A blue vehicle with blue lights

Description automatically generated

**Table of Contents**

[🤗Foreword 5](#_Toc163046166)

[**Workstation Setup** 5](#_Toc163046167)

[**Virtual machine setup** 5](#_Toc163046168)

[**Setup Environment** 6](#_Toc163046169)

[**Synchronize time of robot with laptop** 8](#_Toc163046170)

[**Alias** 9](#_Toc163046171)

[🤔Getting Started 9](#_Toc163046172)

[**The Basics** 9](#_Toc163046173)

[**Autonomous Mobile Robot (AMR)** 9](#_Toc163046174)

[Getting Started - Magni Documentation 10](#_Toc163046175)

[**LiDAR Sensor** 10](#_Toc163046176)

[SICK | Sensor Intelligence 10](#_Toc163046177)

[**Getting Started** 10](#_Toc163046178)

[**Automatic SSH Login** 11](#_Toc163046179)

[**Launch Files** 11](#_Toc163046180)

[**Common Files Types** 12](#_Toc163046181)

[**Troubleshooting** 13](#_Toc163046182)

[**Alias** 16](#_Toc163046183)

[**Report** 16](#_Toc163046184)

[IEP SICK Sol - LiDAR-Enabled Autonomous Mobile Robot System - Brians Tjipto Meidianto (2111182) & Chen Qing Feng Aziel (2124009) - Final Report 17](#_Toc163046185)

[🌊Program Startup Flow 17](#_Toc163046186)

[**Navigation** 17](#_Toc163046187)

[**Mapping** 18](#_Toc163046188)

[**IP Change** 19](#_Toc163046189)

[🏡IP Address and Connections 21](#_Toc163046190)

[**IP Address:** 21](#_Toc163046191)

[**Connections** 22](#_Toc163046192)

[🏎️Drivers 23](#_Toc163046193)

[**Sick Scan Drivers** 23](#_Toc163046194)

[GitHub - SICKAG/sick\_scan\_xd: Based on the sick\_scan drivers for ROS1, sick\_scan\_xd merges sick\_scan, sick\_scan2 and sick\_scan\_base repositories. The driver supports both Linux (native, ROS1, ROS2) and Windows (native and ROS2). 24](#_Toc163046195)

[GitHub - SICKAG/sick\_scan: sick\_scan is an open-source project to support the laser scanner of the company SICK using the ROS-framework 25](#_Toc163046196)

[sick\_scan - Google Drive 25](#_Toc163046197)

[**Starting the LiDAR** 25](#_Toc163046198)

[**Commands** 26](#_Toc163046199)

[**/scan** 27](#_Toc163046200)

[**/sick\_tim\_7xx/lidoutputstates** 28](#_Toc163046201)

[🚢SOPAS ET 30](#_Toc163046202)

[**Field** 30](#_Toc163046203)

[**Troubleshooting** 31](#_Toc163046204)

[👨‍🦽Wheels 32](#_Toc163046205)

[**Documentation** 32](#_Toc163046206)

[Motor Control Board (MCB) - Magni Documentation 32](#_Toc163046207)

[**Teleop** 32](#_Toc163046208)

[**We changed Teleop File in the local machine as shown** 32](#_Toc163046209)

[**Original Teleop from the ROS github** 33](#_Toc163046210)

[**Troubleshooting** 34](#_Toc163046211)

[GitHub - UbiquityRobotics/ubiquity\_motor: Package that provides a ROS interface for the motors in UbiquityRobotics robots 36](#_Toc163046212)

[⏲️Time 36](#_Toc163046213)

[**Time Sync** 36](#_Toc163046214)

[**Time Sync Script** 36](#_Toc163046215)

[**Manunal Time Sync** 38](#_Toc163046216)

[📈Collision Detection 39](#_Toc163046217)

[GitHub - brianstm/botXplorer-scripts: Testing Scripts and Bash File for botXplorer AMR Virtual Machine 39](#_Toc163046218)

[🗺️Map 43](#_Toc163046219)

[**Mapping** 43](#_Toc163046220)

[🤓AMCL 44](#_Toc163046221)

[**AMCL** 44](#_Toc163046222)

[amcl - ROS Wiki 45](#_Toc163046223)

[**Nerd Shit** 45](#_Toc163046224)

[🏃‍♂️Navigation Stack 46](#_Toc163046225)

[**Navigation Stack** 46](#_Toc163046226)

[navigation - ROS Wiki 47](#_Toc163046227)

[**Nerd Shit** 47](#_Toc163046228)

[**Parameters and Configuration** 48](#_Toc163046229)

[**Time Elastic Band (TEB)** 50](#_Toc163046230)

[teb\_local\_planner - ROS Wiki 50](#_Toc163046231)

[**Costmap Conversion** 51](#_Toc163046232)

[teb\_local\_planner/Tutorials/Costmap conversion - ROS Wiki 51](#_Toc163046233)

[🧭Navigation 52](#_Toc163046234)

[**Navigation** 52](#_Toc163046235)

[**Localization** 52](#_Toc163046236)

[**AMCL** 52](#_Toc163046237)

[**Navigation Stack** 52](#_Toc163046238)

[**Magical package** 52](#_Toc163046239)

[**Rviz** 68](#_Toc163046240)

[GitHub - brianstm/botXplorer-navigation: botXplorer AMR Navigation Package 69](#_Toc163046241)

[👾Flask 70](#_Toc163046242)

[**Flask** 70](#_Toc163046243)

[Quickstart — Flask Documentation (2.3.x) 71](#_Toc163046244)

[**Flask App** 72](#_Toc163046245)

[**Index.html** 84](#_Toc163046246)

[Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. 85](#_Toc163046247)

[GitHub - brianstm/botXplorer-flask-app: botXplorer AMR Flask App Package 86](#_Toc163046248)

[😇SICK LiDAR Localization 86](#_Toc163046249)

[SICK | Sensor Intelligence 86](#_Toc163046250)

[**IP Gateway** 86](#_Toc163046251)

[**On the AMR (with IP address 192.168.1.138), we execute the following commands:** 87](#_Toc163046252)

[**On the Virtual Machine, we execute the following command:** 87](#_Toc163046253)

[**Verifying the Gateway Configuration on the AMR** 88](#_Toc163046254)

[**SOPAS Air for Localization and Mapping** 89](#_Toc163046255)

[**Map Conversion with SICK Map ET** 89](#_Toc163046256)

[**Sick Lidar Localization GitHub** 89](#_Toc163046257)

[GitHub - SICKAG/sick\_lidar\_localization 90](#_Toc163046258)

[**Utilizing SICK LiDAR Localization in Navigation** 90](#_Toc163046259)

[GitHub - ehiker/sick\_lidar\_localization2: sick\_lidar\_localization is an open-source project to support the LiDAR-LOC software of the company SICK using the ROS2-framework 92](#_Toc163046260)

# 🤗Foreword

## **Workstation Setup**

### **Virtual machine setup**

1. Download a virtual machine software (VirtualBox, VMWare etc.)

Downloads – Oracle VM VirtualBox

Here you will find links to VirtualBox binaries and its source code.

<https://www.virtualbox.org/wiki/Downloads>

1. Download Ubuntu 20.04 LTS Focal Fossa

Ubuntu 20.04.6 LTS (Focal Fossa)

CD images for Ubuntu 20.04.6 LTS (Focal Fossa)

<https://releases.ubuntu.com/focal/>

1. Set up virtual machine
2. Allocate as much system memory and video memory as you can
3. Allocate at least 25GB of storage
4. Change the Network Adapter settings from NAT to Bridged.
5. Disable any firewalls on your computer
6. Install corresponding ROS version from ROS website, in this case it is ROS Noetic

noetic - ROS Wiki

The ROS Wiki is for ROS 1. Are you using ROS 2 (Humble, Iron, or Rolling)? Check out the ROS 2 Project Documentation Package specific documentation can be found on index.ros.org

<http://wiki.ros.org/noetic>

And you are good to go!

### **Setup Environment**

* Install corresponding ROS version from ROS website, in this case it is ROS Noetic
* Enable zeroconf networking:

sudo apt install libnss-mdns avahi-daemon avahi-utils

* To update the workstation:

sudo apt update

sudo apt upgrade

* To test if zeroconf works:

ping your\_robotname.local // in this case it is ezrobotSP.local

* If the ping works:

You can tell ROS how to communicate with the robot:

export ROS\_MASTER\_URI=http://ROBOTNAME.local:11311

export ROS\_HOSTNAME=$(hostname).local

* To make this environment variable persistent, we append its setting to the file called ~/.bashrc :

echo"exportROS\_MASTER\_URI=http://ROBOTNAME.local:11311" >> ~/.bashrc

echo "export ROS\_HOSTNAME=$(hostname).local" >> ~/.bashrc

* If the ping did not work:

1. Go to the /etc/systemd/resolved.conf file and change #MulticastDNS=no to MulticastDNS=yes
2. Go to the /etc/NetworkManager/conf.d/mdns.conf file. If the file does not exist, create the file in /etc/NetworkManager/conf.d/ and write:

[connection]

connection.mdns=2

* To restart the service:

sudo service NetworkManager restart

sudo service systemd-resolved restart

* Now try to ping the robot from your virtual machine to see if it works

1. To tell ROS how to communicate with the robot:

export ROS\_MASTER\_URI=http://<robot\_ip>:11311

export ROS\_IP=<workstation\_ip> // <workstation\_ip> is your vm ip, to see do the ifconfig command

1. To make this environment variable persistent, we append its setting to the file called ~/.bashrc :

echo "export ROS\_MASTER\_URI=http://<robot\_ip>:11311" >> ~/.bashrc

echo "export ROS\_IP=<workstation\_ip>" >> ~/.bashrc

or you can just change everything to this instead

source /opt/ros/noetic/setup.bash

source ~/catkin\_ws/devel/setup.bash

export ROS\_MASTER\_URI=http://192.168.0.134:11311 # AMR IP

export ROS\_IP=192.168.0.186 # your computer IP

1. Ensure your ROS environment is set up correctly on both your Virtual Machine and Robot. After verifying that the topics are being published from the robot with rostopic list, if you encounter issues receiving topic data using rostopic echo /map, follow these steps to resolve the problem:

* Disable the firewall by running the command sudo ufw disable on both the VM terminal and the Robot terminal. This allows data to flow freely between the two systems.
* Update the hosts file on your VM terminal by executing sudo nano /etc/hosts. Then, add the IP address of the robot and its hostnameubuntu@botXplorer (unless you've changed it using the Ubiquity Website instructions) to the file.

By following these two steps, you should be able to receive topic data using rostopic echo /map or any topic without any issues.

### **Synchronize time of robot with laptop**

1. Install chrony on your laptop and robot:

sudo apt-get install chrony

1. Then do the following command on your laptop:

sudo chronyc -a local stratum 10

sudo chronyc -a allow 0/0

1. ssh into your robot and use these commands:

sudo systemctl stop magni-base

sudo chronyc -a add server <workstation\_ip> iburst // <workstation\_ip> is your vm ip, to see do the ifconfig command

sudo chronyc -a burst 2/4

sudo systemctl start magni-base

1. To set the date and time on both your laptop and robot:

sudo dpkg-reconfigure tzdata

1. To check if ROS is running and you are connected, on your workstation:

rostopic list

### **Alias**

In order for us not to write the long commands we can instead make aliases and then call the aliases. Alias works as to compact the long commands into shorter ones which we can write faster. In order to do this, we will edit the ~/.bashrc file, using the command sudo nano ~/.bashrc, and add these to the file

// The IP Address is based of the robot's IP Address

alias bx="ssh ubuntu@192.168.0.134"

alias bxx="ssh -X ubuntu@192.168.0.134"

// Call the file where it is located, in this case in Downloads

alias rz="rosrun rviz rviz -d /home/user/Downloads/config.rviz"

alias mkmap="rosrun rviz rviz -d /home/user/Downloads/mapmaker.rviz"

alias brc="sudo nano ~/.bashrc"

💡Note: The IP is subject to change based off of the router, SIM card, connection, etc.

# 🤔Getting Started

## **The Basics**

### **Autonomous Mobile Robot (AMR)**

Ubiquity Magni Bot

[[Getting Started - Magni Documentation](https://learn.ubiquityrobotics.com/noetic_overview_need_to_know)](https://learn.ubiquityrobotics.com/noetic_overview_need_to_know" \t "_blank)

## [Getting Started - Magni Documentation](https://learn.ubiquityrobotics.com/noetic_overview_need_to_know" \t "_blank)

[Learn to use our robots with ROS](https://learn.ubiquityrobotics.com/noetic_overview_need_to_know" \t "_blank)

[https://learn.ubiquityrobotics.com/noetic\_overview\_need\_to\_know](https://learn.ubiquityrobotics.com/noetic_overview_need_to_know" \t "_blank)

Suppiled by AIAC (Alan Tan - Office: +6568412311 | Mobile: +6590233211)

### **LiDAR Sensor**

SICK TiM 781-2174101 LiDAR Sensor

## [SICK | Sensor Intelligence](https://www.sick.com/sg/en/tim781-2174101/p/p594148" \t "_blank)

[https://www.sick.com/sg/en/catalog/products/lidar-and-radar-sensors/lidar-sensors/tim/tim781-2174101/p/p594148](https://www.sick.com/sg/en/tim781-2174101/p/p594148" \t "_blank)

💡

**Fun Fact**: SICK AG stands for Sensors, Intelligence, Control, and Knowledge Aktiengesellschaft, which means public limited company in German.

### **Getting Started**

Go to Windows Ubuntu and type out:

ssh ubuntu@192.168.1.114

Regarding the IP address stated above, specifically 192.168.1.114 , it should be noted that this IP is only applicable when used in SP with the mMS4.0 router. In the event that the mMS4.0 router is not being used, it is important to set the IP address to 10.42.0.1 , which is the default IP address of the Raspberry Pi. It is crucial to ensure that the correct IP address is used in order to properly connect to and interact with the device. Failure to do so may result in connectivity issues and hinder the functionality of the device.

when prompt with the password: ubuntu

If the router is unavailable, there is still a way to connect to the AMR. Instead of using the router, you can connect to the AMR's built-in network. However, it's important to note that connecting to the built-in network will not provide the ubiquity bot with Wi-Fi access.

SSID: ezrobotSP7A49 / ezrobot7A49

password: robotseverywhere

### **Automatic SSH Login**

In order to avoid the inconvenience of typing out the SSH command every time you want to log in, you can simplify the process by creating an alias. This alias will automatically log you in and generate a key-token, eliminating the need to enter your password repeatedly. To achieve this, you can use the following code:

ssh-keygen -t rsa

ssh-copy-id ubuntu@192.168.1.114

ssh-keygen -t rsa to generate an RSA key pair, which consists of a private key and a public key. This key pair will be used for secure authentication. After generating the key pair, you can use ssh-copy-id ubuntu@192.168.1.114 to copy your public key to the desired remote server. By doing so, you will be able to log in seamlessly, as the server will recognize your public key and authenticate you without requiring a password every time you establish an SSH connection. Afterwards go to your bash file by doing the following command:

sudo nano ~/.bashrc

And then at the last line of the file, add the following code:

alias <name>="ssh ubuntu@192.168.1.114"

Don't forget to save by doing the Ctrl+O and Ctrl+X command.

### **Launch Files**

Upon turning on the AMR, two launch commands are initiated.

roslaunch sick\_scan sick\_tim\_7xx.launch

roslaunch magni\_bringup base.launch

One is responsible for the LiDAR sensor, while the other manages the AMR's controls, including its OLED, wheels, sonars, and other components. It is crucial that both of these commands run during startup. However, if for any reason the launch files fail to execute, you can manually trigger them by typing out the commands mentioned above.

### **Common Files Types**

These are all the common file types:

1. YAML (.yaml) : YAML stands for "Yet Another Markup Language." It is a human-readable data serialization format commonly used for configuration files in ROS. YAML files store structured data using a combination of lists, dictionaries, and key-value pairs. In ROS, YAML files are often used to define parameters, configuration settings, and other data required by ROS nodes.
2. Xacro (.xacro) : Xacro is an XML macro language used in ROS to generate XML files. It allows you to define reusable XML components and use them in different parts of your robot description. Xacro files typically have the .xacro extension and are processed by the xacro command-line tool to generate corresponding XML files.
3. Launch (.launch) : Launch files are XML-based files used to launch and configure ROS nodes and their interactions. They are used to start multiple ROS nodes with specific parameters, remappings, and other settings in a coordinated manner. Launch files define a collection of nodes, including their names, packages, and required parameters, and provide a convenient way to start complex ROS systems.
4. URDF (.urdf) : URDF stands for "Unified Robot Description Format." It is an XML-based file format used to describe the structure and visual representation of a robot in ROS. URDF files define the robot's joints, links, sensors, and visual properties such as meshes, colors, and textures. URDF files are commonly used for robot modeling and simulation.
5. SRDF (.srdf) : SRDF stands for "Semantic Robot Description Format." It is an XML-based file format used to supplement the URDF with additional semantic information about the robot. SRDF files define groups, allowed collision pairs, and other semantic properties of the robot, which are used for motion planning, collision checking, and other higher-level tasks.
6. RViz Configuration (.rviz) : RViz is a 3D visualization tool in ROS. RViz configuration files store the layout, appearance, and settings of the visualization in RViz. These files have the .rviz extension and are often used to save and load a specific RViz setup for a robot or a particular visualization scenario.

### **Troubleshooting**

If you are getting a ERROR log when running the launch files like the one below:

/home/ubuntu/.ros/extrinsics/lidar\_extrinsics\_top\_plate\_custom.yaml

ERROR failed to parse /opt/ros/noetic/share/magni\_bringup/launch/core\_launch.em

'shell\_installed'

ERROR: Creating launch file did not succeed

where the 'shell\_installed' can be any other thing. You may fix this by checking the robot.yaml file from the directory /etc/ubiquity/robot.yaml and make sure that the contents are as follows:

---

force\_time\_sync: **true**

gps\_installed: none

gps\_position: top\_plate

lidar\_installed: ur50

lidar\_position: top\_plate\_custom

oled\_display:

controller: SH1106

raspicam\_position: tower

shell\_installed: **False**

sonars\_installed: **False**

tower\_installed: **False**

ubiquity\_joint\_publisher:

publish\_rate: 50

type: joint\_state\_controller/JointStateController

ubiquity\_motor:

board\_version: None

controller\_loop\_rate: 20

drive\_type: standard

fw\_max\_pwm: 325

fw\_max\_speed\_fwd: 104

fw\_max\_speed\_rev: -104

pid\_control: 0

pid\_denominator: 1000

pid\_derivative: -110

pid\_integral: 5

pid\_moving\_buffer\_size: 70

pid\_proportional: 5000

pid\_velocity: 0

serial\_baud: 38400

serial\_loop\_rate: 200

serial\_port: /dev/ttyAMA0

wheel\_gear\_ratio: 4.294

wheel\_type: standard

ubiquity\_velocity\_controller:

angular:

z:

has\_acceleration\_limits: **true**

has\_velocity\_limits: **true**

max\_acceleration: 5.0

max\_velocity: 2.0

base\_frame\_id: base\_footprint

cmd\_vel\_timeout: 0.25

enable\_odom\_tf: **true**

left\_wheel: left\_wheel\_joint

linear:

x:

has\_acceleration\_limits: **true**

has\_velocity\_limits: **true**

max\_acceleration: 1.1

max\_velocity: 1.0

pose\_covariance\_diagonal:

- 0.2

- 0.2

- 0.2

- 0.2

- 0.2

- 0.2

publish\_rate: 50

right\_wheel: right\_wheel\_joint

twist\_covariance\_diagonal:

- 0.2

- 0.2

- 0.2

- 0.2

- 0.2

- 0.2

type: diff\_drive\_controller/DiffDriveController

wheel\_radius: 0.1015

wheel\_radius\_multiplier: 1.0

wheel\_separation: 0.33

wheel\_separation\_multiplier: 1.0

## **Alias**

In order for us not to write the long commands we can instead make aliases and then call the aliases. Alias works as to compact the long commands into shorter ones which we can write faster. In order to do this, we will edit the ~/.bashrc file, using the command sudo nano ~/.bashrc, and add these to the file

alias ts='python ~/catkin\_ws/src/script/src/timesync.py'

alias pr='roscd magni\_navigation1/param'

alias lh='roscd magni\_navigation1/launch'

alias fa='roscd flask\_app/src'

alias mb='roslaunch magni\_navigation1 move\_base2.launch'

alias tt='rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py'

alias mpm='roslaunch magni\_lidar magni\_lidar\_mapmaker.launch'

alias gm='rosrun gmapping slam\_gmapping scan:=scan'

alias saver='rosrun map\_server map\_saver -f'

// put <name>-ils when writing the alias

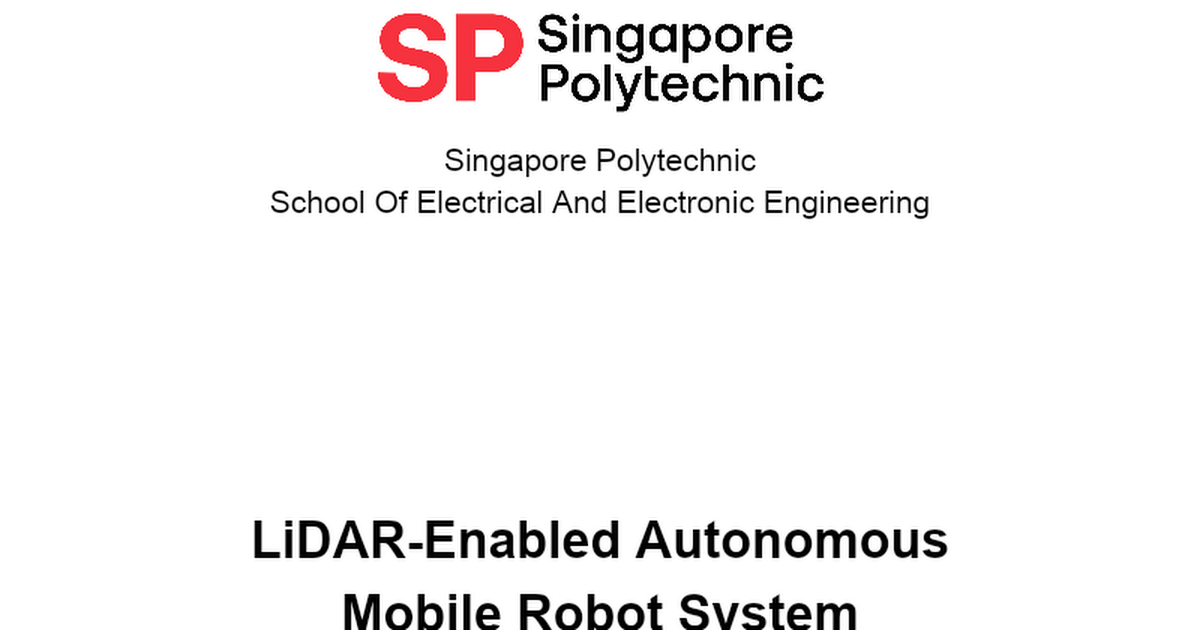
// usage: saver mapname-ils

alias mfd='roscd magni\_lidar/maps'

alias emb='sudo nano ~/catkin\_ws/src/magni\_navigation1/launch/move\_base2.launch'

## **Report**

Our report is accessibe from the link below:

[[](https://docs.google.com/document/d/1QACGV6uIhCqf16Ogfk5Dbks0_wLjD3PT6guiHnzscPs/edit?usp=sharing)](https://docs.google.com/document/d/1QACGV6uIhCqf16Ogfk5Dbks0_wLjD3PT6guiHnzscPs/edit?usp=sharing" \t "_blank)

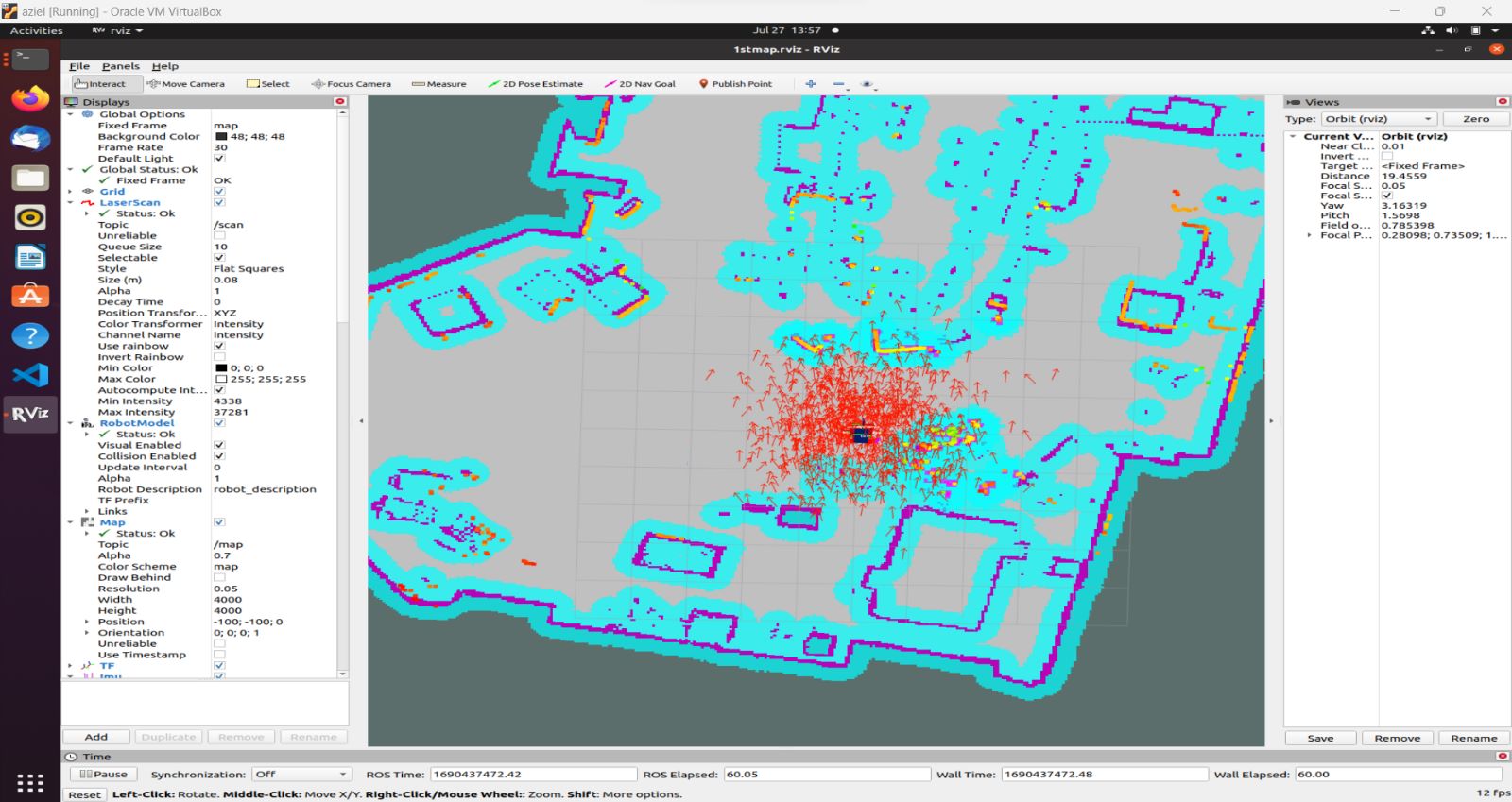
## [IEP SICK Sol - LiDAR-Enabled Autonomous Mobile Robot System - Brians Tjipto Meidianto (2111182) & Chen Qing Feng Aziel (2124009) - Final Report](https://docs.google.com/document/d/1QACGV6uIhCqf16Ogfk5Dbks0_wLjD3PT6guiHnzscPs/edit?usp=sharing" \t "_blank)

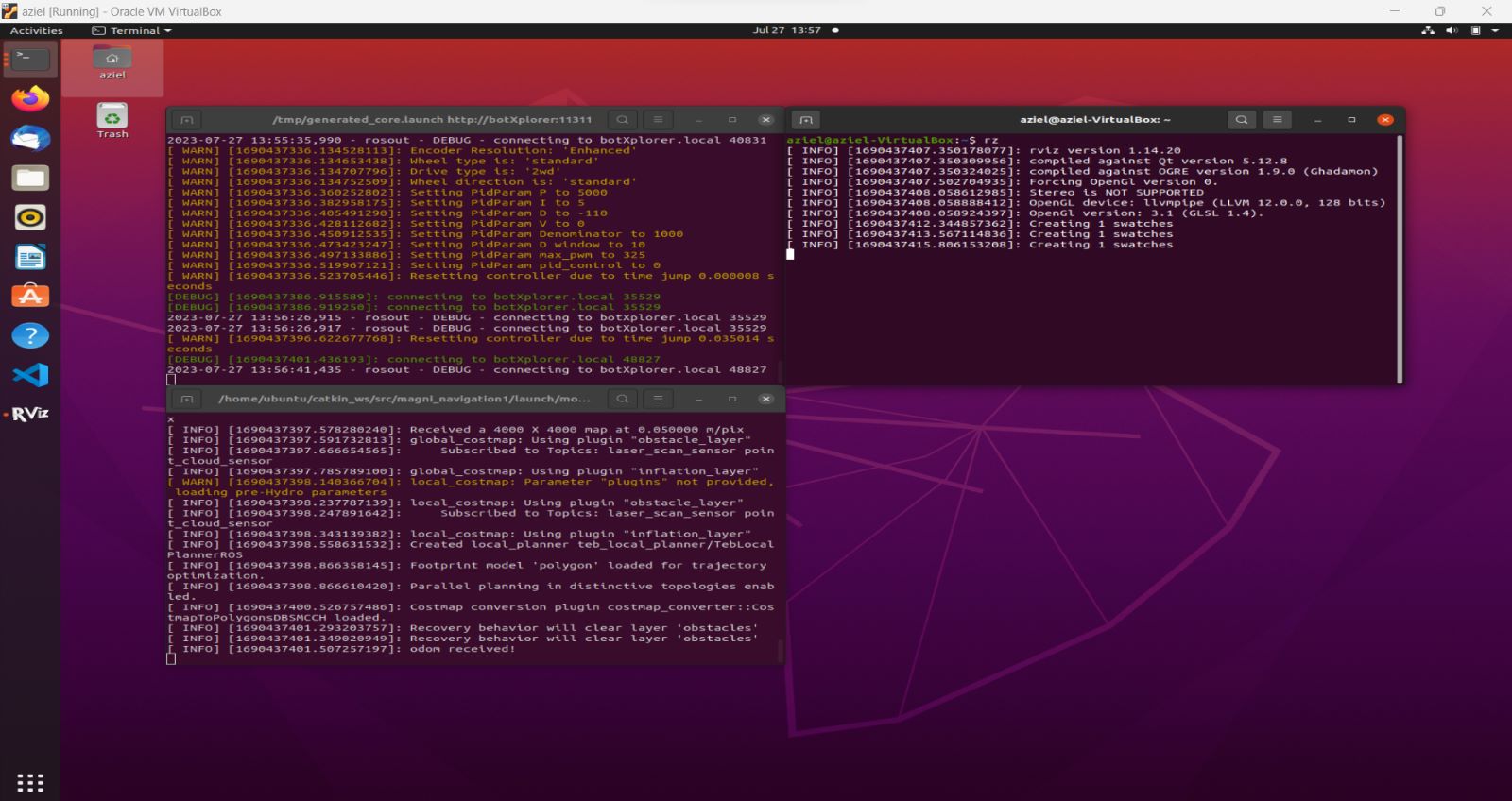
[https://docs.google.com/document/d/1QACGV6uIhCqf16Ogfk5Dbks0\_wLjD3PT6guiHnzscPs/e](https://docs.google.com/document/d/1QACGV6uIhCqf16Ogfk5Dbks0_wLjD3PT6guiHnzscPs/edit?usp=sharing" \t "_blank)

# 🌊Program Startup Flow

### **Navigation**

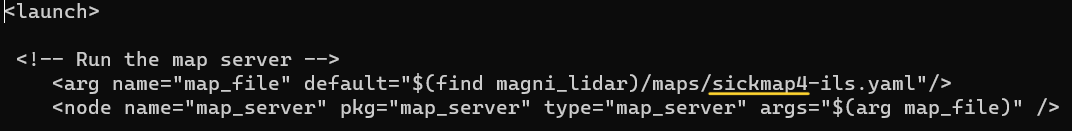
In order to have the [navigation](https://amr-docs-brianstm.vercel.app/navigation) on, we would need to have some nodes running. The following are the node required.  
Steps:

1. Open 3 Ubuntu terminal in the VM
2. First terminal: run roscore  
   Use alias bx to log in to the robot, wait for 3 of the launch files to be fully active (sick driver, bringup / robot wheels, flask app website).
3. Second terminal: run [timesync](https://amr-docs-brianstm.vercel.app/time), ip configuration and navigation software  
   Use alias bx again, then use other alias ts to sync the time. Once the sync is done successfully, use aliasipf to configure the IP for the SOPASair. Once the ip configuration is done successfully, use alias mb
4. Run the SOPASair:  
   Go to the web browser and type in the IP address of the SOPASair: 127.0.0.1
5. Third terminal: run IP receiver and Rviz  
   Without logging into the robot, use alias gtw to receive the gateway and then use the alias rz to start the visual display of the map.



### **Mapping**

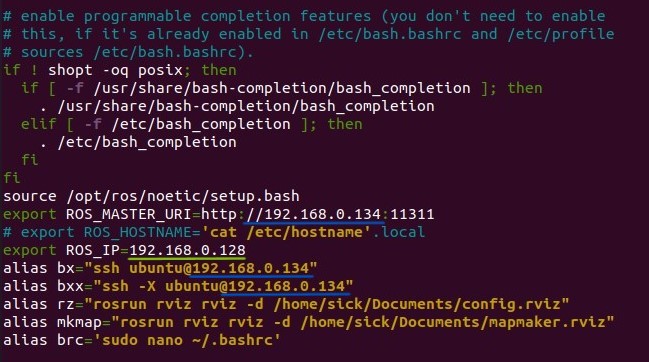
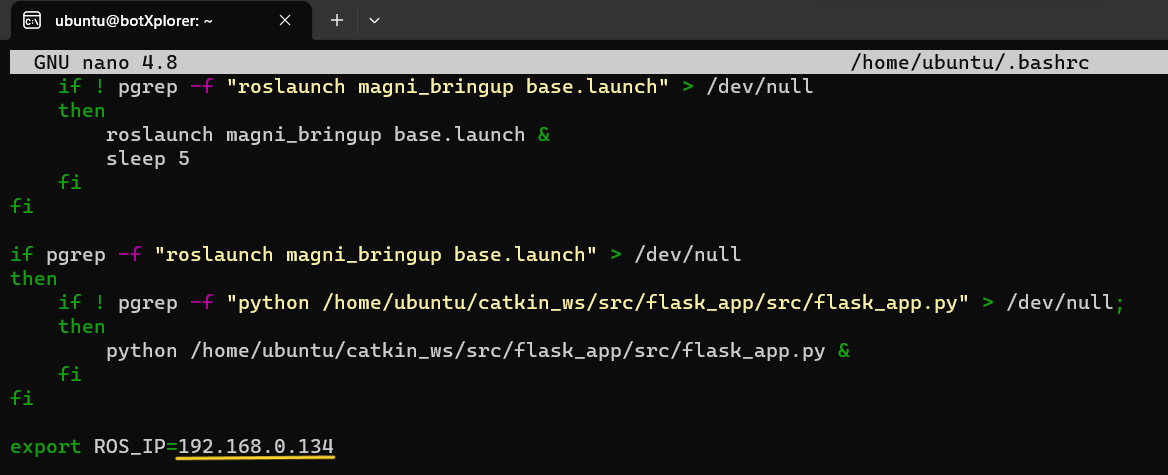
In order to start and do the [mapping](https://amr-docs-brianstm.vercel.app/map), we would need to have some nodes running. The following are the node required.  
Steps:

1. Open 4 Ubuntu terminal in the VM
2. First terminal: run roscore  
   Use alias bx to log in to the robot, wait for 3 of the launch files to be fully active (sick driver, bringup / robot wheels, flask app website).
3. Second terminal: run [timesync](https://amr-docs-brianstm.vercel.app/map) , ip configuration and mapping software  
   Use alias bx again, then use other alias ts to sync the time. Once the sync is done successfully, use aliasipf to configure the IP for the SOPASair. Once the ip configuration is done successfully, use alias mpmgm to start the mapping software gmapping, as well as to start the map making scan process (Note: you can also start them individually using the alias mpm and then gm, mpm starts the mapping software and gm starts the scan process).
4. Run the SOPASair and start mapping:  
   Go to the web browser and type in the IP address of the SOPASair: 127.0.0.1 and then start the mapping in the mapping tab using the legacy format.
5. Third terminal: run IP receiver and Rviz  
   Without logging into the robot, use alias gtw to receive the gateway and then use the alias mkmap to start the visual display of how the robot is scanning.
6. Fourth terminal: run the movement software  
   Use alias bx again, then use other alias mfd to go to the correct folder, then use other alias tt to turn on the movement software, and control the robots movement, using the w, a, s, d keys. For more information on how to use the go to the [wheels](https://amr-docs-brianstm.vercel.app/wheels) page.
7. Fourth terminal: run the map saving software  
   once the mapping is all conpleted, turn off the movement software by pressingCtrl+C before proceeding to the next step, then use the alias saver and the type the name that you want your map to be that ends with -ils for (e.g.,: saver sickmap-ils). Only save the map once you see that the scan is perfect on Rviz.
8. Fourth terminal: modify the navigation software to use the newly created map  
   once the mapping is saved, you need to edit the map in the navigation software to use the new one, use the alias emb and enter in the password and go to the line that says /maps/...-ils.yaml, and change the ... to the new map name that you just saved, then save the file and exit using the command Ctrl+O to save the file and Ctrl+X to exit the file.

modify the yellow colored line to be the name of the new map that you saved previously.

### **IP Change**

In order to change the wifi you would need to first change a few things  
Steps:

1. Start up the robot and connect to it using the old WiFi using the alias bx, if it is unavailable, then use the command ssh ubuntu@10.42.0.1
2. Change the WiFi using the command sudo pifi add MyNetwork password where MyNetwork is the name of the new WiFi and password is the password of the new WiFi
3. Restart the Robot using the command sudo reboot
4. Change the IP in the bash file, use the command brc to change the command both inside the robot and in the VM.
   * In the VM, change the IP where the blue underline is the robot's IP and the green underline is the Computer/VM IP
   * In the Robot, change the IP where the yellow underline is the robot's IP
5. be sure to save everything before exiting the file. using the command Ctrl+O to save the file and Ctrl+X to exit the file.

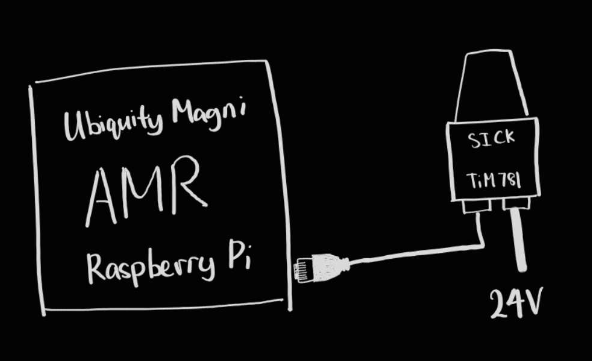
# 🏡IP Address and Connections

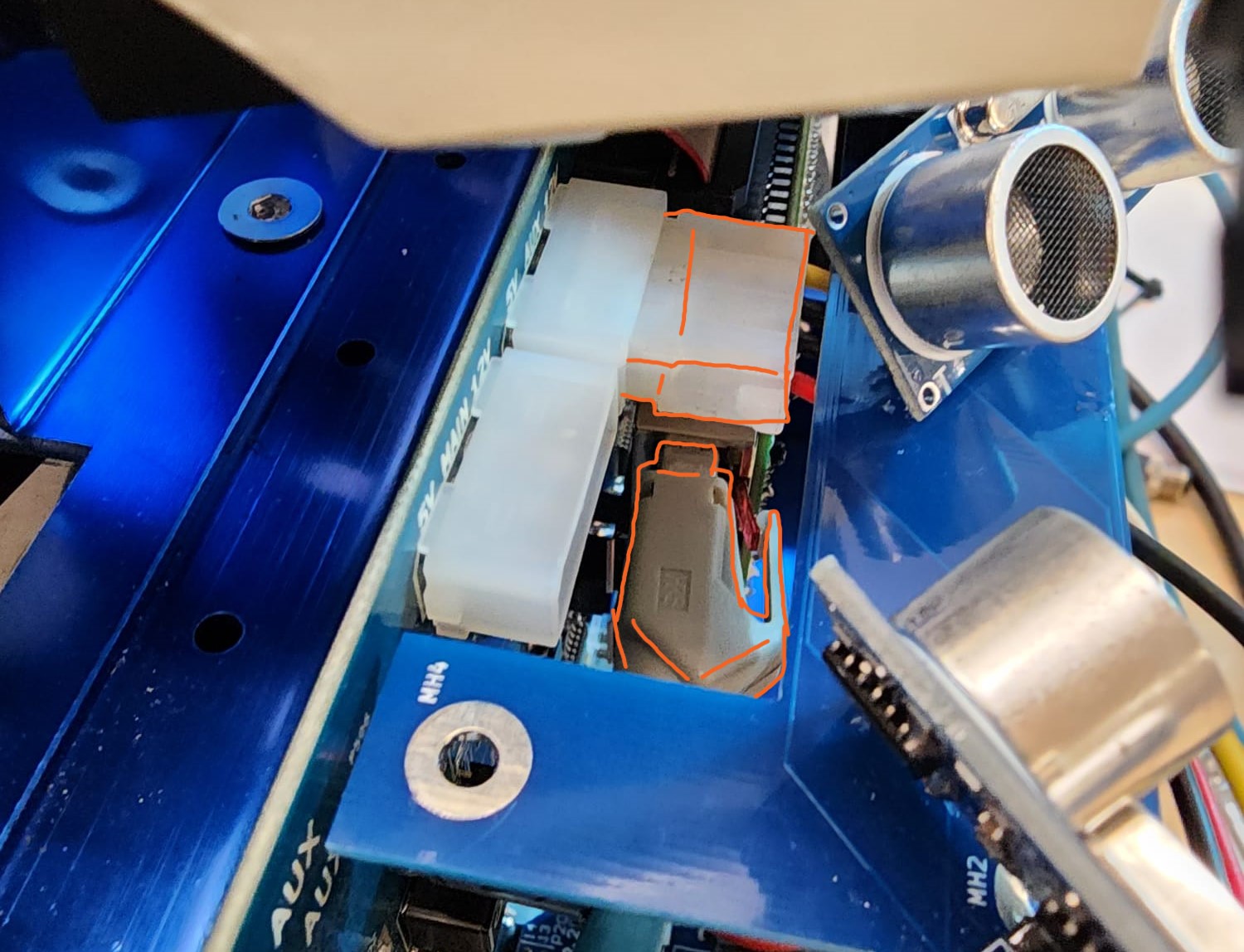
## **IP Address:**

| **NAME** | **IP ADDRESS** |
| --- | --- |
| mMS4.0 WLAN | 192.168.1.114 |
| Raspberry Pi WLAN | 10.42.0.1 |
| Raspberry Pi | 192.168.42.125 |
| LiDAR | 192.168.42.100 |

### **Connections**

The connection between the AMR and the TiM781 is established through the use of an Ethernet cable, as demonstrated in the diagram presented in the Figure below:

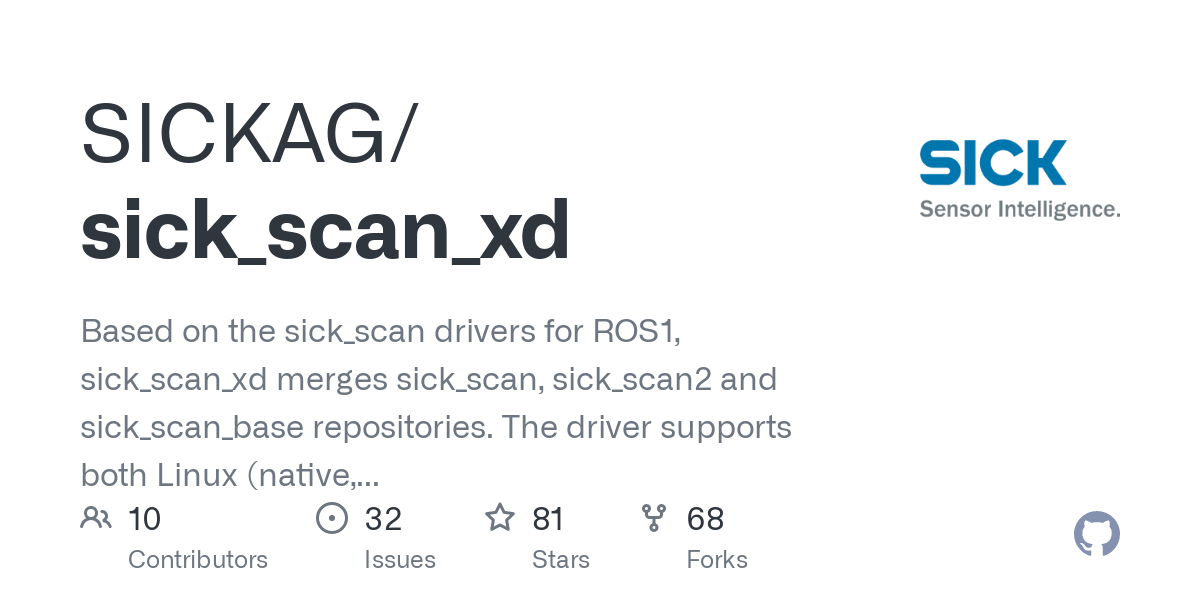




# 🏎️Drivers

### **Sick Scan Drivers**

The LiDAR Sensor is a key component in modern autonomous systems, providing detailed information about the surrounding environment. In order to use the sensor, it is necessary to have a driver installed. Fortunately, obtaining the driver is a straightforward process that can be completed using the GitHub repository provided by the manufacturer. After downloading and installing the driver, users are able to take full advantage of the capabilities offered by the LiDAR sensor.

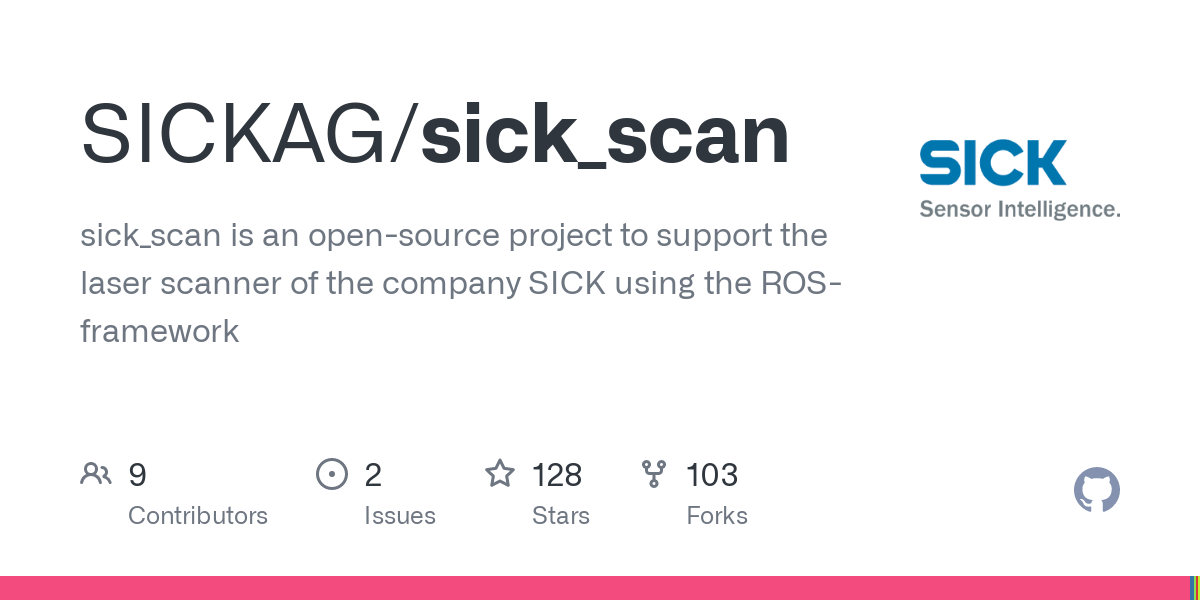
[[](https://github.com/SICKAG/sick_scan_xd)](https://github.com/SICKAG/sick_scan_xd" \t "_blank)

## [GitHub - SICKAG/sick\_scan\_xd: Based on the sick\_scan drivers for ROS1, sick\_scan\_xd merges sick\_scan, sick\_scan2 and sick\_scan\_base repositories. The driver supports both Linux (native, ROS1, ROS2) and Windows (native and ROS2).](https://github.com/SICKAG/sick_scan_xd" \t "_blank)

[Based on the sick\_scan drivers for ROS1, sick\_scan\_xd merges sick\_scan, sick\_scan2 and sick\_scan\_base repositories. The driver supports both Linux (native, ROS1, ROS2) and Windows (native and ROS2)...](https://github.com/SICKAG/sick_scan_xd" \t "_blank)

[https://github.com/SICKAG/sick\_scan\_xd](https://github.com/SICKAG/sick_scan_xd" \t "_blank)

Old repository:

[[](https://github.com/SICKAG/sick_scan)](https://github.com/SICKAG/sick_scan" \t "_blank)

## [GitHub - SICKAG/sick\_scan: sick\_scan is an open-source project to support the laser scanner of the company SICK using the ROS-framework](https://github.com/SICKAG/sick_scan" \t "_blank)

[sick\_scan is an open-source project to support the laser scanner of the company SICK using the ROS-framework - GitHub - SICKAG/sick\_scan: sick\_scan is an open-source project to support the laser...](https://github.com/SICKAG/sick_scan" \t "_blank)

[https://github.com/SICKAG/sick\_scan](https://github.com/SICKAG/sick_scan" \t "_blank)

In the case they removed the repository and you want to use the old one:

## [sick\_scan - Google Drive](sick_scan - Google Drivehttps://drive.google.com/drive/mobile/folders/1_WabeOX_AB4ho6C2V7YyQFomO3pJw5B_?usp=sharing)

[https://drive.google.com/drive/mobile/folders/1\_WabeOX\_AB4ho6C2V7YyQFomO3pJw5B\_?usp=sharing](sick_scan - Google Drivehttps://drive.google.com/drive/mobile/folders/1_WabeOX_AB4ho6C2V7YyQFomO3pJw5B_?usp=sharing)

[sick\_scan.zip83.2MB](https://amr-docs-brianstm.vercel.app/files/sick_scan.zip)

### **Starting the LiDAR**

Run the command:

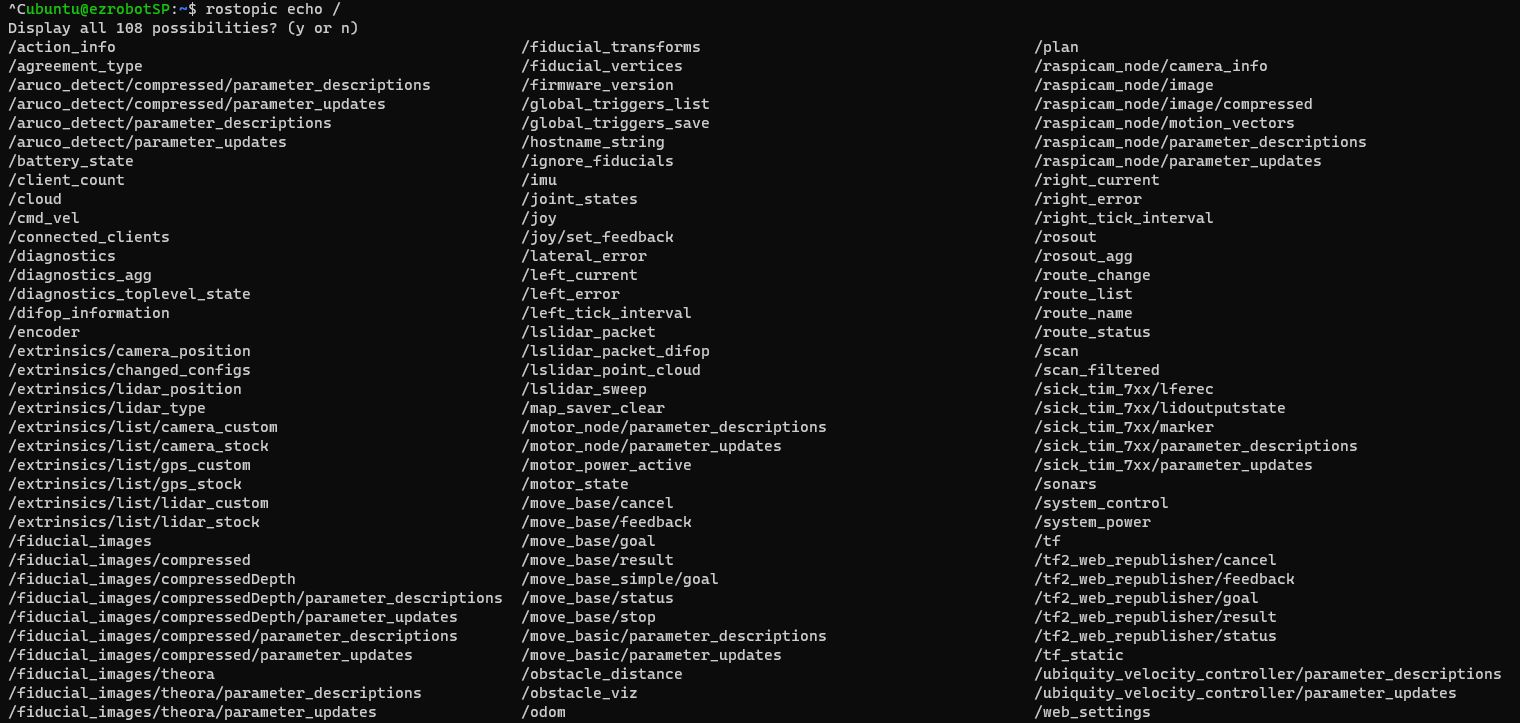
roslaunch sick\_scan sick\_tim\_7xx.launch

If the above command results in an error. Or when ran there is a IP Address other than (NOT) 192.168.42.xxx, run this command instead:

roslaunch sick\_scan sick\_tim\_7xx.launch hostname:=192.168.42.100

### **Commands**

When using the Raspberry Pi ROS and the rostopic echo command, you will have access to a variety of possible commands. The Picture below shows all the possible commands in the AMR.



But we are mainly interested in the ones for the LiDAR, below are all commands for the TiM781.

| **COMMAND** | **DESCRIPTION** |
| --- | --- |
| /scan | This topic provides information about a robot's laser range finder sensor data, which is typically used for mapping and localization. |
| /scan\_filtered | This topic provides the filtered laser range data after applying some filters to remove noise and outlier measurements. |
| /sick\_tim\_7xx/ lferec | This topic might provide information about a LIDAR sensor's range data in a particular format, specific to the SICK TIM 7xx LIDAR sensor. |
| /sick\_tim\_7xx/ lidoutputstate | This topic might provide information about the state of the LIDAR sensor, such as if it is running, scanning or not. |
| /sick\_tim\_7xx/ marker | This topic might provide the marker ID of a SICK LIDAR sensor for identifying the specific sensor in use. |
| /sick\_tim\_7xx/ parameter\_ descriptions | This topic might provide a list of the parameters that can be configured for a SICK LIDAR sensor. |
| /sick\_tim\_7xx/ parameter\_ updates | This topic might provide updated parameter values for a SICK LIDAR sensor. |

### **/scan**

after running rostopic echo /scan you will get the following results:

---

header:

seq: 20823

stamp:

secs: 1682476542

nsecs: 672345192

frame\_id: "cloud"

angle\_min: -2.356194496154785

angle\_max: 2.3557233810424805

angle\_increment: 0.00581718236207962

time\_increment: 6.172839493956417e-05

scan\_time: 0.06666667014360428

range\_min: 0.05000000074505806

range\_max: 100.0

ranges: [0.018000001087784767, 0.0020000000949949026, 0.06000000238418579, 0.0020000000949949026, 0.05000000074505806, 0.04100000113248825, 0.03200000151991844, 0.08700000494718552, 0.07500000298023224, 0.02800000086426735,...]

intensities: [4567.0, 0.0, 5074.0, 0.0, 4705.0, 5522.0, 4204.0, 5520.0, 5007.0, 4741.0, 0.0, 5672.0, 5003.0, 5604.0, 5500.0, 6466.0, 7085.0, 7481.0, 5856.0, 8340.0, 8192.0, 8464.0, 4541.0, 8344.0, 8333.0, 8821.0, 8464.0, 8190.0,...]

---

As you can see, the result is somewhat difficult to read. However, by closely examining the data and using our knowledge of the sensor's operating principles, we can start to piece together what it is trying to convey.

1. header : Contains metadata about the message.
   * seq : The sequence number of the message.
   * stamp : The timestamp of the message.
     + secs : The seconds part of the timestamp.
     + nsecs : The nanoseconds part of the timestamp.
   * frame\_id : The frame ID that this data is associated with.
2. angle\_min : The start angle of the scan in radians.
3. angle\_max : The end angle of the scan in radians.
4. angle\_increment : The angular distance between measurements in radians.
5. time\_increment : The time between measurements.
6. scan\_time : The time between scans.
7. range\_min : The minimum range value the sensor can measure.
8. range\_max : The maximum range value the sensor can measure.
9. ranges : An array of range measurements, representing the distance to an obstacle for each angle in the scan. These values are in meters.
10. intensities : An array of intensity measurements, representing the intensity of the returned laser signal for each angle in the scan.

### **/sick\_tim\_7xx/lidoutputstates**

after running rostopic echo /sick\_tim\_7xx/lidoutputstate you will get the following results:

---

header:

seq: 813

stamp:

secs: 1682560877

nsecs: 565906000

frame\_id: "sick\_tim\_7xx"

version\_number: 0

system\_counter: 1581457000

output\_state: [1, 0, 0, 0]

output\_count: [69, 369, 0, 0]

time\_state: 0

year: 0

month: 0

day: 0

hour: 0

minute: 0

second: 0

microsecond: 0

---

As you can see, the result is similar to the /scan , but instead of giving us RAW data it gives us a more structured data using the help of the field sets (You can learn more about the field and field set here [🚢 SOPAS ET](https://amr-docs-brianstm.vercel.app/sopas)).

1. header : Contains metadata about the message.
   * seq : The sequence number of the message.
   * stamp : The timestamp of the message.
     + secs : The seconds part of the timestamp.
     + nsecs : The nanoseconds part of the timestamp.
   * frame\_id : The frame ID that this data is associated with.
2. version\_number : indicates the version number of the output state message. This field is set to 0 in this example.
3. system\_counter : represents the number of seconds since the device was last powered on.
4. output\_state : a list of four values (by default, there are four fields, but we are only using the first 2), which represent the state of each field in the laser scanner output. If something is detected in the field, it will be high (or 1) until there is nothing detected.
5. output\_count : a list of four values representing the number of output values for each field in the laser scanner output. This count will increment each time something is obstructed or detected on the field.
6. time\_state : represents the time (in microseconds) since the last state change.
7. year , month , day , hour , minute , second and microsecond : represent the current date and time, as reported by the device. In this example, all of these fields are set to 0, since the device does not report date and time information.

# 🚢SOPAS ET

SOPAS ET, or SICK Open Portal Application Studio Engineering Tool, is a powerful software tool developed by Sick to configure, diagnose, and monitor their TiM 781 LiDAR sensors. With the help of SOPAS ET, users can easily set up and customize their sensors to meet their specific needs. The app offers a user-friendly interface that allows for easy access to various sensor parameters, and provides real-time visualizations of sensor data to help users fine-tune their settings. Additionally, SOPAS ET offers advanced diagnostic features that enable users to quickly identify and troubleshoot any issues that may arise with their TiM 781 sensors.

### **Field**

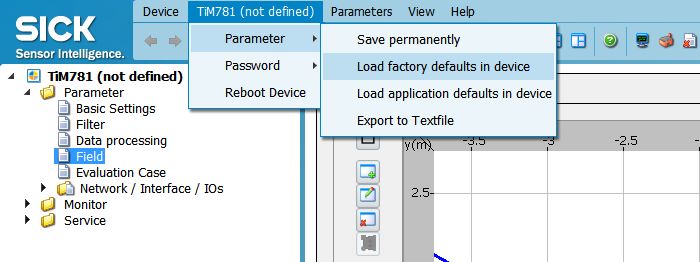
Field sets are pre-defined parameter configurations that allow users to quickly and easily switch between different operating modes of the TiM781 LiDAR sensor. SOPAS ET offers several field sets, each tailored to specific use cases, such as distance measurement, proximity detection, and contour detection. Users can select a field set with just a few clicks, saving time and effort when switching between different applications.

Field monitoring is a feature in SOPAS ET that allows users to view real-time sensor data and track performance metrics. This feature provides a graphical representation of the LiDAR scan area, allowing users to visualize the sensor's field of view and adjust the sensor's parameters accordingly. Users can also monitor the sensor's output data, such as distance measurements and signal quality, to ensure that the sensor is operating within expected parameters. Field monitoring is a powerful tool for fine-tuning sensor performance and ensuring reliable operation in various applications.



### **Troubleshooting**

If you encounter any issues where the Sensor freezes, there are a few things you can do to resolve the issue and get your device back up and running. First, try going to the TiM781 and accessing the Parameter menu. From there, you can select the option to Load factory defaults in device. This will initiate a soft reset of the LiDAR and patch the original settings for both the SOPAS and LiDAR.



# 👨‍🦽Wheels

### **Documentation**

## [Motor Control Board (MCB) - Magni Documentation](https://learn.ubiquityrobotics.com/noetic_magnisilver_mcb" \t "_blank)

[Learn to use our robots with ROS](https://learn.ubiquityrobotics.com/noetic_magnisilver_mcb" \t "_blank)

[https://learn.ubiquityrobotics.com/noetic\_magnisilver\_mcb](https://learn.ubiquityrobotics.com/noetic_magnisilver_mcb" \t "_blank)

### **Teleop**

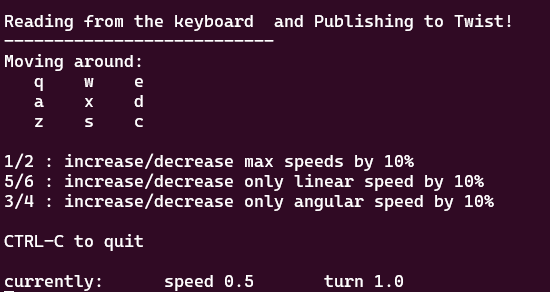
To turn on the teleop package, run the following command in the ssh:

rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

The teleop package is a ROS (Robot Operating System) package that provides a simple way to control a robot using the keyboard. It can be used to move the robot forward, backward, turn left or right, and stop.

## **We changed Teleop File in the local machine as shown**

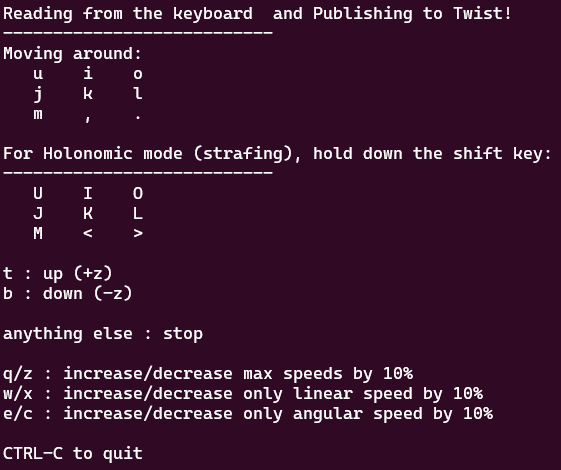
| **KEY** | **DESCRIPTION** |
| --- | --- |
| a | circle left |
| w | straight ahead |
| d | circle right |
| q | rotate counter clockwise |
| k | all stop |
| e | rotate clockwise |
| z | circle backwards left |
| s | straight backward |
| c | cirlce backwards right |
| 1 | increase max speed by 10% |
| 2 | decrease max speed by 10% |
| 5 | increase linear speed by 10% |
| 6 | decrease linear speed by 10% |
| 3 | increase angular speed by 10% |
| 4 | decrease angular speed by 10% |



## **Original Teleop from the ROS github**

| **KEY** | **DESCRIPTION** |
| --- | --- |
| u | circle left |
| i | straight ahead |
| o | circle right |
| j | rotate counter clockwise |
| k | all stop |
| l | rotate clockwise |
| m | circle backwards left |
| , | straight backward |
| . | cirlce backwards right |
| q | increase max speed by 10% |
| z | decrease max speed by 10% |
| w | increase linear speed by 10% |
| x | decrease linear speed by 10% |
| e | increase angular speed by 10% |
| c | decrease angular speed by 10% |

For Holonomic moving mode (strafing), hold the shift key instead, so U, I, O, etc.



### **Troubleshooting**

If on start up there is this error:

ERROR: cannot launch node of type [ubiquity\_motor/motor\_node]: Cannot locate node of type [motor\_node] in package [ubiquity\_motor]. Make sure file exists in package path and permission is set to executable (chmod +x)

You can try the following command:

roscd ubiquity\_motor/ // change directory to ubiquity\_motor

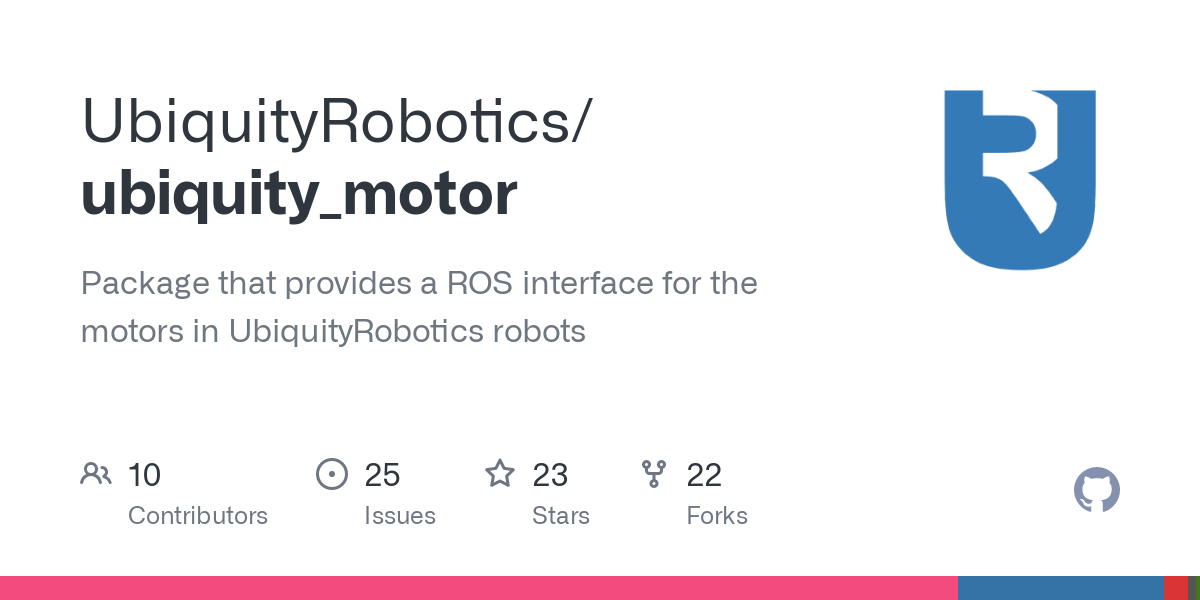
rosdep check package.xml // check for missing dependencies

rosdep install --from-paths . --ignore-src --rosdistro noetic -y // install missing dependencies

If your robot's wheels are no longer functioning properly and it's unable to move, you can try running the following command:

sudo apt-get install ros-noetic-ubiquity-motor

The full motor driver documentation can be found here:

[[](https://github.com/UbiquityRobotics/ubiquity_motor)](https://github.com/UbiquityRobotics/ubiquity_motor" \t "_blank)

## [GitHub - UbiquityRobotics/ubiquity\_motor: Package that provides a ROS interface for the motors in UbiquityRobotics robots](https://github.com/UbiquityRobotics/ubiquity_motor" \t "_blank)

[Package that provides a ROS interface for the motors in UbiquityRobotics robots - UbiquityRobotics/ubiquity\_motor](https://github.com/UbiquityRobotics/ubiquity_motor" \t "_blank)

[https://github.com/UbiquityRobotics/ubiquity\_motor](https://github.com/UbiquityRobotics/ubiquity_motor" \t "_blank)

# ⏲️Time

### **Time Sync**

There is a known, unresolvable time sync error in ROS. This is due to the fact that the NTP server sometimes doesn't work as excepted.

You can see the time error by running the following command in the ssh terminal:

date

### **Time Sync Script**

If you encounter unsynchronized time on your system, you can use the following Python script to synchronize. The default time function in Ubuntu (ROS) may not be accurate, so this script utilizes the reliable worldtimeapi to fetch the current time for the specified location, in this case, Singapore. After synchronizing the time, the script verifies the result by polling the 'date' command.

To run the script, you can simply run the following command or run the alias ts

python ~/catkin\_ws/src/script/src/timesync.py

Below is the script:

import rospy

import subprocess

import requests

import time

from datetime import datetime, timedelta

def time\_sync():

response = requests.get("http://worldtimeapi.org/api/timezone/Asia/Singapore")

if response.status\_code == 200:

api\_data = response.json()

datetime\_str = api\_data["datetime"]

datetime\_obj = datetime.fromisoformat(datetime\_str)

current\_time = subprocess.check\_output(['date']).decode().strip()

rospy.loginfo("Current system time: %s", current\_time)

datetime\_obj += timedelta(seconds=1)

desired\_time = datetime\_obj.strftime("%Y-%m-%d %H:%M:%S")

subprocess.call('echo ubuntu | sudo -S date -s "{}"'.format(desired\_time), shell=True)

rospy.loginfo("System time set to: %s", desired\_time)

else:

rospy.logerr("Failed to retrieve time from the API.")

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('time\_sync\_node')

try:

time\_sync()

for \_ in range(5):

current\_time = subprocess.check\_output(['date']).decode().strip()

rospy.loginfo("Current system time: %s", current\_time)

time.sleep(0.5)

except KeyboardInterrupt:

rospy.loginfo("Time synchronization interrupted by user.")

except subprocess.CalledProcessError as e:

rospy.logerr("Failed to set system time: %s", e.output.decode())

rospy.loginfo("Exiting time\_sync\_node.")

### **Manunal Time Sync**

if you see the time is wrong you can fix it by running the following command in said terminal:

sudo date -s "2023-01-01 00:00:00" // replace the time with the current time

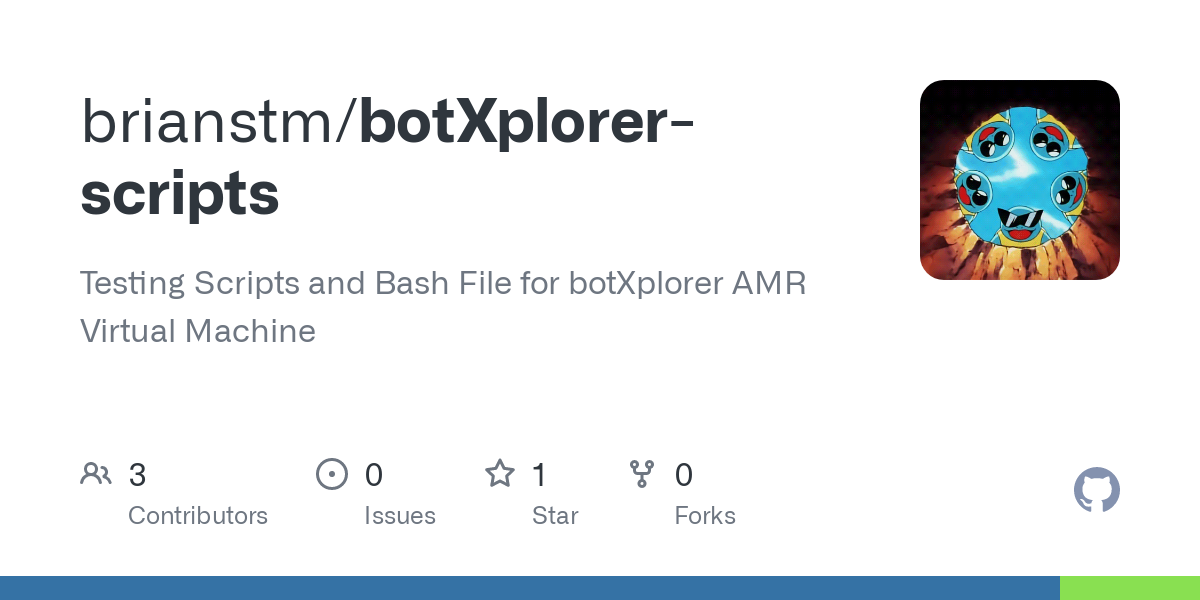
//with the format yyyy-mm-dd hh:mm:ss / "%Y-%m-%d %H:%M:%S"

When you do so, it is advisable to add a slight time delay to the current time. For instance, if the current time is 12:00:00, consider setting the time to 12:00:10. This allows you to press enter precisely when the clock reaches 12:00:10, ensuring accurate time synchronization. It is important to note that you might also be prompted to enter the password for the Ubuntu system during this process.

**Current Time: 2024-04-03 13:59:16**

# 📈Collision Detection

Collision Detection, the first phase of the AMR is to complete the collision detection. Below is the code for the Collision Detection. It is also available on the GitHub Repository:

[[](https://github.com/brianstm/botXplorer_scripts.git)](https://github.com/brianstm/botXplorer_scripts.git" \t "_blank)

## [GitHub - brianstm/botXplorer-scripts: Testing Scripts and Bash File for botXplorer AMR Virtual Machine](https://github.com/brianstm/botXplorer_scripts.git" \t "_blank)

[Testing Scripts and Bash File for botXplorer AMR Virtual Machine - brianstm/botXplorer-scripts](https://github.com/brianstm/botXplorer_scripts.git" \t "_blank)

[https://github.com/brianstm/botXplorer-scripts](https://github.com/brianstm/botXplorer_scripts.git" \t "_blank)

#!/usr/bin/env python

import rospy

from sick\_scan.msg import LIDoutputstateMsg

from geometry\_msgs.msg import Twist

className ObstacleDetector:

def \_\_init\_\_(self):

self.cmd\_pub = rospy.Publisher('/cmd\_vel', Twist, queue\_size=1)

self.lidar\_sub = rospy.Subscriber('/sick\_tim\_7xx/lidoutputstate', LIDoutputstateMsg, self.lidar\_callback)

self.twist = Twist()

self.obstacle\_field1 = False

self.obstacle\_field2 = False

self.last\_obstacle\_field1 = False

def lidar\_callback(self, data):

lidar\_output\_state = data.output\_state

print("0:",data.output\_state[0])

print("1:",data.output\_state[1])

print("2:",data.output\_state[2])

print("3:",data.output\_state[3])

if (lidar\_output\_state[0]==0):

self.obstacle\_field1 = True

else:

self.obstacle\_field1 = False

if (lidar\_output\_state[1]==0):

self.obstacle\_field2 = True

else:

self.obstacle\_field2 = False

def drive(self):

rate = rospy.Rate(10)

while not rospy.is\_shutdown():

if not self.obstacle\_field1 and self.last\_obstacle\_field1:

time\_since\_clear = rospy.get\_time() - self.obstacle\_field1\_time

if time\_since\_clear >= 5:

self.twist.linear.x = 0.7

elif self.obstacle\_field1:

if not self.last\_obstacle\_field1:

self.obstacle\_field1\_time = rospy.get\_time()

self.twist.linear.x = 0

elif self.obstacle\_field2:

self.twist.linear.x = 0.2

else:

self.twist.linear.x = 0.7

self.cmd\_pub.publish(self.twist)

rate.sleep()

self.last\_obstacle\_field1 = self.obstacle\_field1

if \_\_name\_\_ == '\_\_main\_\_':

rospy.init\_node('obstacle\_detector', anonymous=True)

obstacle\_detector = ObstacleDetector()

obstacle\_detector.drive()

TheObstacleDetectorclassName is created to detect obstacles using the LIDAR sensor data. Thelidar\_callbackfunction is called every time new LIDAR data is received, and it checks if there is any obstacle in the two fields of view (obstacle\_field1andobstacle\_field2) by looking at theoutput\_stateof the LIDAR sensor.

If theoutput\_stateof the LIDAR sensor is 0 inobstacle\_field1, it means there is an obstacle in that field, and theobstacle\_field1variable is set toTrue. Similarly, if theoutput\_stateof the LIDAR sensor is 0 inobstacle\_field2, it means there is an obstacle in that field, and theobstacle\_field2variable is set toTrue.

Thedrive()method is the main loop of the node that controls the robot's movement based on the obstacle detection information. It runs at a frequency of 10 Hz, controlled by therospy.Rateobject.

Inside the loop, the method checks the values ofself.obstacle\_field1andself.obstacle\_field2. Ifself.obstacle\_field1isTrueandself.last\_obstacle\_field1isFalse, it means the robot has just detected an obstacle in front of it, so it records the current time usingrospy.get\_time()inself.obstacle\_field1\_time. Ifself.obstacle\_field1isFalseandself.last\_obstacle\_field1isTrue, it means the robot has just cleared the obstacle, so the method calculates the time elapsed since the obstacle was cleared usingrospy.get\_time() - self.obstacle\_field1\_time, and checks if it has been more than 5 seconds. If it has, the robot moves forward with a speed of 0.7 m/s by settingself.twist.linear.xto 0.7. This behavior corresponds to the robot moving forward after clearing an obstacle.

Ifself.obstacle\_field1isTrue, it means the robot is facing an obstacle, so it stops by settingself.twist.linear.xto 0. This behavior corresponds to the robot stopping when it detects an obstacle.

Ifself.obstacle\_field2isTrue, it means the robot is facing an obstacle on its left side, so it moves to the right by settingself.twist.linear.xto 0.2. This behavior corresponds to the robot avoiding an obstacle on its left side.

If none of the above conditions is met, it means the robot has a clear path ahead, so it moves forward with a speed of 0.7 m/s by settingself.twist.linear.xto 0.7. This behavior corresponds to the robot moving forward in a clear path.

Finally, the method publishes theself.twistmessage to the/cmd\_veltopic using theself.cmd\_pubpublisher, and updatesself.last\_obstacle\_field1with the current value ofself.obstacle\_field1.

The code does not explicitly implement collision detection, as it is only checking for the presence of obstacles using LIDAR data. However, by setting the robot's linear velocity to 0 or reducing it when an obstacle is detected, the code is effectively preventing the robot from colliding with any obstacles in its path.

# 🗺️Map

### **Mapping**

To begin the mapping process, there are a few prerequisites that need to be enabled, and rviz must also be started.

* roslaunch magni\_lidar magni\_lidar\_mapmaker.launch

This command launches the ROS node responsible for creating a map using the LiDAR sensor data on the Magni robot.

* rosrun gmapping slam\_gmapping scan:=scan

This command runs the gmapping package's node that performs SLAM (Simultaneous Localization and Mapping) using laser scanner data from the "scan" topic.

* rosrun map\_server map\_saver -f mynewmap-ils
* // mynewmap-ils can be of anyname <name>-ils

This command runs the map\_server node, which reads a map from the "/map" topic and saves it to a file named "mynewmap-ils" in PGM and YAML formats.

* rosrun rviz rviz -d mapmaker.rviz

This command runs the rivz using the configuration file mapmaker.rviz.

[mapmaker.rviz8KB](https://amr-docs-brianstm.vercel.app/files/mapmaker.rviz)

# 🤓AMCL

### **AMCL**

AMCL (Adaptive Monte Carlo Localization) is an advanced localization algorithm widely used in robotics. It plays a vital role in accurately determining the position of a robot within a given environment. AMCL employs the Monte Carlo method, a probabilistic technique, to estimate the robot's position based on sensor measurements and a pre-existing map. By utilizing a particle filter approach, AMCL generates a set of particles that represent potential robot poses. These particles are propagated and weighted based on sensor data, allowing the algorithm to adapt and converge towards the most likely robot position. With its adaptive nature, AMCL can handle uncertainties and handle dynamic environments effectively. It provides reliable and robust localization capabilities, enabling robots like the Magni to navigate autonomously with precise knowledge of their location within the environment.

💡

The name "Adaptive Monte Carlo Localization" (AMCL) originated from the combination of two key concepts: "Monte Carlo Localization" and "Adaptive Localization." Monte Carlo method is a statistical technique that uses random sampling to estimate a system's behavior or state, while Adaptive Localization refers to the ability of a localization algorithm to dynamically adjust its parameters or behavior based on changing environmental conditions or system requirements.

[[amcl - ROS Wiki](http://wiki.ros.org/amcl)](http://wiki.ros.org/amcl" \t "_blank)

## [amcl - ROS Wiki](http://wiki.ros.org/amcl" \t "_blank)

[amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map.](http://wiki.ros.org/amcl" \t "_blank)

[http://wiki.ros.org/amcl](http://wiki.ros.org/amcl" \t "_blank)

### **Nerd Shit**

1. **Initliaztion:**
   * AMCL requires a known map of the environment to begin. The map can be generated using SLAM (Simultaneous Localization and Mapping) algorithms like gmapping.
   * The initial pose estimate of the robot is also required. This can be provided manually or estimated using other localization techniques.
2. **Particle Filter Representation:**
   * AMCL utilizes a particle filter to represent the probability distribution of the robot's pose.
   * The particle filter maintains a set of particles, where each particle represents a possible hypothesis of the robot's pose (position and orientation) in the environment.
   * Initially, the particles are uniformly distributed over the map.
3. **Motion Model:**
   * The motion model predicts the new pose of each particle based on the robot's motion information, usually obtained from wheel encoders or other odometry sensors.
   * The motion model takes into account the control input (e.g., velocity commands) and estimates the displacement and rotation of the robot.
   * However, due to sensor noise and uncertainties, the predicted poses are not entirely accurate, resulting in a spread of particles.
4. **Sensor Model:**
   * When the robot receives sensor measurements (e.g., laser scans from the SICK TiM781 lidar), the sensor model is used to evaluate the likelihood of each particle being the true robot pose.
   * The sensor model compares the expected sensor measurements based on each particle's pose to the actual sensor measurements obtained from the robot's sensors.
   * It assigns a weight to each particle, indicating how well it aligns with the sensor measurements. Particles that generate measurements consistent with the actual measurements receive higher weights.
5. **Weight Normalization:**
   * After evaluating the likelihood of each particle, the weights need to be normalized to ensure they sum up to 1.
   * Normalization allows the particles with higher weights to have a proportionally larger representation in the subsequent resampling step.
6. **Resampling:**
   * Resampling is a crucial step in particle filters, as it determines which particles survive and which ones are replaced.
   * Particles with higher weights have a higher chance of being selected, while particles with lower weights are more likely to be discarded.
   * Resampling concentrates particles around areas that align well with the sensor measurements, improving the accuracy of the pose estimate.
   * Resampling methods can vary, but a common approach is to use the resampling wheel or stochastic universal sampling.
7. **Localization:**
   * The final pose estimate is typically computed as the weighted average of the resampled particles' poses.
   * This estimate represents the robot's localized pose in the map at a given time step.
   * The pose estimate becomes more accurate as the algorithm iterates and refines the particle distribution through the motion and sensor model updates.
8. **Iteration:**
   * Steps 3-7 are repeated as the robot moves and receives new sensor measurements.
   * Iteration helps in continuously updating and improving the pose estimate based on the incoming data.
   * Iteration helps in continuously updating and improving the pose estimate based on the incoming data.

By repeating the steps of motion model prediction, sensor model evaluation, weight normalization, resampling, and localization, AMCL progressively refines the particle distribution, enabling accurate localization of the robot in the environment using the SICK TiM781 lidar sensor and the map generated by gmapping.

# 🏃‍♂️Navigation Stack

### **Navigation Stack**

The Navigation Stack is a collection of ROS packages that work together to enable a robot to navigate autonomously within a given environment. It provides a comprehensive framework for path planning, obstacle avoidance, and localization. The Navigation Stack consists of several components, each serving a specific purpose.

[[navigation - ROS Wiki](http://wiki.ros.org/navigation)](http://wiki.ros.org/navigation" \t "_blank)

## [navigation - ROS Wiki](http://wiki.ros.org/navigation" \t "_blank)

[A 2D navigation stack that takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base.](http://wiki.ros.org/navigation" \t "_blank)

[http://wiki.ros.org/navigation](http://wiki.ros.org/navigation" \t "_blank)

### **Nerd Shit**

1. **Global Planner:**
   * The Global Planner is responsible for generating a high-level global path from the robot's current position to the desired goal position.
   * It takes into account the environment's map, considering obstacles and other constraints, to plan an optimal or feasible path.
   * Common algorithms used in global planning include Dijkstra's algorithm, A\*, or probabilistic roadmaps (PRM).
2. **Particle Filter Representation:**
   * AMCL utilizes a particle filter to represent the probability distribution of the robot's pose.
   * The particle filter maintains a set of particles, where each particle represents a possible hypothesis of the robot's pose (position and orientation) in the environment.
   * Initially, the particles are uniformly distributed over the map.
3. **Local Planner:**
   * The Local Planner focuses on generating a low-level, dynamically feasible trajectory for the robot to follow within a local vicinity.
   * It operates based on the global plan provided by the Global Planner and takes into account the robot's current sensor data.
   * The Local Planner considers obstacles, dynamic changes in the environment, and the robot's kinematic constraints to compute a safe and smooth trajectory.
   * Common algorithms used in local planning include the Dynamic Window Approach, Timed Elastic Band, or trajectory optimization techniques like Model Predictive Control (MPC).
4. **Costmap:**
   * The Costmap is a grid representation of the environment, where each cell contains information about the traversability and cost associated with moving through that cell.
   * There are two types of costmaps: the global costmap, covering the entire environment, and the local costmap, covering the immediate surroundings of the robot.
   * The costmap incorporates sensor data, such as laser scans or point clouds, to update the obstacle information and dynamically adjust the costs.
   * The costmap is used by both the Global Planner and Local Planner to make informed decisions about path planning and obstacle avoidance.
5. **Map server:**
   * The Map Server provides the static map of the environment to the Navigation Stack.
   * It loads and serves the map, which is typically generated using SLAM algorithms like gmapping or acquired from an external source.
   * The map contains information about obstacles, free space, and other relevant features necessary for navigation.
6. **Localization:**
   * Localization is responsible for estimating the robot's pose (position and orientation) within the known map.
   * It uses sensor data, such as odometry, GPS, or visual odometry, along with the map, to determine the robot's position.
   * Common localization approaches include Monte Carlo Localization (MCL), Extended Kalman Filter (EKF), or particle filters like [AMCL (Adaptive Monte Carlo Localization)](https://amr-docs-brianstm.vercel.app/amcl). We are using the [AMCL](https://amr-docs-brianstm.vercel.app/amcl) package for this project.
7. **Trajectory Execution:**
   * Once the Local Planner generates a trajectory, the Trajectory Execution component takes care of executing the planned path on the robot.
   * It sends control commands, such as velocity or motor commands, to the robot's actuators to drive it along the trajectory.
   * The Trajectory Execution component monitors the robot's state and adjusts the commands as necessary to maintain the desired trajectory.

These components work together in a coordinated manner to enable autonomous navigation. The Global Planner generates a high-level plan, the Local Planner refines it into a low-level trajectory, the Costmap provides obstacle information, the Localization module estimates the robot's pose, and the Trajectory Execution component executes the planned trajectory. This integration allows the robot to autonomously navigate while avoiding obstacles and reaching its goal.

### **Parameters and Configuration**

The Navigation Stack requires multiple parameters to be configured, below are all the parameters that we have configured for our robot.

1. **base\_local\_planner\_params.yaml:**

The base\_local\_planner\_params.yaml file is responsible for configuring the behavior of the local planner component within the Navigation Stack. This file contains parameters that define the specific algorithm used by the local planner, such as the Dynamic Window Approach or Timed Elastic Band. The parameters in this file control the behavior of the chosen algorithm, including settings related to trajectory optimization, obstacle avoidance, and the robot's kinematic constraints. By modifying these parameters, you can fine-tune the behavior and performance of the local planner to match the requirements of your robot and its operating environment.

1. **global\_costmap\_params.yaml**

The global\_costmap\_params.yaml file contains parameters specific to the global costmap component within the Navigation Stack. The global costmap is responsible for high-level path planning and representing obstacles at a global scale. This file includes parameters such as the global frame ID, update frequency, and settings related to the map source, inflation, and obstacle layer. It also defines parameters for global planners like Dijkstra's algorithm or A\*, including weights, heuristics, and search limits. By modifying these parameters, you can configure the behavior of the global costmap, including the update rate, the map source (such as a static map or a SLAM-generated map), the characteristics of the global planning algorithm, and the representation and inflation of obstacles in the global costmap.

1. **costmap\_common\_params.yaml:**

The costmap\_common\_params.yaml file contains parameters that are shared between both the global and local costmaps. These parameters define common settings related to the costmap, such as the size and resolution of the costmap grid, the inflation radius for obstacle inflation, and characteristics of the robot's footprint and padding. It also defines the layers of the costmap, such as the obstacle layer and the inflation layer, along with their respective parameters. By modifying the parameters in this file, you can customize the size and resolution of the costmap grid, adjust obstacle representation and inflation, and configure other properties shared between the global and local costmaps.

1. **local\_costmap\_params.yaml:**

The local\_costmap\_params.yaml file contains parameters specific to the local costmap component within the Navigation Stack. The local costmap represents the immediate surroundings of the robot and is used for local path planning and obstacle avoidance. This file includes parameters such as the local frame ID, update frequency, and settings related to the map source, inflation, and obstacle layer. It also includes parameters specific to local planners like the Dynamic Window Approach or Timed Elastic Band. By modifying these parameters, you can customize the behavior of the local costmap, including the update rate, the map source, the representation and inflation of obstacles, and the settings specific to the chosen local planning algorithm.

Since I don't think you have understood anything from the above, I'll make a simple monkey analogy:

Imagine you have a monkey who needs to navigate through a jungle. To help the monkey, you have a bunch of tools and instructions. The base\_local\_planner\_params.yaml file is like a manual that tells the monkey how to move, avoid obstacles, and find the best path in a small area of the jungle. It has settings for the monkey's movements and behavior.

The costmap\_common\_params.yaml file is like a blueprint of the jungle. It tells the monkey about the size of the jungle, where the obstacles are, and how to give them some space. It helps the monkey understand the overall structure of the jungle.

The global\_costmap\_params.yaml file is like a map that shows the monkey the entire jungle and where to find the goal. It has details on the entire jungle, obstacles, and the best paths to take. It helps the monkey plan its long-term journey through the jungle.

The local\_costmap\_params.yaml file is like a magnifying glass that helps the monkey see what's nearby. It focuses on a small area around the monkey and shows the nearby obstacles and paths to avoid them. It helps the monkey make quick decisions to avoid immediate obstacles.

By adjusting the settings in these files, you can teach the monkey how to move, plan paths, and avoid obstacles in the jungle. Each file has its own purpose and helps the monkey navigate the jungle safely and efficiently.

### **Time Elastic Band (TEB)**

[[teb_local_planner - ROS Wiki](http://wiki.ros.org/teb_local_planner)](http://wiki.ros.org/teb_local_planner" \t "_blank)

## [teb\_local\_planner - ROS Wiki](http://wiki.ros.org/teb_local_planner" \t "_blank)

[The ROS Wiki is for ROS 1. Are you using ROS 2 (Humble, Iron, or Rolling)? Check out the ROS 2 Project DocumentationPackage specific documentation can be found on index.ros.org](http://wiki.ros.org/teb_local_planner" \t "_blank)

[http://wiki.ros.org/teb\_local\_planner](http://wiki.ros.org/teb_local_planner" \t "_blank)

The TEB (Timed Elastic Band) Local Planner is a path planning and control algorithm commonly used in robot navigation systems. It is designed to generate dynamically feasible trajectories for a robot to follow in order to reach a goal while avoiding obstacles. The planner takes into account the robot's kinematic constraints, such as maximum velocities and accelerations, to produce smooth and efficient paths.  
TEB Local Planner uses a planning horizon divided into discrete time steps. It optimizes the trajectory by adjusting the robot's velocities and accelerations at each time step while considering various costs, such as proximity to obstacles and adherence to kinematic constraints. The planner aims to find the optimal trajectory that minimizes these costs while satisfying the given constraints.  
The TEB Local Planner also supports features like homotopy class planning, which allows exploration of multiple feasible paths, and costmap conversion, which converts costmap data into more suitable representations for planning.

### **Costmap Conversion**

[[teb_local_planner/Tutorials/Costmap conversion - ROS Wiki](http://wiki.ros.org/teb_local_planner/Tutorials/Costmap%20conversion)](http://wiki.ros.org/teb_local_planner/Tutorials/Costmap%20conversion" \t "_blank)

## [teb\_local\_planner/Tutorials/Costmap conversion - ROS Wiki](http://wiki.ros.org/teb_local_planner/Tutorials/Costmap%20conversion" \t "_blank)

[The ROS Wiki is for ROS 1. Are you using ROS 2 (Humble, Iron, or Rolling)? Check out the ROS 2 Project DocumentationPackage specific documentation can be found on index.ros.org](http://wiki.ros.org/teb_local_planner/Tutorials/Costmap%20conversion" \t "_blank)

[http://wiki.ros.org/teb\_local\_planner/Tutorials/Costmap%20conversion](http://wiki.ros.org/teb_local_planner/Tutorials/Costmap%20conversion" \t "_blank)

Costmap Conversion refers to the process of transforming raw sensor data, such as laser scans or point clouds, into a representation that can be used for planning and obstacle avoidance. In robot navigation, costmaps are grids that represent the environment around the robot, where each cell contains information about the presence or absence of obstacles.  
During costmap conversion, sensor data is processed and analyzed to determine the occupancy or cost values associated with different areas in the environment. For example, obstacles detected by laser scans may be marked as occupied cells in the costmap, while free spaces are marked as unoccupied or low-cost cells.  
Costmap conversion techniques can include clustering algorithms to group nearby points into obstacles, fitting lines or polygons to represent obstacles more efficiently, and applying inflation to create a safety buffer around obstacles.  
The converted costmap is then used by path planning algorithms, like the TEB Local Planner, to generate collision-free paths for the robot. By converting sensor data into a costmap representation, the planner can make informed decisions on how to navigate the environment and avoid obstacles effectively.

# 🧭Navigation

### **Navigation**

Once the map of the surroundings has been created, the AMR requires a precise understanding of its position relative to the map to navigate and traverse the environment effectively. This is where Localization comes in.

### **Localization**

Localization is a crucial process that entails accurately determining the position of a robot within a specific map. This is achieved by leveraging various sensors, including LiDAR, cameras, and odometry, which enable a comparison between the robot's current position and the map. In our scenario, we will employ the AMCL (Adaptive Monte Carlo Localization) package to precisely localize the Magni robot.

### **AMCL**

[AMCL (Adaptive Monte Carlo Localization)](https://amr-docs-brianstm.vercel.app/amcl) is an advanced localization algorithm widely used in robotics. It plays a vital role in accurately determining the position of a robot within a given environment.

### **Navigation Stack**

[Navigation Stack](https://amr-docs-brianstm.vercel.app/navstack) is a collection of ROS packages that provide the ability to navigate a robot within a given environment.

### **Magical package**

To establish a comprehensive navigation system, seamless communication is required between several components. These include the amcl, the navigation stack, the base\_link (representing the robot's body), the base\_footprint(representing the robot's wheels) cmd\_vel (representing wheels movement), and the map. In order to seamlessly integrate these components, we will create a magical package that will effectively bridge the gap and ensure smooth coordination among them, resulting in a harmonious and functional navigation system.

In order to make the package, we will make a new package called navigation (you can name it whatever you want).

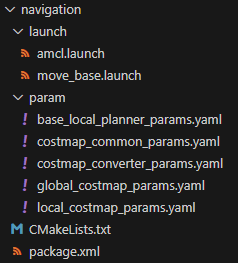
cd ~/catkin\_ws/src

catkin\_create\_pkg navigation roscpp rospy std\_msgs move\_base

cd navigation

mkdir launch param

Folder Tree



move\_base.launch

[move\_base.launch3KB](https://amr-docs-brianstm.vercel.app/files/move_base.launch)

<launch>

<!-- Run the map server -->

<arg name="map\_file" default="$(find map\_pkg)/maps/sickmap-ils.yaml" />

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(arg map\_file)" />

<!--- Run AMCL -->

<include file="$(find navigation)/launch/amcl.launch" />

<!--- Basic TF Transform -->

<node pkg="tf" type="static\_transform\_publisher" name="base\_footprint\_to\_base\_link"

args="0 0 0 0 0 0 base\_footprint base\_link 100" />

<node pkg="tf" type="static\_transform\_publisher" name="base\_link\_to\_cloud"

args="0 0 0 0 0 0 base\_link cloud 20" />

<!--- Move Base Package -->

<node pkg="move\_base" type="move\_base" respawn="false" name="move\_base" output="screen">

<rosparam file="$(find navigation\_pkg)/param/costmap\_common\_params.yaml" command="load" ns="global\_costmap" />

<rosparam file="$(find navigation\_pkg)/param/costmap\_common\_params.yaml" command="load" ns="local\_costmap" />

<rosparam file="$(find navigation\_pkg)/param/local\_costmap\_params.yaml" command="load" />

<rosparam file="$(find navigation\_pkg)/param/global\_costmap\_params.yaml" command="load" />

<rosparam file="$(find navigation\_pkg)/param/base\_local\_planner\_params.yaml" command="load" />

<rosparam file="$(find navigation\_pkg)/param/costmap\_converter\_params.yaml" command="load" />

<param name="base\_local\_planner" value="teb\_local\_planner/TebLocalPlannerROS" />

<param name="controller\_frequency" value="10.0" />

</node>

</launch>

The launch file is used to configure and launch various nodes for the navigation system.

In the code, the launch tag signifies the start of the launch file. The first section includes the include tag, which references and runs another launch file named amcl.launchfrom the navigation package. This allows for the inclusion and execution of the AMCL node, which is responsible for localization.

The subsequent node tags define several nodes that contribute to the navigation system. The static\_transform\_publisher nodes establish static transformations between different coordinate frames, such asbase\_footprint and base\_link, as well as base\_link and cloud. These transformations aid in the correct spatial alignment of different components.

The main node defined in the code is the node tag with the package move\_base and type move\_base. This node represents the core component responsible for the robot's navigation. It loads various parameter files using the rosparam tag, including the common costmap parameters, global costmap parameters, local costmap parameters, move\_base parameters, base\_local\_planner parameters, and AMCL parameters. These parameter files contain configurations for the costmaps, planning algorithms, and other navigation-related settings.

This launch file sets up and runs the necessary nodes for the navigation system, including AMCL, static transformations, and move\_base, with various parameter files providing customized configurations for different components of the navigation system.

amcl.launch

[amcl.launch3KB](https://amr-docs-brianstm.vercel.app/files/amcl.launch)

<launch>

<node pkg="amcl" type="amcl" name="amcl" output="screen">

<param name="initial\_pose\_x" value="0.0" />

<param name="initial\_pose\_y" value="0.0" />

<param name="initial\_pose\_a" value="0.0" />

<param name="initial\_cov\_xx" value="1.0" />

<param name="initial\_cov\_yy" value="1.0" />

<param name="initial\_cov\_aa" value="0.1" />

<param name="odom\_model\_type" value="diff" />

<param name="transform\_tolerance" value="1.0" />

<param name="gui\_publish\_rate" value="10.0" />

<param name="laser\_max\_beams" value="1000" />

<param name="min\_particles" value="500" />

<param name="max\_particles" value="2000" />

<param name="kld\_err" value="0.01" />

<param name="kld\_z" value="0.99" />

<param name="odom\_alpha1" value="0.8" />

<param name="odom\_alpha2" value="0.8" />

<param name="odom\_alpha3" value="0.2" />

<param name="odom\_alpha4" value="0.2" />

<param name="laser\_z\_hit" value="0.95" />

<param name="laser\_z\_short" value="0.05" />

<param name="laser\_z\_max" value="0.05" />

<param name="laser\_z\_rand" value="0.05" />

<param name="laser\_sigma\_hit" value="0.2" />

<param name="laser\_lambda\_short" value="0.1" />

<param name="laser\_min\_range" value="0.08" />

<param name="laser\_max\_range" value="5.0" />

<param name="laser\_model\_type" value="likelihood\_field" />

<param name="laser\_likelihood\_max\_dist" value="5.0" />

<param name="update\_min\_d" value="0.1" />

<param name="update\_min\_a" value="0.1" />

<param name="odom\_frame\_id" value="odom" />

<param name="base\_frame\_id" value="base\_footprint" />

<param name="global\_frame\_id" value="map" />

<param name="resample\_interval" value="1" />

<param name="recovery\_alpha\_slow" value="0.2" />

<param name="recovery\_alpha\_fast" value="0.4" />

</node>

</launch>

The launch file is used to configure and launch nodes for the navigation system, specifically for the AMCL (Adaptive Monte Carlo Localization) algorithm. node responsible for publishing the map. It uses the map\_server package and loads a specific YAML map file.

The next node is the amcl node, which is the main node for the AMCL algorithm. It handles localization using the provided map. The remap tags remap certain topics like scan and map to the same names, as it may find other package for the scan and map.

The following param This launch file sets up the necessary nodes and parameters for the AMCL algorithm to perform localization using a provided map and sensor data.

initial\_pose\_x: Sets the initial x-coordinate of the robot's pose to 0.0.  
initial\_pose\_y: Sets the initial y-coordinate of the robot's pose to 0.0.  
initial\_pose\_a: Sets the initial orientation (yaw angle) of the robot's pose to 0.0 radians.  
initial\_cov\_xx: Sets the initial covariance value for the x-coordinate of the robot's pose to 1.0.  
initial\_cov\_yy: Sets the initial covariance value for the y-coordinate of the robot's pose to 1.0.  
initial\_cov\_aa: Sets the initial covariance value for the orientation (yaw angle) of the robot's pose to 0.1.  
odom\_model\_type: Specifies the type of odometry model to use for motion prediction. In this case, it is set to "diff" for differential drive.  
transform\_tolerance: Sets the tolerance (in seconds) for time-based transformations.  
gui\_publish\_rate: Sets the rate (in Hz) at which the amcl node publishes GUI visualization markers.  
laser\_max\_beams: Sets the maximum number of laser beams used for laser-based particle filtering.  
min\_particles: Sets the minimum number of particles to use in the particle filter.  
max\_particles: Sets the maximum number of particles to use in the particle filter.  
kld\_err: Sets the error threshold used for adaptive resampling of particles in the KLD sampling algorithm.  
kld\_z: Sets the confidence threshold used for adaptive resampling of particles in the KLD sampling algorithm.  
odom\_alpha1: Sets the covariance parameter for the rotational motion noise in the odometry model.  
odom\_alpha2: Sets the covariance parameter for the translational motion noise in the odometry model.  
odom\_alpha3: Sets the covariance parameter for the rotational motion noise in the odometry model (additional parameter).  
odom\_alpha4: Sets the covariance parameter for the translational motion noise in the odometry model (additional parameter).  
laser\_z\_hit: Sets the probability of a laser hit being true given the predicted measurement.  
laser\_z\_short: Sets the probability of a short laser reading being true given the predicted measurement.  
laser\_z\_max: Sets the probability of a maximum laser reading being true given the predicted measurement.  
laser\_z\_rand: Sets the probability of a random laser reading being true given the predicted measurement.  
laser\_sigma\_hit: Sets the standard deviation of the Gaussian distribution used for weighting the likelihood of laser hits.  
laser\_lambda\_short: Sets the decay rate of the exponential function used for weighting the likelihood of short laser readings.  
laser\_min\_range: Sets the minimum range (in meters) for laser readings to be considered valid.  
laser\_max\_range: Sets the maximum range (in meters) for laser readings to be considered valid.  
laser\_model\_type: Specifies the type of laser model to use for likelihood field-based laser matching.  
laser\_likelihood\_max\_dist: Sets the maximum distance (in meters) for the laser likelihood field used for laser-based localization.  
update\_min\_d: Sets the minimum distance (in meters) the robot needs to move before performing a filter update.  
update\_min\_a: Sets the minimum angle (in radians) the robot needs to rotate before performing a filter update.  
odom\_frame\_id: Sets the frame ID used for the odometry data.  
base\_frame\_id: Sets the frame ID used for the base of the robot.  
global\_frame\_id: Sets the frame ID used for the global map.  
resample\_interval: Sets the interval (in filter updates) for resampling particles in the particle filter.  
recovery\_alpha\_slow: Sets the rate at which the filter recovers from a failure when in slow recovery mode.  
recovery\_alpha\_fast: Sets the rate at which the filter recovers from a failure when in fast recovery mode.

base\_local\_planner\_params.yaml

[base\_local\_planner\_params.yaml1KB](https://amr-docs-brianstm.vercel.app/files/base_local_planner_params.yaml)

TebLocalPlannerROS:

odom\_topic: odom

map\_frame: map

teb\_autosize: **True**

dt\_ref: 0.3

dt\_hysteresis: 0.1

global\_plan\_overwrite\_orientation: **True**

max\_global\_plan\_lookahead\_dist: 3.0

feasibility\_check\_no\_poses: 5

max\_vel\_x: 0.4

max\_vel\_x\_backwards: 0.2

max\_vel\_theta: 0.3

acc\_lim\_x: 0.5

acc\_lim\_theta: 0.5

min\_turning\_radius: 0.0

footprint\_model:

type: "polygon"

radius: 0.2

line\_start: [-0.3, 0.0]

line\_end: [0.3, 0.0]

front\_offset: 0.2

front\_radius: 0.2

rear\_offset: 0.2

rear\_radius: 0.2

vertices:

[

[0.2087, 0.21955],

[0.2087, -0.21955],

[-0.2087, -0.21955],

[-0.2087, 0.21955],

]

xy\_goal\_tolerance: 0.2

yaw\_goal\_tolerance: 0.1

free\_goal\_vel: **False**

min\_obstacle\_dist: 0.4

include\_costmap\_obstacles: **True**

costmap\_obstacles\_behind\_robot\_dist: 1.0

obstacle\_poses\_affected: 30

costmap\_converter\_plugin: ""

costmap\_converter\_spin\_thread: **True**

costmap\_converter\_rate: 5

no\_inner\_iterations: 5

no\_outer\_iterations: 4

optimization\_activate: **True**

optimization\_verbose: **False**

penalty\_epsilon: 0.1

weight\_max\_vel\_x: 2

weight\_max\_vel\_theta: 1

weight\_acc\_lim\_x: 1

weight\_acc\_lim\_theta: 1

weight\_kinematics\_nh: 1000

weight\_kinematics\_forward\_drive: 1

weight\_kinematics\_turning\_radius: 1

weight\_optimaltime: 1

weight\_obstacle: 50

weight\_dynamic\_obstacle: 10

selection\_alternative\_time\_cost: **False**

enable\_homotopy\_class\_planning: **True**

enable\_multithreading: **True**

simple\_exploration: **False**

max\_number\_classes: 4

roadmap\_graph\_no\_samples: 15

roadmap\_graph\_area\_width: 5

h\_signature\_prescaler: 0.5

h\_signature\_threshold: 0.1

obstacle\_keypoint\_offset: 0.1

obstacle\_heading\_threshold: 0.45

visualize\_hc\_graph: **False**

odom\_topic: Specifies the name of the topic where odometry information is published.  
map\_frame: Specifies the coordinate frame of the global map.  
teb\_autosize: Enables automatic scaling of the time optimal trajectory.  
dt\_ref: The time resolution (in seconds) used for the discretization of the trajectory.  
dt\_hysteresis: The hysteresis time (in seconds) used for the detection of temporal inconsistencies.  
global\_plan\_overwrite\_orientation: Specifies whether the orientation of the global plan should be overwritten with the current robot orientation.  
max\_global\_plan\_lookahead\_dist: The maximum distance (in meters) to look ahead along the global plan for obtaining a feasible local plan.  
feasibility\_check\_no\_poses: The number of poses along the trajectory to be checked for feasibility.  
max\_vel\_x: The maximum translational velocity (in m/s) in the x-direction.  
max\_vel\_x\_backwards: The maximum translational velocity (in m/s) in the backwards x-direction.  
max\_vel\_theta: The maximum angular velocity (in rad/s) around the z-axis.  
acc\_lim\_x: The maximum translational acceleration (in m/s^2) in the x-direction.  
acc\_lim\_theta: The maximum angular acceleration (in rad/s^2) around the z-axis.  
min\_turning\_radius: The minimum turning radius (in meters) for kinematic constraints.  
footprint\_model: Specifies the shape and dimensions of the robot's footprint.  
xy\_goal\_tolerance: The maximum tolerance (in meters) for the robot's position in the x and y coordinates when reaching the goal.  
yaw\_goal\_tolerance: The maximum tolerance (in radians) for the robot's orientation when reaching the goal.  
free\_goal\_vel: Specifies whether the goal velocity should be set to zero.  
min\_obstacle\_dist: The minimum distance (in meters) to obstacles for the robot to consider while planning.  
include\_costmap\_obstacles: Specifies whether the costmap obstacles should be included in the obstacle checks.  
costmap\_obstacles\_behind\_robot\_dist: The distance (in meters) behind the robot to be considered for costmap obstacles.  
obstacle\_poses\_affected: The number of obstacle poses affected by a single obstacle.  
costmap\_converter\_plugin: Specifies the name of the costmap converter plugin to be used.  
costmap\_converter\_spin\_thread: Enables spinning a separate thread for the costmap converter.  
costmap\_converter\_rate: The rate (in Hz) at which the costmap converter plugin should run.  
no\_inner\_iterations: The number of iterations for optimizing the trajectory without shrinking.  
no\_outer\_iterations: The number of iterations for optimizing the trajectory with shrinking.  
optimization\_activate: Specifies whether the trajectory optimization should be activated.  
optimization\_verbose: Enables verbose output during trajectory optimization.  
penalty\_epsilon: A small value used to prevent division by zero.  
weight\_max\_vel\_x: The weight for maximizing the translational velocity.  
weight\_max\_vel\_theta: The weight for maximizing the angular velocity.  
weight\_acc\_lim\_x: The weight for limiting the translational acceleration.  
weight\_acc\_lim\_theta: The weight for limiting the angular acceleration.  
weight\_kinematics\_nh: The weight for satisfying the non-holonomic kinematics constraints.  
weight\_kinematics\_forward\_drive: The weight for forward-drive kinematics.  
weight\_kinematics\_turning\_radius: The weight for kinematics turning radius.  
weight\_optimaltime: The weight for optimal travel time.  
weight\_obstacle: The weight for avoiding static obstacles.  
weight\_dynamic\_obstacle: The weight for avoiding dynamic obstacles.  
selection\_alternative\_time\_cost: Specifies whether to consider alternative time costs in trajectory selection.  
enable\_homotopy\_class\_planning: Enables homotopy class planning for exploring multiple feasible paths.  
enable\_multithreading: Enables multithreading for faster planning.  
simple\_exploration: Specifies whether to use simple exploration strategies.  
max\_number\_classes: The maximum number of different homotopy classes to explore.  
roadmap\_graph\_no\_samples: The number of samples for creating the roadmap graph.  
roadmap\_graph\_area\_width: The width (in meters) of the area to create the roadmap graph.  
h\_signature\_prescaler: A scaling factor for the h-signature computation.  
h\_signature\_threshold: The threshold value for h-signature similarity.  
obstacle\_keypoint\_offset: The offset (in meters) for obstacle keypoints.  
obstacle\_heading\_threshold: The threshold value for obstacle heading direction.  
visualize\_hc\_graph: Specifies whether to visualize the homotopy class graph.

costmap\_common\_params.yaml

[costmap\_common\_params.yaml1KB](https://amr-docs-brianstm.vercel.app/files/costmap_common_params.yaml)

obstacle\_range: 2.5

raytrace\_range: 3.0

footprint: [[0.4174, 0.43909], [0.4174, -0.43909], [-0.4174, -0.43909], [-0.4174, 0.43909]]

inflation\_radius: 0.55

robot\_radius: 0.6

observation\_sources: laser\_scan\_sensor point\_cloud\_sensor

laser\_scan\_sensor: {sensor\_frame: laser, data\_type: LaserScan, topic: scan, marking: **true**, clearing: **true**}

point\_cloud\_sensor: {sensor\_frame: cloud, data\_type: PointCloud2, topic: cloud, marking: **true**, clearing: **true**}

obstacle\_range: Specifies the maximum range (in meters) at which obstacles are detected by the sensor. Obstacles beyond this range are not considered in the costmap.  
raytrace\_range: Sets the maximum range (in meters) used for raytracing. Raytracing is a technique used to determine if there are obstacles between the sensor and a particular point in space.  
footprint: Defines the robot's footprint as a polygon using its vertices. The footprint represents the physical shape of the robot, and it is used to determine areas of the costmap where the robot cannot pass due to its size.  
inflation\_radius: Specifies the radius (in meters) by which obstacles are inflated in the costmap. Inflation is used to create a buffer around obstacles to ensure safe clearance for the robot.  
robot\_radius: Sets the radius (in meters) of the robot. It is used in combination with the footprint and inflation radius to determine the robot's size in the costmap.  
observation\_sources: Specifies the sources of sensor observations used to update the global costmap. In this case, it includes a laser scan sensor and a point cloud sensor.  
laser\_scan\_sensor: Defines the laser scan sensor configuration. It specifies the sensor frame, data type, ROS topic to subscribe to, and whether the sensor observations should be marked as obstacles or clear space in the costmap.  
point\_cloud\_sensor: Defines the point cloud sensor configuration. It specifies the sensor frame, data type, ROS topic to subscribe to, and whether the sensor observations should be marked as obstacles or clear space in the costmap.

global\_costmap\_params.yaml

[global\_costmap\_params.yaml1KB](https://amr-docs-brianstm.vercel.app/files/global_costmap_params.yaml)

global\_costmap:

global\_frame: map

robot\_base\_frame: base\_link

update\_frequency: 5.0

static\_map: **true**

global\_costmap: Specifies that the following lines define the parameters for the global costmap.  
global\_frame: Sets the coordinate frame for the global costmap to "map". The "map" frame is typically used as the reference frame for the global map.  
robot\_base\_frame: Specifies the coordinate frame attached to the robot's base as "base\_link". This frame represents the origin of the robot.  
update\_frequency: Sets the frequency (in Hz) at which the global costmap should be updated. In this case, it is updated at a rate of 5 times per second.  
static\_map: Indicates that the global costmap should use a static map as a source of information. When set to "true", the global costmap relies on a pre-built map that does not change dynamically.

local\_costmap\_params.yaml

[local\_costmap\_params.yaml1KB](https://amr-docs-brianstm.vercel.app/files/local_costmap_params.yaml)

local\_costmap:

global\_frame: odom

robot\_base\_frame: base\_link

update\_frequency: 5.0

publish\_frequency: 2.0

static\_map: **false**

rolling\_window: **true**

width: 6.0

height: 6.0

resolution: 0.1

local\_costmap: Specifies that the following lines define the parameters for the local costmap.  
global\_frame: Sets the coordinate frame for the local costmap to "odom". The "odom" frame is typically used as the reference frame for odometry information.  
robot\_base\_frame: Specifies the coordinate frame attached to the robot's base as "base\_link". This frame represents the origin of the robot.  
update\_frequency: Sets the frequency (in Hz) at which the local costmap should be updated. In this case, it is updated at a rate of 5 times per second.  
publish\_frequency: Sets the frequency (in Hz) at which the local costmap should be published. In this case, it is published at a rate of 2 times per second.  
static\_map: Indicates that the local costmap should not use a static map as a source of information. When set to "false", the local costmap is primarily based on sensor observations and does not rely on a pre-built map.  
rolling\_window: Enables the rolling window mode for the local costmap. In this mode, the costmap is centered around the robot's current position and moves along with the robot as it navigates.  
width: Specifies the width (in meters) of the local costmap. It determines the extent of the costmap in the x-direction.  
height: Specifies the height (in meters) of the local costmap. It determines the extent of the costmap in the y-direction.  
resolution: Sets the resolution (in meters per cell) of the local costmap. It defines the granularity of the costmap representation, where smaller values provide higher resolution.

costmap\_converter\_params.yaml

[costmap\_converter\_params.yaml1KB](https://amr-docs-brianstm.vercel.app/files/costmap_converter_params.yaml)

TebLocalPlannerROS:

costmap\_converter\_plugin: "costmap\_converter::CostmapToPolygonsDBSMCCH"

# Options for the converter:

# CostmapToPolygonsDBSMCCH / CostmapToLinesDBSRANSAC

# CostmapToLinesDBSRANSAC / CostmapToLinesDBSRANSAC

costmap\_converter\_spin\_thread: **True**

costmap\_converter\_rate: 5

costmap\_converter/CostmapToLinesDBSRANSAC:

cluster\_max\_distance: 0.4

cluster\_min\_pts: 2

ransac\_inlier\_distance: 0.15

ransac\_min\_inliers: 10

ransac\_no\_iterations: 2000

ransac\_remainig\_outliers: 3

ransac\_convert\_outlier\_pts: **True**

ransac\_filter\_remaining\_outlier\_pts: **False**

convex\_hull\_min\_pt\_separation: 0.1

costmap\_converter\_plugin: Specifies the name of the costmap converter plugin to be used. In this case, the plugin is "CostmapToPolygonsDBSMCCH" provided by the "costmap\_converter" package.  
costmap\_converter\_spin\_thread: Enables spinning a separate thread for the costmap converter. This allows the conversion process to run independently in the background.  
costmap\_converter\_rate: The rate (in Hz) at which the costmap converter plugin should run. It specifies how frequently the costmap is converted to polygons or lines.  
costmap\_converter/CostmapToLinesDBSRANSAC: Defines a section for the options specific to the "CostmapToLinesDBSRANSAC" costmap converter.  
cluster\_max\_distance: The maximum distance (in meters) between points to be considered part of the same cluster during clustering.  
cluster\_min\_pts: The minimum number of points required to form a cluster during clustering.  
ransac\_inlier\_distance: The maximum distance (in meters) between a point and a line to be considered an inlier during RANSAC line fitting.  
ransac\_min\_inliers: The minimum number of inliers required for a line to be considered valid during RANSAC line fitting.  
ransac\_no\_iterations: The maximum number of iterations for the RANSAC line fitting algorithm.  
ransac\_remainig\_outliers: The maximum number of remaining outliers allowed after RANSAC line fitting.  
ransac\_convert\_outlier\_pts: Specifies whether to convert the remaining outlier points into lines after RANSAC line fitting.  
ransac\_filter\_remaining\_outlier\_pts: Specifies whether to filter out the remaining outlier points after RANSAC line fitting.  
convex\_hull\_min\_pt\_separation: The minimum separation distance (in meters) between points on the convex hull of an obstacle. Points that are closer than this distance will be removed from the convex hull.

To run the package successfully, navigate to the launch folder and execute the move\_base.launch file. This launch file not only activates the move\_base.launch component but also starts the map\_server and amcl for seamless operation.

cd ~/catkin\_ws/src/magni\_navigation/launch

roslaunch move\_base.launch

### **Rviz**

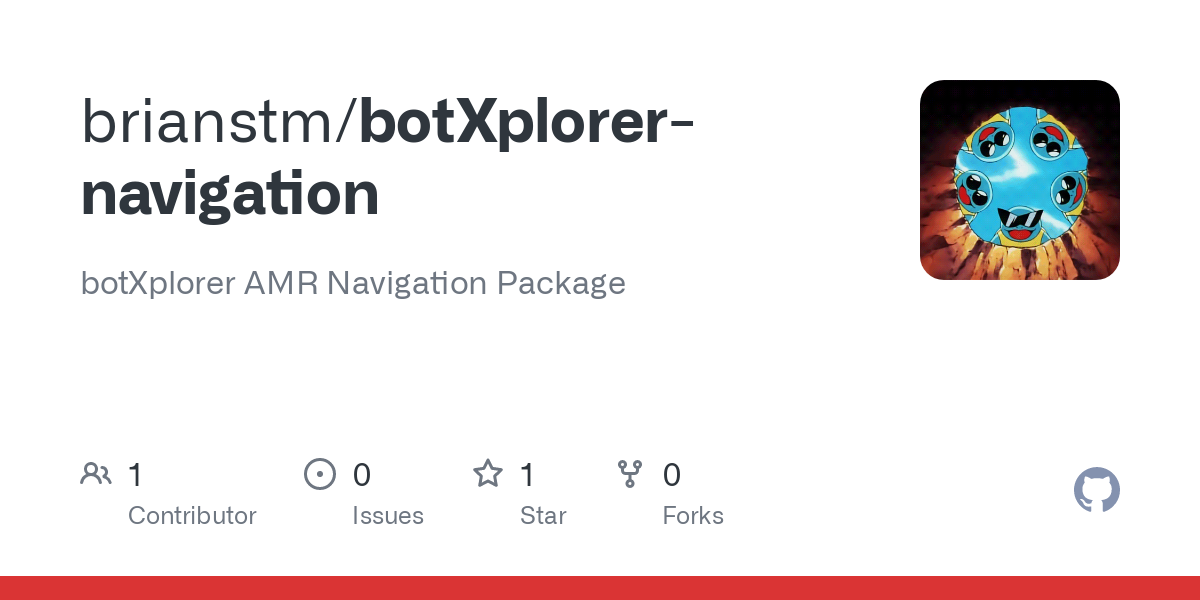
To gain a clear visual understanding of the ongoing processes, it is necessary to activate Rviz—an advanced 3D visualization tool designed for ROS. However, running Rviz directly in the Magni bot terminal may lead to performance issues. Hence, it is crucial to execute Rviz in the Virtual Machine terminal instead. To accomplish this, simply open a new terminal within the Virtual Machine environment and enter the following command:

rosrun rviz rviz

This command launches the versatile ROS visualization tool RViz, which loads a default display configuration. However, this configuration file must be modified to work with both the SICK TiM781, the Magni robot, AMCL, and Navigation Stack. To make this modification, simply click on the File , menu and select Open Config then choose the .rviz file provided below.

[config.rviz11KB](https://amr-docs-brianstm.vercel.app/files/config.rviz)

the updated files can be forked and or pulled from the repository below:

[[](https://github.com/brianstm/botXplorer-navigation.git)](https://github.com/brianstm/botXplorer-navigation.git" \t "_blank)

## [GitHub - brianstm/botXplorer-navigation: botXplorer AMR Navigation Package](https://github.com/brianstm/botXplorer-navigation.git" \t "_blank)

[botXplorer AMR Navigation Package. Contribute to brianstm/botXplorer-navigation development by creating an account on GitHub.](https://github.com/brianstm/botXplorer-navigation.git" \t "_blank)

[https://github.com/brianstm/botXplorer-navigation](https://github.com/brianstm/botXplorer-navigation.git" \t "_blank)

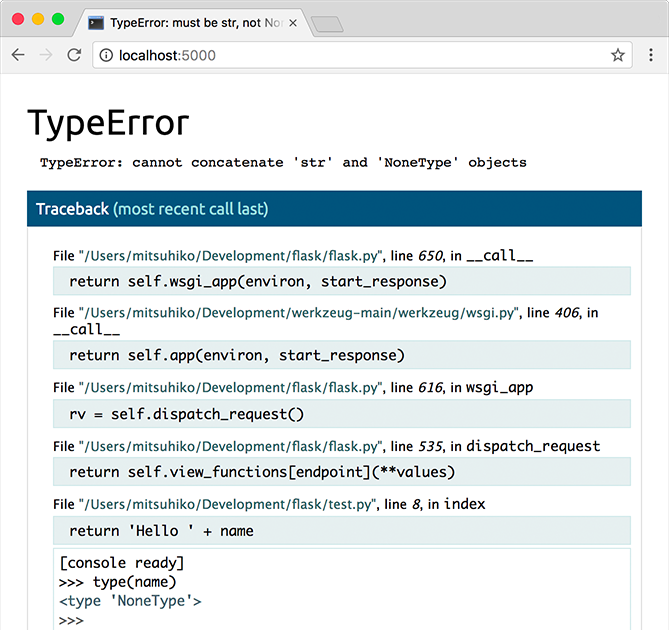
# 👾Flask

### **Flask**

To ensure seamless access to the AMR's system from any location within the network, it was imperative for us to develop a highly versatile application. Thus, we embarked on creating a web application that could cater to our needs. Initially, we considered utilizing a React app in conjunction with Django Python as a backend API to establish a robust infrastructure. However, due to certain practical constraints, implementing the API proved to be unfeasible.

The decision-making process led us to explore alternative solutions, ultimately leading us to leverage Flask—an excellent choice that perfectly aligned with our requirements. By configuring Flask to operate on a specific port, we successfully established accessibility within the network. This implementation allowed us to access the application effortlessly by directing the browser to the local host's IP address and designated port, ensuring a smooth and efficient control mechanism for the AMR.

Additionally, our comprehensive web development efforts involved the creation of a user-friendly website, expertly running on a Raspberry Pi. This setup empowered us to exercise full control over the AMR's movements and navigate it effortlessly by defining waypoints and establishing routes. Furthermore, we incorporated the functionality to dynamically add new points and routes as needed.

[[](https://flask.palletsprojects.com/en/2.3.x/quickstart/)](https://flask.palletsprojects.com/en/2.3.x/quickstart/" \t "_blank)

## [Quickstart — Flask Documentation (2.3.x)](https://flask.palletsprojects.com/en/2.3.x/quickstart/" \t "_blank)

[A minimal Flask application looks something like this:](https://flask.palletsprojects.com/en/2.3.x/quickstart/" \t "_blank)

[https://flask.palletsprojects.com/en/2.3.x/quickstart/](https://flask.palletsprojects.com/en/2.3.x/quickstart/" \t "_blank)

In order to make the package, we will make a new package called flask\_app (you can name it whatever you want).

cd ~/catkin\_ws/src

catkin\_create\_pkg flask\_app roscpp rospy

cd flask\_app

mkdir template static

cd static

mkdir assets

The assets serves as a place for us to store any images necessary for the website (the static is stating that the images will never change hence it is stored in a statuc folder). While the template serves as a place for us to store any the index website.

### **Flask App**

[flask\_app.py10KB](https://amr-docs-brianstm.vercel.app/files/flask_app.py)

from flask import Flask, render\_template, request, redirect, jsonify

import subprocess

import json

import rospy

import os

import time

import logging

import sys

import numpy as np

from geometry\_msgs.msg import PoseWithCovarianceStamped

from actionlib\_msgs.msg import GoalStatusArray

from rospy\_message\_converter import message\_converter

from sensor\_msgs.msg import BatteryState

from geometry\_msgs.msg import PoseStamped

from std\_msgs.msg import Header

from actionlib\_msgs.msg import GoalID

from geometry\_msgs.msg import PoseArray

app = Flask(\_\_name\_\_)

script\_path = os.path.dirname(os.path.abspath(\_\_file\_\_))

filename = os.path.join(script\_path, 'coordinates.txt')

routes\_filename = os.path.join(script\_path, 'routes.txt')

latest\_pose = None

status\_message = {}

coordinates = {}

battery\_state = None

routes = {}

should\_continue\_execution = True

amcl\_covariance = None

particle\_cloud = None

def pose\_callback(msg):

global latest\_pose, amcl\_covariance

latest\_pose = msg

amcl\_covariance = msg.pose.covariance

def status\_callback(msg):

global status\_message

status\_message = message\_converter.convert\_ros\_message\_to\_dictionary(msg)

# rospy.logwarn("Status updated: %s", status\_message)

def battery\_state\_callback(msg):

global battery\_state

battery\_state = {

'voltage': msg.voltage,

'percentage': msg.percentage

}

def cancel\_move\_base():

cancel\_msg = GoalID()

cancel\_msg.stamp = rospy.Time.now()

cancel\_pub.publish(cancel\_msg)

def particlecloud\_callback(data):

global particle\_cloud

particle\_cloud = data.poses

def calculate\_confidence\_level(covariance\_data):

covariance\_data = np.reshape(covariance\_data, (6, 6))

covariance\_position = np.array(covariance\_data[:2, :2])

covariance\_orientation = np.array(covariance\_data[5:6, 5:6])

eigenvalues\_position, \_ = np.linalg.eig(covariance\_position)

eigenvalues\_orientation, \_ = np.linalg.eig(covariance\_orientation)

total\_uncertainty = np.sum(eigenvalues\_position) + np.sum(eigenvalues\_orientation)

confidence\_level = max(0, 100 - total\_uncertainty \* 100)

return confidence\_level

def calculate\_statistics():

global particle\_cloud

if not particle\_cloud:

return None, None, None, None

x\_values = [pose.position.x for pose in particle\_cloud]

y\_values = [pose.position.y for pose in particle\_cloud]

mean\_x = np.mean(x\_values)

mean\_y = np.mean(y\_values)

std\_dev\_x = np.std(x\_values)

std\_dev\_y = np.std(y\_values)

return mean\_x, mean\_y, std\_dev\_x, std\_dev\_y

def estimate\_correctness(std\_dev\_x, std\_dev\_y):

return 1 - (std\_dev\_x + std\_dev\_y) / 2

rospy.init\_node('flask\_app')

logger = logging.getLogger('rosout')

logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(sys.stdout)

formatter = logging.Formatter(

'%(asctime)s - %(name)s - %(levelname)s - %(message)s')

handler.setFormatter(formatter)

logger.addHandler(handler)

rospy.Subscriber('/amcl\_pose', PoseWithCovarianceStamped, pose\_callback)

rospy.Subscriber('/move\_base/status', GoalStatusArray, status\_callback)

rospy.Subscriber('/battery\_state', BatteryState, battery\_state\_callback)

pub = rospy.Publisher('/move\_base\_simple/goal', PoseStamped, queue\_size=10)

cancel\_pub = rospy.Publisher('/move\_base/cancel', GoalID, queue\_size=10)

rospy.Subscriber("/particlecloud", PoseArray, particlecloud\_callback)

def publish\_goal(x, y, z, w):

goal = PoseStamped()

goal.header = Header(stamp=rospy.Time.now(), frame\_id='map')

goal.pose.position.x = x

goal.pose.position.y = y

goal.pose.orientation.z = z

goal.pose.orientation.w = w

pub.publish(goal)

def load\_coordinates():

global coordinates

with open(filename, 'r') as f:

coordinates = json.load(f)

def load\_routes():

global routes

try:

with open(routes\_filename, 'r') as f:

routes = json.load(f)

except FileNotFoundError:

routes = {}

@app.route('/', methods=['GET', 'POST'])

def index():

load\_routes()

load\_coordinates()

if request.method == 'POST':

name = request.form.get('name')

if name in coordinates:

x, y, z, w = coordinates[name]

rospy.logwarn("Publishing goal: %s", (x, y, z, w))

publish\_goal(x, y, z, w)

return render\_template('index.html', coordinates=coordinates, routes=routes)

@app.route('/save\_point', methods=['POST'])

def save\_point():

global latest\_pose

name = request.form.get('name')

x = round(latest\_pose.pose.pose.position.x, 3)

y = round(latest\_pose.pose.pose.position.y, 3)

z = round(latest\_pose.pose.pose.position.z, 3)

w = round(latest\_pose.pose.pose.orientation.w, 3)

coordinates[name] = (x, y, z, w)

with open(filename, 'w') as f:

json.dump(coordinates, f)

return redirect('/')

@app.route('/get\_point', methods=['GET'])

def get\_point():

global latest\_pose

if latest\_pose is not None:

x = round(latest\_pose.pose.pose.position.x, 3)

y = round(latest\_pose.pose.pose.position.y, 3)

z = round(latest\_pose.pose.pose.position.z, 3)

w = round(latest\_pose.pose.pose.orientation.w, 3)

return {'x': x, 'y': y, 'z': z, 'w': w}

else:

return {'error': 'No pose data available'}

@app.route('/get\_status\_message', methods=['GET'])

def get\_status\_message():

global status\_message

return jsonify({'status\_message': status\_message})

@app.route('/get\_battery\_state', methods=['GET'])

def get\_battery\_state():

global battery\_state

if battery\_state is not None:

return jsonify(battery\_state)

else:

return {'error': 'No battery state data available'}

@app.route('/create\_route', methods=['POST'])

def create\_route():

global routes

route\_name = request.form.get('route\_name')

waypoints = request.form.getlist('waypoints[]')

routes[route\_name] = waypoints

with open(routes\_filename, 'w') as f:

json.dump(routes, f)

return redirect('/')

@app.route('/create\_route\_type', methods=['POST'])

def create\_route\_type():

global routes

route\_name = request.form.get('route\_name')

waypoints\_str = request.form.get('waypoints[]')

waypoints = [waypoint.strip() for waypoint in waypoints\_str.split(',')]

routes[route\_name] = waypoints

with open(routes\_filename, 'w') as f:

json.dump(routes, f)

return redirect('/')

@app.route('/execute\_route', methods=['POST'])

def execute\_route():

global routes, status\_message, should\_continue\_execution

route\_name = request.form.get('route')

waypoints = routes.get(route\_name)

if waypoints and should\_continue\_execution:

for waypoint in waypoints:

if waypoint in coordinates and should\_continue\_execution:

if should\_continue\_execution == False:

break

else:

x, y, z, w = coordinates[waypoint]

publish\_goal(x, y, z, w)

time.sleep(5)

while True:

rospy.logwarn(

status\_message['status\_list'][0]['status'])

time.sleep(1)

if status\_message and len(status\_message['status\_list']) > 0:

if status\_message['status\_list'][0]['status'] == 3:

break

time.sleep(7)

return redirect('/')

@app.route('/cancel\_move', methods=['POST'])

def cancel\_move():

global should\_continue\_execution

should\_continue\_execution = False

cancel\_move\_base()

time.sleep(10)

should\_continue\_execution = True

return redirect('/')

@app.route('/clear\_all', methods=['POST'])

def clear():

global should\_continue\_execution

should\_continue\_execution = True

return redirect('/')

@app.route('/get\_confidence\_level', methods=['GET'])

def get\_confidence\_level():

global amcl\_covariance

if amcl\_covariance is None:

return jsonify({'error': 'No covariance data available'})

confidence\_level = calculate\_confidence\_level(amcl\_covariance)

return jsonify({'confidence\_level': confidence\_level})

@app.route('/get\_confidence\_particle\_cloud', methods=['GET'])

def get\_confidence\_particle\_cloud():

mean\_x, mean\_y, std\_dev\_x, std\_dev\_y = calculate\_statistics()

if mean\_x is None:

return jsonify({'error': 'No particle cloud data available'})

correctness = estimate\_correctness(std\_dev\_x, std\_dev\_y)

return jsonify({'correctness': correctness})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=6969, debug=True)

This will call a new coordinates.txt and routes.txt where the coodinates and the routes is stored. Even though both are in a txt file, it is writen in a JSON format where it is an object with an array list, so it will start with a curly brackets, followed with a definition of the object, and then the array list. e.g.,

{

"home": [0.0, 0.0, 0.0, 1.0],

"door": [0.769, 8.166, -0.731, 0.682],

"arm": [7.123, -5.069, 0.989, 0.147],

"charger": [0.829, -4.305, -0.994, 0.114]

}

{

"full-run": [door,home,arm,home],

}

The coordinates.txt array list is showing the coordinate based off of AMCL's pose estimate, where it's home position is 0.0, 0.0, 0.0, 1.0, and it's moving in relation of the home center position routes.txt array list is showing the route that the AMR will take, where it is calling the waypoint in the coordinates.txt file.

1. Importing Required Libraries and ROS Messages: The code starts by importing various Python libraries and ROS messages required for communication and data handling.
2. Global Variables: Several global variables are declared at the beginning of the script. These variables are used to store data received from ROS topics and to maintain robot state.
3. ROS Callback Functions: The script defines several callback functions that are executed when ROS topics receive new data. For example, pose\_callback, status\_callback, battery\_state\_callback, and particlecloud\_callback are callback functions for receiving pose, status, battery state, and particle cloud data, respectively.
4. Helper Functions: There are several helper functions that perform calculations and conversions related to the robot's state and data. For example, calculate\_confidence\_level calculates the confidence level based on covariance data, calculate\_statistics calculates statistics from the particle cloud, and estimate\_correctness estimates correctness based on particle cloud data.
5. ROS Initialization: The script initializes the ROS node and sets up logging for debugging purposes.
6. Flask Web Application: The code creates a Flask web application instance (app) to handle HTTP requests and responses.
7. Flask Routes: The Flask routes are defined using decorators to handle different types of HTTP requests:
   * /: Renders the main web page that displays the robot's current location, saved locations, and available routes.
   * /save\_point: Saves the current robot location with a user-defined name.
   * /get\_point: Retrieves the current robot location and returns it as JSON data.
   * /get\_status\_message: Retrieves the robot's status message and returns it as JSON data.
   * /get\_battery\_state: Retrieves the robot's battery state and returns it as JSON data.
   * /create\_route: Creates a new route with user-defined waypoints.
   * /create\_route\_type: Creates a new route with pre-defined waypoints.
   * /execute\_route: Executes a pre-defined route with saved waypoints.
   * /cancel\_move: Cancels the execution of a route in progress.
   * /clear\_all: Clears any stop execution flags.
   * /get\_confidence\_level: Calculates and returns the confidence level of the robot's pose estimation.
   * /get\_confidence\_particle\_cloud: Calculates and returns the correctness of the robot's pose estimation using particle cloud data.
8. \_\_name\_\_ == '\_\_main\_\_' Block: The script runs the Flask application if it is being executed as the main script (\_\_name\_\_ == '\_\_main\_\_'). The application listens on IP 0.0.0.0 and port 6969.

### **Index.html**

[index.html22KB](https://amr-docs-brianstm.vercel.app/files/index.html)

The index.html is styled using tailwindcss, where it is a CSS library that is used to style the website. The index.html is using a CDN (Content Delivery Network) where it is a network of servers that delivers the content to the user based on their geographic location, the origin of the webpage, and the content delivery server. The CDN is used to reduce the latency of the website, where it is faster to load the website. The CDN is also used to reduce the load on the server, where it is easier to handle the request from the user.

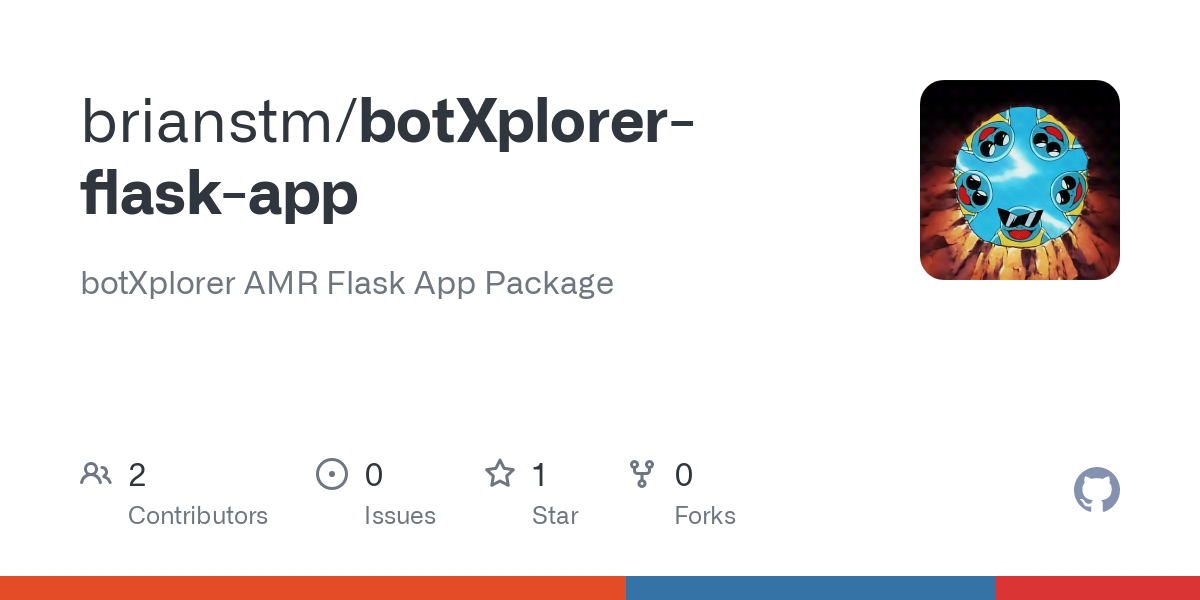
[[](https://tailwindcss.com/)](https://tailwindcss.com/" \t "_blank)

## [Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.](https://tailwindcss.com/" \t "_blank)

[Tailwind CSS is a utility-first CSS framework for rapidly building modern websites without ever leaving your HTML.](https://tailwindcss.com/" \t "_blank)

[https://tailwindcss.com/](https://tailwindcss.com/" \t "_blank)

the updated files can be forked and or pulled from the repository below:

[[](https://github.com/brianstm/botXplorer-flask-app.git)](https://github.com/brianstm/botXplorer-flask-app.git" \t "_blank)

## [GitHub - brianstm/botXplorer-flask-app: botXplorer AMR Flask App Package](https://github.com/brianstm/botXplorer-flask-app.git" \t "_blank)

[botXplorer AMR Flask App Package. Contribute to brianstm/botXplorer-flask-app development by creating an account on GitHub.](https://github.com/brianstm/botXplorer-flask-app.git" \t "_blank)

[https://github.com/brianstm/botXplorer-flask-app](https://github.com/brianstm/botXplorer-flask-app.git" \t "_blank)

# 😇SICK LiDAR Localization

SICK LiDAR Localization is a powerful tool for Localization using SICK Sensors,

## [SICK | Sensor Intelligence](https://www.sick.com/us/en/localization-and-positioning-solutions/lidar-localization/lidar-loc/c/g541272" \t "_blank)

[https://www.sick.com/us/en/catalog/digital-services-and-solutions/application-software/lidar-loc/c/g541272](https://www.sick.com/us/en/localization-and-positioning-solutions/lidar-localization/lidar-loc/c/g541272" \t "_blank)

to fully utilize we would need a license key and also do some pre preparation:

### **IP Gateway**

The Autonomous Mobile Robot (AMR) in our setup is equipped with a LiDAR sensor, which generates /scan data vital for its localization. However, the Raspberry Pi running the robot can't directly handle SICK's LiDAR localization software, SopasAir, due to its hardware limitations. Thus, we install the software on a more robust Virtual Machine (VM). Although the VM can access the /scan data, SopasAir on the VM can't retrieve this data directly. Therefore, we require a network architecture that allows SopasAir to indirectly access the /scan data. To realize this, we establish a network gateway and configure IP routing on the AMR, which serves as an intermediary in this process.  
The LiDAR is connected to a Raspberry Pi, and the Pi's IP address is set to 192.168.42.125. The LiDAR itself is assigned the IP address 192.168.42.100, ensuring that they share the same subnet. This configuration allows for seamless communication between the LiDAR and the Raspberry Pi. However, to access the LiDAR data from our VM, additional steps were necessary. We routed the IP of the LiDAR to the WiFi network, enabling access to the LiDAR from the VM. In our case, the WiFi IP to SSH into the Pi is 192.168.1.138. To do this:

### **On the AMR (with IP address 192.168.1.138), we execute the following commands:**

sudo sysctl -w net.ipv4.ip\_forward=1

This command activates IP forwarding on the AMR, allowing it to accept incoming network packets on one interface and properly forward them to another. This is necessary for the AMR to act as a gateway.

sudo iptables -t nat -A PREROUTING -p tcp --dport 2111 -j DNAT --to-destination 192.168.42.100:2111

This iptables rule redirects incoming traffic intended for port 2111 (the LiDAR's port) to the IP address of the LiDAR (192.168.42.100). It ensures that the LiDAR can receive the traffic intended for it.

sudo iptables -t nat -A POSTROUTING -d 192.168.42.100 -j MASQUERADE

This command uses MASQUERADE to replace the original source IP address with the AMR's IP address before the packets are forwarded to the LiDAR. By doing this, the LiDAR perceives the packets as originating from the AMR, enabling seamless communication between the VM and the LiDAR.

sudo iptables-save | sudo tee /etc/iptables/rules.v4

This command saves the current iptables rules to a file (/etc/iptables/rules.v4), ensuring that the rules can be restored after system reboots. This step is important to maintain the configured IP and port forwarding even after a reboot.

### **On the Virtual Machine, we execute the following command:**

sudo route add -net 192.168.42.0 netmask 255.255.255.0 gw 192.168.1.138

These commands were executed to establish a network architecture where SopasAir running on the VM can indirectly access the LiDAR data through the AMR acting as a gateway. The iptables rules redirect incoming traffic to the LiDAR and modify the source IP address before forwarding the packets, ensuring seamless communication between the VM and the LiDAR.

💡

**Note**: The IP addresses used in this example are for the SICK Camera WiFi. You will need to change the IP addresses to match your network setup.

### **Verifying the Gateway Configuration on the AMR**

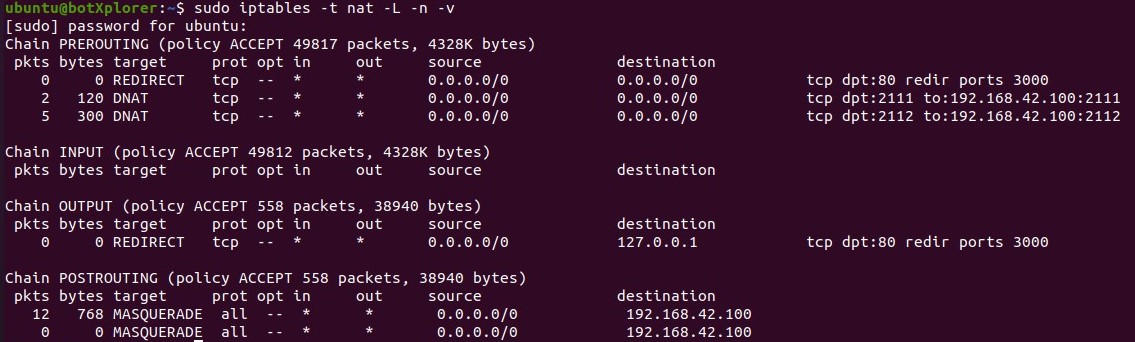
Once the gateway is established on the AMR, it's crucial to verify that the configuration has been applied correctly. To achieve this, you can use the command:

sudo iptables -t nat -L -n -v

This command allows you to list the rules in the nat table of the iptables firewall in a verbose manner. Here is a breakdown of the command:

* iptables -t nat: Accesses the 'nat' table in iptables. This table is responsible for Network Address Translation, which includes port forwarding rules we've set up.
* -L: Lists all the rules in the specified table.
* -n: Shows numerical addresses instead of trying to resolve hostnames, making the output faster and more concise. -v: Stands for 'verbose'. This option makes the command output more detailed, providing extra information such as the network interface names and the packet and byte counters.

Running this command will display the iptables rules currently in place for network address translation, including the PREROUTING, POSTROUTING, and OUTPUT chains. You can then verify if the rules we've set up are present and correct.



### **SOPAS Air for Localization and Mapping**

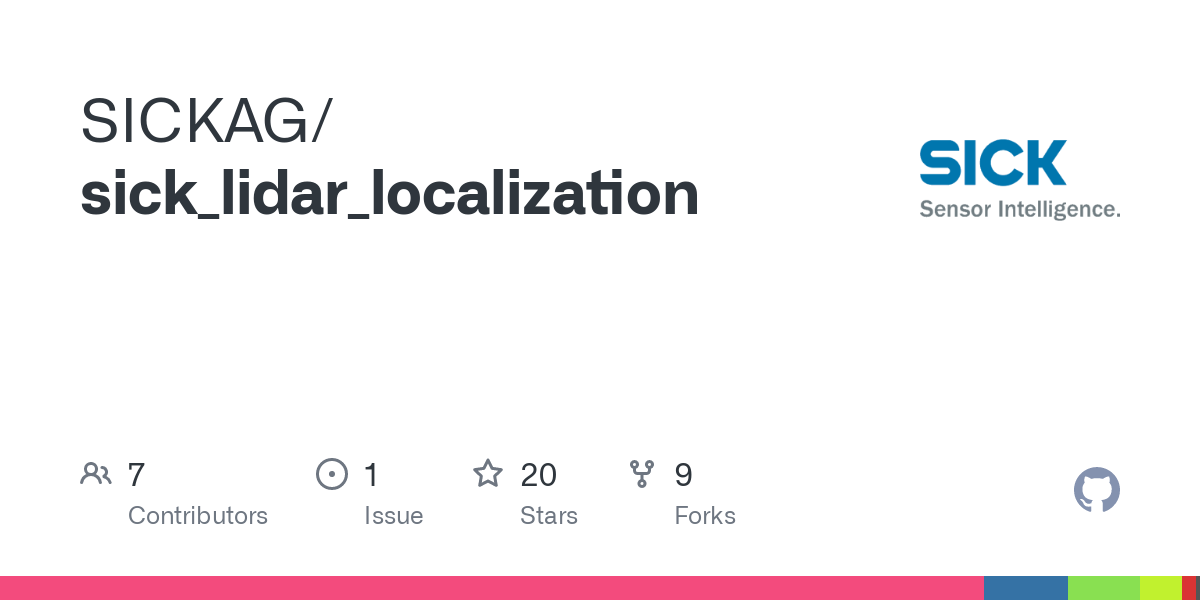
The SICK SOPAS Air software is used for localization and mapping tasks with the LiDAR. SOPAS Air provides a user-friendly interface to configure and control the LiDAR's functionalities. With SOPAS Air, you can initiate the localization process, configure mapping parameters, and visualize the generated maps. Localization involves LiDAR capturing data from its surroundings and using this information to estimate its own position within the environment. Mapping, on the other hand, focuses on building a representation of the environment by combining multiple scans obtained from the LiDAR.

### **Map Conversion with SICK Map ET**

Once a map is generated in the LiDAR localization software, it is stored in a .rawmap format. To utilize this map in our system, it needs to be converted to the .vmap format. This is where the SICK Map ET (Map Editor Tool) comes into play. SICK Map ET is a powerful tool that enables the conversion of map files between different formats, ensuring compatibility with various applications.

### **Sick Lidar Localization GitHub**

For detailed information on SICK LiDAR localization including configuration, SOPAS Air usage, and map conversion, you can refer to the GitHub repository at:

[[](https://github.com/SICKAG/sick_lidar_localization)](https://github.com/SICKAG/sick_lidar_localization" \t "_blank)

## [GitHub - SICKAG/sick\_lidar\_localization](https://github.com/SICKAG/sick_lidar_localization" \t "_blank)

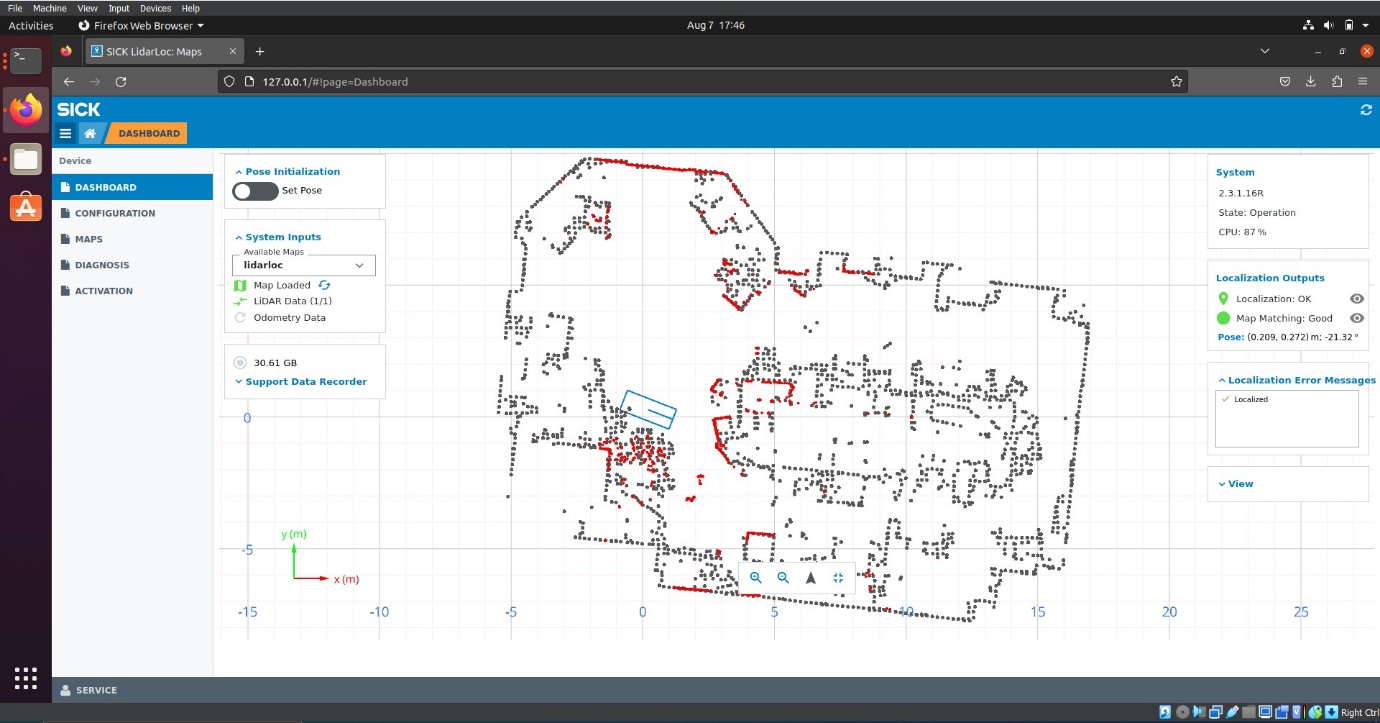
[Contribute to SICKAG/sick\_lidar\_localization development by creating an account on GitHub.](https://github.com/SICKAG/sick_lidar_localization" \t "_blank)

[https://github.com/SICKAG/sick\_lidar\_localization](https://github.com/SICKAG/sick_lidar_localization" \t "_blank)

This repository provides comprehensive documentation and resources to guide you through the setup and implementation of SICK LiDAR localization into the navigation software.

### **Utilizing SICK LiDAR Localization in Navigation**

The SICK LiDAR localization software also provides PointCloud data, which is a set of data points in a coordinate system. These points represent the external surface of the objects in the robot's environment. This data can be useful for visualization and debugging purposes



In our system, the data obtained from the SICK LiDAR localization process is utilized in the /tf (transform) component of the Move Base. The /tf transform manages the coordinate transformation between various frames of reference, essential for accurate navigation and localization of the AMR. The transformed data ensures that the AMR's movements are accurately mapped to its actual position in the environment, contributing to reliable navigation and obstacle avoidance.

Another thing we did is we installed a package calledpointcloud-to-laserscan, which converts the pointcloud data to laserscan data, which is used by the amcl. This is done by the following command:

sudo apt-get install ros-$noetic-pointcloud-to-laserscan

and then running this launch code:

<launch>

<node pkg="pointcloud\_to\_laserscan" type="pointcloud\_to\_laserscan\_node" name="pointcloud\_to\_laserscan">

<remap from="cloud\_in" to="/pointCloud"/>

<remap from="scan" to="/scan"/>

<param name="transform\_tolerance" value="0.01"/>

<param name="min\_height" value="0.0"/>

<param name="max\_height" value="1.0"/>

<param name="angle\_min" value="-1.57"/>

<param name="angle\_max" value="1.57"/>

<param name="angle\_increment" value="0.01"/>

<param name="scan\_time" value="0.033"/>

<param name="range\_min" value="0.45"/>

<param name="range\_max" value="4.0"/>

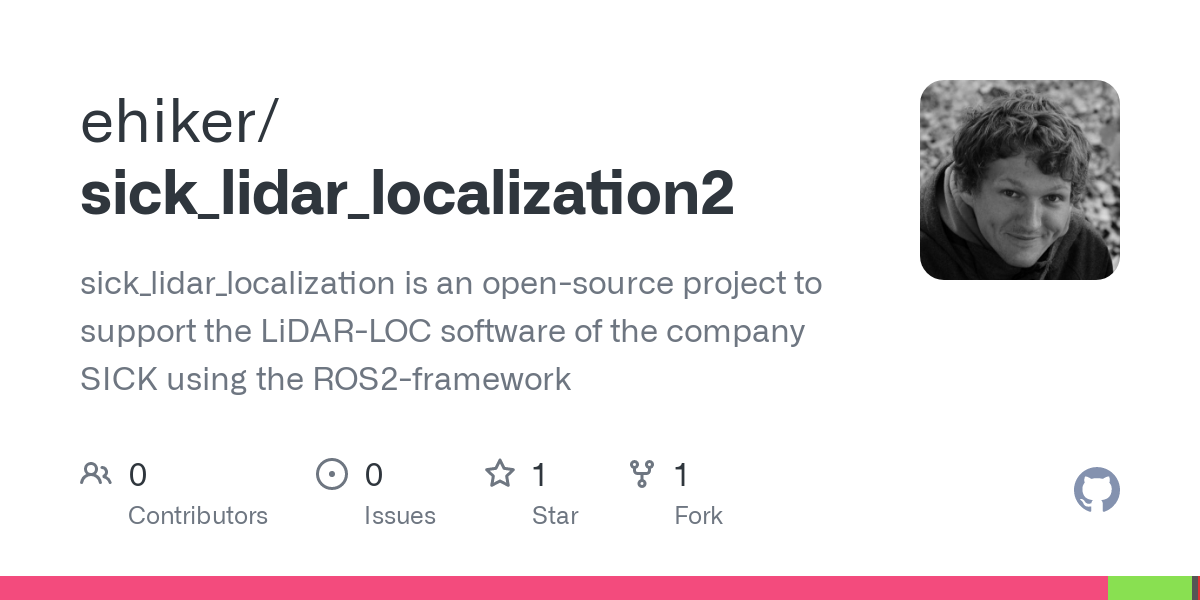
<param name="use\_inf" value="true"/>

</node>

</launch>

By running both the above launch file as well as the amcl launch file, it will convert the pointCloud data which amcl cant read to a more readable /scan data, which is used by the amcl.

Another thing we found is that we can also use another repository fromehiker instead of from the official SICK\_AG found here:

[[](https://github.com/ehiker/sick_lidar_localization2)](https://github.com/ehiker/sick_lidar_localization2" \t "_blank)

## [GitHub - ehiker/sick\_lidar\_localization2: sick\_lidar\_localization is an open-source project to support the LiDAR-LOC software of the company SICK using the ROS2-framework](https://github.com/ehiker/sick_lidar_localization2" \t "_blank)

[sick\_lidar\_localization is an open-source project to support the LiDAR-LOC software of the company SICK using the ROS2-framework - ehiker/sick\_lidar\_localization2](https://github.com/ehiker/sick_lidar_localization2" \t "_blank)

[https://github.com/ehiker/sick\_lidar\_localization2](https://github.com/ehiker/sick_lidar_localization2" \t "_blank)

This is an enhanced version of the existing SICK LiDAR Localization software, from this you can get the direct data called the result\_telegram from the following topic:

rostopic echo "/sick\_lidar\_localization/driver/result\_telegrams"

This data can be used to get the position of the robot from the telemetry data, from here you can add a line of code in your amcl launch file to read this as a intitial pose location like so:

<remap from="initialpose" to="/sick\_lidar\_localization/driver/result\_telegrams"/>

💡**Note**: This package was designed for ROS Moelodic, this package only worked on our system because we are still able to use Python 2 libraries in addition to Python 3 libraries.