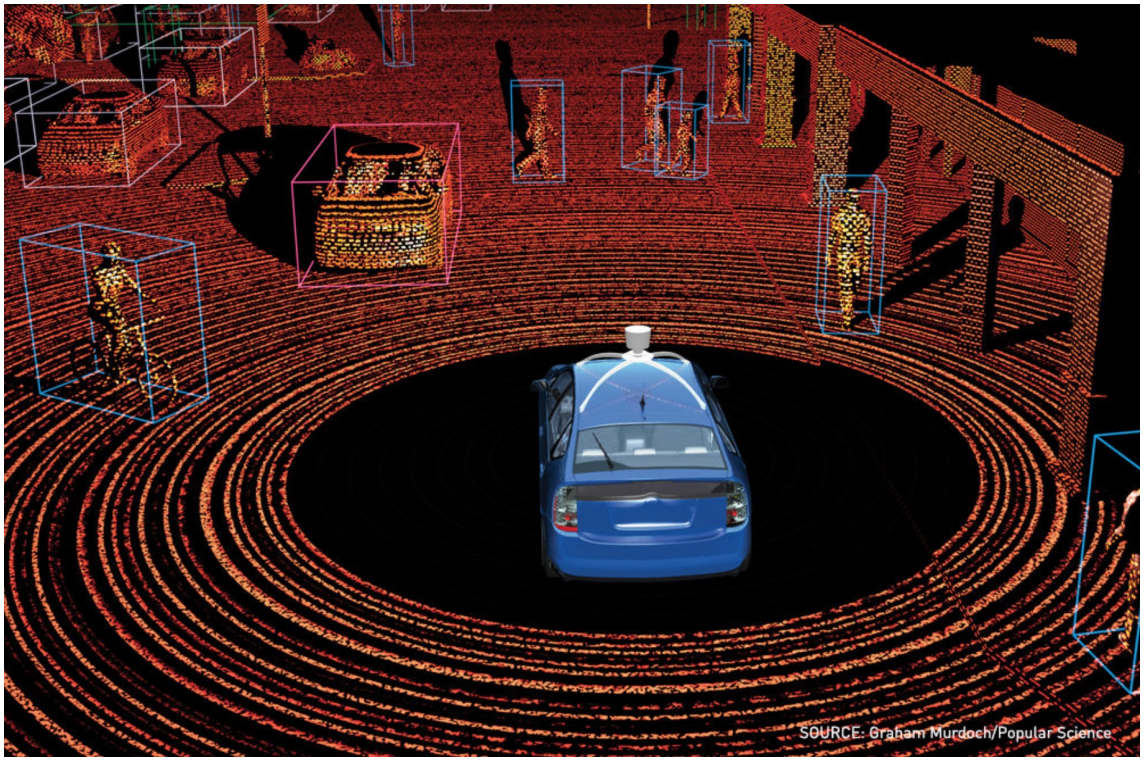

SLAM with 3D LiDAR

Romain Henry, Florent Willemin, Zhi Yan and Yassine Ruichek

June 15, 2018

Contents

1 Introduction **4**

2 Context **4**
2.1 Specifications . . . 4
2.2 LiDAR . . . 5
2.2.1 How LiDAR works . . . 5
2.2.2 Why LiDAR . . . 5
2.2.3 Our LiDAR . . . 6
2.3 ROS . . . 7
2.3.1 ROS Definition . . . 7
2.3.2 ROS purpose . . . 7
2.3.3 ROS history . . . 8
2.3.4 ROS general organization . . . 8
2.3.5 Languages to program ROS . . . 9
2.3.6 ROS basics . . . 10
2.3.7 ROS for Laser Odometry And Mapping (LOAM) . . . 11
2.4 SLAM algorithm : LOAM . . . 11
2.4.1 Lidar Odometry . . . 12
2.4.2 Lidar Mapping . . . 13

3 Results **14**
3.1 Setting Up the environment . . . 14
3.1.1 How to build and run your catkin workspace . . . 14
3.1.2 How to debug . . . 15
3.2 Entire map without optimization . . . 15
3.3 With Octomap . . . 18
3.3.1 About Octomap . . . 18
3.3.2 Octomap integration . . . 19
3.3.3 Octomap resolution and range . . . 22
3.4 2D LIDAR . . . 25

4 Conclusion **27**

List of Figures

1 Kanban system . . . 4
2 Explanation of the LiDAR principle . . . 5
3 Explanation of the LiDAR principle . . . 5
4 Comparison of acquisitions . . . 6
5 The Velodyne models . . . 6
6 Comparison of the Velodyne models . . . 7
7 ROS.org logo . . . 7
8 Willow Garage . . . 8
9 Kinetic Kame . . . 9
10 ROS overview . . . 11
11 LOAM software system overview . . . 12
12 Exemple of extracted landmarks points . . . 12
13 Feature point correspondence . . . 13
14 Mapping algorithm . . . 13
15 Qt logo . . . 14
16 Rqt Graph not optimized . . . 16
17 Entire map not optimized . . . 17
18 Begining of the resources usage during LOAM processing by keeping the entire map 18
19 End of the resources usage during LOAM processing by keeping the entire map . . 18
20 Limit of map height . . . 19
21 Octomap with wrong orientation . . . 20
22 About Octomap Z axis issue . . . 21
23 Max double value . . . 21
24 Octomap launchfile : node code . . . 22
25 Map with small range and high resolution . . . 22
26 Map with big range and low resolution . . . 23
27 Octomap result of the entire 3D map . . . 23
28 Octomap result of a 2D map . . . 24
29 Begining of the resources usage during LOAM processing by using Octomap 24
30 End of the resources usage during LOAM processing by using Octomap 24
31 Rviz view with LOAM processing using two lidar . . . 25
32 Rqt_graph with two lidar . . . 25
33 Rqt_graph for octomap with two lidar . . . 26

1 Introduction

During TO52 we have to develop a computational project by merging all our skills. Thus, the aim of this course is to acquire skills in techniques and organization to carry out an academic development project. This project is highly supervised by a professor in order to help us when needed thanks to his strong background. In our case, we chose to develop the project proposed by Ruicheck Yassine (Professor and Director of the System and Transport Laboratory at the University of Technology of Belfort-Montbéliard (UTBM)) and Zhi Yan (Assistant Professor in the LE2I-CNRS laboratory at the UTBM). Our goal was to understand the general principle and test solutions of the fundamental problem we can find in mobile robotics, that is to say the simultaneous localization and mapping (SLAM). It is a field which has been intensively studied in the robotics community in the past. To work on this project we got data from a Velodyne HDL-32E LiDAR and a SLAM source code which provided by LOAM. We chose to work on this subject because we are doing our best to improve our skills in embedded software and image processing in order to have a challenging career in autonomous vehicle. Indeed, we thought that this project is so strongly linked with our life goal. So in order to fulfill the project expectations, we had firstly to request the specifications, resume them and build a roadmap, then we had to take in charge the context, and especially to see how the software and hardware (LOAM and LiDAR) provided are working, finally we could start developing and test solutions.

2 Context

2.1 Specifications

First of all we needed more informations about the specifications. The subject given was only a brief of the mission. Our goal was here to ask the right questions to our superviser which was Zhi Yan. It allowed us to see how it is in the real world. Indeed, we could compare Zhi Yan with a client which need a solution to his problem. So our questions were well prepared and after the meeting with the client we got a better overall view of the project. Indeed, we understood that the first issue was the fact that the entire map was not kept with the published LOAM algorithm. So we had to find a way to display the entire map by optimizing the use of resources (especially the RAM).That's why we could directly divide the project in multiple steps and the idea here was create a kind of kanban system in order to visualize our progression. Kanban system is a way to limit the different process in progress. It comes from the Japonese management methods call "the lean management", the translation could be "reporting cards". The well known GitHub provide us a tab which allow us to create a Kanban system -like. The drawback of this tab is that we can't put deadline which is so much important in the Kanban system and in software development. Nevertheless, despite this drawback, we chose to create the board you can find below which could summarize all the steps we have to work on.

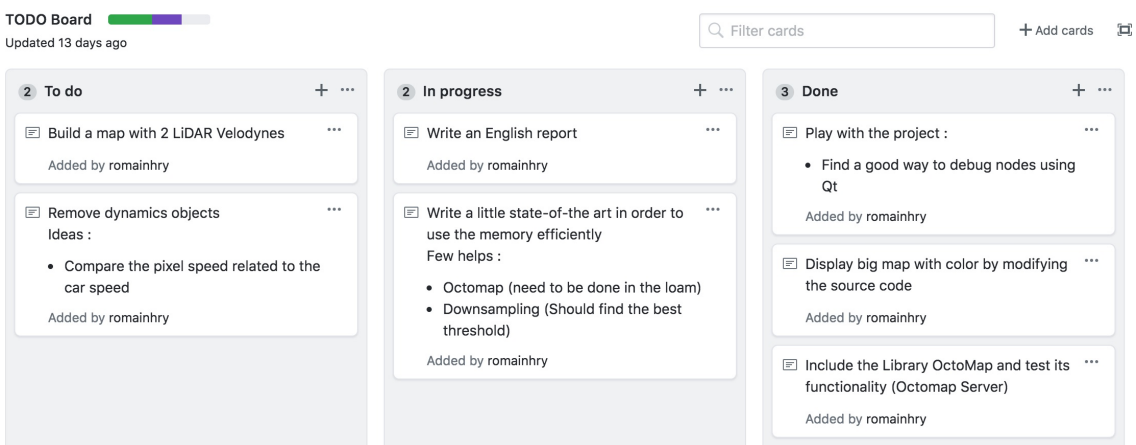

Figure 1: Kanban system

2.2 LiDAR

Since we are working with data coming from a 3D LiDAR, it was interesting for us to understand how this technology work, why is it so strongly used compared to other technologies.

2.2.1 How LiDAR works

LiDAR, which stands for Light Detection And Ranging, uses laser pulses to take measurements and generate a map of an environment. After the unit sends out a laser pulse, a sensor on the instrument measures the amount of time it takes for the pulse to bounce back. As light moves at a constant speed, the LiDAR unit is able to accurately calculate the distance between itself and the target. To summarize it is like a telemeter placed on a motor that will give the rotation. It can build a 3D map or a 2D map. To build a 3D map the telemeter should have another degree of freedom (DOF), that's why another motor is often placed in order to change the tilt of the laser. Below we can see that the motor will translate a piece in order to change the angle of the mirror.

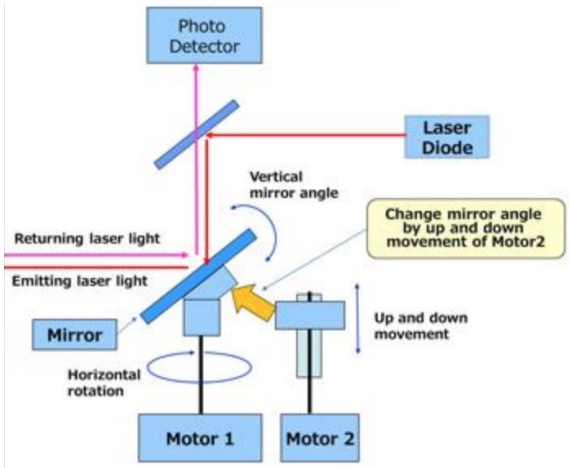

Figure 2: Explanation of the LiDAR principle

As we could understand, only the mirror is moving, thus it is impossible to have all the environment because it will be hidden by the system itself. So the 3D LiDAR models can have more or less plans varying by the angle of the mirror. And more we have plan, more it is expensive.

2.2.2 Why LiDAR

LiDAR has its own drawbacks and its own advantages. First of all we would like to show you a standard sensors set we can find on a self-driving, and we will then study why all this sensors are set.

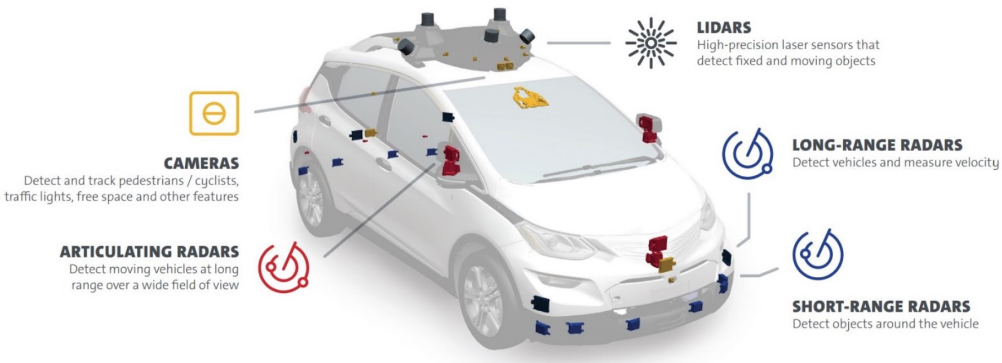

Figure 3: Explanation of the LiDAR principle

Thanks to the figure 3 above, we can see that self-driving cars use a lot of other sensors to see, notably radars and cameras, but laser vision is hard to match. Radars are reliable, but don't offer

the resolution needed to pick out things like arms and legs. Cameras deliver the detail, but require machine-learning-powered software that can translate 2-D images into 3-D understanding. Lidar, by contrast, offers hard, computer-friendly data in the form of exact measurements.

Rating: H = High, M=Medium, L = Low	Camera	Radar	LiDAR
Object Detection	M	H	H
Classification	H	M	L
Close-Proximity Detection	M	H	L
Speed Detection	L	H	M
Lane Detection	H	L	L
Traffic Sign Recognition	H	L	L
Range	H (200m)	H (250m)	M (120m)
Work in Rain, Fog, Snow	L	H	M
Work in Low Light	L	H	H
Work in Bright Light	M	H	H
Size	Small	Small	Medium
Cost	$	$$	$$$$

Figure 4: Comparison of acquisitions

The table above show us the different drawbacks and advantages between all the technologies. We can imagine thanks to that why we need to merge the 3 types of sensor. But nowaday, some car manufacturer are trying to take off LiDAR, because it can't be well integrated into the car chassis and therefore it is not aesthetic, and consumers are attached to the beauty of the car.

2.2.3 Our LiDAR

Our LiDAR come from the most famous LiDAR company : Velodyne. It is good to know that the best known self-driving cars (Toyota, Uber car, Renault-nissan, Uber car, Cruise, Navya) are all equipped by at least 3 velodyne LiDAR. Only Waymo is using its own LiDAR. Velodyne has 3 types of LiDAR, the cheaper will build 3D map with 16 plans, midrange will produce 3D map with 32 plans (HDL-32E), and last one and the most expensive will build 3D map with 64 plans but the sensor is quite big (HDL-64E). Find below the different models, and a comparative table.

Figure 5: The Velodyne models

Feature/Parameter	VLP-16	HDL-32E	HDL-64E
Laser/Detector Pairs	16	32	64
Range	100 m	100 m	120 m
Data Rate	300k points/s	700k points/s	1,300k points/s
Accuracy	±3.0 cm	±2.0 cm	±2.0 cm
Data Output	Distance, Calibrated Reflectivity		Distance, Intensity
Scan Frequency	5 Hz to 20 Hz	5 Hz to 20 Hz	5 Hz to 15 Hz
Horizontal FoV	360°		
Vertical FoV	30° (-15° to +15°)	40° (-30° to +10°)	26.8° (-24.8° to +2°)
Azimuth Resolution, 5 to 20 Hz	0.10° to 0.40°	0.08° to 0.35°	0.08° to 0.35°
Vertical Sampling Resolution (spacing between laser/detector pairs)	2.0°	1.3°	0.4°
Laser Safety	Class 1, Eye Safe		
IR Wavelength	905 nm		
Power Consumption	8 W	12 W	60 W
Operating Temperature	-10°C to +60°C	-10°C to +60°C	-10°C to +50°C
Mass	0.83 kg	1.00 kg	15.00 kg

Figure 6: Comparison of the Velodyne models

2.3 ROS

Figure 7: ROS.org logo

2.3.1 ROS Definition

As the name suggests, ROS (Robot Operating System) is an operating system for robots. As well as operating systems for PCs, servers or standalone devices, ROS is a complete operating system for service robotics [1]. ROS provides services close to an operating system (hardware abstraction, management of competition, processes ...) but also high-level functionalities (asynchronous calls, synchronized calls, centralized data database, system setting of the robot ...).

2.3.2 ROS purpose

Before the robotic OS, each robot designer, each researcher in robotics spent a significant amount of time to physically design his robot and the embedded software associated. It required skills in mechanics, electronics and embedded programming. Generally, the programs conceived corresponded more to embedded programming, close to electronics, than to robotics, as we can meet it today. The reuse of the programs was not easy because strongly related to the underlying material. The main idea of a robotic OS is to avoid reinventing the wheel each time and to offer standardized features that disregard hardware, just like a classic PC OS. Another advantage of the robotic OS like ROS is to bring together know-how from different disciplines. Indeed, to design and program a robot is:

- Manage the material by writing the drivers
- Manage memory and processes
- Manage competition and data fusion
- Propose abstract reasoning algorithms that rely heavily on artificial intelligence

Robotics therefore requires very different skills, skills usually beyond the reach of a single person.

2.3.3 ROS history

There are many robotic frameworks made for specific purposes, for prototyping purposes. ROS was wanted more general even though its designers think that it is not the ultimate OS capable of doing everything. ROS is developed and maintained by a California company, Willow Garage, founded in 2006 by Scott Hassan, one of Google's earliest employees who has been involved in the development of search engine technology [2]. The CEO of Willow Garage is Steeve Cousin, a former IBM.

Figure 8: Willow Garage

Willow Garage develops both ROS software and hardware with their PR2 and TurtleBot robots. In all cases, what is produced is open source (BSD licenses). Their course of action is to accelerate research by providing solid hardware and software bases that are open source in order to see robots living with us, in our homes. It seems that Willow Garage wants to build the robotics community rather than robotics itself.

2.3.4 ROS general organization

The philosophy of ROS can be summed up in the following 5 main principles:

- **Peer to peer**
- **Based on tools (microkernel)**
- **Multi languages**
- **Lightweight**
- **Free and open source**

Peer to Peer: A sufficiently complex robot is composed of several computers or embedded boards connected by Ethernet as well as sometimes external computers to the robot for intensive computation tasks. A peer to peer architecture coupled to a buffer system (buffering) and a lookup system (a name service called master in ROS), allows each actor to interact live with another actor, synchronously or asynchronously depending needs.

Multi languages: ROS is neutral from a language point of view and can be programmed in different languages. The specification of ROS intervenes at the message level. Peer to peer connections are negotiated in XML-RPC that exists in a large number of languages. To support a new language, either we rewrap the C++ classes (which was done for the Octave client for example), or we write the classes to generate the messages. These messages are described in Interface Definition Language (IDL).

Based on tools: Rather than a monolithic runtime, ROS has adopted a microkernel architecture that uses a lot of small tools to build and run the various ROS components. When you browse the ROS tutorials, you learn to use several commands to manipulate nodes and messages. Each command is actually an executable. The advantage of this solution is that a problem on an executable does not affect the others, making the system more robust and scalable than a system based on a centralized runtime.

Lightweight: In order to fight against the development of algorithms more or less related to the robotic OS and therefore difficult to reuse later, the developers of ROS wish that the drivers and other algorithms are contained in independent executables. This ensures maximum reusability and especially keeps it small. This mechanism makes ROS easy to use, the complexity being in the libraries. This organization also facilitates unit testing. Finally, ROS uses code (drivers and algorithms) from other open source projects:

- Player / stage simulator
- Image Processing and Artificial Vision Library: OpenCV
- Planning Algorithm: OpenRave
- And much more

Free and open source: We have already explained the reasons for this choice before. Note however that the chosen architecture is consistent with this choice: ROS passes data through interprocess communication. As a result, the modules do not need to be linked in a single process, thus facilitating the use of the different licenses.

2.3.5 Languages to program ROS

ROS is language independent. Three main libraries have been defined for ROS which make it possible to program respectively ROS in Python, in Lisp or in C ++. In addition to these three libraries, two experimental libraries are proposed, which allow to program ROS in Java or Lua.

The ROS file system ROS resources are organized in a hierarchical structure on disk. Two important concepts stand out:

- **The package**: This is the main software organization unit of ROS. A package is a directory that contains the nodes (we'll see below a node), external libraries, data, configuration files, and an xml configuration file named manifest.xml.
- **The stack**: A stack is a collection of packages. It offers an aggregation of features such as navigation, localization ... A stack is a directory that contains the package directories and a configuration file named stack.xml.

In addition to these two very important notions, there is also the notion of distribution. A Distribution, as in Linux, is a set of versioned stacks.

For our project, we use ROS kinetic Kame, the best for ubuntu 16.04

Figure 9: Kinetic Kame

2.3.6 ROS basics

The basic principle of a robotic OS is to run in parallel a large number of executables that must be able to exchange information synchronously or asynchronously. For example, a robotic OS must interrogate at a defined frequency the sensors of the robot (ultrasonic or infrared distance sensor, pressure sensor, temperature sensor, gyroscope, accelerometer, cameras, microphones ...), retrieve this information, process it (do what we call data fusion), move them to processing algorithms (speech processing, artificial vision, localization and simultaneous mapping, ...) and finally control the engines back. This whole process is done continuously and in parallel. On the other hand, the robotic OS must manage the competition to ensure efficient access to the robot's resources. We describe below the concepts grouped in ROS under the name of "ROS Computation Graph" and which make it possible to reach these objectives. These are the concepts used by the running system while the "ROS FileSystem" described in the previous section corresponds to the static concepts.

Nodes

ROS answers all this problem thanks to simple basic notions. The first notion is the notion of node. In ROS, a node is an instance of an executable. A node can correspond to a sensor, a motor, a processing algorithm, a surveillance algorithm ... Each node that launches declares itself to the Master. Here we find the microkernel architecture where each resource is an independent node.

The master

The Master is a node declaration and registration service that allows nodes to know each other and exchange information. The Master includes a much used subpart which is the Parameter Server. This one as its name suggests is a kind of centralized database in which the nodes can store information and thus share global parameters.

The topics

The exchange of information is done either asynchronously via a topic or synchronously via a service. A topic is an information transport system based on the subscribe / publish system. One or more nodes will be able to publish information on a topic and one or more nodes will be able to read the information on that topic. The topic is somehow an asynchronous information bus. This notion of many-to-many asynchronous bus is essential in the case of a distributed system. The topic is typed, that is, the type of information that is published (the message) is always structured in the same way. Nodes send or receive messages on topics.

The messages

A message is a composite data structure. A message is composed of a combination of primitive types (character strings, Boolean, integer, floating ...) and message (the message is a recursive structure). For example a node representing a servomotor of the robot, will certainly publish its state on a topic (according to what you have programmed) with a message containing for example an integer representing the position of the engine, a float representing its temperature, another floating representing its speed... The description of the messages is stored in package_name / msg / myMessageType.msg. This file describes the structure of the messages.

Services

The topic is an asynchronous communication mode that allows many-to-many communication. The service, on the other hand, responds to another need, that of synchronous communication between two nodes. The service description is stored in package_name / srv / myServiceType.srv. This file describes the data structures of queries and responses.

Bags

"Bags" are formats for storing and replaying messages exchanged. This system makes it possible to collect, for example, the data measured by the sensors and then replay them as many times as desired in order to make simulation on real data. This system is also very useful for debugging a system a posteriori.

Rviz

Rviz (ROS visualization) is a 3D visualizer for ROS. You can display infrared distances measurements, sonar data, and much more.

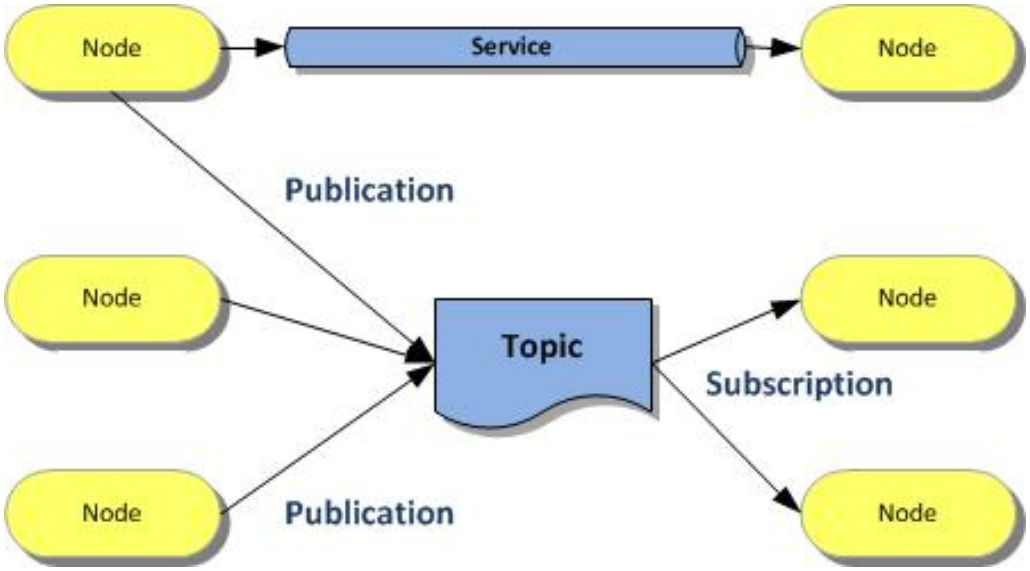

Figure 10: ROS overview

2.3.7 ROS for Laser Odometry And Mapping (LOAM)

- loam velodyne.launch will :
  - start the master (allow the nodes to find each other, exchange messages and use services)
  - start the nodes (processes) written in cpp (or py) and compiled using gcc + rviz (a node) (exemple for a robot : 1 node laser range-finder, 1 node controls the wheel motors, 1 node performs localization, 1 node performs path planning, 1 Node provides a graphical view of the system ...)
- the node, after reporting their registrations to the master, are able to speak directly together by sending messages (request or response) to a subscribed topic (as the services are stored in the master), it uses TCPROS (based on TCP/IP socket)

2.4 SLAM algorithm : LOAM

SLAM (for Simultaneous Localization And Mapping) was originally introduced by Hugh Durrant-Whyte and John J. Leonard [3]. Its aim is to estimate the position of a mobile robot and to build the map of its environment during the navigation of the robot. We can imagine easily the complexity of this issue : How can the robot know its position without a map ? And how can the robot build the map without its position ? It is a chicken-and-egg situation. To resolve the problem SLAM consists of multiple parts : Landmark extraction, data association, state estimation, state update and landmark update. SLAM is very important in robotics because we can't just trust nether the wheel odometry because its error is cumulative nor the IMU (Inertial Measurement Unit) due to its drift. There are many algorithms to solve each of those parts which I just spoke about few lines above. In our case, we will focused on the LOAM algorithm which mean Localization Odometry And Mapping.

LOAM was introduced by Ji Zhang and Sanjiv Singh [4] in 2014. They propose in their paper a real-time method for odometry and mapping using range measurements from a 2-axis lidar moving in 6-DOF (Degree Of Freedom). Odometry is a technique for estimating the position of a moving vehicle. The two first advantages of their methode is that it achieves both low-drift and low-computational complexity without the need for high accuracy ranging or inertial measurements. It sounds that we could reduce the high price of self-driving car which are currently between 300000 and 400000 dollars. The LOAM is divided in two part. One will perform odometry (called Lidar Odometry) at a high frequency but low fidelity to estimate velocity of the lidar. Another algorithm called Lidar Mapping will run at a frequency of an order of magnitude lower for fine matching and registration of the point cloud. Combination of the two algorithms allows the method to map in real-time.

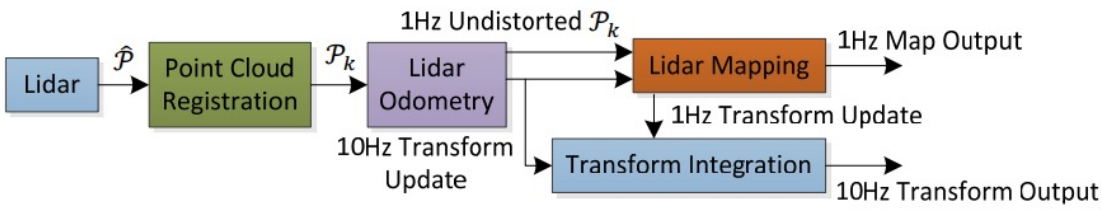

Figure 11: LOAM software system overview

2.4.1 Lidar Odometry

The first step of this algorithm will have to extract the landmarks points, called "features point" by the writer. There is two kinds of landmarks points : the edge point and the planar point. They are selected on sharp edges and planar surface patches. Based on their coordinates, referred by the Lidar, a value is computed for each point by comparing their coordinates to all the other points in the same scan. One scan is one sweep of the Lidar (360°). The point with the minimum computed value will be the planar point landmark, and the one with the maximum computed value the edge point landmark. Below you can find in red the extracted planar point, and in yellow the edges ones.

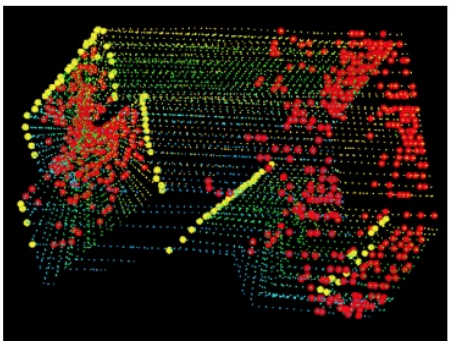

Figure 12: Exemple of extracted landmarks points

The second step of the algorithm has to find feature point correspondence between the sweep t and the sweep t-1. Thus, the goal is to find an edge lines in the scan t as the correspondence for an edge landmark point(a), and a planar surface as the correspondence for a planar landmark point (b). In both (a) and (b), j is the closest point to the feature point, found in the scan t-1. The orange colored lines represent the same scan of j, and the blue colored lines are the two consecutive scans. To find the edge line correspondence in (a), we find another point, l, on the blue colored lines, thus the edge line is represented as (j, l). To find the planar patch correspondence in (b), we find another two points, l and m, on the orange and blue colored lines, respectively, thus the planar surface is represented as (j, l ,m). You can find below the figure that represent this method.

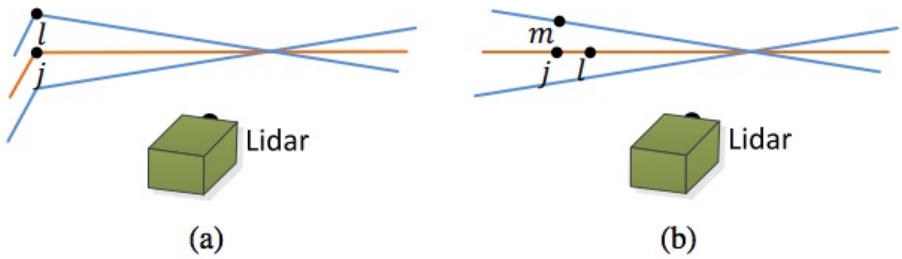

Figure 13: Feature point correspondence

The last step of odometry algorithm has to estimate the motion of the lidar thanks to the previous steps in order to correct the distortion. To do that it uses maths that go beyond our areas of expertise.

2.4.2 Lidar Mapping

The lidar mapping algorithm is called only one per sweep. It gets as parameters the undistorted points of the sweep computed by the odometry algorithm. The algorithm has to register the point cloud to create a map as we can see in the figure below.

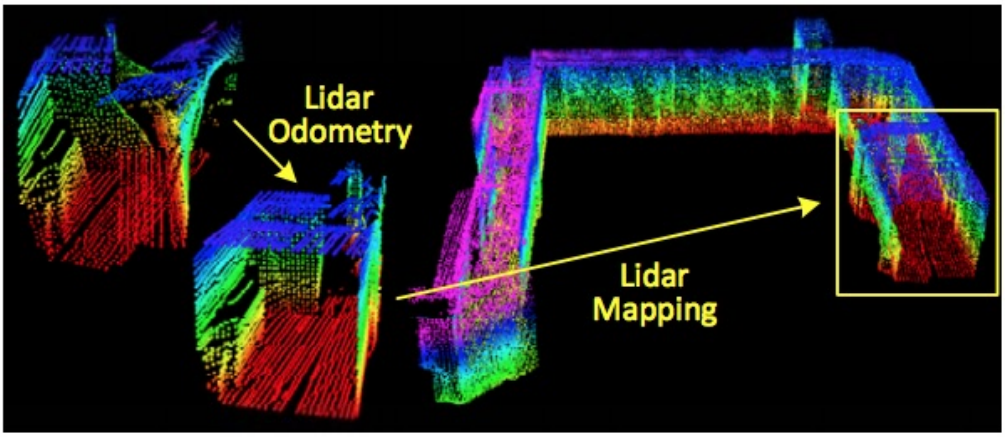

Figure 14: Mapping algorithm

To conclude this part, as we divide one big algorithm in two algorithms before combining them, it can be solved in real-time easily.

3 Results

3.1 Setting Up the environment

The first step to achieve before codding is to getting a proper environment. Thus, we followed a lot of tutorials in order to get the best environment we could. And lot of them didn't work or were not well explained. That's why we decided to do our own tutorial, and we would like to share it with our readers. As we work on Qt each time we are coding in c++, we decided to maintain the tutorial only for Qt. Tha advantage of using Qt is firstly its powerful development kit management and its debugging mode. But if you are not an IDE fan, feel free to do it by using the command lines and by adding all the nodes you need in the ".launch" used by ROS.

Figure 15: Qt logo

3.1.1 How to build and run your catkin workspace

- First off all : Don't install Qt for the moment !
- Follow this tutorial first : https://ros-industrial.github.io/ros_qtc_plugin/_source/How-to-Install-Users.html
- Open Qt, create new project by using "other project" and selecting "ros workspace" and your dedicated workspace
- manage your compilation and run tool chain (click on projects on the left)
- in run, you may have :
  - 2 launch step :
    - one with package loam_velodyne and loam_velodyne launch as target (will launch all the nodes needed) (note that in our last pushes we included all nodes needed in our ".launch", so only the this step is needed)
    - the other velodyne_pointcloud and 32e_points.launch (will convert the lidar data into PointcloudMessage)
  - a run step :
    - with package rosbag target play and argument your bag file

3.1.2 How to debug

To debug, only one object (one node) could be debug :

- Don't forget to allow ptrace ! by following these instructions :
  - Open file: sudo gedit /etc/rc.local
  - Add this line before the exit 0 line: echo 0 | tee /proc/sys/kernel/yama/ptrace_scope
- Insert your breakpoint
- Menu Bar > Debug > Start Debugging > Attach to Unstarted Application...
- Browse to the executable (devel/lib/loamvelodyne/multiScanRegistration) by exemple
- select start watching
- run your project (Ctrl + R)
- depending on where the breakpoints it should be stop when it reaches one

If you want to have an overview of your ROS nodes and topics we suggest you to install and use the rqt plug-in that provides some awesome commands line. It is a Qt-based framework for GUI development for ROS.

By example the command line " rqt_graph " will show us interactions between the nodes on the ros system

- The rectangles in the the window show the topics currently available on the system.
- The ovals are ROS nodes. Arrows leaving the node indicate the topics the node publishes, and arrows entering the node indicate the topics the node subscribes to.

3.2 Entire map without optimization

After having setting up the environment we could start to play with the given algorithms and to play a little bit with the ROS system which was not so easy to understand. A command that helped us a lot to understand the sequence of the program is : *rqt_graph* We could see the given result below (Figure 16) and we could understand that circles represent active nodes and squares represent active topics. Moreover the diagram provide the sequence of nodes and topics used by odometry, mapping and all that is used by the different algorithms. to see the final map, we understood that Rviz had to subscribe to the topic: **laser_cloud_surround**

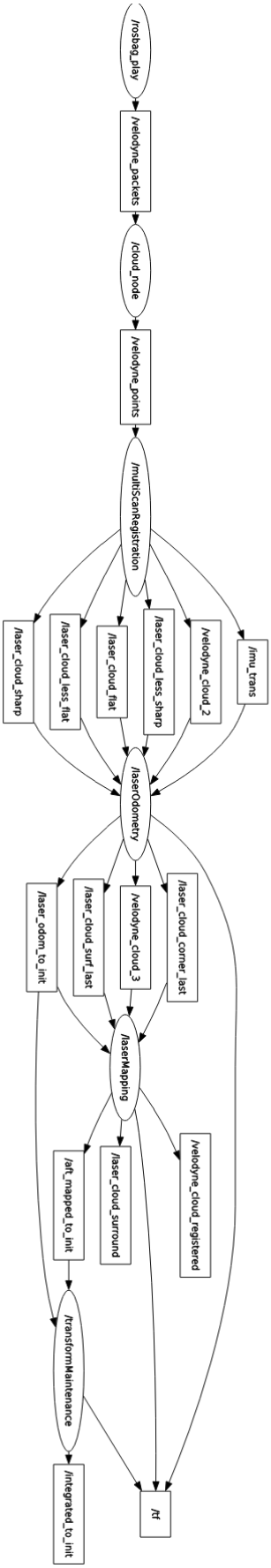

Figure 16: Rqt Graph not optimized

We quickly understood that the LOAM algorithm cleared the map in order to keep the memory safe. After studying a lot the algorithm we find where this last one clear the map. In addition to this, it reduces the resolution of the farthest points (laser_pointcloud_surround) by using the down sampling technique, and moreover it removes the color of these farthest points. We would like then to keep the entire map even if the memory will not support. That's what we did by removing the lines (see below) which cleared the map in the LaserMapping.cpp before publishing the results (function publishResult).

```
_laserCloudSurround->clear();
```

Then we added the color and replaced the down sampling cloud by the original cloud (see code line below) in order to have the best quality of the map.

```
publishCloudMsg(_pubLaserCloudSurround, *_laserCloudSurround,
                _timeLaserOdometry, "/camera_init");
```

This experiment allows us to see the worst case of the memory state. We got the result we can find below.

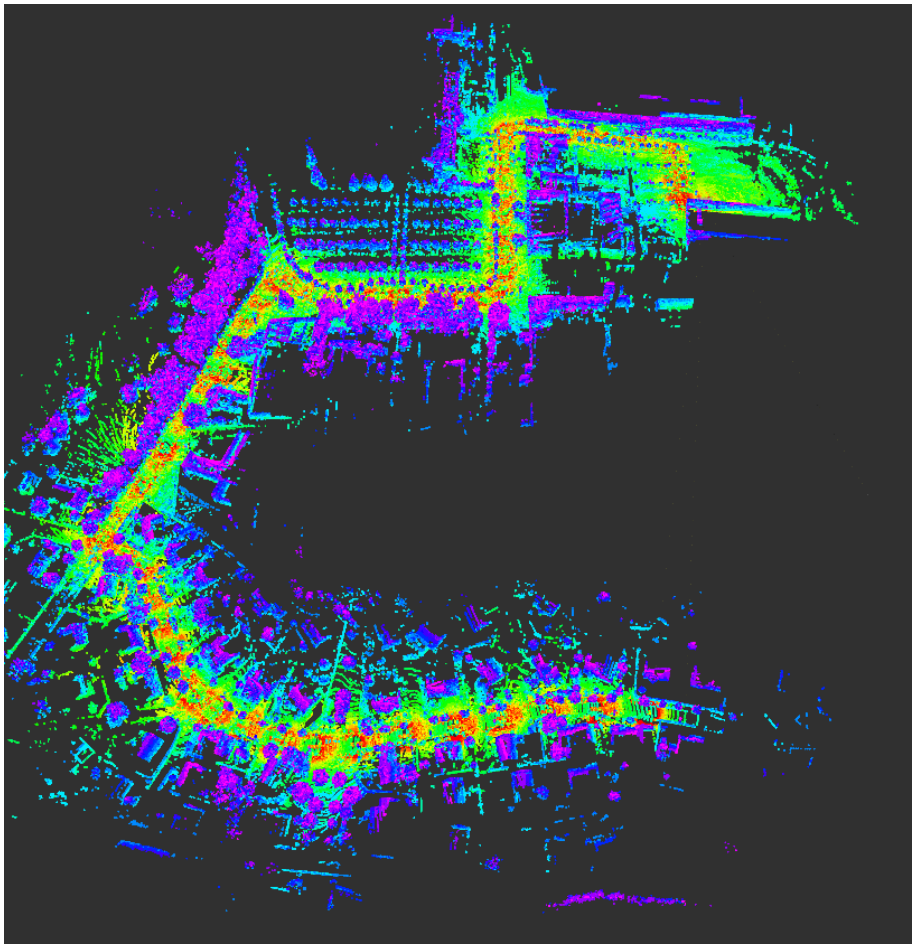

Figure 17: Entire map not optimized

However, as we could imagine the ram and the cpu will not well support these modifications. We checked that on the Ubuntu system monitor and see a constant progression of the ram until a threshold where the ram is a little bit released in order to survive. The cpu is following the same policy, and this could be explain by the number of computation it has to do with the entire map. You can find below our experience on the monitor system with a computer equipped by an Intel i7 and 8 GB of RAM. In this case, the main problem is RAM that becomes almost full. The CPU is really busy but is not at the maximum of these capabilities.

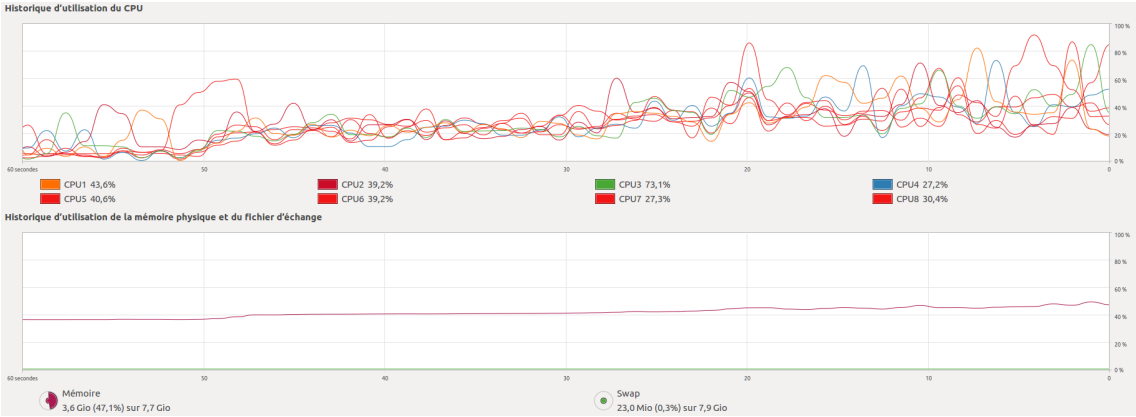

Figure 18: Begining of the resources usage during LOAM processing by keeping the entire map

Figure 19: End of the resources usage during LOAM processing by keeping the entire map

3.3 With Octomap

To reduce the use of RAM, we decided to use the Octomap library, this library allows us to choose the resolution of the map and thus reduce the number of points to display. Reduce the number of points to keep in memory also reduce the RAM used.

3.3.1 About Octomap

- **Full 3D model :** [5] The map is able to model arbitrary environments without prior assumptions about it. The representation models occupied areas as well as free space. Unknown areas of the environment are implicitly encoded in the map. While the distinction between free and occupied space is essential for safe robot navigation, information about unknown areas is important, e.g., for autonomous exploration of an environment.

- **Updatable :** It is possible to add new information or sensor readings at any time. Modeling and updating is done in a probabilistic fashion. This accounts for sensor noise or measurements which result from dynamic changes in the environment, e.g., because of dynamic objects. Furthermore, multiple robots are able to contribute to the same map and a previously recorded map is extendable when new areas are explored.

- **Flexible :** The extent of the map does not have to be known in advance. Instead, the map is dynamically expanded as needed. The map is multi-resolution so that, for instance, a high-level planner is able to use a coarse map, while a local planner may operate using a fine resolution. This also allows for efficient visualizations which scale from coarse overviews to detailed close-up views.

- **Compact :** The map is stored efficiently, both in memory and on disk. It is possible to generate compressed files for later usage or convenient exchange between robots even under bandwidth constraints.

3.3.2 Octomap integration

After watching reading lot of faq and papers on the octomap server, we understood that the octomap node need either the fixed frame (which is in our case the odometry) as the point cloud to integrate in order to build the entire map. Camera_init is a fixed frame fixed on the odometry, that's why we linked it with the frame_id parameter requested by octomap server.

```
<param name="frame_id" type="string" value="camera_init" />
```

The data to integrate was the last point cloud computed by the loam before the mapping algorithm. That's why, thanks to rqt_graph, we remapped the requested "cloud_in" to the velodyne_cloud_3 topic.

```
<remap from="cloud_in" to="/velodyne_cloud_3"/>
```

We get then the desired result except that the map was cut of when it arrived at a certain distance of the origin as we can see below.

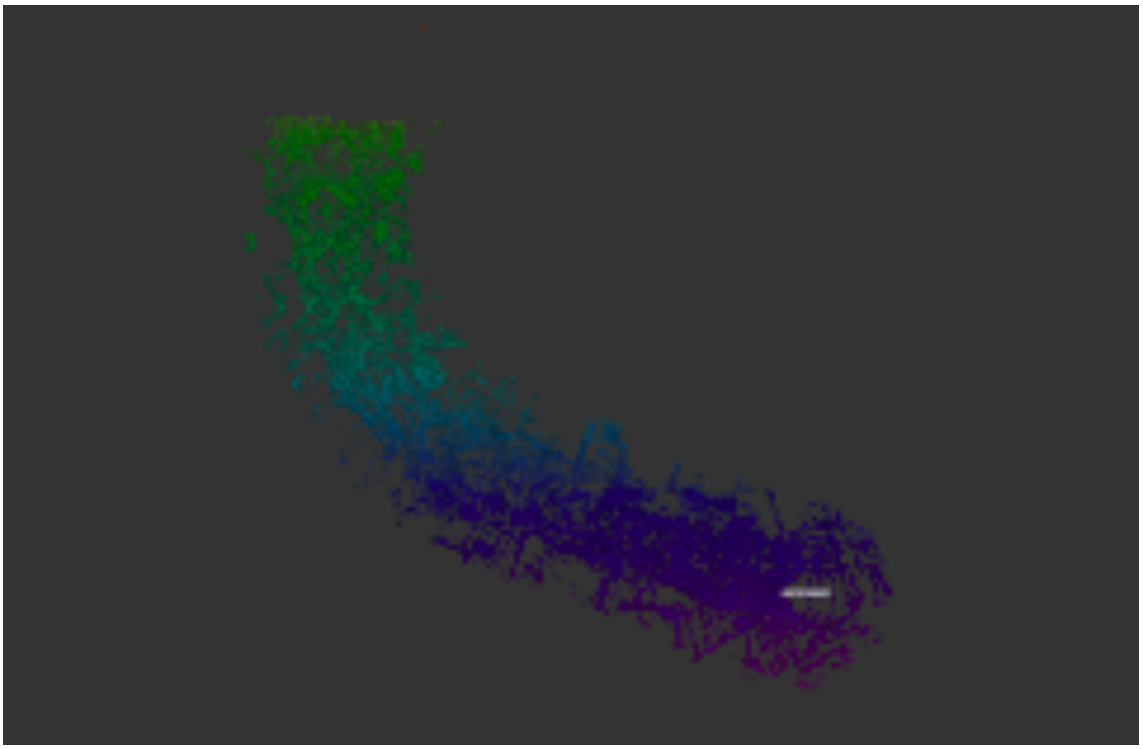

Figure 20: Limit of map height

Thanks to the color which should be from blue for a lower cube altitude to red at the higher altitude (the altitude is defined by the z axis), we realized that our cubes are going higher and higher and could make us believe that the car is taking off. But by looking at it more closely, we realize that our pointclound is just misdirected as we can see below. Note that the grid is included in the x,y plan and should be then collinear to the road or even mingled.

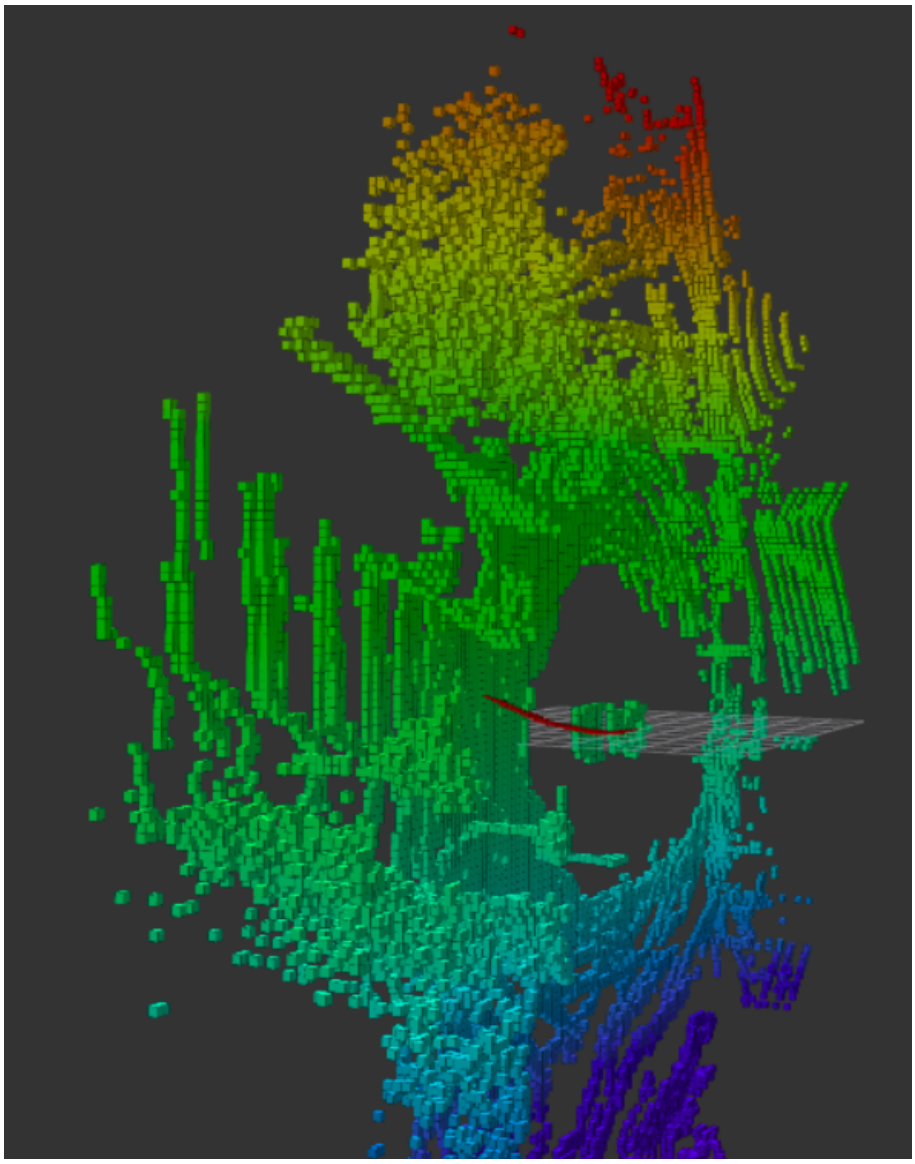

Figure 21: Octomap with wrong orientation

Thus, first of all we tried to avoid this limitation by searching on the net some help. However by looking on the parameters that we can give to the octomap server we found that their is a parameter that give the minimum and maximum height of occupied cells to be consider in the final map.

```
~occupancy_[min|max]_z (float, default: -/+ infinity)
```

However infinity doesn't exist in computer science, that's why we would like to see how it is coded, and we found that it uses the max and the min value of a double as we can see below.

```
m_maxTreeDepth(0),
m_pointcloudMinX(-std::numeric_limits<double>::max()),
m_pointcloudMaxX(std::numeric_limits<double>::max()),
m_pointcloudMinY(-std::numeric_limits<double>::max()),
m_pointcloudMaxY(std::numeric_limits<double>::max()),
m_pointcloudMinZ(-std::numeric_limits<double>::max()),
m_pointcloudMaxZ(std::numeric_limits<double>::max()),
m_occupancyMinZ(-std::numeric_limits<double>::max()),
m_occupancyMaxZ(std::numeric_limits<double>::max()),
m_minSizeX(0.0), m_minSizeY(0.0),
```

Figure 22: About Octomap Z axis issue

So we thought that it could explain the limitation in height that we had, however by looking the value of the max and min (see below) we were and still are not able to be entirely sure of this explanation.

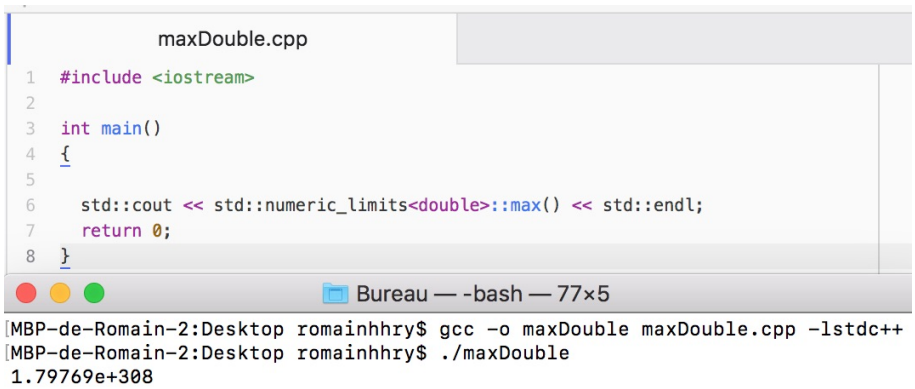

Figure 23: Max double value

So the goal was to rotate the pointcloud published in the point_cloud_3 topic before be given to the octomap server before they are used by the octomap server.
Thus, thanks to a rotation matrix around the x axis of $\pi/2$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\pi/2) & -sin(\pi/2) \\ 0 & sin(\pi/2) & cos(\pi/2) \end{pmatrix}$$

we tried to rotate the points computed in laser_odometry.cpp just before the are published in the topics. However, the computed odometry was not rotated, that induce a result that was not the one expected. Rather than rotating everything, we thought to rotate the pointclouds received by the multi_scan_registration.cpp. We realized then that they were all rotate willingly.

```
point.x = laserCloudIn[i].y;
point.y = laserCloudIn[i].z;
point.z = laserCloudIn[i].x;
```

We just let them as they should be and removed this rotation. Moreover, we had to affect some other lines of the function that were working for the rotated points.
Finally, the map was as expected, with the red on the top and the blue on the bottom, and mostly, it was not cut off again and displayed entirely.

3.3.3 Octomap resolution and range

With octomap you can adapt the resolution and the maximum distance of the map from the vehicle. That reduces the quantity of "points" which are in reallity cubes and then reduces the CPU usage. So decreasing resolution and/or the max range of the cumputed map induces a lost in quality but let to the CPU more time to be focused on other proccesses. Below we can find the code of the **octomap** node and the interesting paramaters.

```
<node pkg="octomap_server" type="octomap_server_node" name="octomap_server" output="screen">

        <!-- Resolution in meter for the map when starting with an empty map.
        Otherwise the loaded file's resolution is used. -->
        <param name="resolution" value="0.7" />

        <!-- fixed map frame (set to 'map' if SLAM or localization running!) -->
        <param name="frame_id" type="string" value="camera_init" />

        <!-- Maximum range in meter for inserting point cloud data when dynamically building a map.
        Limiting the range to something useful (e.g. 5m) prevents spurious erroneous points far away from the robot. -->
        <param name="sensor_model/max_range" value="35" />

        <!-- Whether topics are published latched or only once per change.
        For maximum performance when building a map (with frequent updates), set to false.
        When set to true, on every map change all topics and visualizations will be created. -->
        <param name="latch" value="false" />

        <!-- Whether visualization should encode height with different colors -->
        <param name="height_map" value ="true" />

        <!-- data source to integrate (PointCloud2) I would integrat velodyne_cloud_registered -->

        <remap from="cloud_in" to="/velodyne_cloud_3"/>
</node>
```

Figure 24: Octomap launchfile : node code

Thus, the differences between a good resolution and a bad one can be seen below. Note that their are we used for the both the same capture and we can see that the computed maps are quit different.(Figure 24 and 25).

In the below we put as parameter:

- Resolution = 0.2
- Max_range = 10

So a pretty good resolution with a small map.

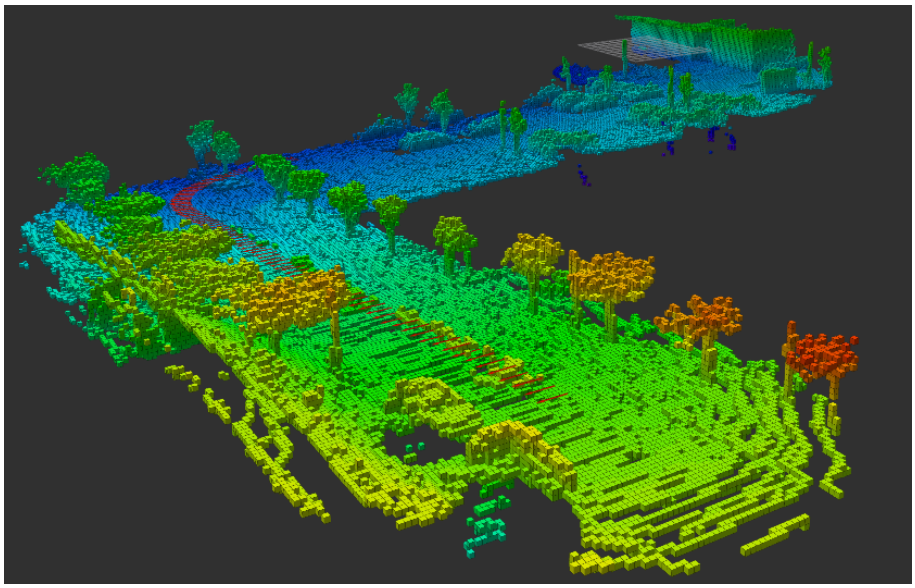

Figure 25: Map with small range and high resolution

In the other figure below :

- Resolution = 0.7
- Max_range = 35

So a less good resolution but a bigger map.

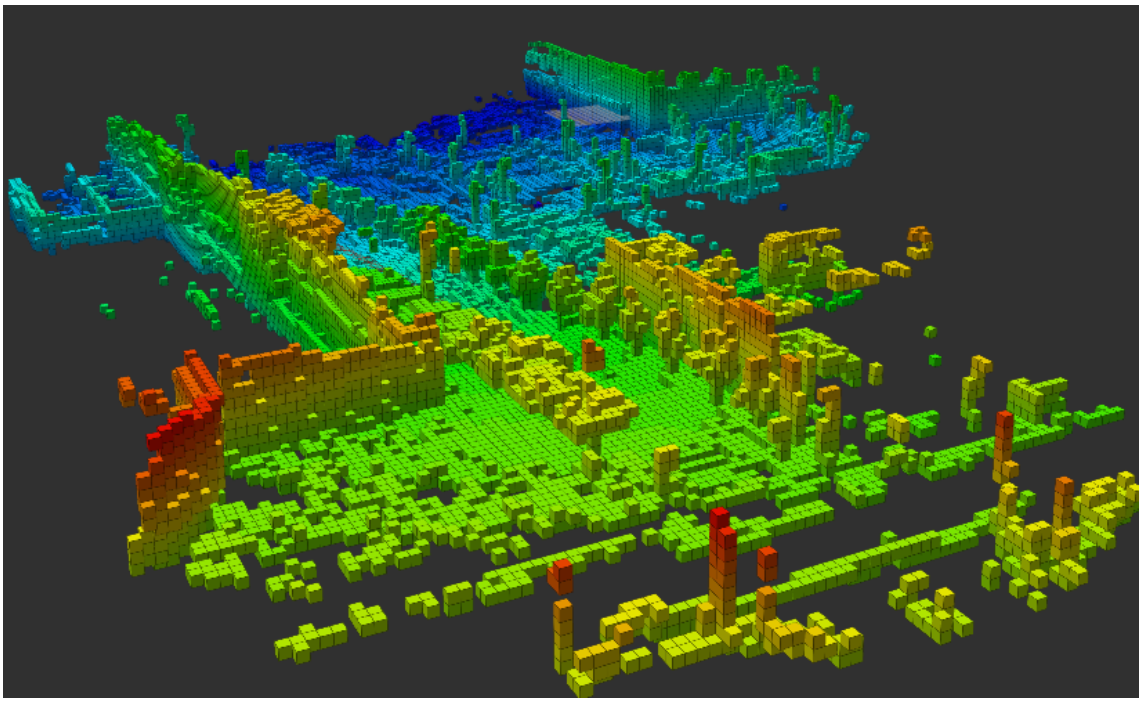

Figure 26: Map with big range and low resolution

On the first figure we can easily discern trees whereas on the second one we can't because the resolution is to bad. On the other hand figure one show a smaller map, thus the second figure shows more informations thanks to a wider map.

Thanks to octomap we were finally able to display the entire map by reducing the use of RAM. Below is the result of an entire map displayed with octomap.

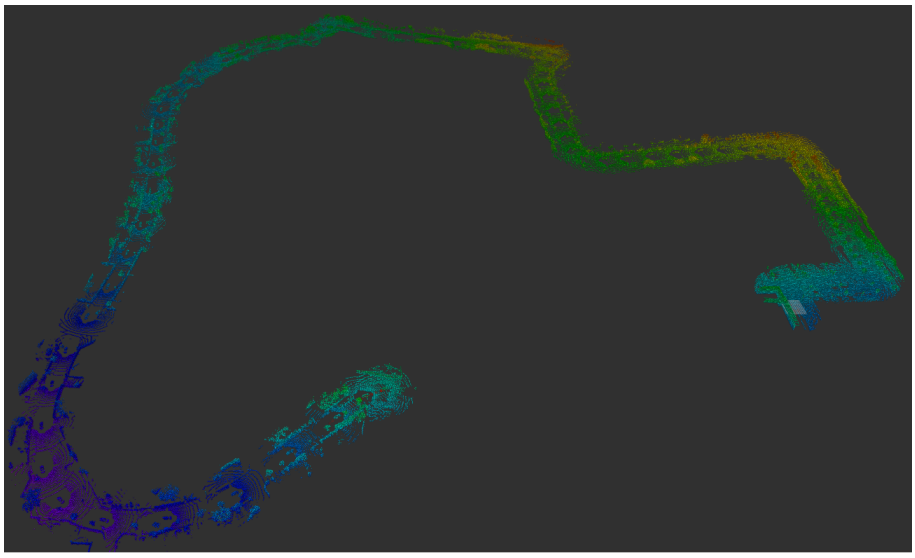

Figure 27: Octomap result of the entire 3D map

It is also possible to display it in 2D, by adding the corresponding toppic in Rviz.

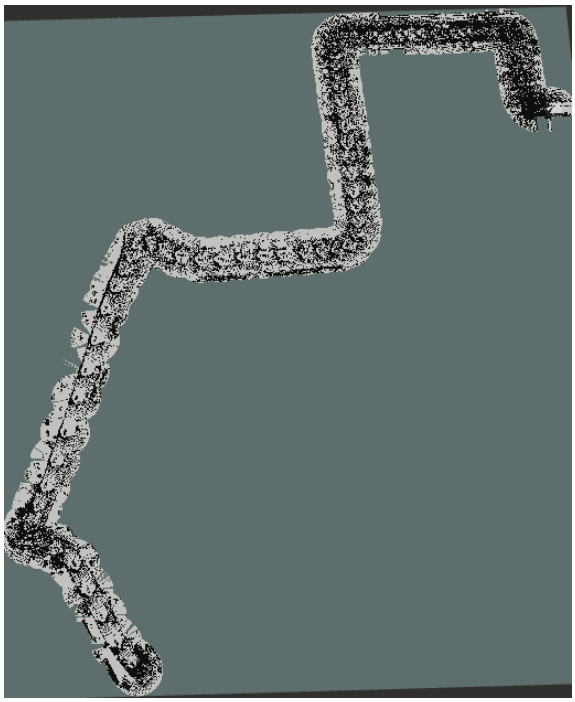

Figure 28: Octomap result of a 2D map

Unfortunately with octomap we only moved the problem of RAM to the CPU. Indeed, the memory present in a classical PC is now enough, but the CPU reached its maximum capacity. We can see with the figures below that the memory remains sufficient but the CPU reaches its limit

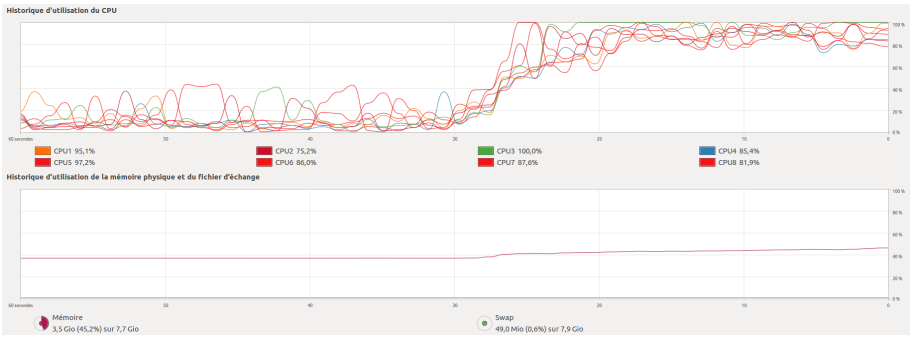

Figure 29: Begining of the resources usage during LOAM processing by using Octomap

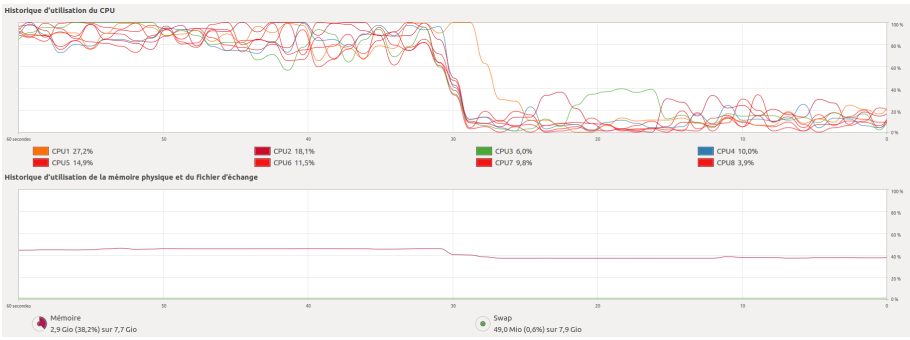

Figure 30: End of the resources usage during LOAM processing by using Octomap

3.4 2D LIDAR

The last step of our work was to display the data captured by two LiDAR. As the LOAM algorithm is waiting for a unique topic with pointcloud inside, we thought to remap the two resulting topics of each lidar to the entry point of the LOAM and that worked properly to see the surround pointclouds. But to get the entire map thanks to octomap server it was not enough due to a bad computation of the odometry. As this semester was very short and as we had to dome some other work, we have decided to not go further on this step even if it was very interesting for us. We can find below the result of computed surround of the two Lidar with rviz after having understand and succeed to integrate it into our project. We can see that we see more details than with only one LiDAR, which was quit logic as get 1 400 000 points per second instead of 700 000.

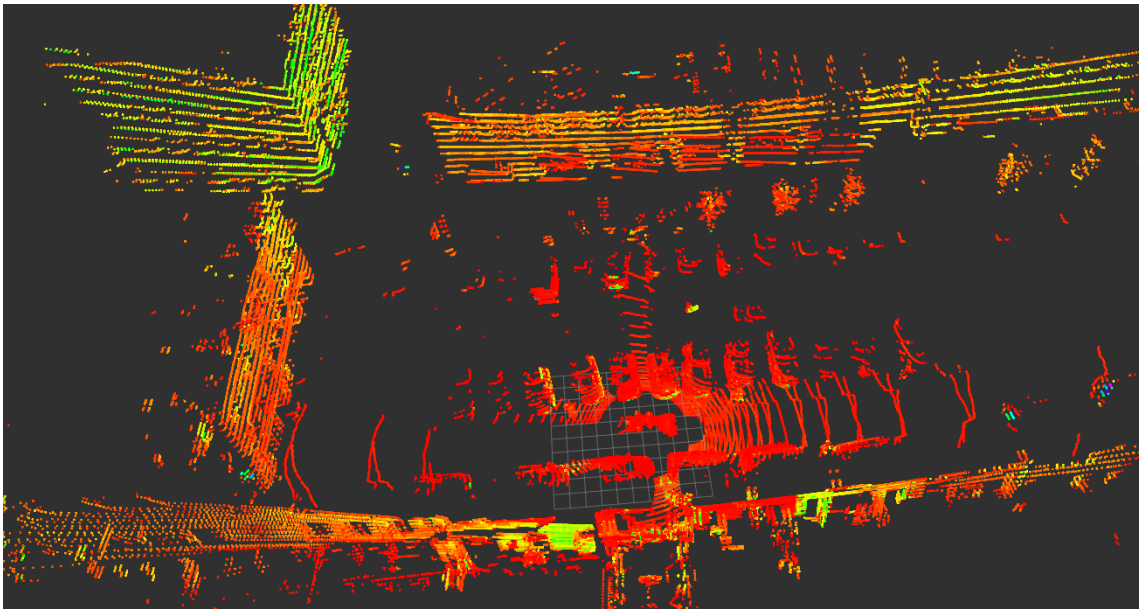

Figure 31: Rviz view with LOAM processing using two lidar

And here is the sequence of subjects and nodes. In this example we have not yet integrated octomap.

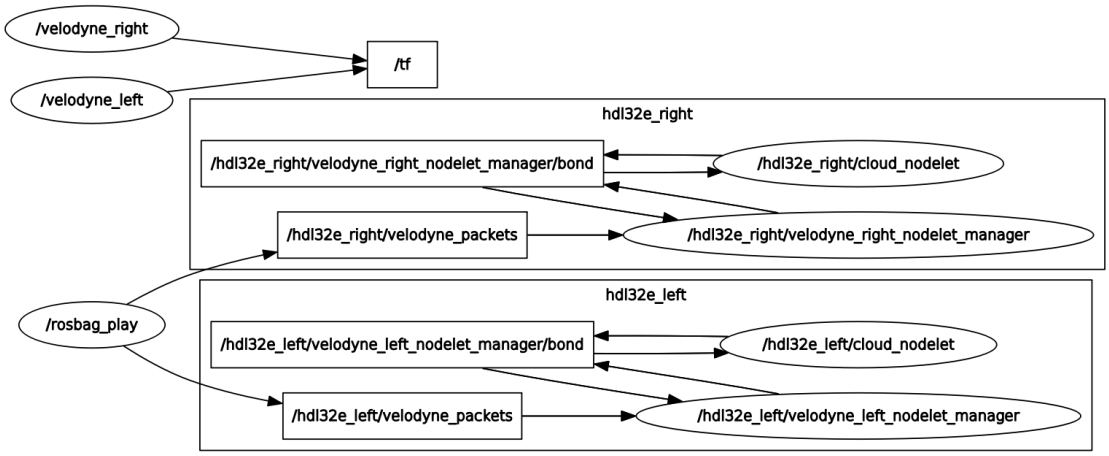

Figure 32: Rqt_graph with two lidar

We still added the scheme of operation to the next page even if octomap is not working.

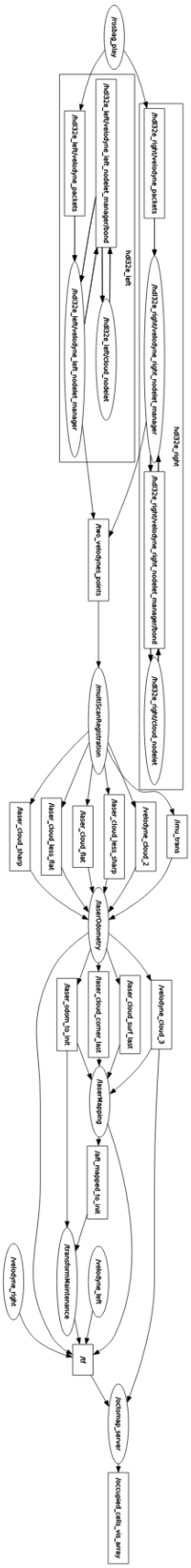

Figure 33: Rqt_graph for octomap with two lidar

4 Conclusion

For this project we first had to understand the operation of ROS and how the loam worked. We started with a lot of documentation and then we installed the tools to take control of the current project. Finally a lot of tests were done to take control of the project and we managed to integrate octomap.

ROS is a very powerful tool. Although difficult to handle, once the basics are understood it opens up a lot of possibilities. We worked a lot together and it was really usefull to understand how the project base works because we could help each other.

We encountered a lot of problems in this project, but it gives us a glimpse of what lies ahead in business.

Git has been very useful for us to develop in parallel on the different project parts and to oversee its progression over time. We proceeded by validating each of the previously defined steps by the Professor Zhi Yan which was very helpful.

We enjoyed to learn to use a tool like ROS because we are both very interested in robotics, automation and more specifically on autonomous cars. Indeed, ROS is so much used in these areas and it will be a very good skill to have to be competitive in the labor market. Although Romain will start his work (in French called "CDI") with Bosch on the Jaguar self-driving car early July.

So to go further with this work, the main point should be to make the LOAM algorithm working directly with octree instead of pointcloud, that will significantly reduce the RAM and the CPU usage because it will avoid useless computation and it should avoid the usage of octomap server to transform pointcloud to octree. Moreover, the next step should be to make the 2 LiDAR working with the octomap and to make some quantitative analysis on these results.

References

[1] ROS.org. Ros/concepts, 2014.

[2] Génération Robots. Ros – robot operating system, 2016.

[3] J.J. Leonard and H.F. Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE*, 1991.

[4] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. *Robotics: Science and Systems Conference (RSS)*, 2014.

[5] Armin Hornung · Kai M. Wurm · Maren Bennewitz · Cyrill Stachniss · Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 2013.