Week 9 Studio 1 – Algorithm with the Best Name

Core Objectives:

C1. Introduction to SLAM
C2. Introduction to ROS
C3. Running Hector SLAM under ROS
C4. SLAM or no SLAM?

Preparation (Before the studio):

- Download the image from
  http://www.comp.nus.edu.sg/~boyd/RPi_ROS_Mar2020.img
  Note:
  - The file size is **15,931,539,456 bytes** (~15Gb)
  - The MD5 checksum is **cd6b11b2fd01f50807aa5f9b4acf5b3d**

- **Only 1 image is needed per project team.** Please arrange for a representative in the team to download the image so as not to put undue stress on the web server.
- **Remember to backup any work you have on the Pi (e.g. push to Git) as the image will replace all files on the Pi.**

Studio Setup:

- Work with your project team for this studio as you need both the Pi and Lidar.
- Remember to bring the following:
  - Raspberry Pi
  - PowerBank (fully charged ☺)
  - RPLidar Unit with the given microUSB-to-USB cable

C1. Introduction to SLAM

For a robotic vehicle platform, two of the main tasks associated with navigation are mapping and localization. Constructing the floor plan of the immediate surrounding is known as the mapping problem, while localization refers to the problem of identifying the robot's location on the map. On their own, the two problems have a number of good solutions. However, it become much trickier (and much "funner") if we need to solve both mapping and localization problems at the same time.

Consider the case where the robotic platform is placed in an unexplored locale, e.g. the cleaner bot Roomba is used for the first time in your house, the robot has no floor plan to rely on, but needs to navigate in order to explore. You should realize that this is a chicken and egg problem: to navigate, we need the map; to get the map, we need to navigate!

This is where SLAM algorithms come into play. SLAM stands for **S**imultaneous **L**ocalization **A**nd **M**apping, which is a collection of algorithms to perform both localization and mapping at the same time. This is an active research area where new optimizations, tweaks or even new approaches are being discovered / proposed regularly.

Although the implementation of SLAM algorithm is perhaps too involved for you at this stage, the key ideas are quite intuitive. We will give a high level overview of the algorithm main steps to give you some insights to the working of SLAM.

Before we start, here are a few common terms used in SLAM algorithm description:

Odometry / Telemetry	Information about the position of the robot. The information usually come from motor controls.
Landmark	Environment features that are **re-observable, stable (does not move)** and **easily distinguishable.** For example, walls and large furniture are examples of good landmark.

The key steps of a typical SLAM algorithm is shown below:

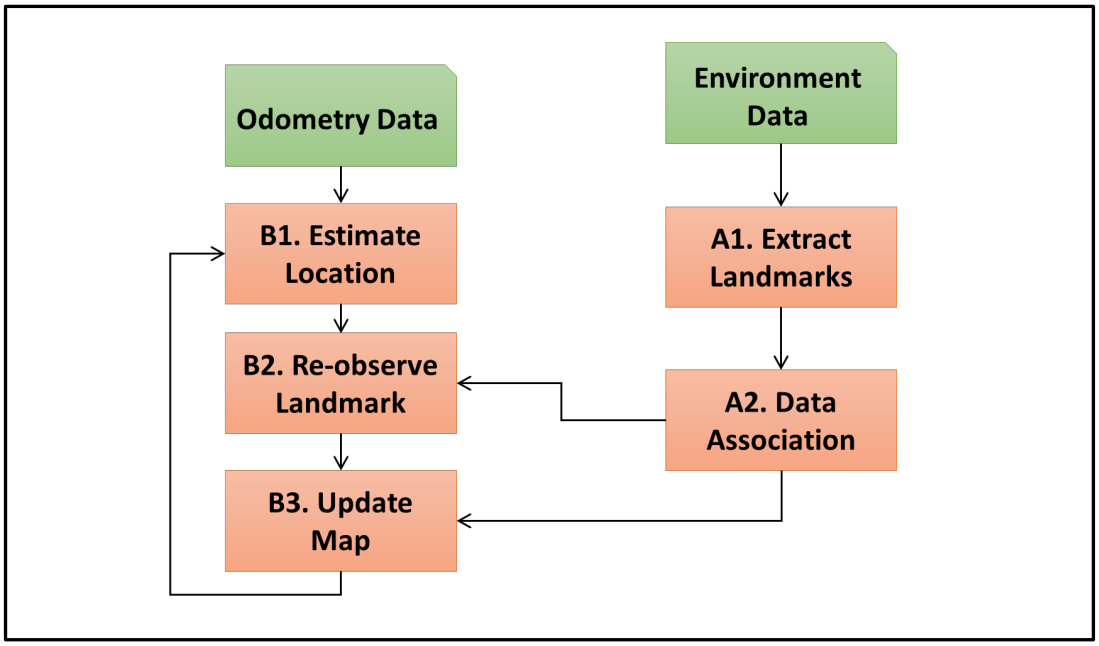
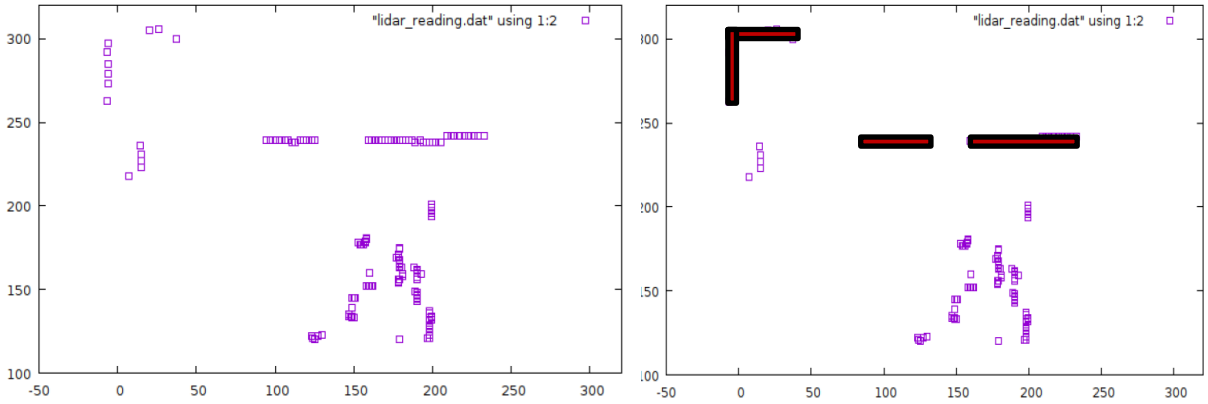


Figure 1. SLAM Algorithm Main Steps

Brief description of the main steps:

A1. Extract Landmarks. Receive environment data from sensor, e.g. LiDAR sensor. From the data, extract likely landmarks. Landmark can be extracted in many ways, for example, grouping data points that are close together, using abrupt change in data to detect the edge of landmark etc. For example, given the LiDAR data on the left, we can detect a few probable landmarks (likely to be wall) as highlighted in the right diagram:



A2. Data Association. Attempt to link observed data points with landmarks. For example, we may observe the same wall segments in subsequent scans, but the data points may be slightly different (e.g. in LiDAR, each revolution of the sensor may not "ping" the same points as the motor rotation is not perfectly aligned with the light transmission and detection unit). The landmarks are then passed to the other part of algorithm for further processing.

B1. Estimate Location. Using the movement information, e.g. from the motor control, estimate the current location of the robot. Since the motor control data is not accurate (e.g. the wheels may not be perfectly aligned ➔ moving at a slightly skew angle, etc), the estimate is usually not very trustworthy.

B2. Re-observe Landmarks. Using the estimated location and observation data, we attempt to match the observed environment to the known landmarks. For example, if we think there is a wall 40cm ahead of us and we have moved 20cm straight, then the wall should be 20cm in front of us now. Difference in the estimation and observation can then be used to update where we think we are. For example, from actual observations we found that the wall is actually 25cm in front of us and slight skewed to the left, we can conclude that our estimated position is not entirely correct and update accordingly.

B3. Update Map. With the observation and estimated location of the robot, we can then extend the map with the observation. The map will be used in the next round of algorithm.

Popular variants of SLAM

At the moment, **GMapping**[2] and **Hector SLAM**[3] are among the more popular SLAM variants. Hector SLAM in particular is quite special as it **does not require odometry data!** What Hector SLAM does is rather ingenious, instead of using odometry data to estimate location, it uses the new observation data and attempt to match it to the known map in order to figure out where the robot is. For example, instead of asking "I've moved forward 20cm, what should I see?", **Hector SLAM** asks "I see a wall to my left now, so how have I moved in order to have the wall there?".

By removing the reliance on odometry data, Hector SLAM enables various interesting application, e.g. handheld mapping device (where odometry data is hard to generate), drone mapping device (where odometry data may be inaccurate, etc). The main drawback of Hector SLAM is that it cannot handle big movement between updates. So, **we need to move at a pace dictated by the speed of Hector SLAM.**

C2. Introduction to ROS

If you try to explore robotic related development on the web, it is almost inevitable that you will bump into a platform known as **ROS**[1] sooner or later. **ROS** (**R**obotic **O**perating **S**ystem) is one of the most commonly used and well developed platform for robotic projects (like ours!). At the moment, there are over 3,000 ROS packages out there, each with unique functionality. For example, you can find ROS package that provides motor control, route planning, teleoperation via keyboard, teleoperation via gamepad, map visualization, various SLAM implementations and even "turtle" simulation!

ROS is a HUGE playground, but also quite daunting place for beginner. For EPP 2, we hope to bring you into this crazy world, but would not demand you to be an "expert" in such a short time frame. We will cover a few key ideas of ROS, show you how to use RPLidar and SLAM on it (Section C3) and leave you the final say on how much (or even whether to use) ROS. The pros / cons are discussed in section C4. (Btw, you may want to know that ROS is only introduced in advanced robotic courses, usually at level 3000 and above ☺).

IMPORTANT

As there is no pre-built ROS Core system packages for Raspbian Stretch (i.e. you cannot do "apt-get install" to get ROS), we have to build ROS from source code manually. Although we would very much love to share this painful process with you (it tooks merely 10+ hours to compile on Pi ☹), it would be dead boring to have the entire studio (actually the entire day) just staring at the compilation progress.

Hence, we **have pre-built** ROS binary for you in the form of a SD card image, i.e. we are giving you a "clone" of our SD card. As the SD card image is 14+ Gb, there is no good way to distribute it other than downloading it over internet. It is cumbersome, but still beat 10+ hours of compilation…….

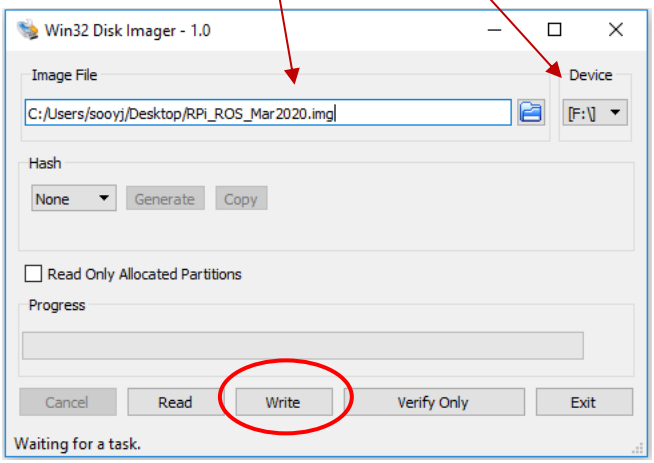
1. **Backup any important programs you have written on Pi,** especially those you have written in previous studios and any progress you have made on your project. (This would be a great time to use your GitHub repository, eh?).

2. For your reference, the ROS version we are using is **Kinetic Karma** (released in 2016).

3. Setup **windows disk imager** program. We recommend **Win 32 Disk Imager** (installer is given in Luminus). Install and run the Disk Image Writer. Another option is: Etcher https://www.balena.io/etcher/

4. Remove the SD Card from your Pi and use a USB card reader / microSD adapter to access the card on your laptop.

5. Launch the Disk Imager, select the drive letter that correspond to the SD card. If there are multiple drive letters assigned (e.g. E:, F:, G:, H:), then select the **lowest** letter in alphabetical order (i.e. E: in this example).
6. Select the SD Card image **`RPi_ROS_Mar2020.img`** to **overwrite** the current content of your SD card.



7. The image writing process takes about 20 minutes to complete. Once it is done, you can place the SD card back into your Pi and boot up.

8. The disk image contains **most of the setup** we covered so far in past studios, including:
   - SSH, VNC enabled
   - Arduino installation (on desktop)

9. The key change is that the **password for "pi"** user has been change to "**`epp2wins`**". This is just to boost our ego (and more importantly, to ensure you have the right disk image).

10. Ensure you have Wifi access to the internet before proceeding.

Verify ROS

Let's verify whether ROS is working:

1. Open a terminal.
2. Enter "**`roscore`**", which is the ROS core enginer (the master program).
3. You should see a bunch of startup messages but no error message.
4. [**`Ctrl-C`**] to terminate **`roscore`**.

You are now ready to try out ROS.

Another backup tool

The disk imager is also a great way to **backup the entire SDCard** for your Pi. You can use the "**Read**" function to store the SDCard image as a file on your laptop / PC.

How does it work?

ROS, despite its name, is not an operating system in the traditional sense. Instead, it is more like a **middle-ware** that enables independent ROS packages to communicate with each other. The communication is implemented using a **message broadcast/subscription** model. Before we continue, let's take a look at a few common terms used by ROS:

ROS Terminology	Meaning
Node	An independent ROS package. Can generate message / consume message.
Message	Essentially formatted information. Can be anything generated by a Node.
Topic	Classification of message (i.e. message type).

In essence, a ROS node can generate (broadcast) output in the form of messages under a certain "topic", for example the LiDAR can broadcast its reading continuously as "/scan" messages. Other ROS nodes can subscribe to message topics and receive messages under that topic, for example, a visualization node can subscribe to "/scan" and plot the points, a SLAM node can use the LiDAR messages to perform localization and mapping, etc. We will see this in action by using the famous **turtle simulation**.

Turtle, all the way down

1. **`cd ~/Desktop/ros/`**
   You can browse around to see the structure of the folder:

```
ros/
    src/
        turtlesim/
```

2. Enter **`catkin_make`**
   `catkin_make` is a project building utility (similar to "**`make`**"), which is configured to look for directories under the **`src/`** sub-folder and build them one by one. In this case, there is only one package, namely "**`turtlesim`**".

   You should see compilation messages on screen. There may be a few compilation warnings which you can safely ignore.

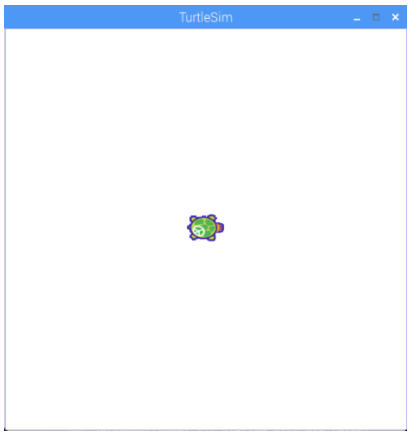
3. Enter **`source devel/setup.bash`** **[Note: Have to do this for all new terminals]**

   This step add the newly compiled packages to the ROS searching directory list. As mentioned above, you need to do this for all **new terminals you open** as the configuration only work for the current terminal.

4. Enter **`roscore`** to start the ROS master program.

5. Open another terminal, browse to the **`ros/`** folder and **perform step 3 again**. Then, Enter **`rosrun turtlesim turtlesim_node`**

   You should see a similar windows titled "TurtleSim" on screen:



   This is a simulation of a turtle (duh!). Note that the Turtle drawing is random, so your turtle may be much cuter (or uglier) than the one shown. Do NOT close the **`turtlesim`**.

6. Let us find out how many ROS nodes are involved. Open a new terminal, and enter

   `rosnode list`

   You should see the following output:

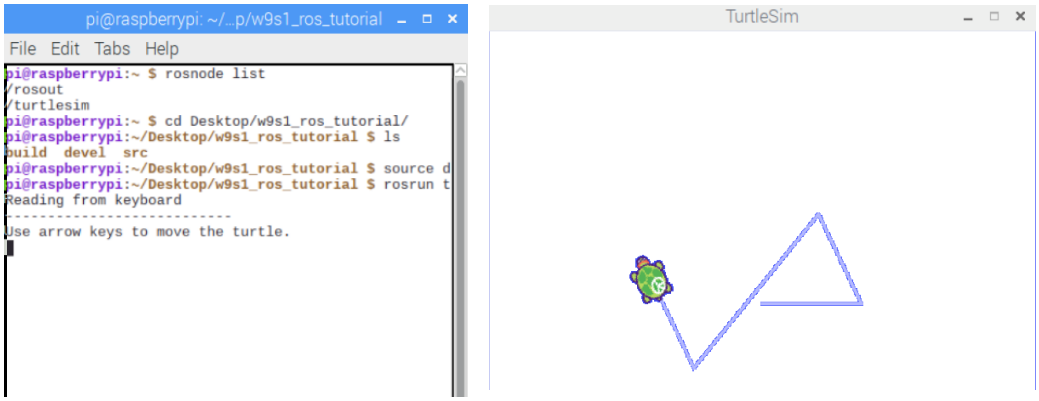
```
/rosout
/turtlesim
```

   `turtlesim` is the visualization we see, while **`rosout`** is the default console log output (the messages you see on the terminal from step 3).

7. Use the new terminal, **perform step 3**, then enter:

   `rosrun turtlesim turtle_teleop_key`

   This starts another ROS node "**`turtle_teleop_key`**" which allows you to "tele-operate" the turtle simulation using keyboard. Press the arrow keys to move your turtle around:



8. Let's find out the number of ROS nodes now. Open yet another new terminal and enter

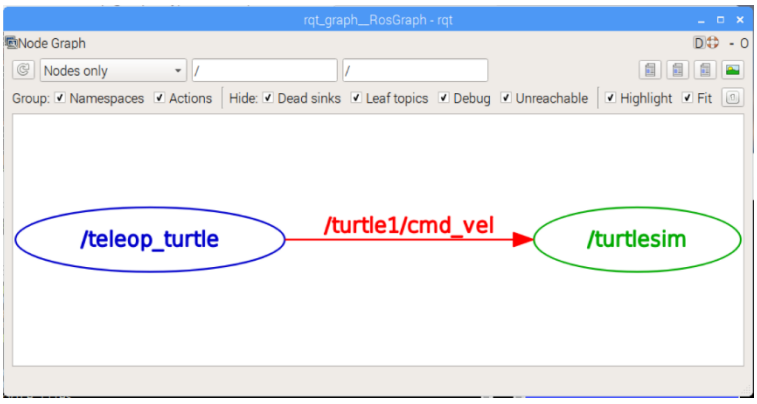
   `rosnode list`

```
/rosout
/teleop_turtle
/turtlesim
```

   As you can see there are now 3 nodes, with the new addition "**`teleop_turtle`**".

9. We can also visualize the relationship between all ROS nodes by using the "**`rqt_graph`**" ROS package:

   `rosrun rqt_graph rqt_graph`



As you can see, the "**`teleop_turtle`**" node generate messages under the topic "**`/cmd_vel`**" (velocity command) which is subscribed by the "**`turtlesim`**". Hence, any key press will be picked up by the "**`turtlesim`**" and processed (i.e. move the turtle).

So, is message broadcast / subscription a good idea?

You are right to think that we can rewrite the entire teleoperation + simulation / visualization as a single and much more efficient program. However, you should also appreciate the flexibility of the ROS setup.

By separating the nodes, you are now free to do "plug and play" with other nodes. For example, instead of a keyboard teleoperation, you can write a new node that accept hand gesture as the control mechanism. The **`turtlesim`** node will not be affected / changed in anyway even though you have rewritten the entire control mechanism. Similarly, you can image how we can change the turtle simulation node to be a VR simulation or even better, as a node to actually control a real world turtle bot.

ROS can be appreciated as showcase of good software engineering in addition to its obvious benefit to the robotic field.

10. **You can safely close all the terminals now.**

C3. Running Hector SLAM under ROS

Key idea

We are going to compile and run multiple ROS nodes in this section. Basically, we need:

a. A RPLidar node to broadcast the LiDAR readings.
b. A SLAM node to use LiDAR readings for environment mapping.
c. A Visualization node to render the environment map.

Other than (c), where we will be using a standard ROS visualization node, known as ***rviz***, we will have to compile and run (a) and (b) on our own.

Compiling RPLidar ROS node

1. Check and make sure your Pi has internet access.

2. Check whether you have **git** installed on Pi. You can just type "**`git`**" to find out. If needed, use: **`sudo apt-get install git`** to install.

3. Open a terminal, and **`cd ~/Desktop/slam/src`**

4. Clone the RPLidar (ROS node) remote repo by:

   `git clone -b slam https://github.com/robopeak/rplidar_ros`

5. Change directory by **`cd ~/Desktop/slam/`**(i.e. one level above the **`src/`** subfolder) then perform **`catkin_make`**

   The compilation takes a bit of time, but should complete successfully.

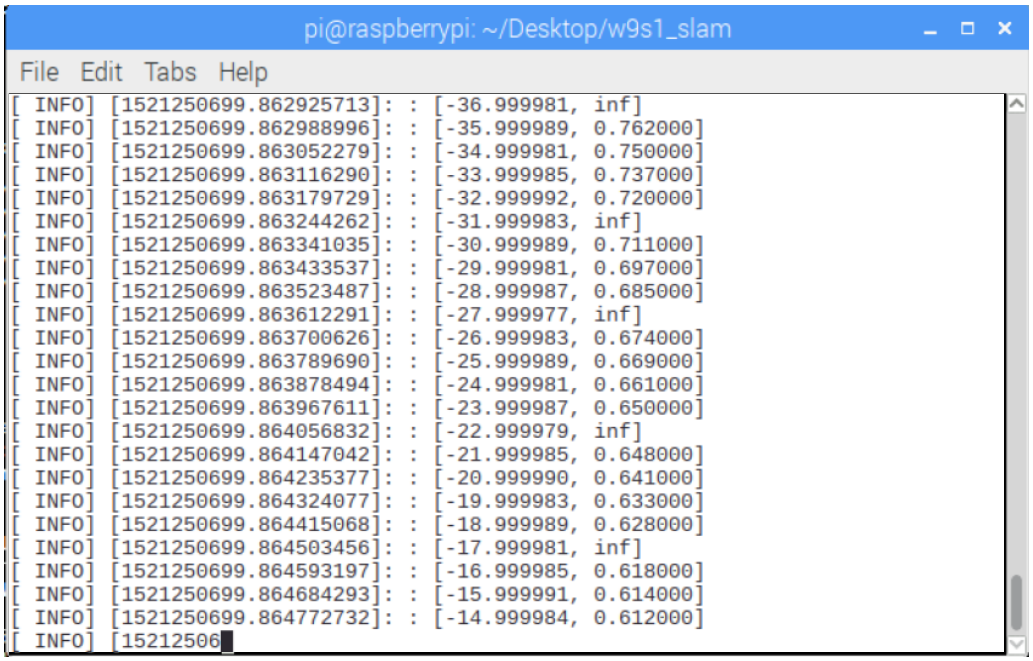
6. **`source ~/Desktop/slam/devel/setup.bash`**

   Remember to perform this step if you open a new terminal. Alternatively, you can add this line to the end of **`.bashrc`** under the **`/home/pi`** folder. The file "**`.bashrc`**" (note the "**`.`**" in front) is executed automatically every time you open a new terminal.

7. Plug in your RPLidar A1 unit, then run the following two commands in **two separate terminals:**

```
roslaunch rplidar_ros rplidar.launch
rosrun rplidar_ros rplidarNodeClient
```

On the terminal that executed the second command, you should see RPLidar readings similar to the following:



8. Once you verified that the RPLidar readings are shown similar to the above, you can safely close all terminals.

Compiling Hector SLAM ROS node

Hello Hector! As discussed in Section C1, there are many SLAM variants. We have chosen **Hector SLAM** for our studio as it does not require odometry data.

1. Open a terminal and **`cd ~/Desktop/slam/src`**

2. Clone the remote repository with Hector SLAM on ROS:

   `git clone` **https://github.com/tu-darmstadt-ros-pkg/hector_slam**

3. Change directory by **`cd ~/Desktop/slam/src/hector_slam/`**

4. Checkout and older version of hector_slam: **`git checkout 3f63834`**

5. Change directory by **`cd ~/Desktop/slam/`**(i.e. one level above the **`src/`** subfolder) then perform **`catkin_make`**

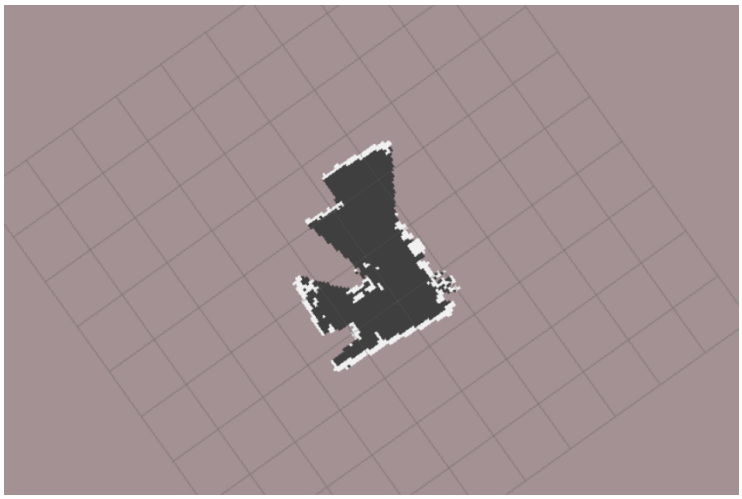
   The compilation is even longer, so be patient.

6. Once the compilation is done, perform
   `source ~/Desktop/slam/devel/setup.bash`

7. Now, this is the moment of truth. Execute:

   `roslaunch rplidar_ros view_slam.launch`

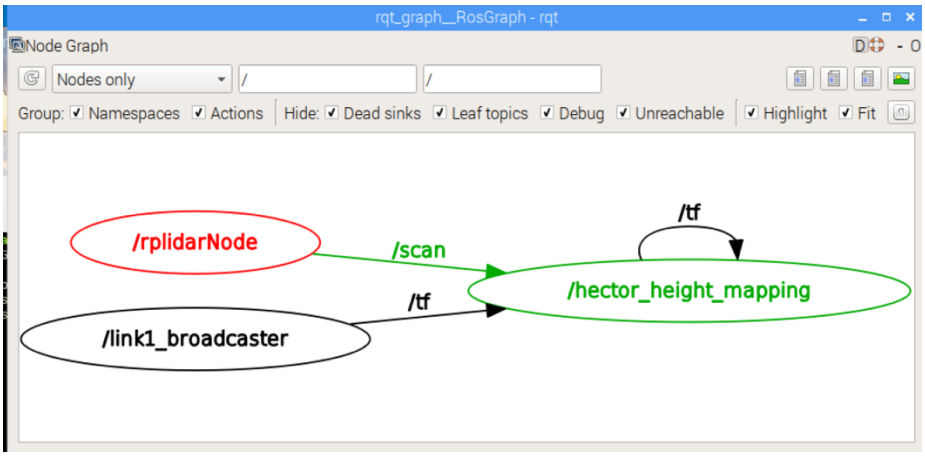
8. You should see a map generated by the visualizer program **`rviz`**:



9. **Slowly** rotate the RPLidar A1 unit on the spot and observe the map. You should see the map is being refined continuously. If you have a wifi setup with Pi, you are encouraged to move the RPLidar unit around in the lab **slowly** and see how the map build up, though don't cross over into others sections. Show us the best map you made!

10. To understand the ROS nodes involved in this process, you can open a new terminal and run:

    `rosrun rqt_graph rqt_graph`



You can see that **`rplidarNode`** broadcast the LiDAR reading for the **`hector mapping`** node to perform SLAM algorithm. Note the **`rviz`** node is not shown.

C4. SLAM or no SLAM?

Once you are calm enough to have rational thoughts ☺, take a look at the CPU load at the upper right corner of the Pi desktop. The SLAM + visualization processes take ~90% of the CPU load and you can even feel the Pi heating up after a while.

So, although SLAM definitely has the wow factor, it may not be very practical for your project. Discuss with your team mates whether you want to use SLAM in the final demo. The bottom line is that you **must use LiDAR** reading to understand the environment during the navigation. Whether you want to use something simple like **gnuplot** from week 7 studio 1 or SLAM is entirely your call. Several common options:

Gnuplot	Just use Week 7 studio 1.
Enhanced Gnuplot	Figure out how to add distance information on the plot (e.g. use/improve on W7S1: C6)
ROS visualization of RPLidar	Figure out how to use **rviz** on rplidar reading only. No SLAM.
SLAM	i.e. this studio. You can look into ways to reduce CPU load by shutting down non-essential services etc.

ROS is red, Alex blew… up

Our aim in this studio is to expose you to "cool stuff", whether ROS is actually used for your project is secondary. Our ultimate "evil" hope is that you are enthralled by ROS and continue to explore / build stuff using it after the semester end. If you have any cool ideas (with / without ROS), please contact Maker@SoC Lab for resource and help.

Use the ROS website [1], [4] to learn more!

References / Resources:

1. ROS (http://www.ros.org)
2. GMapping (http://wiki.ros.org/gmapping)
3. Hector SLAM (http://wiki.ros.org/hector_slam)
4. ROS Tutorials (http://wiki.ros.org/ROS/Tutorials)