publisher sonarCloudPublisher_
  ros::Subscriber velSubscriber_
  intensityCloud::Ptr laserCloud
  intensityCloud::Ptr soinarCloud_
  ros::NodeHandle nh_
  tf::TransformListener listener_
  tf::MessageFilter<avora_msgs::SonarScanLine>* tf_filter
---------------------------------------------
public:
  OnInit()
  SonarToCloud()
  ~SonarToCloud()
  laserInit()
  sonarInit()
  beamCallback(avora_msgs::SonarScanLineConstPtr scanLine)
  velCallback(geometry_msgs::Twist::Ptr msg)
```

```
sonar_processing::Thresholder
---------------------------------------------
private:
  string mode_
  int fixedThresholdValue_
  double maxThresholdProportion_
  double minThresholdProportion_
  double  OTSUMultiplier_
  int minThresholdValue_
  int maxBinValue_
  ros::Subscriber cloudSubscriber_
  ros::Publisher cloudPublisher_
  ros::Publisher rosCloudPublisher_
  ros::NodeHandle nh_
  avora_msgs::StampedIntensityCloud stampedCloudMsg_
---------------------------------------------
public:
  onInit()
  Thresholder()
  ~Thresholder()
  cloudCallback(avora_msgs::StampedIntensityCloudPtr cloudMessagePtr)
  pcl::IndicesConstPtr thresholdCloud(intensityCloud::Ptr cloudPtr)
  double getOTSUThreshold(intensityCloud::Ptr cloudPtr)
  publishCloud(intensityCloud::Ptr cloudPtr, std::vector<double> timestamps)
```

```
sonar_processing::OutlierRemover
-----------------------------------------------------------------
private:
  string mode_
  avora_msgs::StampedIntensityCloud stampedCloudMsg_
  int minNeighbouts_
  double minRadius_
  ros::Subscriber cloudSubscriber
  ros::Publisher cloudPublisher
  ros::Publisher rosCloudPublisher
  ros::NodeHandle nh_
-----------------------------------------------------------------
public:
  onInit()
  OutlierRemover()
  ~OutlierRemover()
  cloudCallback(avora_msgs::StampedIntensityCloudPtr cloudMessagePtr)
  removeOutliertsFromCloud(intensityCloud::Ptr cloudPtr)
  publishCloud(intensityCloud::Ptr cloudPtr, std::vector<double> timestamps)
```

```
sonar_processing::Wall2DDetector
-----------------------------------------------------------------
private:
  string mode_; // RANSAC
  int nWalls_;
  intensityCloud::iterator cloudIterator_;
  intensityCloud::Ptr sonarCloud_;
  ros::Subscriber cloudSubscriber_;
  ros::Publisher wallPublisher_;
  ros::Publisher wallCoefficientPublisher_;
  ros::Publisher markerPublisher_;
  avora_msgs::Wall wallMsg_;
  double RANSACDistance_;
  double RANSACProbability_;
  int RANSACMaxIterations_;
  ros::NodeHandle nh_;
-----------------------------------------------------------------
public:
  onInit();
  Wall2DDetector();
  ~Wall2DDetector();
  cloudCallback(avora_msgs::StampedIntensityCloudConstPtr cloudMessagePtr);
  detectWall(intensityCloud::ConstPtr cloud);
  publishWall();
```


MLSMCore
private:
double resolution_;
double sizeXMeters_;
double sizeYMeters_;
geometry_msgs::Twist meanVelocity_;
unsigned long int velSamples_;
bool matching_;
tf::TransformBroadcaster br_;
visualization_msgs::MarkerArray markers;
MLSM map_;
ros::Subscriber cloudSubscriber_;
ros::Subscriber positionSubscriber_;
ros::Subscriber matchingSubscriber_;
ros::Subscriber velSubscriber_;
ros::Publisher markerPublisher_;
ros::Publisher mapPublisher_;
ICP ICPSolver_;
public:
MLSMCore(ros::NodeHandle* n);
~MLSMCore();
int addPointCloudToMap(avora_msgs::StampedIntensityCloudPtr cloudMsg);
void publishMapMarkers();
void cloudCallback(avora_msgs::StampedIntensityCloudPtr cloudMsg);
void matchingCallback(const std_msgs::Bool msg);
void velCallback(nav_msgs::Odometry msg);
MLSM
private:
double resolution_;
int spanX_;
int spanY_;
double sizeXMeters_;
double sizeYMeters_;
boost::shared_ptr<QuadGrid> grid_;
int markersId;
struct kdtree* kdtree_;
public:
MLSM();
MLSM(double resolution, double sizeXMeters, double sizeYMeters);
~MLSM();
void init(double resolution, double sizeXMeters, double sizeYMeters);
int getSpanX();
int getSpanY();
int addPointCloud(intensityCloud::Ptr cloud);
visualization_msgs::MarkerArray getROSMarkers(string frameId);
sensor_msgs::PointCloud2 getROSCloud(string frameId);
double getResolution();
bool checkCell(int i, int j);
Block* findSuitableBlock(int i, int j, pcl::PointXYZI point);
Block* findSuitableBlock(cellPtr cellP, pcl::PointXYZI point);
Block* findClosestBlock(pcl::PointXYZI point);
QuadGrid
grid_type gridPP_;
grid_type gridNN_;
grid_type gridPN_;
grid_type gridNP_;
gridIndex gridIndex_;
int pxSize_;
int pySize_;
int nxSize_;
int nySize_;
QuadGrid();
QuadGrid(int pxSize, int pySize, int nxSize, int nySize);
~QuadGrid();
cell* operator() (int x, int y);
cell* insert(pcl::PointXYZI point);
int getXSize();
int getYSize();
ICP
private:
int maxIterations_;
double errorThreshold_;
int nSamples_;
double sampleStep_;
public:
ICP();
ICP(int maxIterations, double errorThreshold);
ICP(int maxIterations, double errorThreshold, ros::Publisher* debugPublisher);
~ICP();
void setDebugPublisher(ros::Publisher *debugPublisher);
void setMaxIterations(int maxIterations);
void setErrorThreshold(double errorThreshold);
void setWidth(double width);
void setNSamples(int nSamples);
void setSampleStep(double sampleStep);
std::vector<Vector3d> applyTransformation(intensityCloud::Ptr P, Matrix4f R, std::vector<Vector3d> transforms);
std::vector<BlockInfo> closestPoints(intensityCloud::Ptr P, MLSM *X,std::vector<double> timestamps, Vector3d eV);
intensityCloud::Ptr getTransformation(avora_msgs::StampedIntensityCloudPtr P0, MLSM *X, Vector3d eV, Vector3d eT,
Matrix4f eR, Vector3d *Tv, Vector3d *T, Matrix4f *R);
double registration(intensityCloud::Ptr P, std::vector<BlockInfo>* Y, std::vector<double> timeStamps,
std::vector<Vector3d> *Tv, Vector3d *T, Matrix4f *R, Vector3d direction);

5 | Economical and Legal Aspects

5.1 License

There are several licesing factors that intervene on this project. There are several licenses that must be uphold:

- *Third party software*: In this case we can go through the licenses of the third party software used:
  - *ROS*: As stated in `http://www.ros.org/is-ros-for-me/`,

    > The core of ROS is licensed under the standard three-clause BSD license. This is a very permissive open license that allows for reuse in commercial and closed source products. [...] While the core parts of ROS are licensed under the BSD license, other licenses are commonly used in the community packages, such as the Apache 2.0 license, the GPL license, the MIT license, and even proprietary licenses. Each package in the ROS ecosystem is required to specify a license,[..]

  - *Gazebo*: As stated in their webpage,

    > Gazebo is open-source licensed under Apache 2.0

    which can be found at `http://www.apache.org/licenses/LICENSE-2.0`.
  - *KdTree library* The kdTreelibrary used specifies that

    > kdtree is free software. You may use, modify, and redistribute it under the terms of the 3-clause BSD license.

  - *Laser based range sensor* In the case of the software that was used as base for the imaging sonar simulator using laser scans, the authors specify their of license which is the following:

    > // Copyright (c) 2012, Johannes Meyer, TU Darmstadt // All rights reserved.
    > // Redistribution and use in source and binary forms, with or without // modification, are permitted provided that the following conditions are met: // * Redistributions of source code must retain the above copyright // notice, this list of conditions and the following disclaimer. // * Redistributions in binary form must reproduce the above copyright // notice, this list of conditions and the following

disclaimer in the // documentation and/or other materials provided with the distribution. // * Neither the name of the Flight Systems and Automatic Control group, // TU Darmstadt, nor the names of its contributors may be used to // endorse or promote products derived from this software without // specific prior written permission.
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND // ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED // WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE // DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY // DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES // (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; // LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND // ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT // (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS // SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- *AVORA team software* The first avora team stated that the software developed by the team was licensed under a BSD Licensed. The license is the following:

  * Copyright (c) 2012, The Avora I Project, All rights reserved. * * The Avora I Project is composed by the following members: Anil Motilal * Mahtani Mirchandani, AarÃşn MartÃŋnez Romero, Luis SÃąnchez Crespo, Daniel * GarcÃŋa Pereira, David Morales Ventura, Federico Maniscalco MartÃŋn, * Enrique FernÃąndez Perdomo * * Redistribution and use in source and binary forms, with or without * modification, are permitted provided that the following conditions are met: * 1. Redistributions of source code must retain the above copyright notice, * this list of conditions and the following disclaimer. * 2. Redistributions in binary form must reproduce the above copyright notice, * this list of conditions and the following disclaimer in the documentation * and/or other materials provided with the distribution. * 3. All advertising materials mentioning features or use of this software * must display the following acknowledgement: * This product includes software developed by The Avora I Project. * 4. Neither the name of The Avora I Project nor the names of its contributors * may be used to endorse or promote products derived from this software * without specific prior written permission. * * THIS SOFTWARE IS PROVIDED BY THE AVORA I TEAM "AS IS" AND ANY EXPRESS OR * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF * MERCHANTABILITY AND

FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO * EVENT SHALL THE AVORA I TEAM BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The rest of the software is licensed under the same BSD license with changes only regarding authors.

- *Software developed as part of this project*: The software developed for this project, which can be found at `https://github.com/Aridane/underwater_map_construction/` contains its own LICENSE file which is a three clause BSD license.
- *Documentation* Finally this document, is licensed under a Creative commons Attribution-NonCommercial-NoDerivs License (CC BY-NC-ND).

5.2 Expenses

The equipment used for this project did not involve the acquisition of new elements. In case of using a simulation environment it does not involve the purchase of any devices nor licenses. For real world testing some platform should be used, so we can actually go through an estimate of what the platform built by the AVORA team costs:

Component	**Unit cost**	**Amount**	**Total cost**
Sony CCD 700TVL	25 €	1	25 €
Servo Futaba s3001	11.00 €	1	11.00 €
Arduino Mega 2560	12.00 €	1	12.00 €
Sonar Imagenex 852	3955.00 €	1	3955.00 €
DUB-H7 7 port USB 2.0 Hub	25.58 €	1	25.58 €
Pressure sensor U5100 series transducer	150.38 €	1	150.38 €
H-38120S Headway LiFePO4	14.86 €	24	356.64 €
H-38140S Headway LiFePO4	19.62	8	156.96 €
DC-DC USB Intelligent buck-boost DC-DC	70.51 €	3	211.53 €
Seaconn Connectors (Various)	70 € - 50 €	11	650 €
10A 5-25V Dual Channel DC Motor Driver	50.54 €	2	101.8 €
SG90 Servo	3.50	2	7 €
Seabotix BTD150	422.63 €	4	1690.52 €
Odroid Xu4 + eMMc 32Gb	130 €	1	130 €
Total Cost			7483,41 €

It is important to note that the cost of testing at sea is not included due to the fact that is minimal, considering that it was by deploying the vehicle by hand from a dock ramp.

6 | Development

6.1 Introduction

We have defined our objectives and analysed the problems we might face. In this chapter the approach used to fulfil our objectives and deal with the existing problems will be addressed. Separate modules performing different tasks can be identified. Among these the following, which does not mean they have to belong to different packages or nodes, we only highlight the stages that are important and can be differentiated from the rest:

- Simulation
- Transform sensor output to cloud data
- Clean cloud data
- Detect shapes
- Build map
- Match clouds to map

6.2 Multi Level Surface Maps

Since the structure chosen for this project is the MLSM we will go a bit deeper on its structure and characteristics.

6.2.1 Structure and manipulation

The MLSM is basically a grid with cells at each position of the grid c_{ij}. Each cell contains a list of blocks(b^k_{ij}) that correspond to measurements made at different heights for the cell position i and j. Following the work in [11] the blocks are identified by a tuple with the following elements:

- *Mean* μ^k_{ij}: Corresponding to the mean of all the measurements gathered in that block.
- *Variance* σ^k_{ij}: Corresponding to the variance of the measurements gathered.
- *Height* h: This is the value of the highest measurement belonging to the block.

- *Depth* d: The depth of a block is the difference between the highest and lowest measurement contained.
- *Type* π: The type allows us to characterise the distribution of the measurements in a block. In [11] the blocks are divided in horizontal and vertical. Since this work intended to be compatible with plane detection, these two block types are preserved. Also a "free" type has been introduced to characterise the detected free. This will allow us to discern between unknown/unmapped areas and free areas.
- *Intensity*: The blocks also store the mean intensity of the measurement, not exactly as a separate field, but as part of the mean and variance, thus having a mean intensity and intensity variance. This data is also important since they can be a factor on determining the certainty of the existence of an obstacle in that block, or, in future work, it could help characterise the object it belongs to, or the type of terrain that produced those echoes.

Every time a measurement $p = \{p_x, p_y, p_z, p_i\}$ is collected, the cell c_{ij} which satisfies that $p_x \geq j \cdot cell_size$ and $p_x < j \cdot (cell_size + 1)$ and $p_i \geq i \cdot cell_size$ and $p_y < i \cdot (cell_size + 1)$ is iterated to obtain a list of candidate blocks to hold the measurement. These candidate blocks b_{ij}^k must satisfy that $|p_z - height_{ij}^k| < cell_size$ and $|height_{ij}^k - depth_{ij}^k - p_z| < cell_size$. Then we have the following situation:

- *No candidate*: This case is the simplest one, we create a new block with mean $\mu_{ij}^k = p, \sigma_{ij}^k = 0, d = 0, h = p_z, \pi = Horizontal$.
- *One candidate*: We update the parameters of that block.
- *Several candidates*: In this case the blocks are fused into one vertical block, combining the means and variances of all candidate blocks.

6.3 Iterative Closest Points (ICP)

The ICP algorithm is, as its names points out, an iterative algorithm for point cloud matching. The original algorithm is described in [5]. It aims to find the transformation between a point cloud and some reference surface (or another point cloud), by minimizing the square errors between the correspondences established. The iterative part of ICP comes from the fact that the correspondences are recalculated on each iteration as the algorithm converges to a local minimum on the error function. ICP is a gradient descendent method, and being such it depends on a relatively good starting point in advance, otherwise it can be easily trapped in a local minimum and stay there providing a worthless solution. In the field of mobile robots, ICP has been extensively used to match 2D laser scans. This is due to the fact that laser sensors can provide a full scan of an area at high speed and accuracy. This way we will have successive point clouds with small errors that can be matched for map building and also providing localization since we know the transformation between each scan.

The ICP algorithm was originally stated as follows:

- The point set P with N_p points $\{\vec{p}_i\}$ from the data shape and the model shape X (with N_x supporting geometric primitives: points, lines, or triangles) are given.
- The iteration is initialized by setting $P_0 = P, \vec{q}_0 = [1, 0, 0, 0, 0, 0, 0]^t$ and $k = 0$. The registration vectors are defined relative to the initial data set P_0 so that the final registration represents the complete transformation. Steps 1, 2, 3, and 4 are applied until convergence within a tolerance τ. The computational cost of each operation is given in brackets.
  a Compute those closest points: $Y_k = C(P_k, X)$(cost $O(N_p N_x)$ worst case, $O(N_p log N_x)$ average).
  b Compute registration: $(\vec{q}_k, d_k) = Q(P_0, Y_k)$ (cost: $O(N_p)$).
  c Apply the registration:
  d Terminate the iteration when the change in mean-square error falls below a threshold $\tau > 0$ specifying the desired precision of the registration: $d_k - d_{h+1} < \tau$.

6.3.1 ICP Implementation

The ICP algorithm has been implemented in the ICP class under the mlsm_manager package. The ICP solver is used every time the mapper node receives a point cloud and the matching topic is publishing *true*.

The in-depth analysis of the implementation and method followed can be found in the *mlsm_manager package*.

6.4 Gazebo simulation

Gazebo provides an easy way to work from already implemented sensors, i.e. Plugins. This is a appropriate if we want to take an existing type of sensor and modify it to suit our purposes. The issue arises when what we want to do is directly make gazebo provide the raw data our sensor would produce, and such sensor is not among the already provided by Gazebo. These two approaches have been followed in order to provide a simulated imaging sonar and are addressed next.

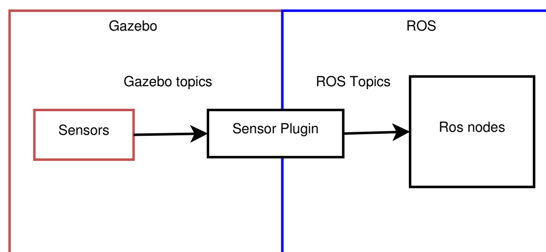


Figure 6.1: Gazebo-ROS Sensor interaction

6.4.1 Collision based simulation

The first implementation went through adding a new sensor to Gazebo, not just adding a pluging to transform incoming data. The easiest way to do so was to install gazebo form source and add the required modifications to it. These modifications can be summarized as follows:

- Adding the sensor message
- Adding the sensor to gazebo infrastructure
- Implementing the sensor
- Implement Gazebo sensor plugin
- Implement Gazebo-ROS sensor plugin

Published data

The sensor produces a Gazebo message in a topic the imaging sonar ros plugin subscribes to. The plugin basically makes a conversion from the Gazebo message to our *avora_msgs::SonarScanLine* so there is not much to say about the plugin. The sensor message created was similar to the one defined for ROS since it was supposed to contain the same data as a sonar beam.

```
package gazebo.msgs;

/// \ingroup gazebo_msgs
/// \interface Sonar
/// \brief Message for a sonar value

import "pose.proto";
import "vector3d.proto";

message SonarScanLine
{
required string frame                = 1;
required gazebo.msgs.Pose world_pose  = 2;
required double range_min                   = 3;
required double range_max                = 4;
required double radius                   = 5;
required double range                    = 6;
required double angle                    = 7;

repeated int32  intensities                = 8;
}
```

6.4.2 XML parameters

To the already defined XML parameters of the Sonar sensor we added some extra ones by modifying the source code of SDF 1.5, since we have a source installation of both Gazebo and SDF. These parameters are set as children of the *<sonar>* element:

- *step*: Defines the step the sonar head takes each time it rotates.
- *opening*: This is the aperture width of the cone. Instead of using the radius of the cone at maximum distance, it is easier to have a parameter like this, since the imaging sonar specification will directly give us this parameter.
- *binCount*: Number of bins existing for each beam. This ir also a parameter in the imagenex 852.

The Sensor

The sensor developed was based on the *SonarSensor* provided by Gazebo. This sensors calculates the closest collision between an established shape (a cone) and the environment and publishes the data according to that collision (3D point). Our sensor is a bit different, first, we do not consider just a collision, we need all the collisions along the length of the cone so we can fill the array of slant ranges with their intensities from the collisions that are feasible to come from an echo return. And second, we need the sensor to internally rotate the sensor so it produces a reading at each angular position, so we can simulate the head rotation of the original imaging sonar.

The sensor basically sets up a collision filter and then updates and publishes its data when a update event occurs. For the collision filter we need to first create the shape we will use for collision search and then feed it to the collision element. Here we set up the shape used for collisions using the parameters specified in the sdf file of the robot model.

```
this->sonarShape = boost::dynamic_pointer_cast<physics::MeshShape>(
this->sonarCollision->GetShape());
// Use a scaled cone mesh for the sonar collision shape.
this->sonarShape->SetMesh("unit_cone");
this->sonarShape->SetScale(math::Vector3(this->radius*2.0,
this->radius*2.0, range));
// Position the collision shape properly. Without this, the shape will be
// centered at the start of the sonar.
math::Vector3 offset(0, 0, range * 0.5 + 0.2);
offset = this->pose.rot.RotateVector(offset);
this->sonarMidPose.Set(this->pose.pos - offset, this->pose.rot);
```

Then we set up the collision element from the shape defined before:

```
this->sonarCollision->SetRelativePose(this->sonarMidPose);
this->sonarCollision->SetInitialRelativePose(this->sonarMidPose);
// Don't create contacts when objects collide with the sonar shape
this->sonarCollision->GetSurface()->collideWithoutContact = true;
this->sonarCollision->GetSurface()->collideWithoutContactBitmask = 1;
this->sonarCollision->SetCollideBits(~GZ_SENSOR_COLLIDE);
this->sonarCollision->SetCategoryBits(GZ_SENSOR_COLLIDE);

// Create a contact topic for the collision shape and filter using our collision
std::string topic =
this->world->GetPhysicsEngine()->GetContactManager()->CreateFilter(
        this->sonarCollision->GetScopedName(),
        this->sonarCollision->GetScopedName());
```

Finally we set up a callback for every time the collisions are updated. Even though the collisions are updated constantly, the sensor only publishes the results at the specified rate in the SDF file.

```
// Subscribe to the contact topic
this->contactSub = this->node->Subscribe(topic,
        &ImagingSonarSensor::OnContacts, this);
```

The *ImagingSonarSensor::OnContacts* function will update a vector containing the current collisions so that we can access them when the update event triggers, which is where the processing is made. In this function we use the array of collision elements to keep a count of how many collisions take place at each slant range. Then we assign an intensity depending on how many collisions occurred at each range. In order to make the readings a it more realistic we add more contacts around each contact using a Gaussian distribution.

```
// Iterate over all the contact messages
for (ContactMsgs_L::iterator iter = this->incomingContacts.begin();
iter != this->incomingContacts.end(); ++iter)
{
  // Iterate over all the contacts in the message
  for (int i = 0; i < (*iter)->contact_size(); ++i)
  {
    for (int j = 0; j < (*iter)->contact(i).position_size(); ++j)
    {
      pos = msgs::Convert((*iter)->contact(i).position(j));
      normal = msgs::Convert((*iter)->contact(i).normal(j));

          math::Vector3 relPos = pos - referencePose.pos;
          double len = pos.Distance(referencePose.pos);
          /*
             Here we calculate the corresponding index on the slant range array
             from the distance from the sensor to the collision. We don't really
```

```
                 need its 3D position
               */
               index = ceil((len * (this->binCount/this->rangeMax))) - 1;
               if (index > (this->binCount-1)) continue;
               scan->set_intensities(index,scan->intensities().Get(index) +1 );
               if (scan->intensities().Get(index) > maxContactCount){
                 maxContactCount = scan->intensities().Get(index);
               }
               /*
                 Then we add 10 more collisions around each collision using a
                     using a gaussian distribution with a standard deviation of 0.6
                     The idea was for both the number of poitns and the standard
                     deviation were configurable with a parameter in the sdf file
               */
                 double auxLen;
               for (int k=0;k<10;k++){
                 auxLen = gazebo::math::Rand::GetDblNormal (len, 0.6);
                 index = ceil((auxLen * (this->binCount/this->rangeMax))) - 1;
                 if (index > (this->binCount-1)) continue;

                 scan->set_intensities(index,scan->intensities().Get(index) +1 );

                 if (scan->intensities().Get(index) > maxContactCount){
                   maxContactCount = scan->intensities().Get(index);
                 }
           }
         }
       }
     }
```

After that we just use the maximum collision count as the maximum strength point for computing the echo streng at each slant range.

```
for (int i = 0; i < this->binCount; ++i){
        scan->set_intensities(i,(127*scan->intensities().Get(i)) / maxContactCount);
}
```

Then, the only thing left to do is rotate the sonar head. The logical step here would seem to rotate the collision shape around the sensor origin, but, since the access to that rotation and the effect it has on the collision filter is a bit obscure, the easiest thing to do is rotate the sensor frame *SonarDato* using a continuous joint which is not accessible form the charle_control package. The main problem here is that the joint is hardcoded so in order to make it usable in any other model this is set as a parameter.

```
// We update our angular position
this->angle = fmod((this->angle + (this->step * M_PI/180.0)),2*M_PI);
// And then set the positon of our continuous rotation joint
```

```
physics::ModelPtr model = this->world->GetModel("robot1");
physics::JointPtr joint = model->GetJoint("sonarDato_to_sonar");
joint->SetPosition(0,this->angle);
joint->Update();
```

Even though the results are quite similar to what a clean sonar image would look like, the proper thing to do would be to use the collision element to obtain the normal of the collision surface (which would require to operate with the shape and orientation of the whole element it collides with) and also use the material defined for the element from the map. These two considerations would be a great addition to the simulation, but there are several issues. First, it would require more processing on a piece of code that should run as fast as it cans and it should not be too computationally heavy. Second, we would have to define each element of the map as a separate element with its own material, which would mean that we can not just take a 3D model from another 3D modelling tool, since they provide the whole map as one element. Still, it is possible to do it and it is a great idea for future work.

Sonar cone bug

When the sensor simulation was complete, some strange elements appeared on the data that were being published. Basically some points that yielded no collision and collisions detected where there was nothing at all. This issues can be observed in figure 6.2.

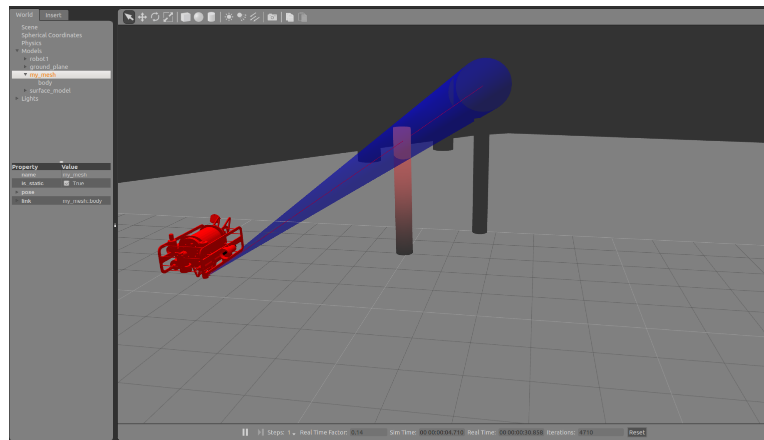


Figure 6.2: Bug where collision is ignored

This issue makes the collision based sonar unusable for the moment, but, since I believe is the best way of implementing it, it should be tested with other gazebo versions when support for ROS Jade is available.

6.4.3 Laser cone based simulation

Team Hector [4] from the *Technic University of Darmstadt* has developed some Gazebo-ROS plugins for their robot simulation. These packages can be found at the *hector_gazebo_plugins* [2] package wiki and provide several modified tools for simulation. Such as GPS, IMU, Range sensor (laser based), etc. The imaging sonar sensor explained here is a modification of their *GazeboRosSonar*. The parameters in this case are exactly the same as the ones for the regular laser sensor. The only difference is data are treated, since we have a number of rays that form a pyramid that we could treat as an approximation to a cone and then produce our intensity arrays from that data.

In this case we have not added a new sensor to Gazebo, but instead work through a plugin that takes the laser pyramid data and turns it into beams. In this case we also have the *step* and *binCount* parameters. The good thing about this plugin is that it is much easier to modify since we rely on an already configured sensor. The iteration through readings is pretty much the same:

```
for(int i = 0; i < num_ranges; ++i) {
  double ray = sensor_->GetLaserShape()->GetRange(i);
  int index = ceil((ray * ((double)binCount_/(double)sensor_->GetRangeMax()))) - 1;
  if (ray == sensor_->GetRangeMax()) continue;
  if (index < 0) index = 0;
  contactCount[index] += 1;
  if (contactCount[index] > maxContactCount){
    maxContactCount = contactCount[index];
  }
}
```

Here we do not generate the Gaussian points when the collisions are received, instead we accumulate each contact in its according intensity array index so we can generate the Gaussian points around each contact. In order to generate the sparse data, we use the contact count accumulated on the array to generate points around te collisions. The more collisions we have, the more sparse points we generate around it.

```
for (int i=0;i<binCount_;i++){
  if (contactCount[i] != 0){
    for (int k=0;k<contactCount[i];k++){
      double ray = i * sensor_->GetRangeMax() / binCount_;
          double auxRay= gazebo::math::Rand::GetDblNormal(ray, 1);
          index = ceil((auxRay * (binCount_/sensor_->GetRangeMax()))) - 1;
          if (index < 0) index = 0;
          if (index > (binCount_-1)) continue;
          intensities[index] += 15;
          if (intensities[index] > 127) intensities[index] = 127;
        }
  }
}
```

After that we also generate some noise in each beam to be a little more realistic. Here we generate 15 points using a distribution with an standard deviation of 4 meters and then generate the point with an intensity that follows another Gaussian distribution.

```
for (int k=0;k<15;k++){
  double auxRay = gazebo::math::Rand::GetDblNormal (range_.range, 4);
  index = ceil((auxRay * (binCount_/sensor_->GetRangeMax()))) - 1;
  if (index < 0) index = 0;
  if (index > (binCount_-1)) continue;
    scanLine_.intensities[index] = gazebo::math::Rand::GetDblNormal (53, 20);
  }
```

6.5 ROS

6.5.1 Nodes and nodelets

As we have mentioned before, the scan data acquired from the sonar (simulated or real) undergoes a process that takes it through different processing stages, that will end up producing point cloud data useful for the mapping node (or any other node that listens to the corresponding topic). This pipeline, consist of some simple and functional nodes that can be linked to the stages of processing defined previously, these being:

- Sonar to cloud conversion
- Cloud thresholding
- Outlier removal
- (Optional) Basic shape detection

6.5.2 Sonar to cloud conversion

This node is the "SonarToCloud" nodelet, we have mentioned before that this node is in charge of taking the sonar scans and turning them into point clouds. There is a bit more than "just" that in the process of converting the point clouds. This process can be summed up in a few steps:

- Node initialization.
- Beam acquisition, transformation, and storage.
- Scan publish.

Node initialization

This node has several configurable parameters that we have conveniently stored into a yaml file. Among the most important parameters we can find the following:

- *Mode*: Allows us to select how the result cloud should be built. There are two modes *SONAR* and *LASER*. And we can also add *CONT* to the mode. *SONAR* mode will produce a point cloud where each group of points corresponds to a slant range and the intensity of that slant range. Meanwhile, the *LASER* mode will produce *laser like* data. Instead of considering all the points, the result cloud will only contain one point for each received beam, so that the result can be interpreted the same way a laser scan would be, where we have one point representing the distance for each laser line. The additional *CONT* part lets us decide whether the result should be published each time a beam is received (letting as process the whole scan every time there is new data), or if the scan cloud should only be published once for each complete scan.

- *Scan size*: Defines how many measurements should be gathered before publishing a result. By default this is set to 120, which is the number of beans we can find in a scan with 3°step. If we wanted to process each beam separately, we could set the scan size to 1.

- *Velocity topic*: This topic helps us know which beams of the scan were measured while moving. This will help with the processing when the data arrives to the mapping node.

- *Target frame*: Indicates to which tf frame the data should be converted. In general this should be set to "odom" or "map" since these are the frames that contain the latest transforms to world coordinates.

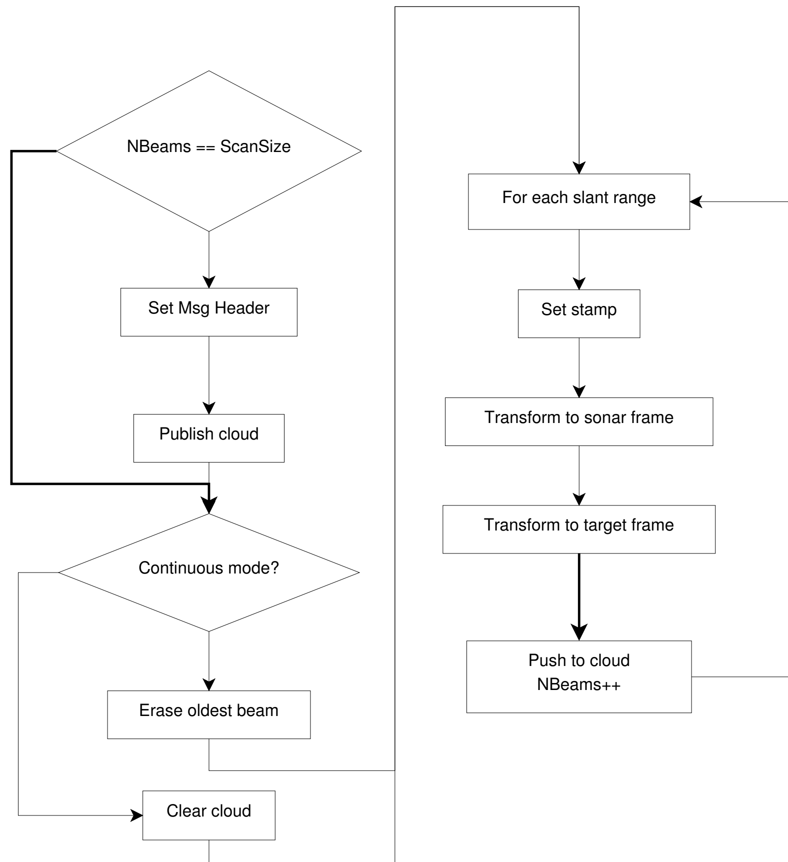
Another important step of the initialization is the creation of the tf filter as show in the following snippet:

```
tf_filter_ = new tf::MessageFilter<avora_msgs::SonarScanLine>(scanLine_sub_,
                                listener_, targetFrame_, 120);
tf_filter_->registerCallback( boost::bind(&SonarToCloud::beamCallback, this, _1) );
```

This will synchronize the beam callbacks with the tf broadcasting, this way we can assure that the available transformation to the target frame is the one corresponding to when the measurement was made, otherwise, when we change the coordinate system, we could be transforming into the future or into the past, something we do not desire.

Cloud transformation

Here is where the processing takes place.Depending on the mode configuration, some different things will happen, but in general, for both *SONAR* and *LASER* modes, the steps taken are pretty much the same and can be synthesized in the following flow chart.



6.5.3 Thresholding

The thresholding nodelet is the first one to start transforming our cloud. The operation this node performs is a simple thresholding based on intensity. The main idea is to perform a thresholding operation similar to the local maximum search performed in [14]. But, our approach does not rely on local maximum intensities but in a global intensity threshold applied to the cloud. In order to provide certain level of customization and also being able to test which options work better there are several modes we can select through the parameters. Before talking about this mode lets take a look at the most important parameters we have:

- *Mode*: The thresholding mode. This parameters can take one of two values plus an optional extra parameter, similarly to the *SonarToCloud* nodelet. The possible values are *OTSU* for OTSU based thresholding and *FIXED* for a fixed threshold value that is set in another parameter. The optional part is *PROPORTIONAL* This indicates that the value calculated with the *OTSU* method should be multiplied by a factor specified in another parameter. There is also the choice to set only the proportional parameter, which will use the maximum and minimum intensities to calculate the threshold using the *Max Threshold Proportion* and *Min Threshold proportion* with the following formula: $\theta = maxProportion \cdot maxIntensity + minProportion \cdot minIntensity$
- *OTSU Multiplier*: This is the multiplier applied to the threshold calculated with the *OTSU* algorithm.
- *Min Threshold*: This parameters sets a bottom value for the thresholding operation, since sometimes we might find that *OTSU* gives a value that is too low, or that there are not enough strong intensities to calculate a proper value using other modes.

OTSU thresholding

Fixed and proportional thresholding modes are simple enough, and there is not much to say about them. So we will focus on the *OTSU* algorithm which is much more interesting.

Otsu's method described in [12] is an algorithm used to automatically perform clustering-based image thresholding, and produce a binary image. This algorithm assumes that there are two classes of pixels following a bi-modal histogram (foreground and background pixels), it then calculates the optimum threshold that separates the two classes so that their combined spread is minimal so that their intra-class variance is minimal.

In this method we search for the threshold value that minimizes the intra-class variance, which is defined as the sum of variances of the two classes:

$$\sigma_\omega^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t) \tag{6.1}$$

Weights ω_i are the probabilities of the two classes separated by a threshold t and σ_i^2 are the variances of these classes. Expressing it in terms of the probabilities ω_i and each class mean μ_i we have the equation:

$$\sigma_b^2(t) = \sigma^2 - \sigma_\omega^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2 \tag{6.2}$$

Which shows that minimizing the intra-class variance is the same as maximizing the inter-class variance. The class probability $\omega_1(t)$ is computed from the calculated histogram as t:

$$\omega_1(t) = \sum_0^t p(i) \tag{6.3}$$

while the clas mean $\mu_1(t)$ is:

$$\mu_1(t) = [\sum_0^t p(i)x(i)]/\omega_1 \tag{6.4}$$

where $x(i)$ is the value at the centre of the ith histogram bin. $\omega_2(t)$ and μ_2 can be computed similarly on the right-hand side of the histogram for bins greater than t.

The following algorithm can be extracted for the stated method:

1. Compute the histogram and probabilities of each intensity level
2. Set up initial $\omega_i(0)$ and $\mu_i(0)$
3. Step through all possible thresholds $t = 1 : MaxIntensity$
   (a) Update ω_i and μ_i
   (b) Compute $\sigma_b^2(t)$
4. The desired threshold corresponds to the maximum $\sigma_b^2(t)$
5. Two maxima, and two corresponding thresholds, can be calculated. $\sigma_{b1}^2(t)$ is the greater maximum and $\sigma_{b2}^2(t)$ is the grater or equal maximum
6. Our desired threshold is $threshold = \frac{threshold_1 + threshold_2}{2}$

In general this algorithm yields good results if we have a population of noisy elements, since it will help remove elements that have an insignificant intensity compared to what the algorithm classifies as foreground. In case we had a big population of zero elements that are fed into the algorithm we could calculate the threshold feeding only the non zero intensity values, this way, we can get rid from the effect of having too many zero intensity readings.

6.5.4 Outlier removal

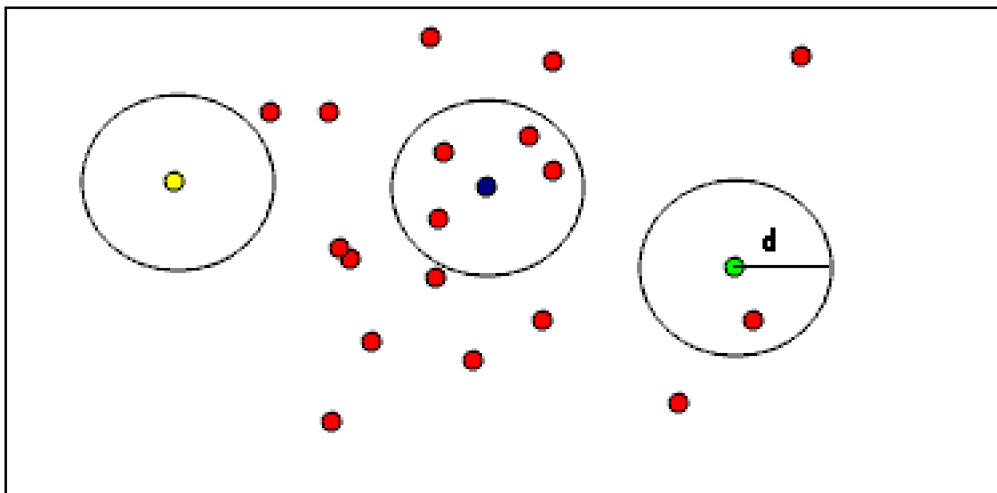
This node has the same mechanics as the thresholder node, we take an input cloud, transform it and publish it. Its goal is to get rid of isolated noisy points, i.e. outliers. This is done by means of the *pcl::StatisticalOutlierRemoval<pcl::PointXYZI>* and *pcl::RadiusOutlierRemoval<pcl::PointXYZI>* classes. Before going into how they works lets take a look at the parameters of this node:

- *Mode*: This parameter lets us decide which kind of outlier removal should be used. The values it can take are *STATISTICAL* and *RADIUS* which are self explanatory.
- *Min radius*: Minimal radius for the radius outlier remover.
- *Min neighbours*: This parameter specifies the minimum number of neighbours a point must have not to be erased in the removal process

Radius outlier removal

As its name suggest, the *radius outlier remover* gets rid of outliers based on a radius. It basically searches for a minimum number of neighbours in a specified radius. Even though it is a simple approach when correctly tuned it can yield quite good results, since our valid points should appear in more or less populated clusters. The usage is quite simple as the following code snippet shows:

```
//Set source points
radius_outlier_removal.setInputCloud(cloudPtr);
// Set radius for neighbor search
radius_outlier_removal.setRadiusSearch(minRadius_);
// Set threshold for minimum required neighbors neighbors
radius_outlier_removal.setMinNeighborsInRadius(minNeighbours_);
// Do the filtering
radius_outlier_removal.filter(*cloudPtr);
```



Statistical outlier removal

This other outlier remover follows a more elaborate process than the *radius outlier remover*. This outlier remover performs a statistical analysis on each point's neighbourhood, and trims the points that do not meet certain criteria extracted from the previous analysis. The PCL statistical outlier remover bases its filtering on the computation of the distribution of point to neighbours distances in the input dataset. For each point, it computes the mean distance from it to all its neighbours. By assuming that the resulted distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered outliers and trimmed from the dataset. Like the previous outlier remover, this class does not need any sort of difficult configuration:

```
// Set source points
sor.setInputCloud (cloudPtr);
// Set the number of nearest neighbors to use for mean distance estimation.
sor.setMeanK (20);
/* Set the standard deviation multiplier for the distance threshold
   calculation.
   The distance threshold will be equal to: mean + stddev_mult * stddev.
   Points will be classified as inlier or outlier if their average neighbour
   distance is below or above this threshold respectively.
*/
sor.setStddevMulThresh (1);
sor.setKeepOrganized(false);
sor.filter (*cloudPtr);
```

6.5.5 Line and circle detection

This nodes aim to perform shape recognition on the clouds resulting form the cleaning process that takes place in previously addressed nodes. This nodes detect shapes in the input clouds using RANSAC. For the case of each detection nodelet the parameters used are the topics for subscription and the ones for advertisement. In the case of the line detector we have the parameter *nWalls* which indicates how many lines (walls in the case of the 2D horizontal scan) we would like to find.

RANSAC

The RANdom SAmple Consensus (RANSAC) is a general parameter estimation algorithm for sets containing outliers. Assuming that the data set is comprised by outliers and inliers, this algorithm uses an iterative approach to match subsets with a particular model. The subsets, and the final solution, are built upon the smallest possible set and then are expanded with sets of consistent data points by iteratively selecting a random subset from the original data and then checking if they are inliers or ourliers as follows:

1. The model is fitted to the hypothetical inliers.

2. All other data are tested against the fitted model and, if a points fits well to the estimated model, also considered as a hypothetical inlier.

3. The estimated model is found reasonably good if enough points are classified as hypothetical inliers.

4. The model is estimated again using all the hypothetical inliers, since it was estimated using only the initial set of inliers.

5. Finally, the model is evaluated by estimating the error of the inliers relative to the model.

The procedure explained above is repeated for a number of specified iterations, each one providing a rejected model which contains too few inliers or a refined model with a corresponding error measure. After each iteration we will keep the model if the error is lower than the last saved model. The main advantage we have by using RANSAC is that it can estimate the parameters with high accuracy even when the dataset contains a significant number of outliers. On the other hand, there is no upper bound on the time the algorithm will take to compute these parameters. By specifying the number of iterations we can bound the execution time, but the solution might not be the best one. Because of this, we have to keep in mind that by having more iterations we increase the probability of producing a reasonable model, but will also increase the execution time. RANSAC only estimates one model for a particular dataset, something we have to deal with when trying to fit a model to a dataset where two or more models exists, since RANSAC may fail to find either one.

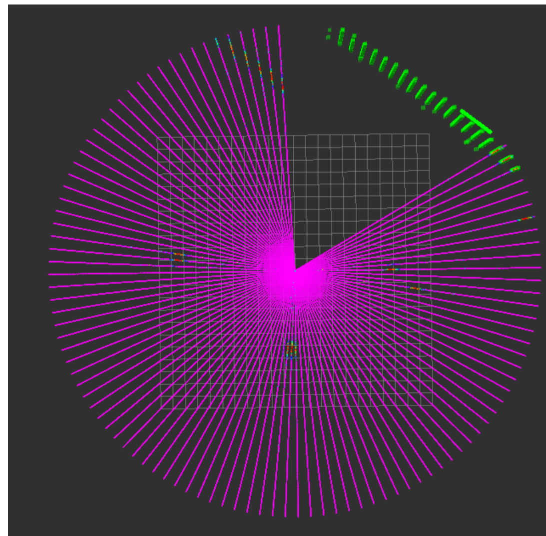


Figure 6.3: RANSAC line detection

Detection

The operation of this nodelet is very similar to the ones already explained. For the detection we try to find one set of points that are classified as inliers for the model we have set up and then proceed to eliminate these points from the input cloud and then try to detect again, until we have detected as many elements as the parameter indicated or until we are unable to find anything that matches our model. In the case of RANSAC, we set up the model, compute the best inliers and extract them from the point cloud so we can run it again for subsequent elements.

```
// Input model for RANSAC
pcl::SampleConsensusModelLine<pcl::PointXYZI>::Ptr
        model_l(new pcl::SampleConsensusModelLine<pcl::PointXYZI>(cloud));
pcl::RandomSampleConsensus<pcl::PointXYZI> ransac(model_l);
```

```
// RANSAC parameters
ransac.setDistanceThreshold(RANSACDistance_);
ransac.setMaxIterations(RANSACMaxIterations_);
ransac.setProbability(RANSACProbability_);
// Computation of model and extraction of result
ransac.computeModel();
ransac.getInliers(inliers);
ransac.getModelCoefficients(coefficients);
// Copies all inliers of the model computed to another PointCloud
pcl::copyPointCloud(*cloud,inliers,line);
pcl::ExtractIndices<pcl::PointXYZI> extractor(true);
extractor.setInputCloud (cloud);
extractor.setIndices (boost::make_shared<vector<int> >(inliers));
extractor.setNegative (true);
extractor.filter(*cloud);
inliers.clear();
[...]
result += line;
line.clear();
```

The issue with RANSAC is that it can only estimate one model for a particular data set, that explains why we remove the data and perform successive detection attempts. As said before, RANSAC may fail to find either one. Still, since the data resulting from the cleaning process does not contain many outliers, this algorithm yields good results that can be easily usable for localization or data tagging.

6.5.6 Mapping node

This node, which is actually called MlsmCore, is the one in charge of building the map and matching incoming data. We could synthesize the procedure of this node the same way as the others, but that would not be enough since here we have a more complex structure. Instead, we will go from higher levels, i.e. the MLSMCore class, to lower levels where the actual processing happens, i.e. MLSM ans ICP classes. The class diagram of these elements has been already seen in the analysis section, so we will jump into the MLSMCore node, which is the one doing the interfacing with ROS.

Data structures

- *Block*: Blocks are implemented as structs containing the parameters seen before. The mean and standard deviation are stored as *pcl::PointXYZI* containing the values for each axis.
- *Cell*: Cells are defined as *std::vector <boost::shared_ptr<mlsm::Block> >*
- *Grid*: The grid of cells that will form the MLS map is actually a tow dimensional multi_array (*boost::multi_array<cell, 2>*) which allows us to easily resize the array using the functions provided by the *boost::multi_array* class.

- *Quadgrid*: The *Quadgrid* is the class used for solving the problem of having a grid centred in the world that also needed to grow on different sides in different ways. What we basically have is a structure with four grids that are divided using the sign of their x and y indexes. Thus, we have: Positive(x)-Positive(y), Positive(x)-Negative(y),Negative(x)-Positive(y),Negative(x)-Negative(y)

Kd tree

A kd tree (k-dimensional tree) is a space partitioning data structure for organizing points in a k-dimensional space.

The main idea behind the construction of a kd Tree is using non-leaf nodes as splitting hyperplanes dividing the space in two parts. Points of the left subtree represent points to the left of the hyperplane and the points of the right subtree represent points to the right of the hyperplane.

In the case of the 2d tree used for cell partitioning, we have the cell positions which could be considered as a 2 dimensional space (x, y) and the planes used for division as planes aligned with the z axis.

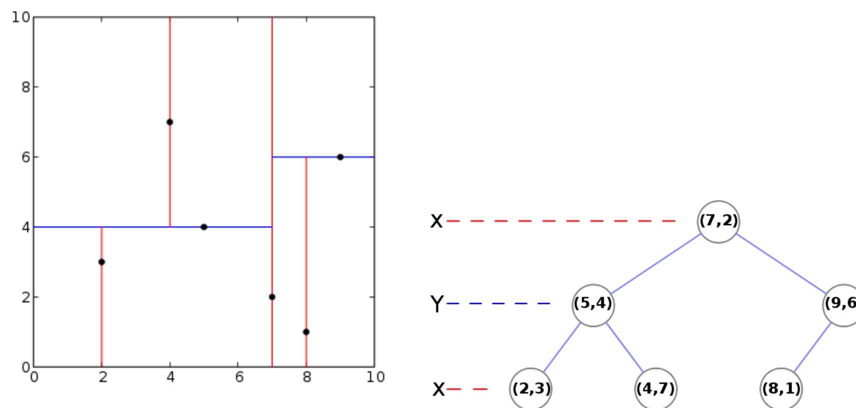


Figure 6.4: Planar representation of a 2dTree (Wikipedia)

Using a kdTree to organise the data was not an initial requirement, but it became one the moment we begun testing the ICP algorithm. For the closest point search we have to iterate through the cell structure of dimensions $m \times n$ where the search is $O(nxm)$. This slowed the algorithm too much, so the cells are also inserted into a 2d tree, where the search cost is $O(log_(n \times m))$.

MLSMCore

This is the node in charge of interfacing with ROS. It contains parameters related to the ROS side of the node, such as topics for subscribing and publishing, and callbacks that provide us with data or flag some event. Among the most important parameters we can find the following:

Map parameters:

- **Resolution**: Resolution of the map, it determines the size of each cell.
- **SizeXMeters**: Size in meters along the X axis. This, together with SizeYMeters and resolution define the initial number of cells the map contains.
- **SizeYMeters**: Size in meters along the Y axis.

ICP parameters:

- **Max. iterations**: Maximum number of iterations for the ICP algorithm.
- **Error threshold** Threshold for the iterations, when the error equals or is below this value we can accept the transformation as valid. (This is τ in the ICP algorithm).
- **N. samples**: Number of samples that should be taken in the direction of movement when running the algorithm while the robot is moving.
- **Sample step**: Proportion of the step taken. The step is calculated by multiplying the sub-estimated translation by this factor.

These nodes have two important features, one is the *StartMatching* topic subscription. The other is the callback where the cloud is added to the map. The *StartMatching* topic is a boolean topic that indicates whether we should run our input through ICP or not. This is useful for the initial mapping, but also for when we already have a known pose and we don't really need to consume resources by executing the algorithm for clouds we already have located.

The cloud callback just executes the *addPointCloudToMap* function, which will just add the point cloud to the map or calculate the estimated direction of movement vector and use ICP to try to undistort the data and provide the transformation before adding the corrected cloud to the map.

MLSM

This class is where the mapping takes place. By using the *addPointCloud(intensityCloud::Ptr cloud)* function, the MLSMCore node provides this class the point clouds that will be used for building the map itself. Logically, we will start by analysing the *addPointCloud* function, which is the core function of the map creation. Lets take a look at a general flow diagram that shows the steps taken in this function:

A good thing of this type of map (based on grid) is that adding a point somewhere is just a matter of calculating the indexes (for the 2D cell, working with the cell structure adds complexity). Calculating the indexes is as simple as doing the following:

```
index.x = floor(cloudIterator->x / resolution_);
index.y = floor(cloudIterator->y / resolution_);
```

The simplest case we have in this function is when the cell that should hold the block corresponding to the point does not even exist which is when this condition is not met.

```
if (((cellP = (*grid_)((int) index.x, (int) index.y)) != NULL)
        && (cellP->size() != 0)) {
```

In that case the only thing to do is to create a new cell with only one existing block with the values of the current point:

```
blockPtr = boost::make_shared<Block>(newMean,newVariance, 1,
cloudIterator->z, 0.0, FLOOR);

cellP->push_back(blockPtr);
double pos[2] = { round(index.x), round(index.y)};
assert(kd_insert( kdtree_, pos,0) == 0);
```

As shown in the code snippet, we do not only insert the block in the cell, but also add the cell indexes to the kdtree to allow for a faster search of occupied cells.

When the cell is not empty we have to go over the blocks, evaluating whether the measurement belongs to a block or not.

```
if ((fabs(cloudIterator->z - iterator->get()->height_) < resolution_)
        ||(fabs(iterator->get()->height_ - iterator->get()->depth_
                - cloudIterator->z)          < resolution_)
        || ((cloudIterator->z < iterator->get()->height_)
        &&(cloudIterator->z > (iterator->get()->height_
                - iterator->get()->depth_))))
{
```

If we check the MLS map section, the theory says that

> These candidate blocks b_{ij}^k must satisfy that $|p_z - height_{ij}^k| < cell_size$ and $|height_{ij}^k - depth_{ij}^k - p_z| < cell_size$.

This means that, for a point to be considered as belonging to a block, it must be at least at a distance of *resolution* from both the highest point of the block and the lowest point, which means that the block should have a maximum span of *resolution*. Instead of doing that, we consider that a point belong to a block when is at a distance closer than the resolution of the map. The other condition incorporated *((cloudIterator->z < iterator->get()->height_)&&(cloudIterator->z > (iterator->get()->height_ - iterator->get()->depth_))* starts working when we have points that lie inside fused blocks that span a column larger than *resolution*.

For each block in the cell we will evaluate whether the measurement could belong to a block or not, finding the following three situations:

- If we find a candidate block, we update its parameter using *addObservationToBlock* which updates its mean, variance and the rest of parameters.
- In case we keep finding blocks after one has been found, we use the *fuseBlocks* function to fuse the candidate we had with the one we just found.

- When no blocks are found, we create a new block and insert it in the cell the same way we did when the cell was empty.

The *addObservationToBlock* function synthesises what otherwise would be a blob of code coming from the mean and standard deviation update rules which are implemented according to the following equations:

$$\mu_{n+1} = \frac{n\mu_n}{n+1} + \frac{x_{n+1}}{n+1} \tag{6.5}$$

$$V_{n+1} = \frac{nV_n}{n+1} + \frac{(x_{n+1} - \mu_{n+1})^2}{n+1} \tag{6.6}$$

The *fuseBlocks* function works similarly to the *addObservationToBlock* function, the main difference is that the update of mean and variance is made using the mean and variance of the blocks that are going to be fused.

The last case, where we only have to create a new block does not involve any complications, we just initialise the new mean and variance and create the new block that will be inserted into the cell. This procedure is the same applied when the cell corresponding to the measurement is empty, the only difference is that if the cell was empty the indexes are also inserted into the kdtree:

```
// New mean and variance
newMean.x = cloudIterator->x;
newMean.y = cloudIterator->y;
newMean.z = cloudIterator->z;
newMean.intensity = cloudIterator->intensity;
newVariance.x = 0;
newVariance.y = 0;
newVariance.z = 0;
newVariance.intensity = 0;
// Block creation
blockPtr = boost::make_shared<Block>(newMean,newVariance, 1,
cloudIterator->z, 0.0, FLOOR);
// Insertion into cell and kdtree
cellP->push_back(blockPtr);
double pos[2] = { round(index.x), round(index.y)};
assert(kd_insert( kdtree_, pos,0) == 0);
```

The other important functions of the *MLSM* class are *findSuitableBlock* and *findClosestBlock*. The first one uses the same rules for searching the block where a point be inserted to find if a given point belongs to an existing block. The second one is used to find the closest cell to a given position using the kdTree. Both these functions are used together to find the closest blocks to the points fed to the ICP algorithm that will be addressed next.

ICP

The ICP class performs the ICP algorithm which tries to match incoming clouds to the existing map. The core procedure of this class is the *getTransformation* function, which executes the main loop of the ICP algorithm. If we ignore the initialisation of variables, the main loop is implemented as follows:

```
while ((iterations < maxIterations_) && (error > errorThreshold_)){
        (*R) = Matrix<float, 4, 4>::Identity();
        // Calculate closest points
        Y = closestPoints(P,X, P0->timeStamps,eV);
        // Calculate transformation
        error = registration(P,&Y,P0->timeStamps,&transforms, T, R,eV);
        /* Iterate through P applying the correct transformation depending
           on Tv and the stamp */
        applyTransformation(P,P0->timeStamps, *R, transforms, *T);
        // Calculate error
        error = calculateError(P, &Y);
        iterations++;
        // Accumulate transformations
        accumulatedT += *T;
        accumulatedTv += *Tv;
        accumulatedR = accumulatedR * (*R);
}
```

As we can see, each step of the ICP iterations has been programmed into a function, so we can easily address each of the functions separately to check how each of the steps are implemented.

The *closestPoints* function takes the sensor cloud (P), the map (X), the timestamps of each beam (*P0->timeStamps*) and the estimation of current velocity (eV) as inputs. In this function we try to find the closest point to each point of the input cloud in the map with a small modification where the *nSamples* ICP parameter and the estimation of velocity are used. For each measurement we first find the closest block conventionally, recurring to the *findClosestBlock* function from the *MLSM* class. After that we check if we have a velocity estimation and if the *nSamples* parameter has been set. After that we find *nSamples* closest points, but, instead of using the original point, we search for the closest point to the original point plus a translaction in the direction of movement multiplied by the sample index and the *sampleStep* parameter. This way we have a series of positions in the direction of movement that could provide better matches since the measurements are displaced due to the vehicle's motion. In order to do this, we use the difference between the movement direction and the line from the sample/measurement to the point found as closest point:

```
unitSpeed[0] = eV[0];
unitSpeed[1] = eV[1];
unitSpeed[2] = eV[2];
```

```
unitSpeed.normalize();
for (int j=0;j<nSamples_;j++){
        p[j].x = p0.x + movingTime* eV[0] * j * sampleStep_;
    p[j].y = p0.y + movingTime* eV[1] * j * sampleStep_;
    p[j].z = p0.z + movingTime* eV[2] * j * sampleStep_;
    blockPtr = X->findClosestBlock(p[j]);
    if (blockPtr == NULL) break;
    // Find angle between the block and p0
    unitDir[0] = blockPtr->mean_.x - p0.x;
    unitDir[1] = blockPtr->mean_.y - p0.y;
    unitDir[2] = blockPtr->mean_.z - p0.z;
    unitDir.normalize();
    angle = acos(unitSpeed.dot(unitDir));
    if (fabs(angle) < closestAngle){
            bestBlockPtr = blockPtr;
        closestAngle = angle;
    }
}
```

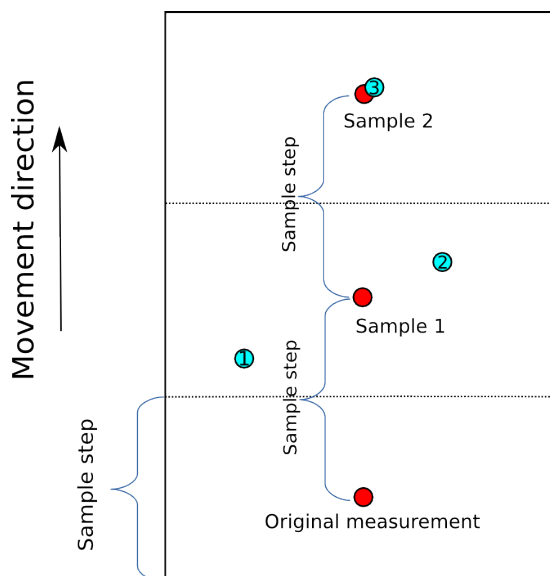
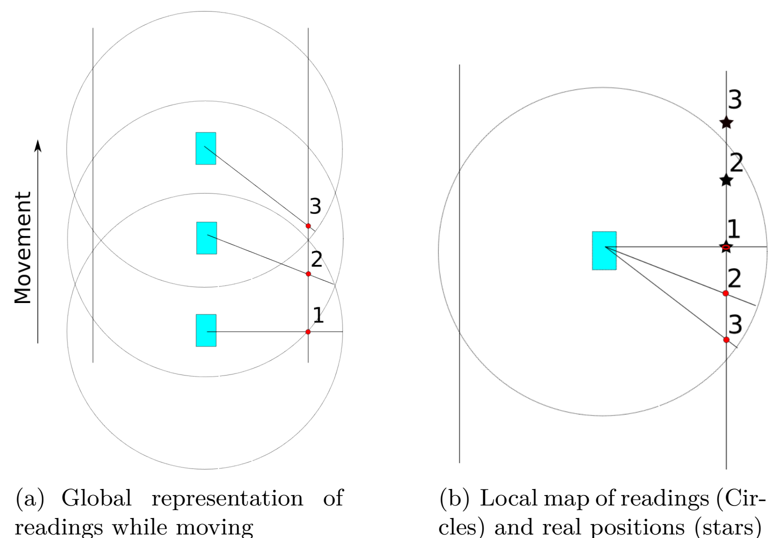


Figure 6.5: This image is an example of the directional search. As we see, the closest point to the original measurement would be point 1. If we search one sample further in the direction of movement we will have point 2 as the closest point. And finally after three samples we would get point three, which is the point that best aligns with the direction of movement

The *registration* function has three input parameters: The point cloud from the sensor, the closest points from the map, and the timestamps of each measurement. The output parameters are the transformation for each point, the rotation of the point cloud (for still measurements), and the translation suffered by the vehicle. The original ICP registration function uses the centroid of the input dataset and the centroid of



(a) Global representation of readings while moving

(b) Local map of readings (Circles) and real positions (stars)

Figure 6.6: Global and local map representations

the closest points to calculate the translation from one to the other. This works if we assume that we are capable of getting a reading of our surroundings instantaneously while moving, or that we are not in motion when gathering the readings. In the case of our imaging sonar, we are able to gather a scan of the environment and then match it using the original ICP, there is not much of a problem there. The problem arises when we are moving but also want to use the ICP algorithm to know our position.

The original ICP assumes that the translation error between a point and its closest match is the same for all of the input points. In our context, this is only possible when static, gathering readings while moving would produce each point to have different translations, which are directly related to the speed and duration of the movement. In order to solve this we have introduced two modifications of the original ICP algorithm. The first one has already been pointed out in the *closestPoints* function. The second one is made in the *registration* function, where we use the slope of error to estimate the speed and provide transformations for each point based on speed.

Figures 6.6(a) shows how the measurements are taken while moving, and 6.6(b) shows how the measurements are represented in the local map (red) and where they should belong(blue). The transformation that takes the measurements from the local map to the global map is the one we try to find on the *registration* function. I order to do so, we use the error between each point and its correspondence. If we consider the initial time t_0 as the moment where both movement and scan begin, and have $t_{0..n}$ as the timestamps of each measurement, and $e_{0..n}$ as the error in the direction of movement of samples taken at each time increment. We can assume that the initial measurements will have less translation than the last ones. More so, the last measurements' translation, should be the same as the displacement suffered by the vehicle from t_0 to t_n.

The most basic situation when calculating the transformations would be when the vehicle is static, this produces a flat line of errors, producing the same as the centroid difference calculation. The next basic situation would be where the vehicle is moving at

a more or less constant speed, the curve representing the error between measurements and corresponding points should be a line with equation $y = mx + n$ where the slope is actually the speed of the vehicle. If we combine these two cases, we can have a scan that is gathered from measurements made while static and while on movement, producing what we could call translation segments similar to the ones on figure 6.7. Following the example on the figure, the points from $t = 0$ to $t = 5$ would have an error of 0, the points from $t = 5$ to $t = 10$ would have a translation represented by $y = \frac{3}{5}(t - 5) + 0$. Jumping a bit forward, the points between $t = 15$ and $t = 20$ have a translation of 8 and the ones from $t = 20$ to $t = 25$ have the error represented as $y = \frac{20}{25}(t - 20) + 8$.

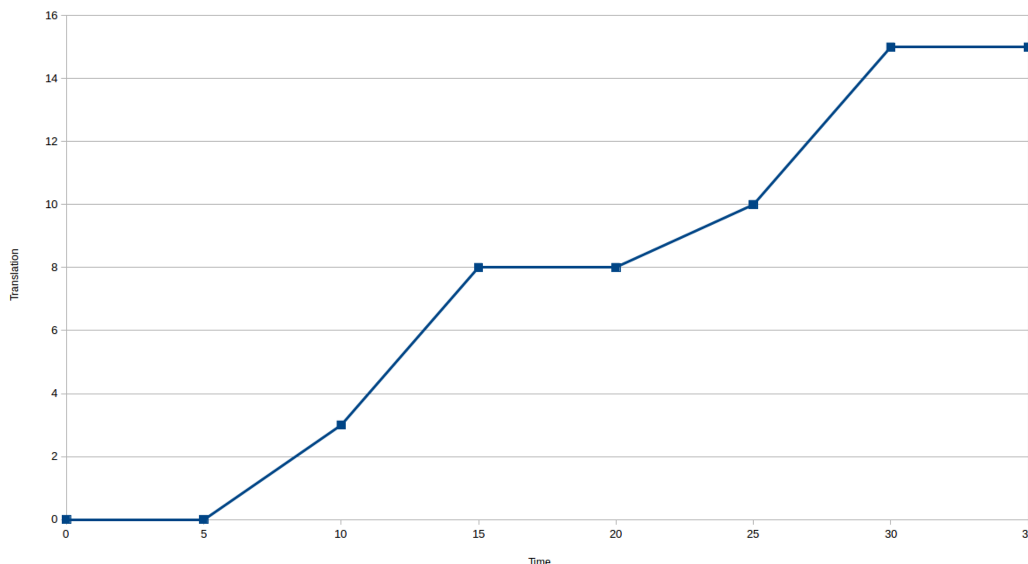


Figure 6.7: Example or time and translation representation

Figure 6.5.6 shows how the translation are calculated by segments. We just accumulate data for each type of segment and process it differently when we reach the end it. The diagram shown does not cover the calculation of the last segment, which basically uses the same if condition and calculations depending on the segments' type.

The registration process described in the former paragraph is then followed by the *applyTransformation* function, which is very simple. It just takes the list of transformations provided by the *registration* method and iterates through the input cloud transforming each point. Even though there are methods to directly apply a transformation to a point cloud, i.e. *pcl::transformPointCloud*, they use a transformation matrix, and we don't want to use the same transformation. Thus, we end up moving points "the hard way" as shown in the following snippet.

```
P->at(i).x = P->at(i).x + transforms.at(i)[0];
P->at(i).y = P->at(i).y + transforms.at(i)[1];
P->at(i).z = P->at(i).z + transforms.at(i)[2];
```

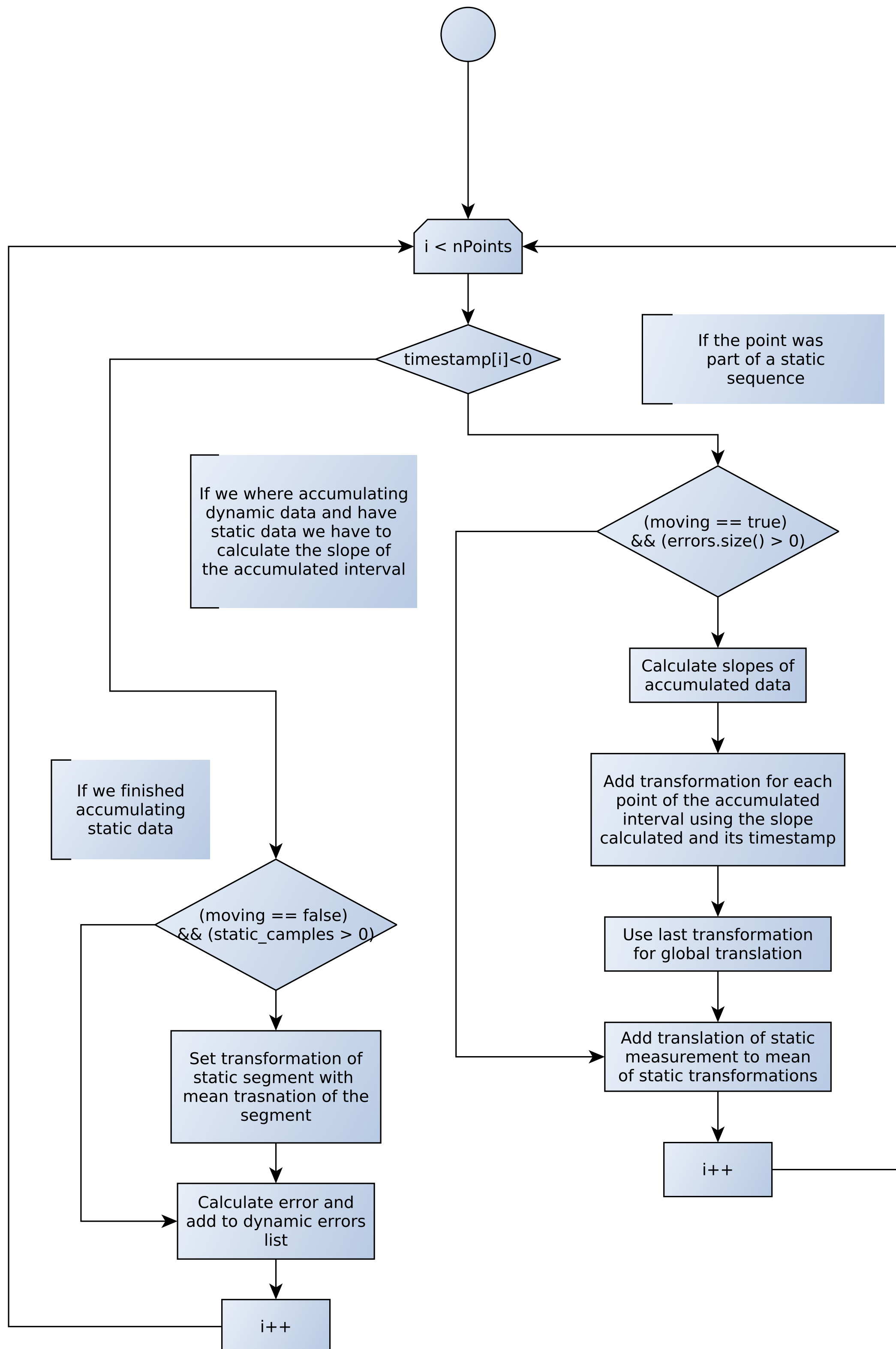


Figure 6.8: Flow chart for translation calculation by segments

The next step of the ICP algorithm is error calculation. This step is performed by the *calculateError* function, as its name points out. This step has been preserved the same as the one in the original ICP algorithm. We calculate the error as the mean euclidean distance from the transformed points to their correspondences. There is only a slight difference, we also have a threshold that sets the error to zero for a given point in case it is below a threshold. This is done due to the fact that blocks actually span an area, so when a measurement has its block correspondence, the measurement could actually belong to anywhere inside the area spanned by the block, since measurements from different positions could yield points that are located in a different area of the block rather than its mean.

Finally we accumulate the results of the current iteration and proceed to the next one:

```
iterations++;
accumulatedT += *T;
accumulatedR = accumulatedR * (*R);
```

6.6 Processing distribution

After the last version, the components where a bit crumbled up, there were only two nodes carrying out all the work to produce the final result. There was the processing node, with quite different processing stages taking part in the same place, and the mapper node, which basically took the product of the processing node and detection nodes to integrate them in the map. I was already happy with the structure for the mapper, but the other node was not so right. The other option could have been separating each processing step into a node. The main drawback there is that there would be around four nodes, transmitting sonar data among them, what creates a high system overhead due to the cost of constantly sending large amounts of data. Due to this, I went for turning each processing step into a nodelet, taking advantage of the zero copy feature of the nodes, that makes it possible to share information at no cost, instead of having to send it through the TCP/IP connection of regular nodes. This way I could keep the efficiency of the monolithic processing node, with the flexibility of having several nodes.

After this decision each component was separated into different nodes and nodelets so that each component could be easily tested, modified, or even replaced. Each of the "processing" components (SonarToCloud, Thresholder...) were built into nodelets.

In general, the reasons for this "extra" reorganization were the following:

- First of all, separating the processing stage into different execution units would allow us to directly get rid of the processes that were not needed. Instead of having several conditions to skip the unwanted processing stages.
- The code and the system would be easier to follow and debug. The stages are clearly divided and they are also easily debugged since we can just check the output of each stage by subscribing to it.

- We do not incur in extra system overhead. Each stage has to transmit to the next one the processed data, which is a point cloud message. If I had implemented each stage in a separate node there would have been some overhead due to the point clouds coming and going through the topics that connected each node. Since nodelets allow us zero copy data transmission the system won't get any overhead.
- Adding new filters or processing stages is as simple as adding a new nodelet to the pipeline.

7 | Results

This chapter addresses the results gathered from testing the developed tools. Since these tools can be checked out independently if followed from the bottom up, meaning that there is no point on testing the mapping tool using noisy data since the filtering steps work properly.

In order to properly understand the images shown in this chapter the following guidelines should be taken into account:

- Sonar scans are viewed from top, so, what is being observed is a horizontal scan.
- MLSM images show cylinders and cubes that build vertical structures. When this structures are comprised of vertical blocks, they have a lighter colours than the rest.
- The images showing the ICP algorithm result show the global map and the result of the matching algorithm.

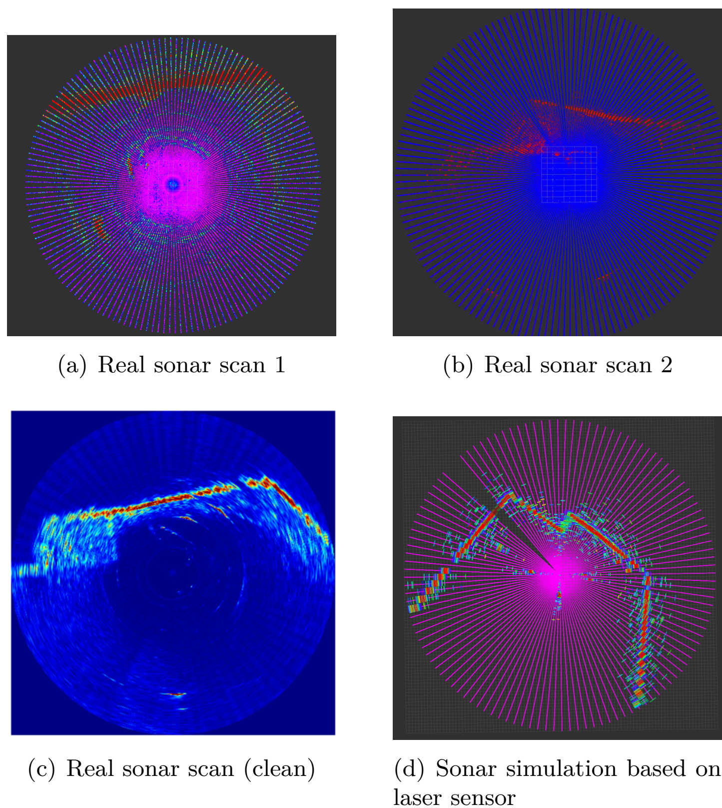
Important notice

The lack of real world testing is due damages caused to the platform during one of the competitions, which prevented the vehicle from being used for testing. Even though, this project will serve as a starting point for future projects that will test and expand the software in real conditions.

7.1 Sonar simulation

A lot of factors intervene in the data returned by the beams of the sonar, as seen in section 4. Some of these parameters where not simulated due to limitations both in complexity and ability to extract the features needed from the tools used (Gazebo). Figure 7.1 shows some data gathered from the sensor and from the simulator.

Figure 7.1(a) represents data gathered from the same wall (with a shorter range) as figures 7.1(b) and 7.1(c). The main reason of this noisy image was that the gain was not properly adjusted. Even though, posterior filtering stages can easily deal with this kind of noise. Figures 7.1(b) and 7.1(c) show properly tuned scans, which yield more accurate results. If we were to compare them with the one produced by the simulator (7.1(d)), the intensities might seem a bit more concentrated and a bit noisier than it should regarding proper tuning. Still, it can be used for simulation without great concerns due to two reasons. The first one, and most obvious, is that this is a simulation using a simplification of the sonar's behaviour, developed for testing. The second one is that after the processing stages, the result ends up being quite similar when we have the processed data from the sensor or from the simulator.



(a) Real sonar scan 1

(b) Real sonar scan 2

(c) Real sonar scan (clean)

(d) Sonar simulation based on laser sensor

Figure 7.1: Comparison of real and simulated scans

7.2 Sonar processing

The filtering steps work reasonably well on both original sensor data and simulated data as figures 7.2,7.3, and 7.4 show.

For clouds similar to the ones on figure 7.4 the Otsu thresholder provides a value around *66* which, considering that the maximum value is *127*, is an intermediate value. This is appropriate for Otsu filtering, i.e. background separation, but in general it is good to tune this value up a bit by using the *OTSUMultiplier* parameter, but it should be tuned for each environment accordingly.

7.3 Mapping

As shown on figures 7.3 and 7.6 the map is built properly. In general it is easy to see the grid structure created. We can also notice the block fusion at it's best on figure 7.5(b), where the small distance made possible to fuse blocks with almost the entire height of the wall (large bright red cylinders).

On figure 7.6 we can better appreciate how the map is built when using long range scans. We can see that the closest vertical structures contain fused blocks (light yellow pillars) while further measurements are harder to combine due to the opening between scan angles.

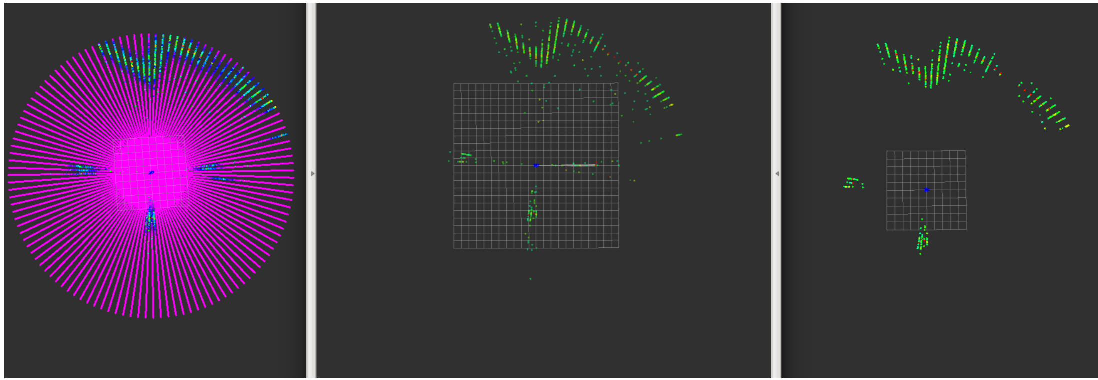


Figure 7.2: Thresholding and outlier removal with simulated data

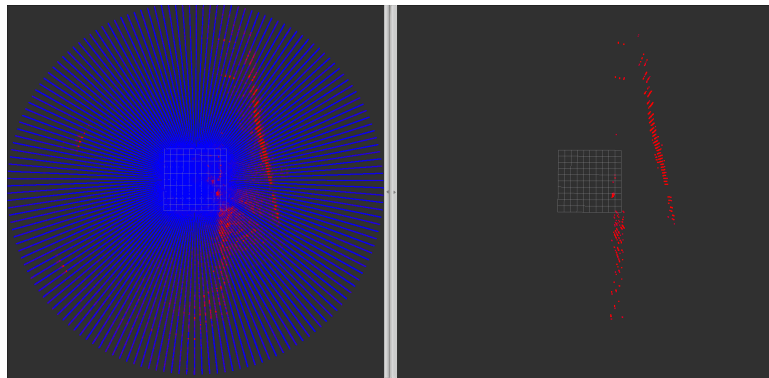
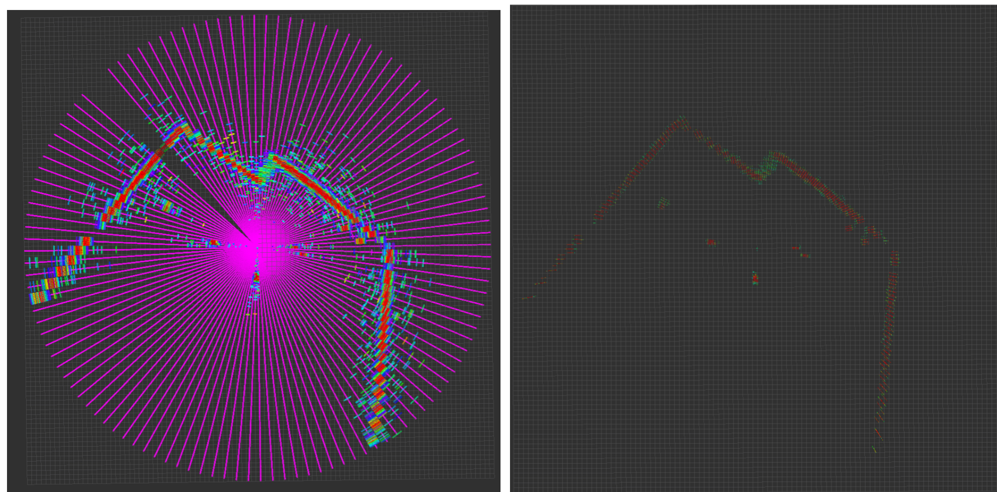


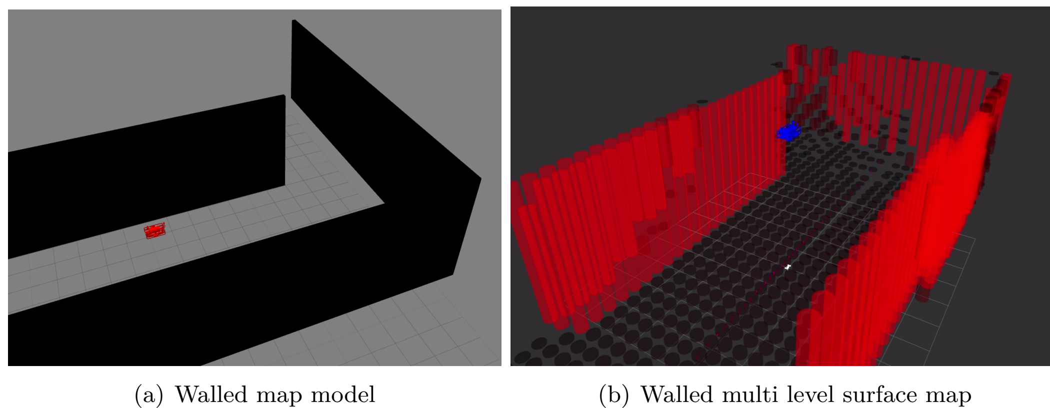
Figure 7.3: Cleaning and line detection from sensor data



(a) Cleaning and line detection from sensor data

(b) Cleaning and line detection from sensor data

Figure 7.4: Long range simulated scan 50m



(a) Walled map model (b) Walled multi level surface map

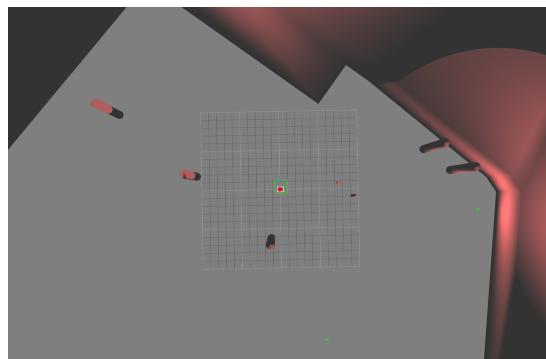


Figure 7.5: Taliarte model

7.4 ICP

The static ICP has yielded the expected results, which was to be expected, since there is not much complication on using the centroid of the clouds for translation calculation.

The screenshots of figures 7.7, 7.8, 7.9 and 7.10 show the dynamic ICP working after different situations: Moved 5 meters on X, moved another 4 meters on Y, and then, -10 meters on X. Instead of directly matching the clouds with no approximation, we have used a more realistic approach for testing. Odometry is extracted from the simulator but, since this odometry is perfect, we induce an error of 30%, which means that for a distance of 4 meters in odometry, there is a correction of 1.2 meters to be corrected. Each of the figures specifies how many meters the robot has moved, and also include the correction given by the algorithm. For figure 7.7 the correction should be around 1.2 meters (4 meters in X plus a 30% correction). For figure 7.8 we would have around 1.2 meters in X and 0.9 in Y. And finally, for figure 7.9 we should have -1.2 in X and 0.9 in Y.

On figure 7.10 we can see the correction applied to transform from *odom* frame to *map* frame. This is also a good visual example of how the transformation between frames work.

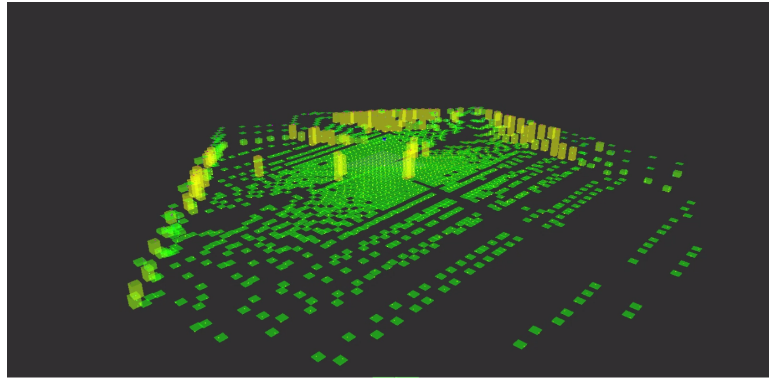


Figure 7.6: MLSM generated Taliarte model

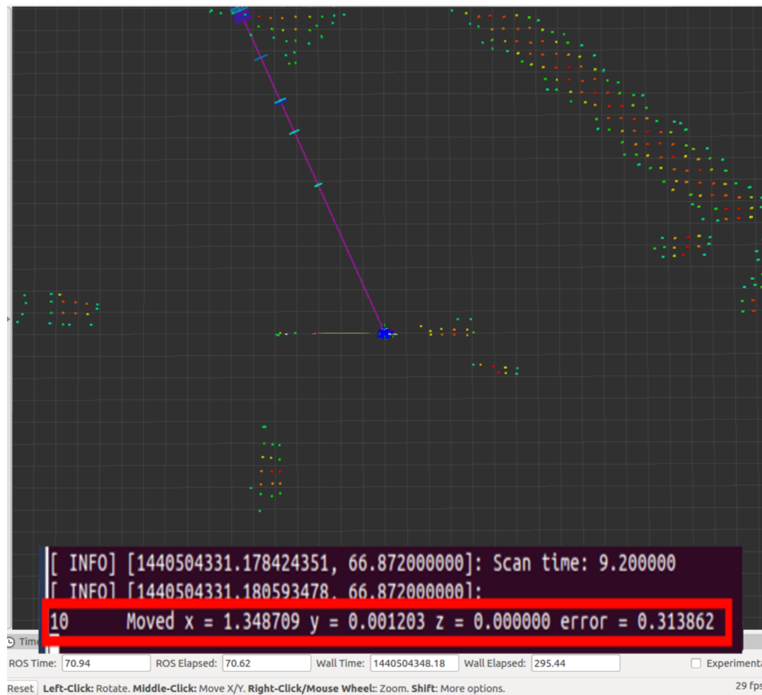


Figure 7.7: Moved 4 odom meters on X (Correction on X = 1.348, Y = 0.001, Error = 0.313)

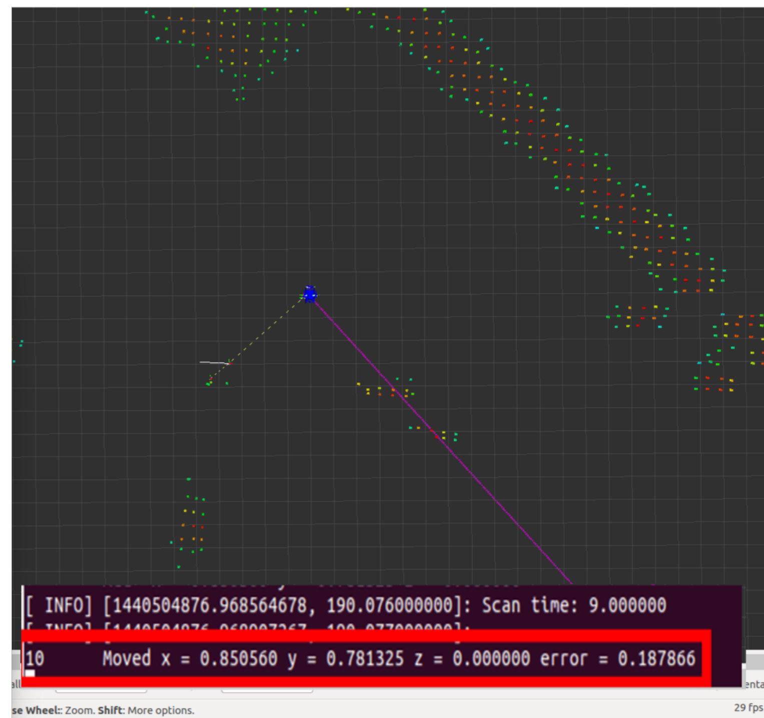


Figure 7.8: Moved 4 odom meters on X and 3 odom meters on Y (Correction on X = 0.851, Y = 0.781, Error = 0.188)

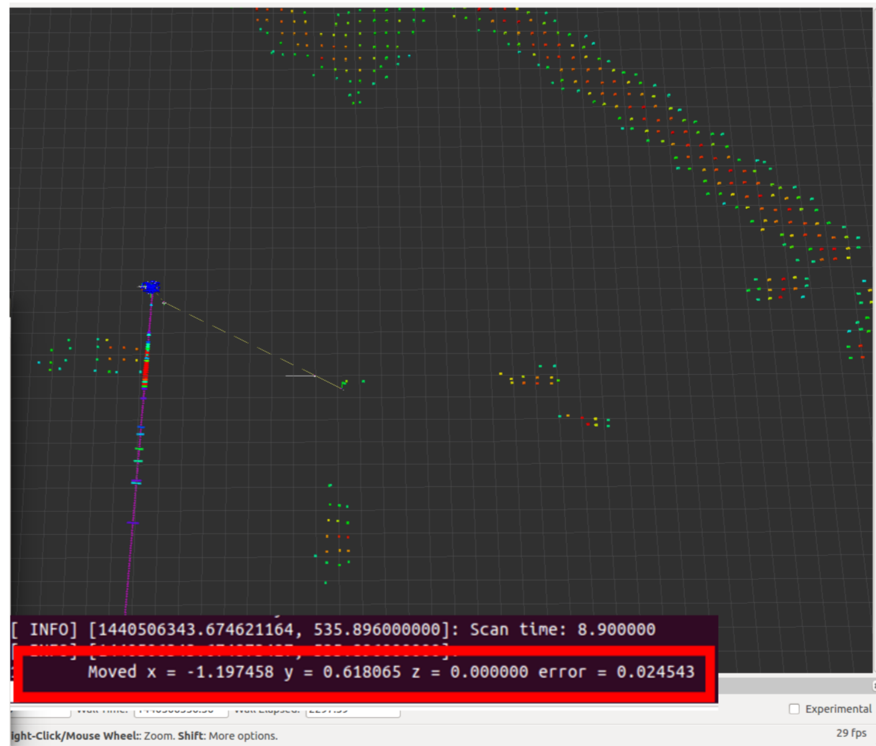


Figure 7.9: Moved -8 odom meters on X after 7.8 (Correction on X = -1.197, Y = 0.618, Error = 0.025)

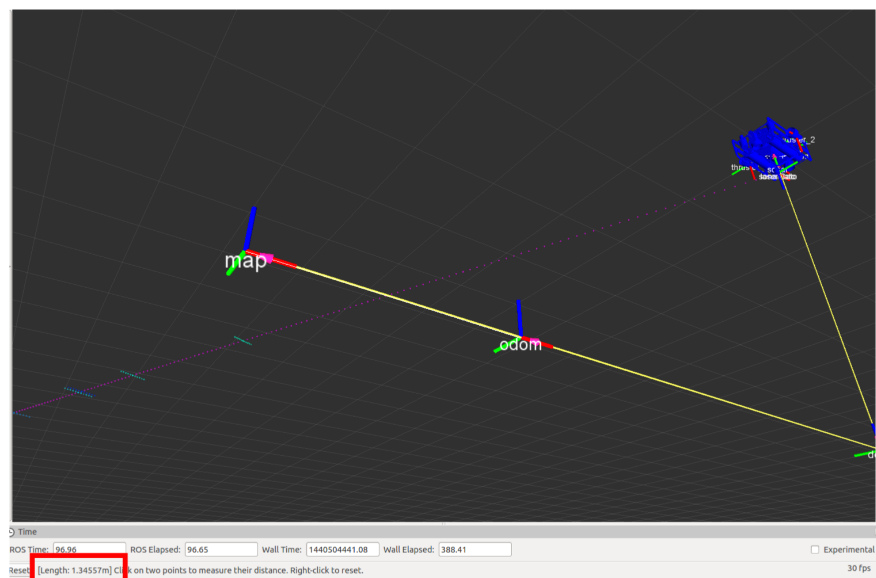


Figure 7.10: Transformation visualization for image 7.7

8 | Conclusions and future work

8.1 Conclusion

A realistic simulation tool for an Imaging Sonar sensor has been developed and integrated into Gazebo, along with a robot model, successfully, this will allow future developments to be tested without the need of the robotic platform or the sonar itself.

There are many factors that intervene into the formation of echo returns on a Sonar image, with our objectives in mind the developed simulation is only an approximation to reality.

A Sonar data processing pipeline has been established and developed successfully. The processing pipeline implemented in the *sonar_processing* package yields good results and can be used with real world data.

The graph architecture ROS provides is a great way of tackling both small and large systems that require asynchronous processes. Also nodelets present a good solution to the problem of communicating large chunks of data between nodes, like the ones we can find when communicating point clouds.

Several mapping structures have been analysed. Each structure tends to present a better solution for different situations, and we have to find the one that best balances all the features we look for.

A mapping tool has been developed using point clouds as an input, so the mapping section can be actually used with whatever sensor is desired as long as they meet the format of message used for input, which is another benefit from ROS also.

The ICP algorithm has been analysed and also modified to suit the platform and sensor used, which is another of the strong points of this project. The solution is not ideal or perfect (but still, localisation algorithms do not tend to be perfect), but provides a good solution to track our position, which works even better if we are able to stay more or less static.

As side benefits for the developer himself, the development of a project that comprises of simulation, data processing and mapping altogether has been very enlightening, providing good background for future developments of this sort.

8.2 Future work

There is always room for improvement, and among the possible ones regarding this project we can find the following:

- The simulation node could be checked out after further releases of Gazebo-ROS to check if Gazebo's native sonar sensor can be used for simulation. This would also provide room for considering more parameters in the simulation, such as the surface of collision (its material and normal according to the sonic pulse) and any other parameters that require gathering information on the collision itself.
- For the processing nodelets, some improvements could be made, regarding the use of organised point clouds as input. Currently, the methods provided by PCL for filtering require unorganised point clouds, so, if we want to keep the output of the processing stage as organised, the fact that the filtering methods should be implemented from scratch should be taken into account.
- The previous improvement relates also to the possibility of converting clouds into images, thus giving the possibility of adding OpenCV methods in the processing pipeline. Or making it easier to use OpenCV detection methods as Hough Transform.
- The mapped package could be expanded to hold several types of map, providing support for Octomap, which seems could provide optimisation space-saving wise.
- The ICP matching function for dynamic data could be improved by adding considerations of the nature of the movement, since speed will not be linear.
- Finally, the pipeline conceived for data processing + matching + mapping could be altered following the work on [14].

Bibliography

[1] Boost web page. http://www.boost.org/.

[2] gazebo_hector_packages. http://wiki.ros.org/hector_gazebo_plugins.

[3] Ros wiki. http://wiki.ros.org/.

[4] Team hector. http://www.gkmm.tu-darmstadt.de/rescue/.

[5] P.J. Besl and H.D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1992.

[6] D. Scaramuzza C. Forster, M. Pizzoli. Svo: Fast semi-direct monocular visual odometry. *IEEE International Conference on Robotics and Automation (ICRA)*, 1992.

[7] Jason Gillham. Underwater sonar and laser measuring. an experimental comparison, 2011.

[8] Jun Hua Zhenqiang Yaoa Hong Shen, Maoli Rana. An experimental investigation of underwater pulsed laser forming. *Optics and Lasers in Engineering Volume 62, Pages 1-8*, 2014.

[9] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. Software available at http://octomap.github.com.

[10] Z. Li, Q. Zhu, and C Gold. *Digital terrain modeling: principles and methodology.*

[11] V. Prieto Marañon, J. Cabrera Gámez, A. C. Domínguez Brito, D. Hernández Sosa, J. Isern González, and E. Fernández Perdomo. Efficient plane detection in multilevel surface maps. *Journal of physical agents, Vol. 5, NO. 1*, 2011.

[12] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Trans. Sys., Man., Cyber. 9 (1): 62âĂŞ66*, 1979.

[13] Patrick Ptaff, Rudolph Triebel, and Wolfram Burgard. An efficient extension to elevation maps for outdoor terrain mapping and loop closing. *The International Journal of Robotics Research, Vol. 26, No. 2*, 2007.

[14] David Ribas Romagós. *Underwater SLAM for Structured Evironments Using an Imaging Sonar.* PhD thesis, Universitat de Girona, Department of Computer Engineering, May 2008.

[15] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.