Bertrand Gallet
1st Year Master CSE
galletb@minatec.inpg.fr

Internship Report :

2nd year of engineering school

9 weeks
9/7/2011 to 7/9/2012

Subject :

Evaluate the use of a Roomba robot as a mobile node in a wireless sensor platform

Host company :

INRIA GRENOBLE - RHÔNE-ALPES
Inovallée
655, avenue de l'Europe
38334 Montbonnot
France
Tél : 04 76 61 54 04
http://www.inria.fr/centre/grenoble-rhone-alpes

Supervised by :
Nicolas Turro
nicolas.turro@inrialpes.fr

Preface :

The 2nd year internship of the engineering school allow the students to be introduced to new companies and more importantly to discover direct applications of the elements learned during courses. This report explains the work I have done during the period of internship and what I have learned from it.

Acknowledgments :

I would like to thank the entire SED team for welcoming me the way they have. I would also like to thank my supervisor Nicolas for sharing his knowledge with me, getting me back on the right path as I proceeded through my research and guiding me through my report and presentation. Many thanks also goes Jean-François for always being available and ready to help, to the HiKoB members for sharing their passion for robotics and to Léo and Liam for sharing great moments during the period of internship.

Table of Contents

I.Internship Environment -- 1

I.1. Presentation of INRIA
I.2. Presentation of SED
I.3. my working place

II. The subject of the internship --- 2

III. The platform and software --- 3

III.1 Devices -- 3
III.2 Software --- 5
III.2.1 ROS
III.2.2 Navigation

IV. My work --- 7

IV.1 Presentation -- 7
IV.2 Setting up the robot -- 8
IV.3 Autonomous charge of batteries -- 8
IV.4 Going from point A to B using odometry ------------------------------------ 9
IV.5 Integrating a gyrometer --- 10
IV.6 Going from point A to B using navigation in a map -------------------- 11
IV.6.1 creation of a map
IV.6.2 programming a scenario using navigation

V. Results -- 13

V.1 Using ROS --- 13
V.2 Testing the “dock” function --- 13
V.3 Evaluating the precision of odometry ------------------------------------- 14
V.4 Testing navigation as a solution -- 15
V.5 Conclusion -- 16

VI. Personal assessment -- 17
VII. Summary --- 18
VIII. Glossary -- 19
IX. Bibliography -- 20
IX. Appendices --- 20

I Internship Environment

I.1 Presentation of INRIA

INRIA is a public national research center specialized in applied Informatics. It conducts advanced research in computer science, telecommunication,multimedia, robotics, and signal processing. There are 8 INRIA research centers in France. The Rhône-Alpes research unit was created in 1992 and has more than 400 employees, has partnerships with other research centers and collaborates strongly with local universities (UJF and INPG).
INRIA also seeded some start-ups such as HiKoB or Blue Eye Video.

I.2 Presentation of SED

Each INRIA has a service called SED providing support for experimentation to the research teams. This service is composed of 10 engineers and technicians, who study the feasibility of the researchers' ideas and help them to build experimental platforms. They therefore conceive and develop devices in order to get as close as possible to the researchers' needs.

I.3 My working environment

I worked in the SED service and was at the heart of experiments since my desk was inside the laboratory where all the devices are kept and worked on. This was great as I could see all the current projects being developed. Nicolas Turro helped me with research and programming problems while Jean-François Cuniberto helped me with mechanical problems. Other students and engineers were also available for any questions I had.

During my internship I had a great sens of autonomy since the working hours were very flexible and the supervision of my research and work wasn't systematic. However a schedule of tasks was given to me at the beginning which oriented my work and fixed the amount of time I should spend on certain topics. This schedule can be seen in the Gantt diagram (Appendix).

II Subject of the internship

The Senslab project (http://www.senslab.info/) is composed of four wireless sensor platforms present in four different cities in France including Grenoble. Each one of these platforms is composed of 256 fixed nodes used to study network protocols through low power consumption sensors. Engineers and researchers experiment with these platforms by using a Web application that executes their program and the nodes they chose.
There is now a need to study the possibility of adding mobile nodes to the platform to study network protocols in a dynamic environment. The solution retained by the engineering team is to place the sensor on a mobile robot and to make the robot move inside the sensor network which will be placed in the ground and ceiling of the corridors of INRIA.

The robot must fulfill the following **requirements** to be used effectively as a mobile node:

- Start the robot from distance (through internet)
- The robot runs autonomously
- The robot must be able to go back and forth from point A to point B in the office corridors of INRIA and be aware of its environment (avoiding possible obstacles, crossing humans and other robots, …)
- Run for one hour without recharging its batteries. When they are low, it must go recharge and then continue its tasks.
- Retrieve information on the robot and the sensor from distance

The aim of the internship is to first investigate the chosen devices and software which I will present next to see what we are capable of doing and then find out if it is possible to meet the above requirements using our equipement.

III The platform and software

The hardware and software to be used for our platform were chosen before hand by the engineering team. These elements are described below as well as the reasons for choosing them.

III.1 Devices

The robot I worked on is based on the same structure as the Turtlebot (http://turtlebot.com/). This robot isn't commercialized in France nevertheless it was decided to copy this robot because the drivers to control it (ROS and the turtlebot stack) are open-source and compatible with the robot we chose.
I'll first present here the elements that our robot platform is equipped with.

Image 1: the turtlebot

Roomba 531:

- Low cost (370 €)
- Robust
- Possible to control the robot through serial commands.

Image 2: Roomba 531

Sensors :

- 2 bumpers up front
- 6 IR bumpers up front
- 4 cliff IR sensors
- Left, right and omni-directional IR data receivers
- Various buttons

Actuators :

- Left and right wheel motors
- Main and side brush motors and vacuum motor
- Rudimentary music playback
- Various lights

Portable Computer :

-Linux: Ubuntu with Xfce
-WiFi IEEE *802.11*
- Dual processor : Intel Centrino
This laptop is compact enough (12.1” screen) to fit on top of the Roomba and powerful enough to run several complexe applications at the same time. It is also equipped with two batteries providing a long autonomy.

Kinect :

Our main usage of the Kinect is the depth sensor made with an infrared laser projector and CMOS sensors. The Kinect was chosen because it is a low cost (140 €) and very popular device.
Properties of depth sensor :
 - range of 0.8 to 6 meters
 - angular view : 57 ° horizontally and 43 ° vertically
- depth image resolution of 640 x 480 and 11-bits per pixel.

Image 3: Microsoft Kinect

Depth sensors with better characteristics exist but they become rapidly very expensive. The properties of the Kinect's depth sensors are thought to be sufficient for our application.

Gyrometer :

The gyrometer used is the one integrated on the HiKoB FOX , it has a 3 axis angular rate measurement. The micro-controller present on the card was programmed to send back only the gyro's data using JTAG.
Sensitivity : 8.75 / 17.50 / 70 mdps/digit
Measurement range : 250, 500 or 2000 dps
Zero-rate level : 10, 15, 20 (depending on measurement range)

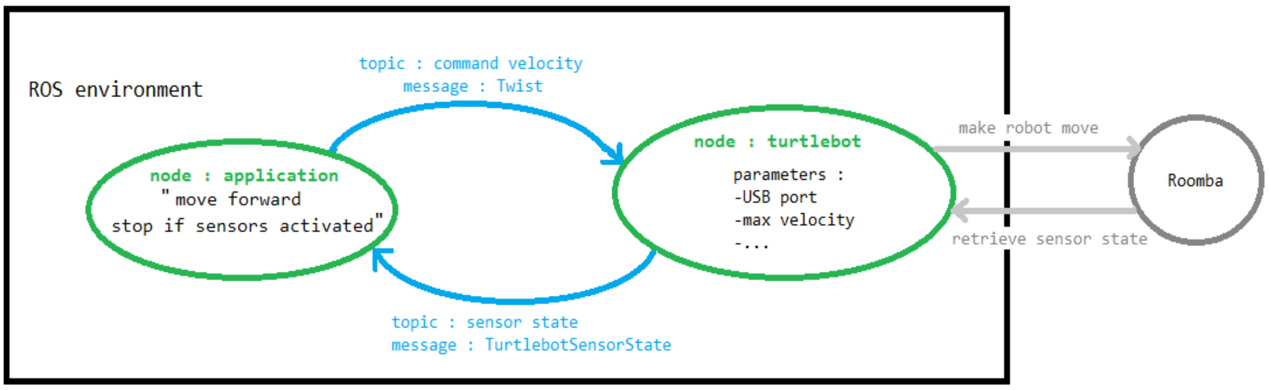
III.2 Software

III.2.1 ROS (Robot Operating System) :

http://www.ros.org/wiki/

Robot Operating System is a framework for robot software development created by Willow Garage. The platform aims all robotic systems providing device drivers, visualizers, libraries and much more. It is open-source and free to use.
ROS's goal is to help software developers achieve more capable robot applications quickly and easily. This is made possible in part thanks to its architecture.
The ROS environment is composed of four main elements :
- **Nodes** : every program running on ROS is called a node. Each one of them is registered in the master node.
- **Parameters** : each node has different parameters that can be adjusted.
- **Topic** : Data flow through canals that are called topics. Each topic has only one particular type of data defined by the message type. A node transmitting data on a topic is a Publisher while a node receiving data is a Subscriber.
- **Services** : One node can ask another to perform an action or to give back the value of a certain object. It is therefore a Question/Reply structure.

This structure makes ROS very powerful as it makes it easy for many programs to read the same data at the same time. Nodes can even be on different computers as long as they are connected to each other through a network.
More importantly, ROS makes it possible to run your application on different devices since the message types and the topics are always kept the same and are **standardized**.
ROS applications can be written either in Python or in C++.



Schematic 1: example of ROS architecture

In this case, the “application” node is publishing on the topic “command velocity” telling it to go forward and is subscribed to the “sensor state” topic to check if the sensors detect something. The “turtlebot” node is then in charge of executing these requests to the Roomba itself.

A stack (cluster of programs) called Turtlebot already exists in ROS and provides the basic necessary programs needed to control and run all our chosen devices together. The main topics and services I will be using can be found in *Appendix A*.

ROS also provides SLAM, AMCL and Navigation applications :

Simultaneous localization and mapping (SLAM) is a technique used in robotics to create or update a map in an unknown environment and at the same time keep track of the current location. The robot uses a depth sensor to observe fixed obstacles in the environment and then uses the data of the depth sensor combined with odometry to place the objects on the map being created. This can be done in 2D or 3D.

Adaptive Monte Carlo Localization (AMCL) is a Monte Carlo method to localize a robot in a given map. It works by finding out where the robot would need to be in order for its depth sensor to match with this map. It is based on a particle filter, each particle represents the estimated position and orientation of the robot in the map. The estimated position of the robot converges as the robot moves.

The **Navigation program** of ROS provides the necessary algorithm to move the robot from point A to point B inside a given 2D map. The map is in a gray scale file format : PGM (Portable Gray Map). Black pixels represent an obstacle, white represent free space in which the robot can navigate and gray is unknown space.
Navigation is then achieved by sending an estimated initial position followed by the position of the goal in the map.
A laser scan is obtained thanks to a ROS program (pointcloud to laserscan) that makes a 2D projection of the depth image of the Kinect. The laserscan is used to detect dynamic obstacles and to localize itself using the AMCL method.

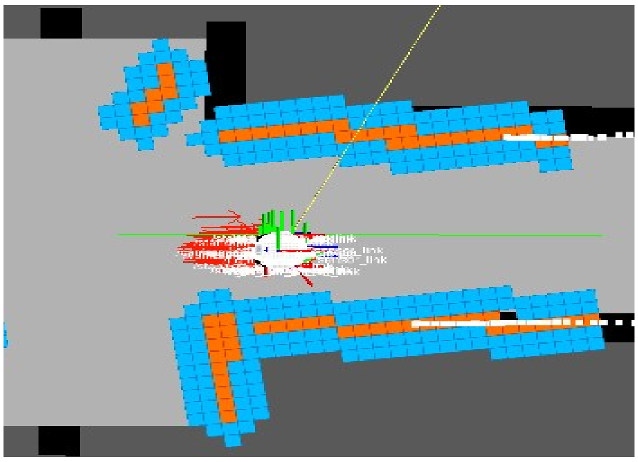


Image 4: Visualization of the different elements involved in running navigation

This image provides support to understand how the navigation stack works :
When a goal is sent to the stack, an initial global path from the current position to the goal is computed based on the provided static map. The robot then follows small portion by small portion this path. On each of these small portions, the “costmap” program enables the robot to avoid obstacles by continuously recalculating the best local path corresponding to the one with the lowest cost. Each obstacle has a cost that increases as you get closer to it. The blue pixels add low costs while the orange a greater cost. (*Appendix B : schematic of the programs necessary for navigation*).

IV My work :

IV.1 Presentation

The objective of my internship is to evaluate the chosen materials and software and determine if they can be adapted to satisfy the requirements of the Senslab platform.

The first part of my work consisted in a vast research on ROS to firstly learn how it works and then find out which programs already created can be used for our application. I also spent many hours getting familiar with the world of robotics and learning how to program in Python.

Secondly, with the help of Jean-François Cuniberto and Léo Statkus we made a platform in order for the robot to function with all the devices connected and mounted to it.

After having installed and configured the software, I worked on one important requirement which is charging the batteries of the Roomba and the PC.

Three different solutions were then retained and tested to make the robot go from point A to point B :

- using odometry
- adding an inertial measurement unit to improve localization
- Using navigation in a map of INRIA

I will then present the results and robustness of these solutions.

Learning to program in **Python** and make **ROS applications**

Discovering the world of **Robotics**

IV.2 Setting up the robot:

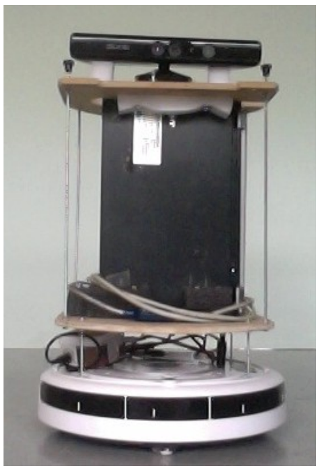


Image 5: our platform

This is the platform we came up with for the robot to carry along the portable computer, Kinect and gyro. The PC and the Roomba communicate using the interface called iRobot Roomba Serial Command Interface (manual). The Roomba's connector is a 7 pin Mini-Din and is connected to the PC through USB (the cable includes an adapter). The Kinect is connected to the PC through USB but also needs an external power source as the USB can't deliver enough power to run the Kinect. Therefore a 12V power source was made using a voltage converter which is connected to the batteries of the Roomba.

A desktop computer communicates with the PC using the ssh protocol. On both the computer I installed ROS with the following main stacks :
- turtlebot stack
- openni_camera
- navigation

I learned to used these programs following the ROS tutorial provided for most programs.

C. Autonomous Charge of the batteries :

A first requirement of the project is the autonomous charge of the batteries, I present here my investigations and solutions.

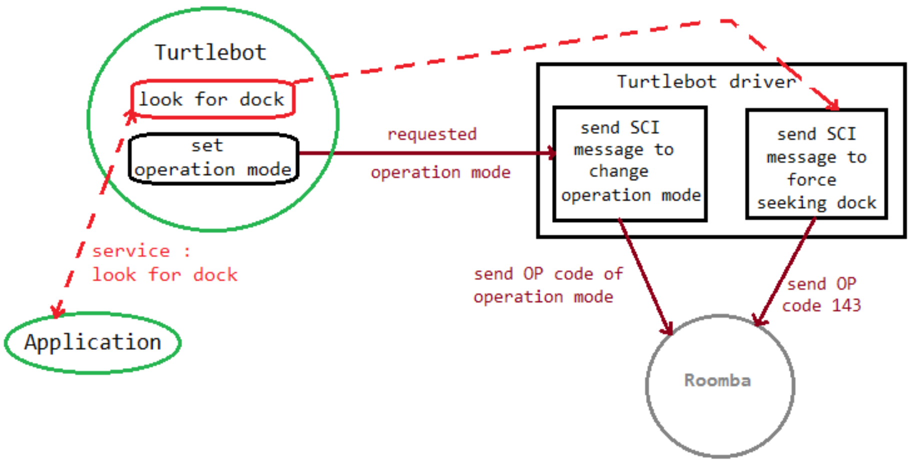
One of the built in functions of the Roomba is the “dock” function (which can be executed by pressing the “dock” button on the robot). Once called, the robot looks autonomously for the charging base and docks. This is achieved thanks to the continuous IR-signals that the base transmits.

This is the function we want to call to recharge the robot's battery.

Image 6: Roomba while docking

Contribution : creation of a service to call the “dock” function and modification of code to enable charge

No services existed to call this function in ROS however a function did exist at a low level.
Therefore the solution was to create a “ look for dock” function in the turtlebot node in the same way as the “set operation mode “ function is implemented.
There is now a service called “look_for_dock”, when called the robot seeks for the dock and returns “True” once it has docked.



Schematic 2: implementation of the “dock” function

Charge of the Roomba's batteries :
Problem : In order for the batteries to charge, the robot must be in passive operation mode. However when the robot's batteries are low and the robot is dock, the ROS program is made such that the program will reboot changing it to safe mode.
Solution : I modified this part of the code for the robot to stay passive while charging.

Charge of the laptop's battery :
Jean-François Cuniberto added to the robot's dock a charging mechanism for the PC. Therefore when the robot is charging, so is the computer.

D. Going from point A to point B using odometry :

After learning how to make the robot move at a desired velocity using ROS tutorials, I can now apply this to go from a point A to a point B.
To determine if the robot has reached the designated points, I will be using the odometry of the robot. As the wheels turn, a sensor keeps track of how much they have turned and therefore can estimate the position x and y of the robot.
The Turtlebot stack has the topic called /odom with messages of the type : nav_msgs/Odometry which contain the estimated position.

To make sure that we can rely on these values, I made a series of tests (see Results : testing odometry)

E. Integrating a gyrometer

Source code : Appendix C

To try to improve the odometry in angular movement, a gyrometer was installed on top of the robot. The gyrometer returns the speed in 3 direction, but we are only interested in the value of the yaw angle.

Contribution : Creation of a ROS driver in Python that reads the gyrometer's values and converts them

The gyrometer returns a value between −32768 and 32767. This corresponds to the intensity of the angular speed. We chose a measurement range of 2000 dps which gives us a sensitivity of S = 70 mdps/digits and a zero rate level of 20 digits.

Converting the digital value to dps :
- Calculate the average value when the gyro isn't moving.
- Subtract this average to the output values of the gyro.
- Set to zero the the values between -20 and 20.
- Multiply the output values of the gyro by S.

$$R_t \begin{cases} = SC \times (R_m - R_0) \\ = 0 \quad \text{if } |(R_m - R_0)| < 20 \end{cases}$$

R_t (dps): true angular rate
R_m (LSBs): gyroscope measurement
R_0 (LSBs): average zero value
SC (dps/LSB): sensitivity

To obtain the pose orientation, we multiply the speed by the time it spent at that speed.

I made a program in Python language which reads the values returned by the gyrometer through the USB cable by using the serial library of Python. The program then converts these values and sends them to the ROS environment in the corresponding topic called “imu/data”. A built-in ROS node then takes the values of the gyro and odometry and inputs them in an Extent Kalman Filter resulting in a more precise localization.

The integration of the gyrometer **wasn't completed** because a 2 degrees per minute drift was detected and couldn’t be compensated. Since time was limited, finding a solution was postponed. There was therefore no objective in testing if this improved odometry as we knew already that the values of the gyro were false.

F. Going from point A to point B using Navigation :

The first step in using navigation is to create a map of the corridors of INRIA to then be able to use it as a landmark. There are two ways of obtaining this map, using SLAM and creating it from scratch based on plans.

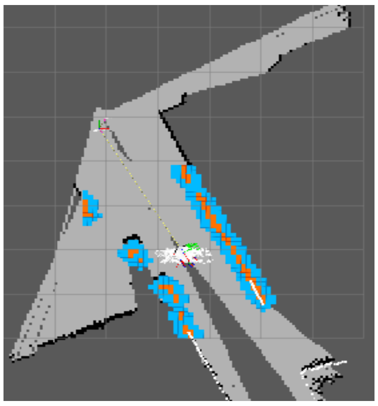


Image 7: process of creating a map

Contribution : Creating a map of the corridors using SLAM

I obtained a final map by navigating in the INRIA corridors by using teleoperation. The visualizing program used is Rviz and the all the necessary elements to create a map is provided by the gmapping package.

The map created is far from being perfect. Some walls aren't straight, some areas aren't discovered, some obstacles are found in the middle of the corridors..

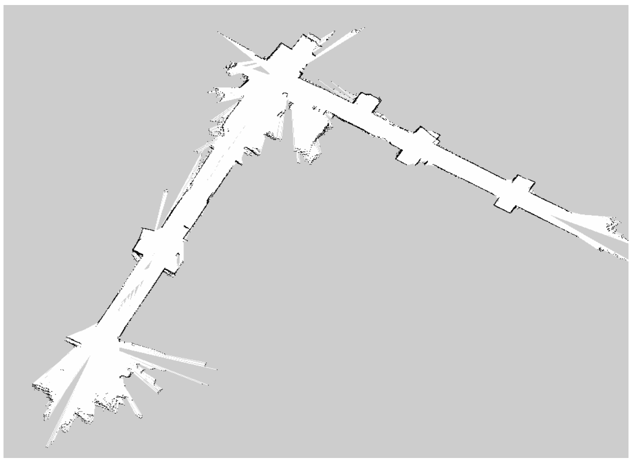


Image 8: result of a map made using SLAM

Contribution : Creating a map from construction plans

To obtain a more decent map, I created created one based on the construction plans of INRIA. The original file was a Bitmap image 2326 x 2211 pixels with a resolution of 20 pels/meters (*see Appendix D*). Using GIMP, I edited this file by removing all elements that would create a problem (openings of doors,..) and drew the obstacles (walls, tables, shelves, ..) in black and the free space in white.

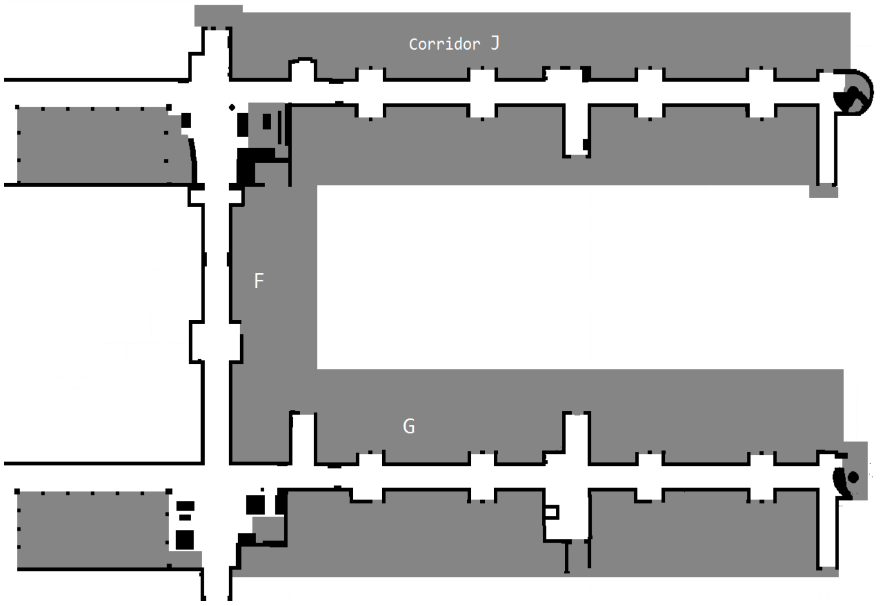


Image 9: Map I created from plans

Contribution : Programming a scenario using navigation satisfying the requirements *(Source code : Appendix E)*

Now that we have a valid map, the navigation algorithm can be run and tested.
I therefore created a program in Python that runs a scenario corresponding to the requirements :
The robot's dock is at the intersection of corridor F and J. Point A is the end of the corridor J and point B the end of corridor G, creating a total traveling distance of : 123 meters. This cycle is done for 60 minutes.
If the batteries are low, the robot goes back near the dock and then the dock function is called to dock. It can then start its cycles again.
The program also backs up if the bumpers hit something, can play a sound if it passes in a certain area, take a picture, and many more options can be added.

The program is 400 lines long and based on a **class** I created called nav_goal. Each method of the class has a precise role such as defining the goals, backing up from the dock , playing a sound,.. These methods are then called in the right order in the main function.
Callback functions are called when messages are received from the Roomba or the Kinect. They are received at a frequency of 30Hz. It is in these functions that the variables are updated and also where I stop running the robot if the bumpers hit something or the batteries are low.

Subscribed to 4 topics : /turtlebot_node/sensor_state , /laptop_charge , /move_base/feedback and /move_base/current_goal
Publishes in 2 topics : cmd_vel and /initialpose
Client of : move_base action.

In the Appendix F you can find my program while running in the ROS environment.

I then ran this program many times to determine to what extent the navigation can be trusted (results can be found in the V. D section).

V. Results

A. Using ROS

ROS is a very powerful and flexible software that I find very appropriate to use in this context. It provides many drivers that are compatible with many devices including ours and it is even possible to modify them thus obtaining exactly what we wish for. Tutorials also exist which is great to start using ROS.

The drawbacks I found on ROS is that it lacks information when ones wants to modify the already created functions. Moreover, some variables aren't available directly on the user interface and you have to look for them in the code to have them published.

B. Testing the dock function

After making adaptations to be able to call the “dock” function which docks autonomously the robot, a series of tests were made to determine its robustness by calling it from different positions and different situations.

Problems found :

If something interferes with the IR receiver or transmitter of the robot or the base, the robot won't dock correctly.
If the robot is too far away from the base, it will not receive the IR-signals of the base and will thus not detect its presence. In this case the robot will go in random directions in order to detect the base's signals. It even has a tendency to brush against walls as it knows that the dock is up against one. This scenario is very bad for us because as you can see on the picture of our robot, the devices on top of the robot are larger than the base of the Roomba, therefore the PC would hit the walls along with the USB connectors. A tragic event where all the elements of the robot were found disconnected and scatter on the ground led us to change the structure of the robot :

Conclusion :
The dock function can be trusted 99% of the time if the robot is facing the dock base at approximately one meter away (in order to receive the IR-signals) and nothing interferes with the IR

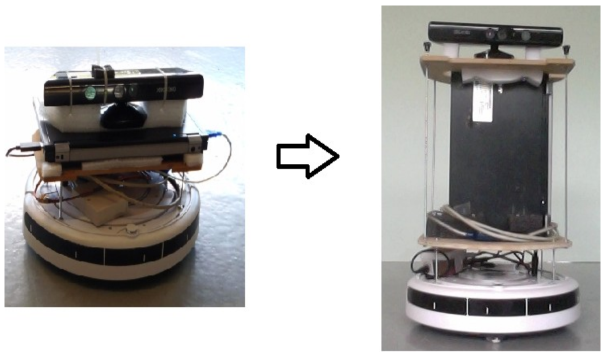


Image 10: upgrading our platform

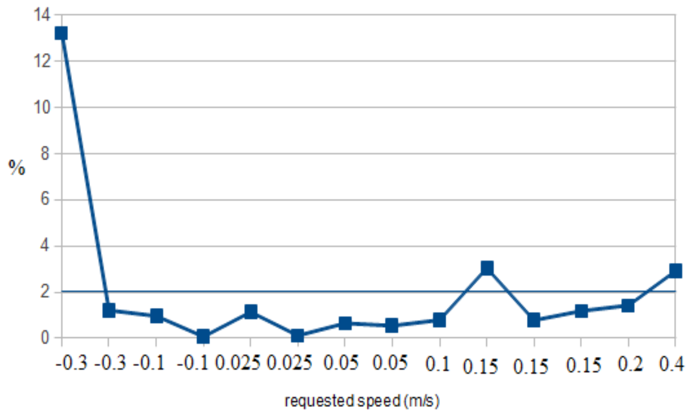
transmitter and receiver.
The charging state of both the Roomba's battery and the laptop's battery is updated in ROS.
The requirement to recharge the battery is therefore met.

C. Evaluating the precision of odometry

To be sure that we can rely on the odometry values and determine a confidence interval I ran a series of test by measuring the distance the robot has really moved and comparing it with the value return by odometry. These test were made with linear and then rotating movements, with different velocities and distances.

Source code : Appendix G

1) Linear movements test :

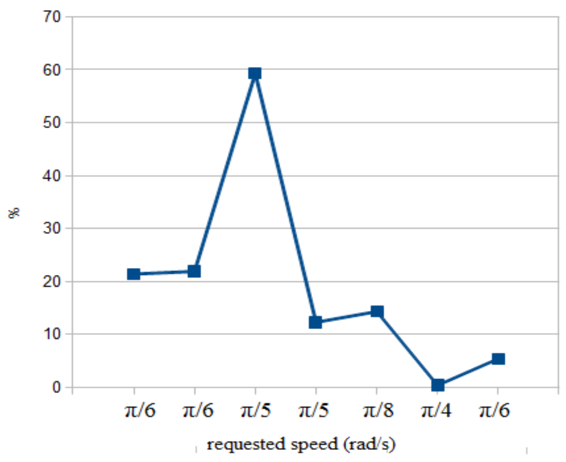


Graph 1: Error percentage between measured and odometry values in straight line

The error has a 2% average and even lower if we take out the first value which is very high (maybe due to an error of initialization).
It is important also to notice that the measured value is higher than the odometric values, this may be due to the fact that sometimes, the odometric sensors skip a signal.
Moreover, the error seems to increase as the requested speed increases.

We can therefore conclude that in a straight line, we can trust the odometric values with 2% precision.

2) Rotating movement test :



Graph 2: difference in percentage between measured and odometry values while rotating

Here the error is much greater than in linear command.
But this time a greater error comes from the measurement as I am less precise measuring angles then measuring a straight line.
Nevertheless the difference is still considerably large and the odometric values for the angle therefore, **can't be trusted**.

Conclusion:
The odometry isn't robust enough for our application. Indeed as the robot goes back and forth from point A to point B, the odometry error increases since the error accumulates in time. A one hour cycle isn't conceivable.

D. Testing the use of Navigation as a solution

I will first analyze the map I created for the robot to navigate in and then give my conclusion elaborated from tests concerning the use of navigation to respond to the requirements.

1) Using the created map :

The map will never be perfect as some obstacles represented can be moved such as the couches and tables (rectangular and square shapes). One solution would be to mark these areas as unknown but this will reduce the amount of landmarks with which the robot uses to localize itself. The laserscan can also detect elements hanged on the wall such as fire extinguishers or radiators. These elements weren't represented on the construction map and need to be added manually, not all were added on this map and the position of the ones added isn't very accurate.
Nevertheless if one spends more time on the map, we can obtain something very precise.

Conclusion : the created map is largely sufficient to run tests on it, for the real application it may be needed to spend more time improving it.

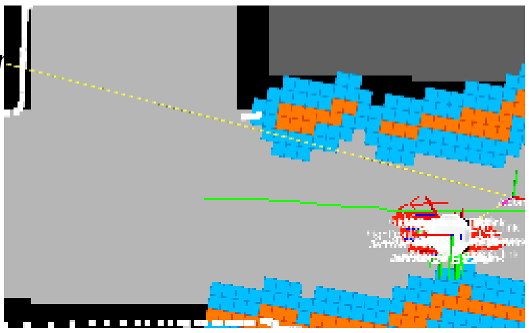
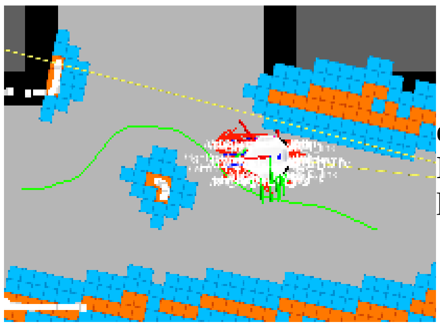
2) Running the program:

I ran my program previously presented many times in the aim of testing my program and testing the robustness of navigation.

testing period	**times launched**	**distance covered**	**running time**
two weeks	> 40	~ 2,8 km	21 hours

Not surprisingly, this complex program didn't run perfectly the first time, but running these test allowed me to understand the ways to improve navigation as well as its limitations. At the end of each test, improvements can be made leading little by little towards a stable program.
Here is some issues I came across :

Behavior when obstacle on its path :

When the robot is faced with a dynamic obstacle detect by the Kinect, it simply recalculates its local path and avoids the obstacle.

The main issue is to make sure the robot detects objects that are close to the ground. The parameter “min_height” of the pointcloud to laserscan can be adjusted to improve this detection but we have to make sure not to lower this value too much or the laser will detect the ground as an obstacle.
A problem that can't be fixed is that objects close to the robot won't be detected due to the limitation of the minimum range of the Kinect.
Possible improvement : Add to the navigation program the “light bumper signals” of the Roomba that can detect objects that are just in front of it. It would fill the some blind spots of the Kinect.

Wi-fi problem :

Since the robot is moving in the building, it periodically looses the Wi-Fi signal and disconnects as it moves away from routers. This is a major problem for ROS as it always needs an internet connection. A solution is to set the ROS IP address as the local host. But this disables communication through ROS with the robot. I therefore created a program that sends information on the platform through UDP packets using the socket library of Python.

Contribution : Creation of map showing the Wi-Fi signal intensity in the corridors of INRIA (*See Appendix H*)

This map could then be used to improve the wireless network quality and prevent the robot from disconnecting from the network. This is an important problem as this means that sometimes the user won't be able to communicate with the robot.
Possible improvement : Adding a Wi-Fi antenna to the laptop, changing the Wi-fi device with one that has a greater gain or improving the quality of the wireless network.

Turning problem :

Due to the narrow horizontal angular view of the Kinect (57°) , when the robot executes a right angle turn at the intersection of two perpendicular corridors, it can sometimes hit the corner of the wall as it doesn't see it. To reduce the chances of collision, the Kinect was placed as far behind as possible to have a wider view.
Possible improvement : Increase the parameter : “ inflation radius” of the costmap which will force the robot to make a wider turn. But this will also reduce the number of possible paths and may prevent the robot from finding an open path.

Conclusion :

Using AMCL is the best solution retained to satisfy the requirements. The robot can move from point A to be B autonomously and take into consideration its environment while sending back information to the user concerning the robot's state or of his sensor. The batteries last more than one hour while running and can be easily recharged.

Nevertheless these requirements aren't always met every time the scenario is launched. It has faced some problems and still does but it hasn't come across an impasse. The majority of the problems were programming mistakes, Wi-Fi disconnection errors, device configuration errors and positioning errors of devices. Thus these can be fixed. In some other cases however, I couldn't find where the source of error was. Therefore it is necessary to run more tests to make this program more robust and eventually run it during a whole 24 hours. Making the program more robust seems possible by finding the right combination to the more than 30 parameters that exist to adapt the navigation system.

In my opinion, if one spends one more month on the project, a very reliable scenario satisfying the requirements can be found using ROS and the chosen devices.

VI. Personal assessment

I discovered countless elements and improved my knowledge and usage of many tools during this internship. I discovered the world of robotics and its complex domain that covers the grounds of electronics, informatics and mechanics. I learned to read and program in the Python language and use many of its library such as the serial, sound, graphical and socket libraries. I also improved in using the Linux environment and navigating in it. Finally I discovered new techniques and methods like AMCL or SLAM.

During this internship, I came across some difficulties. I found difficulty in orienting my research and my solutions towards an easier path and sometimes persisted in wanting to solve a problem with the first solution I came about without checking other methods. On a personal scale, I was surprised to find myself quiet uneasy being surrounded by people with a lot of knowledge on the elements I was working with. It made be lack confidence in supporting and bringing out my own ideas.

I was very interested in the work I was doing and found even a new passion for robotics. These two months spent working on this project thus will help me in choosing my future orientation and will remain as a very rich first experience working with an engineering team.

VII. Summary

I had the chance of doing my 2nd year of engineering school internship in the public research center of mathematics and informatics called INRIA. I worked in the service called SED which is mainly composed of engineers who help establish experimental platforms for researchers.

My internship was part of the Senslab project which reunites 256 fixed nodes in a wireless sensor network platform. The subject was to study the feasibility of adding mobile nodes to the platform in Grenoble in order to study network protocols in a dynamic environment.

The solution retained by the engineering team to make mobile nodes was to place a sensor on a robot and to make this robot move inside the platform. The devices they chose are : the vacuum cleaning robot Roomba from iRobot, the Microsoft Kinet, a portable computer and a gyrometer. And the software to work with was ROS. My work was aimed at figuring out how well these chosen elements could be adapted to the following application : the robot must move autonomously from a given point A to point B while being able to localize itself and send data at all times. The robot must be aware of its environment (the office corridors of INRIA) and to be able to recharge its battery on its own.

After getting familiar with the devices and software, I tested different programs of the ROS framework to see if they corresponded for our application, if not , I made many small contribution to adapt them when possible. Once the robot was set up, I added a function in Python programming language which enable an API to make the robot recharge autonomously. I then tested the use of odometry to localize the robot but quickly abandoned due to a lack of precision. Adding a gyrometer to improve the precision wasn't a success as a drift of 2 degs/minute couldn't be fixed.

I then used the AMCL method to localize the robot in a map of INRIA I created. I created a program that runs a scenario which follows to a maximum the requirements. It was run it during two weeks to determine how robust it is and little by little I made small modifications to make it more stable.

Although more time needs to be spent to make this solution more reliable, I concluded that the AMCL provided by ROS is a method that can largely be used to respond to the desired application.

This internship taught me a lot in both an educational and personal way. I learned how to program in Python and to write applications in the ROS software. I also discovered the world of robotics and the different techniques that are associated to it. Overall, my internship was a very rich first experience as part of an engineering team.

VIII. Glossary

Full-scale range : defines the measurement range. The higher the value, the higher the speed we can detect but the step will be larger, making the gyro less precise.

Sensitivity : this value links the output of the gyro to the real value in deg/s. The value given by the datasheet may have been altered when the MEMS was soldered to the PCB. So it is important to find the new corresponding value in order for our gyro to be correct.

Zero-rate level : when the gyroscope isn't moving , values are still being sent on the output. This value defines the offset level.
Rviz : A 3d visualization environment for robots using ROS.
ROS : software framework for robot software development, providing operating system-like functionality on a heterogenous computer cluster
AMCL : Adaptive Monte Carlo localization is a Monte Carlo method to determine the position of a robot given a map of its environment
SLAM : Simultaneous localization and mapping is a technique used by robots and autonomous vehicles to build up a map within an unknown environment while at the same time keeping track of their current location.

IX. Bibliography

ROS wiki - http://www.ros.org/wiki/
Wikipedia – Wikipedia foundation - http://en.wikipedia.org/wiki/
“Introduction to MEMS gyroscopes” - ElectroIQ -
http://www.electroiq.com/articles/stm/2010/11/introduction-to-mems-gyroscopes.html

X. Appendices

Appendix A : main topics and services used

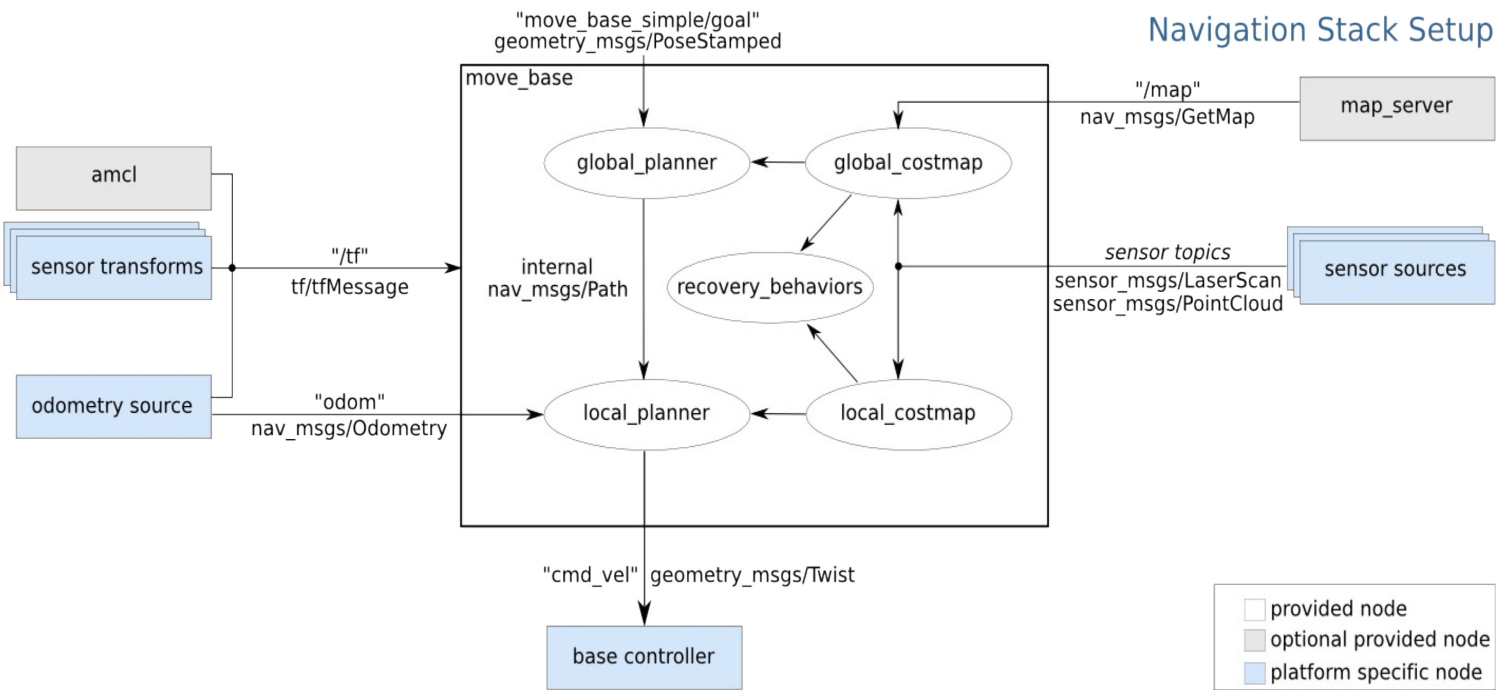
Topics :

/cmd_vel
/laptop_charge
/my_imu/data
/odom
/robot_pose_ekf/odom
/rosout
/tf
/amcl_pose
/camera/depth/image
/initialpose
/map
/move_base/cancel
/move_base/current_goal
/move_base/feedback
/move_base_simple/goal
/particlecloud
/turtlebot_node/sensor_state

Services :

/kinect_laser/set_parameters
/move_base/global_costmap/set_parameters
/move_base/local_costmap/set_parameters
/move_base/set_parameters
/turtlebot_node/set_operation_mode

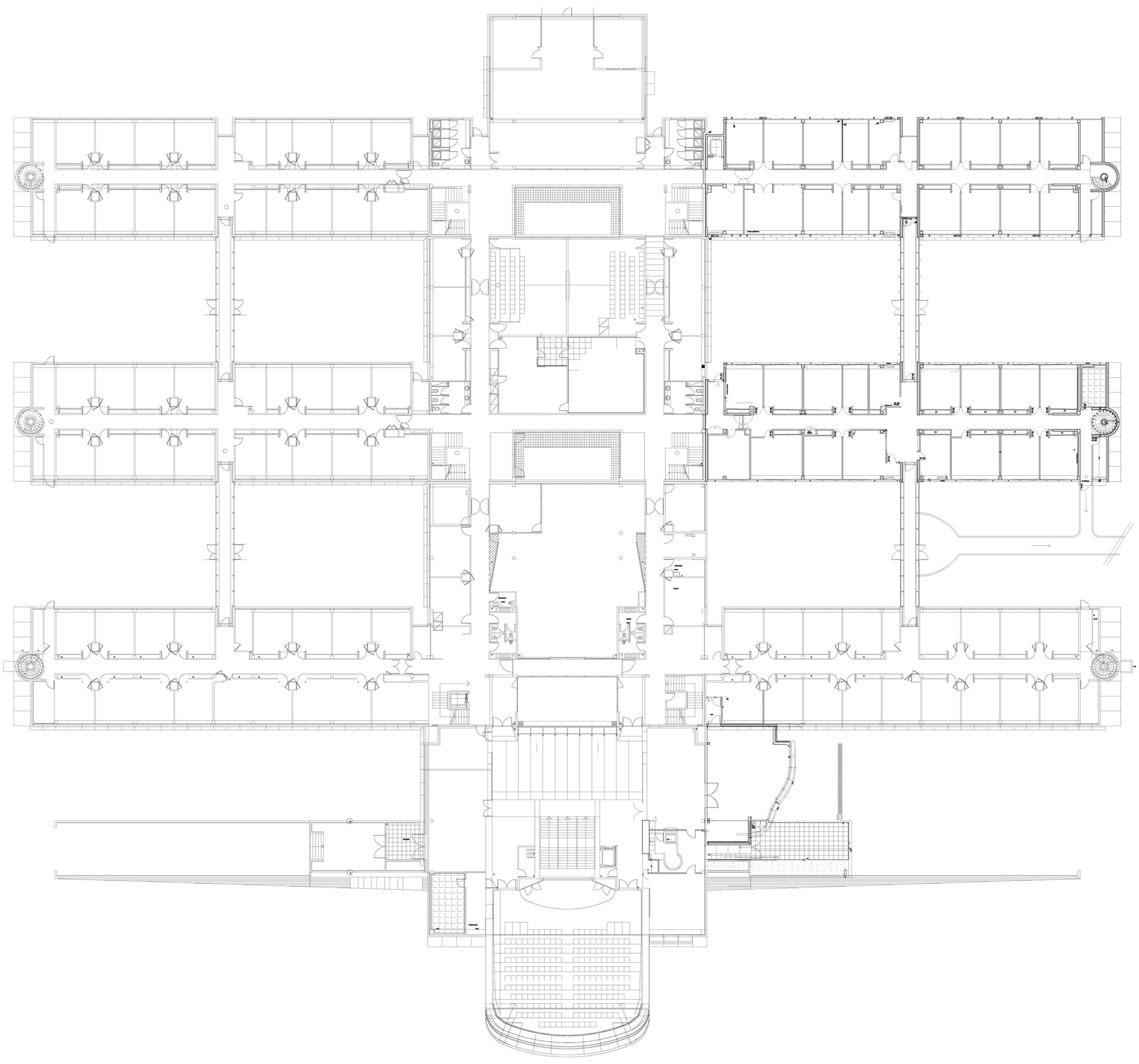
Appendix B : Schematic of the interaction between the different programs running during navigation



Appendix C : The program treating with the values of the gyrometer

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('turtlebot_gyro')
import rospy
import sensor_msgs.msg
import serial
import shlex
from datetime import datetime
import PyKDL
from math import pi
from math import fabs
def gyro_pub():
   start = datetime.now()
   teta = 0.0
   time_pub = 0.0
   imu_pub = rospy.Publisher('my_imu/data', sensor_msgs.msg.Imu)
   rospy.init_node('imu_publisher')
   data = sensor_msgs.msg.Imu(header=rospy.Header(frame_id="gyro_link"))
   data.orientation_covariance = [1e6, 0, 0, 0, 1e6, 0, 0, 0, 1e-6]
   data.angular_velocity_covariance = [1e6, 0, 0, 0, 1e6, 0, 0, 0, 1e-6]
   data.linear_acceleration_covariance = [-1,0,0,0,0,0,0,0,0]
   ser = serial.Serial('/dev/ttyUSB2',500000,8,'N',1)
   ser.open()
   line=ser.readline() #read and discard first line in case not complete
   old_t = datetime.now()
   i =0
   moy_rec =0
   moy_signal = 0

   while 1 :
     line=ser.readline()
     data.header.stamp =  rospy.get_rostime()
     new_t = datetime.now()
     dt = (new_t - old_t)
     old_t = new_t
     data_gyro = shlex.split(line)
     data.angular_velocity.z = ((((float)(data_gyro[2]))*0.07)*(2*pi/360)) - 0.0055 # average calculated before hand
     if  abs(data.angular_velocity.z) < 1.4 :
                        data.angular_velocity.z = 0
     teta += ((float)(dt.microseconds)/1000000) * data.angular_velocity.z
     (data.orientation.x, data.orientation.y, data.orientation.z, data.orientation.w) =
PyKDL.Rotation.RotZ(teta).GetQuaternion()
     time_pub += dt.microseconds
     #moy_rec = (i * moy_rec + data.angular_velocity.z)/(i+1)
     #moy_signal = (i * moy_signal + (float)(data_gyro[2]))/(i+1)
     i += 1

     if time_pub > 33000 :  #time at which we publish
          imu_pub.publish(data)
          rospy.loginfo(teta)
          time_pub = 0

if __name__ == '__main__':
   try:
     gyro_pub()
   except rospy.ROSInterruptException: pass
```

Appendix D : plans of INRIA used to build map



Appendix E : The program running the scenario

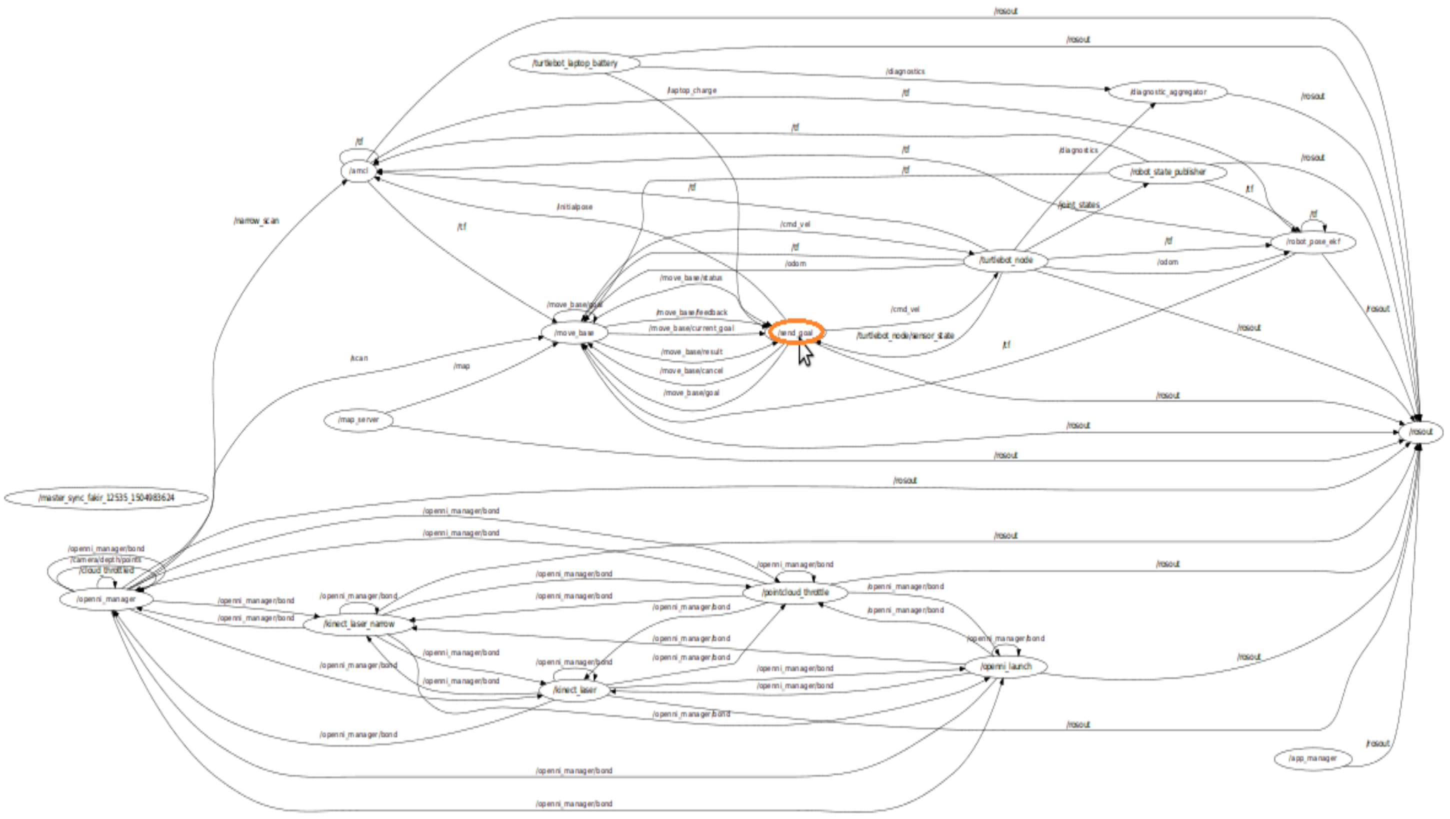
```
#!/usr/bin/env python

import roslib; roslib.load_manifest('circuit')
import rospy
import actionlib
from move_base_msgs.msg import MoveBaseAction
from move_base_msgs.msg import MoveBaseGoal
from actionlib_msgs.msg import GoalStatus
from turtlebot_node.srv import SetTurtlebotDocking
from turtlebot_node.srv import SetTurtlebotMode
from turtlebot_node.msg import TurtlebotSensorState
from turtlebot_node.msg import LaptopChargeStatus
from geometry_msgs.msg import PoseWithCovarianceStamped
from geometry_msgs.msg import Twist
from move_base_msgs.msg import MoveBaseActionFeedback
from geometry_msgs.msg import PoseStamped
from math import sqrt
from math import pi
import os
from datetime import datetime
from datetime import timedelta
from Tkinter import *
import tkSnack
import cv
from std_msgs.msg import String
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError



class nav_goal(object):


    def __init__(self):

            rospy.init_node('send_goal')
            self.sensor_msg = TurtlebotSensorState()
            self.sensor_msg.light_bumper = 0
        self.battery_weak = False
        self.docking = False
            self.pc_battery_info = LaptopChargeStatus()
            self.feedback_info = MoveBaseActionFeedback()
            self.current_goal = PoseStamped()
            self.d = 0.0
            self.bridge = CvBridge()
            self.new_pic = ''
        self.times_hit_bumpers = 0
        self._def_position_and_velocity()
            self.nbr_cycle = 0
            self.print_info = True

            self._init_pub_sub_action()
    def _init_pub_sub_action(self):
            #define a client of the move_base node and the action called MoveBaseAction
            self.client = actionlib.SimpleActionClient("move_base" , MoveBaseAction)
            #wait for connection to server
            self.client.wait_for_server()
            # receive sensor information roomba
            self.sensor = rospy.Subscriber('/turtlebot_node/sensor_state', TurtlebotSensorState,
self.update_roomba_sensors,queue_size=1)
             # receive PC battery info
            self.pc_batt = rospy.Subscriber('/laptop_charge', LaptopChargeStatus, self.update_pc_batt)
            # receive feedback information
            self.feed_back = rospy.Subscriber('/move_base/feedback', MoveBaseActionFeedback , self.update_feedback)
            #receive current goal location
            self.where_is_current_goal = rospy.Subscriber('/move_base/current_goal' , PoseStamped ,
self.update_current_goal)
            #publish velocity commands
            self.force_move = rospy.Publisher('cmd_vel', Twist)
            #publisher of the initial pose
            self.pub = rospy.Publisher('/initialpose', PoseWithCovarianceStamped)


    def _def_position_and_velocity(self) :

            self.command_back = Twist()
            self.command_spin = Twist()

            self.command_back.linear.x = -0.1
            self.command_spin.angular.z = -pi/8

            self.obj = MoveBaseGoal()
            self.obj2 = MoveBaseGoal()
            self.start = MoveBaseGoal()

            self.init_pose = PoseWithCovarianceStamped()

        self.save_goal = PoseStamped()

            #define the position of the dock
            self.init_pose.header.frame_id ="/map"
            self.init_pose.pose.pose.position.x = 17.7
            self.init_pose.pose.pose.position.y = -5.6
            self.init_pose.pose.pose.orientation.z = 0.7071067811
            self.init_pose.pose.pose.orientation.w = 0.7071067811
            self.init_pose.pose.covariance = [0.01, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06853891945200942]


            #define location in front of dock
            self.start.target_pose.header.frame_id = "/map"
            self.start.target_pose.header.stamp =  rospy.get_rostime()
            self.start.target_pose.pose.position.x = 17.7
            self.start.target_pose.pose.position.y = -6.4
            self.start.target_pose.pose.position.z = 0.0
            self.start.target_pose.pose.orientation.x = 0.0
            self.start.target_pose.pose.orientation.y = 0.0
            self.start.target_pose.pose.orientation.z =  0.7071067811
            self.start.target_pose.pose.orientation.w =  0.7071067811

    #define location of objective and position of roomba
```

```
        self.obj.target_pose.header.frame_id = "/map"
        self.obj.target_pose.header.stamp =  rospy.get_rostime()
        self.obj.target_pose.pose.position.x = 59.6
        self.obj.target_pose.pose.position.y = -6.3
        self.obj.target_pose.pose.position.z = 0.0
        self.obj.target_pose.pose.orientation.x = 0.0
        self.obj.target_pose.pose.orientation.y = 0.0
        self.obj.target_pose.pose.orientation.z =  0.7071067811
        self.obj.target_pose.pose.orientation.w =  0.7071067811


    #define location of objective 2 and position of roomba
        self.obj2.target_pose.header.frame_id = "/map"
        self.obj2.target_pose.header.stamp =  rospy.get_rostime()
        self.obj2.target_pose.pose.position.x = 15.45
        self.obj2.target_pose.pose.position.y = -6.3
        self.obj2.target_pose.pose.position.z = 0.0
        self.obj2.target_pose.pose.orientation.x = 0.0
        self.obj2.target_pose.pose.orientation.y = 0.0
        self.obj2.target_pose.pose.orientation.z =  0.7071067811
        self.obj2.target_pose.pose.orientation.w =  0.7071067811


    def play_sound(self,sound_num) :
        root = Tk()
        tkSnack.initializeSnack(root)
        if sound_num == 1 :
            mysound = tkSnack.Sound(file='/home/adminsed/Musique/R2D2_1.mp3')
        mysound.play(blocking=1)
        elif sound_num == 2 :
            mysound = tkSnack.Sound(file='/home/adminsed/Musique/R2D2_4.mp3')
        mysound.play(blocking=1)

    def start_robot(self) :

        #publish the position (5 times to be sure message received)
        for i in range(5):
            self.pub.publish(self.init_pose)
            rospy.sleep(1.0)
        # if docked, backup
        if self.sensor_msg.charging_sources_available != 0 :
        for j in range(50) :
            self.force_move.publish(self.command_back)
            rospy.sleep(0.1)
        for j in range(30) :
            self.force_move.publish(self.command_spin)
            rospy.sleep(0.1)
         self.docking =False

    def cycles(self,nbr_cycles_to_do,time_limit):

      beginning = rospy.get_time()
        stop_time = beginning + time_limit
        rospy.loginfo("Roomba starting %s cycles at : %s , time limit : %s ", nbr_cycles_to_do
,datetime.now(),datetime.now() + timedelta(seconds=time_limit)
)

        #start cycles
        while self.nbr_cycle < nbr_cycles_to_do :
            rospy.loginfo("Roomba moving to goal 1 : %s", datetime.time(datetime.now()))

            self.client.send_goal(self.obj)


            while self.client.get_state() != GoalStatus.SUCCEEDED :
       if self.battery_weak:
                        self.client.cancel_goal()
          break
        if self.times_hit_bumpers >= 40 :
                        self.client.cancel_goal()
            break
       current_time = rospy.get_time()
       if  current_time > stop_time :
            rospy.loginfo(" TIME LIMIT : %s" , datetime.time(datetime.now()))
            self.client.cancel_goal()
            break
       rospy.sleep(1)
       self.dist()
        rospy.sleep(10)


        if self.battery_weak :
            break
        if self.times_hit_bumpers >= 40 :
            break
        if current_time > stop_time :
            break


        rospy.loginfo("Roomba arrived at goal 1 : %s", datetime.time(datetime.now()) )
        rospy.loginfo("Laptop battery charge : %s percentage", self.pc_battery_info.percentage )
        rospy.loginfo("Roomba battery charge : %s ", self.sensor_msg.charge)
            rospy.sleep(2)

#going to obj 2

        self.client.send_goal(self.obj2)
        rospy.loginfo("Roomba going to goal 2 : %s ", datetime.time(datetime.now()))

            while self.client.get_state() != GoalStatus.SUCCEEDED :
        if self.battery_weak :
           break
         if self.times_hit_bumpers >= 40 :
             break
        current_time = rospy.get_time()
        if  current_time > stop_time :
            rospy.loginfo(" TIME LIMIT  %s" , datetime.time(datetime.now()))
            self.client.cancel_goal()
                    break
        rospy.sleep(1)
        self.dist()
                rospy.sleep(10)


        if self.battery_weak :
            break
        if self.times_hit_bumpers >= 40 :
            break
        if current_time > stop_time :
                break
        rospy.loginfo("Roomba arrived at goal 2 : %s", datetime.time(datetime.now()) )
        rospy.loginfo("Laptop battery charge : %s percentage", self.pc_battery_info.percentage )
        rospy.loginfo("Roomba battery charge : %s ", self.sensor_msg.charge)
        rospy.sleep(2)
        self.nbr_cycle += 1


  def end(self):

        if self.times_hit_bumpers >= 40 :
      rospy.loginfo("exiting : %s",datetime.time(datetime.now()))
    else :
      rospy.loginfo("Robot going near dock")
      rospy.loginfo("Laptop battery charge : %s percentage", self.pc_battery_info.percentage )
      rospy.loginfo("Roomba battery charge : %s ", self.sensor_msg.charge)
      self.client.send_goal(self.start)
      while self.client.get_state() != GoalStatus.SUCCEEDED :
        self.dist()
        rospy.sleep(5)
      rospy.loginfo("Roomba arrived at starting position : %s", datetime.time(datetime.now()) )



  def godock(self):
    self.docking = True
    rospy.loginfo("Roomba docking")
        rospy.wait_for_service('/turtlebot_node/look_for_dock') #wait for service to be available
        try :
            go_dock = rospy.ServiceProxy('/turtlebot_node/look_for_dock', SetTurtlebotDocking )  #start service
            resp = go_dock(1)
            return resp.docked
        except rospy.ServiceException, e:
        print "Service call failed: %s"%e

  def set_mode(self,mode_num):
        rospy.wait_for_service('/turtlebot_node/set_operation_mode') #wait for service to be available
        try :
            set_the_mode = rospy.ServiceProxy('/turtlebot_node/set_operation_mode', SetTurtlebotMode )
            resp = set_the_mode(mode_num)
            return resp.valid_mode
        except rospy.ServiceException, e:
        print "Service call failed: %s"%e



  def update_roomba_sensors(self,data_received):

    self.sensor_msg.light_bumper = data_received.light_bumper
    self.sensor_msg.charge = data_received.charge
    self.sensor_msg.capacity = data_received.capacity
    self.sensor_msg.charging_state = data_received.charging_state
    self.sensor_msg.bumps_wheeldrops = data_received.bumps_wheeldrops
    self.sensor_msg.charging_sources_available = data_received.charging_sources_available
    if self.docking == False :
      if (self.sensor_msg.bumps_wheeldrops == 1 or self.sensor_msg.bumps_wheeldrops == 2 or
self.sensor_msg.bumps_wheeldrops ==3) :
              self.save_goal = self.current_goal
          self.client.cancel_goal()
          self.times_hit_bumpers += 1
```

```
                self.play_sound(1)
            self.take_picture()
            rospy.loginfo("Roomba hit something , trying to back up : %s" , datetime.time(datetime.now()))
            for j in range(40) :
               self.force_move.publish(self.command_back)
               rospy.sleep(0.1)
            if self.save_goal.pose.position.x == self.start.target_pose.pose.position.x :
              self.client.send_goal(self.start)
                    rospy.sleep(1)
                if self.save_goal.pose.position.x == self.obj.target_pose.pose.position.x :
              self.client.send_goal(self.obj)
                    rospy.sleep(1)
                if self.save_goal.pose.position.x == self.obj2.target_pose.pose.position.x :
              self.client.send_goal(self.obj2)
                    rospy.sleep(1)

   if  self.sensor_msg.charge < 1000 and self.current_goal != self.start and self.docking == False:
     if self.print_info :
       rospy.loginfo("Roomba : low battery : %s" , datetime.time(datetime.now()))
       self.print_info = False
       self.battery_weak = True


 def update_pc_batt(self,data_received):

    self.pc_battery_info.percentage = data_received.percentage
    if self.pc_battery_info.percentage < 10 and self.docking==False and printed_info == False:
      if self.print_info :
       rospy.loginfo("PC : low battery : %s" , datetime.time(datetime.now()))
       self.battery_weak = True
        self.print_info = False

 def charge_batteries(self):

        while self.pc_battery_info.percentage < 90 and self.sensor_msg.charge < 2400 :
          rospy.sleep(120)
    self.battery_weak = False



 def update_feedback(self,data_received) :

   self.feedback_info.feedback.base_position.pose.position.x = data_received.feedback.base_position.pose.position.x
   self.feedback_info.feedback.base_position.pose.position.y = data_received.feedback.base_position.pose.position.y


 def update_current_goal(self,data_received):

   self.current_goal.pose.position.x = data_received.pose.position.x
   self.current_goal.pose.position.y = data_received.pose.position.y

 def dist(self):

       x1 = self.feedback_info.feedback.base_position.pose.position.x
       y1 = self.feedback_info.feedback.base_position.pose.position.y
       x2=  self.current_goal.pose.position.x
       y2=  self.current_goal.pose.position.y
       d =  sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
   rospy.loginfo("Distance from goal : %s at %s" , d , datetime.time(datetime.now()))
```

```
    def update_pic(self,data):
          try:
            cv_image = self.bridge.imgmsg_to_cv(data, "bgr8")
          except CvBridgeError, e:
            print e
          self.new_pic = '/home/adminsed/ros_workspace/image_hit/pic_'
          self.new_pic += (str)(int(self.feedback_info.feedback.base_position.pose.position.x)) + '_'
          self.new_pic += (str)(int(self.feedback_info.feedback.base_position.pose.position.y)) + '_'
          self.new_pic += '.jpg'
          cv.SaveImage(self.new_pic,cv_image)

    def take_picture(self):

            image_sub = rospy.Subscriber("/camera/rgb/image_color",Image,self.update_pic)
            rospy.sleep(4)
            image_sub.unregister()

  def main(argv) :
       u = nav_goal()
       try:
                  rospy.sleep(20)
                  i = 0
                  while i < 3 :
                          #set robot to full mode
                          u.set_mode(3)
                          # back away from dock
                          u.start_robot()
                          #number of cycles and time limit
                  u.cycles(2,1800)
                          # go in front of dock
                          u.end()
                          # dock
                          if u.times_hit_bumpers < 40 :
                          if u.godock() :
                                  rospy.loginfo("Robot docked, did %s cycles " , u.nbr_cycle)
                          u.set_mode(1)
                          # if batteries really weak, wait for full recharge
                  if u.battery_weak :
                                  u.charge_batteries()
                          u.nbr_cycle = 0
                          #recharging time before restarting cycles
                  rospy.sleep(450)
                          i = i+1
       except KeyboardInterrupt:
        print "Shutting down"
 if __name__ == '__main__':
    main(sys.argv)
```

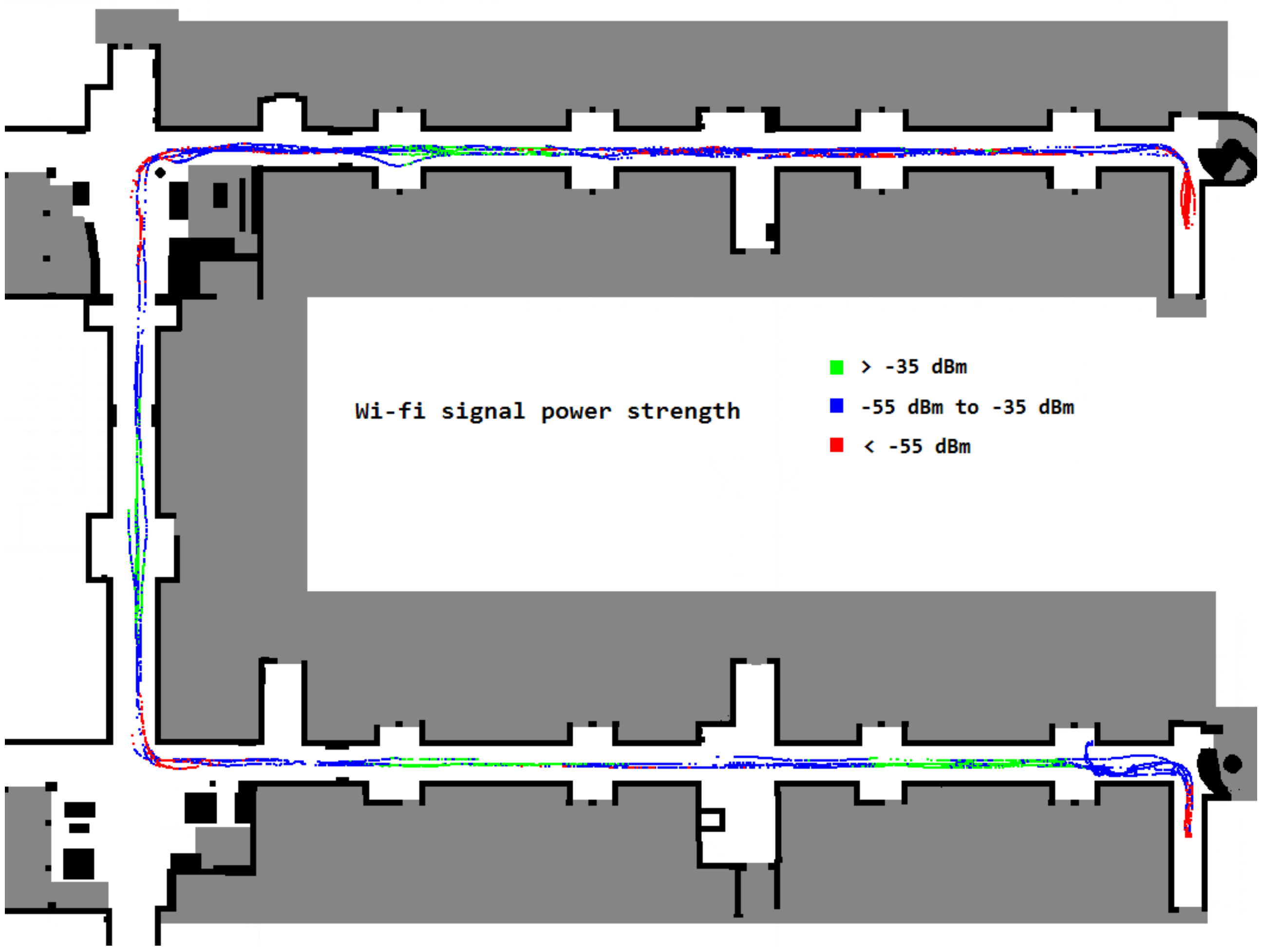
Appendix F : Schematic of ROS nodes and topics while my program is running



Appendix G: The program to evaluate odometry

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('roomba')
import rospy  #imports python library for ROS
from math import pi
from geometry_msgs.msg import Twist #import message structure
def command_vel():
    #define which topic we want to publish in and the type of message we want to publish
    pub = rospy.Publisher('cmd_vel', Twist)
    #initialise the node used to publish data
    rospy.init_node('command_vel')
    #define variable of the type "Twist" with initial values at 0
    cmd = Twist()
    #change de value of the angular speed
    cmd.angular.z=pi/8 #rad/s

# we run a loop to keep on publishing the message onto the topic.
# if the command is only published once, the message will be executed for 0.6 seconds
# it is possible to change this value by changing the parameter : cmd_vel_timeout
# rosparam set /.../cmd_vel_timeout


    for i in range(160):
        pub.publish(cmd)
        rospy.sleep(0.1)

# rospy.sleep stops the program for the number of seconds indicated
# but the message cmd is still being executed by the robot.
# therefore using a loop, the message is executed for: imax * nbr_sec_sleep

    cmd = Twist()  #stop : the values of the message are set to zero and published
    pub.publish(cmd)


if __name__ == '__main__':
    try:
        command_vel()
    except rospy.ROSInterruptException: pass
```

Appendix H : Wi-fi signal strength where the robot has been


Wi-fi signal power strength
🟩 > -35 dBm
🟦 -55 dBm to -35 dBm
🟥 < -55 dBm