Philadelphia Flight Control Lab

Final Report

Supervised by Professor Daniel Zelazo

Written by Yoav Houri
yoavhouri26@hotmail.fr

Table of contents

Introduction

This year, the lab worked on two brands different mecanum wheel robots, the **Moebius mecanum wheel** robot and the **Yahboom X3**. These two robots are equipped with **mecanum wheels**. The specific **design** of the wheels allows the robots to move not only **forward** and **turn** as we are familiar, but also move **sideways** and in **diagonal**.

ROS Basics

What is ROS?

Our different robots (Moebius, Yahboom and Gazebo simulated robots) are controlled by the **ROS** (Robot operating system), it's a set of software libraries and tools destined for robot applications. Basically, ROS need a **ROS master** to coordinate the robots and peripheral devices for **remote control**.

In our case, the **ROS master** will be the **computer** in which we are going to build our Simulink models and upload them into the robot (IRL or Simulation).

To coordinate and command the different robots, **ROS need to be installed on each robot**. Each robot comes with his own **.img** file, [1]which is what we are going to upload into the Micro SD card that sits inside the Raspberry pi 4 Model B. We will further see that in order to control the robot using Simulink, the host computer doesn't need to install ROS. This makes the task easier for non-experienced Linux user. Indeed, ROS is supported on Ubuntu Linux only (for ROS 1).

In the Lab, the host computer runs on **Ubuntu 20.04.6 LTS** Linux OS and the installed ROS version is **ROS noetic**.

NOTE: The Ubuntu OS version of the lab computer (or any other computer) doesn't need to match to the Ubuntu OS version of the robot for remote control.

Here are some useful **links** to **learn about** ROS1 or ROS2:

Official ROS site (free): http://wiki.ros.org/noetic/Installation/Ubuntu
Courses site (subscription fees) : https://www.theconstructsim.com

[1] All the corresponding files are accessible from the computer in the lab or from the Github of the PFCL:

How ROS works?

Below, a **general picture** of the **functioning** of **ROS**:

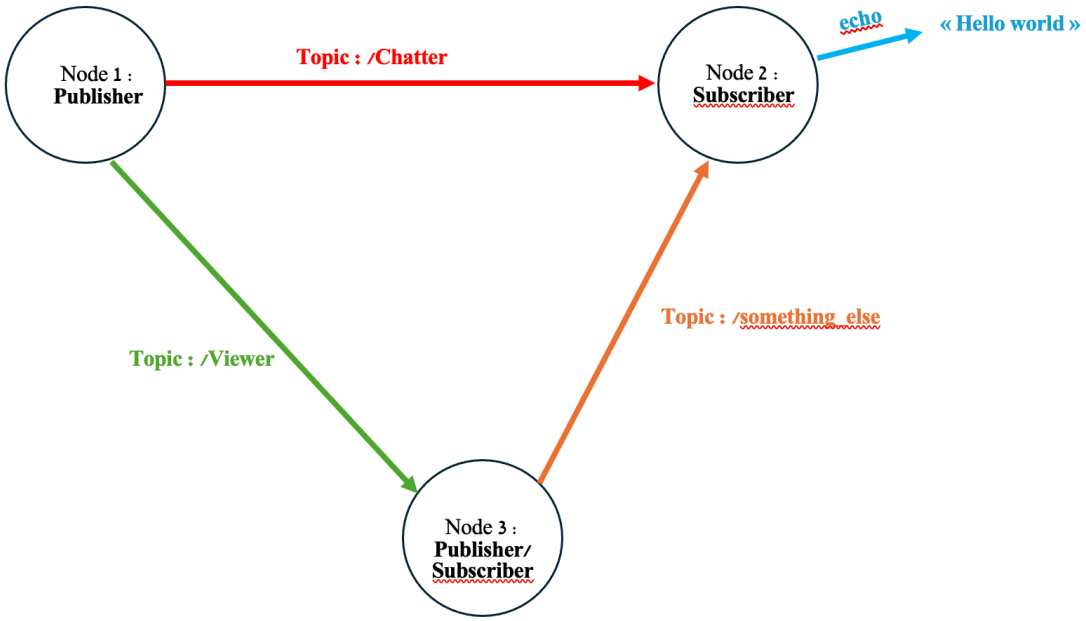


Figure 1: Fundamental functionning of ROS

When using **ROS**, a specific **terminology** is employed.
The following concepts need to be known:

- **Nodes:** Nodes are basically programs made in ROS that communicate each other.
- **Publisher:** A publisher is a node that writes information to a topic.
- **Subscriber:** A subscriber is a node that reads information from a topic.
- **Topics:** ROS handles almost all its communications through topics.

A topic it's like a pipe. Nodes use topics to publish/read information to/from other nodes. The robot will keep listening to the last message that is being published on the topic.

- **Messages**: Messages are the ROS data type used when subscribing or publishing to a topic.
- **Services:** Services allow to code a specific functionality for the robot and then provide for everyone to call it.

Services are synchronous. When your ROS program calls a service, you program can't continue until it receives a result from the service.

- **Actions:** Actions are like services.

The only difference is that actions are asynchronous. When your ROS program calls a service, your program can perform other tasks while the action is being perform in another thread.

Let's take an **example**[2] for better understanding:

Let say that we programmed a **node** called "IMU" and another one called "Fusion".

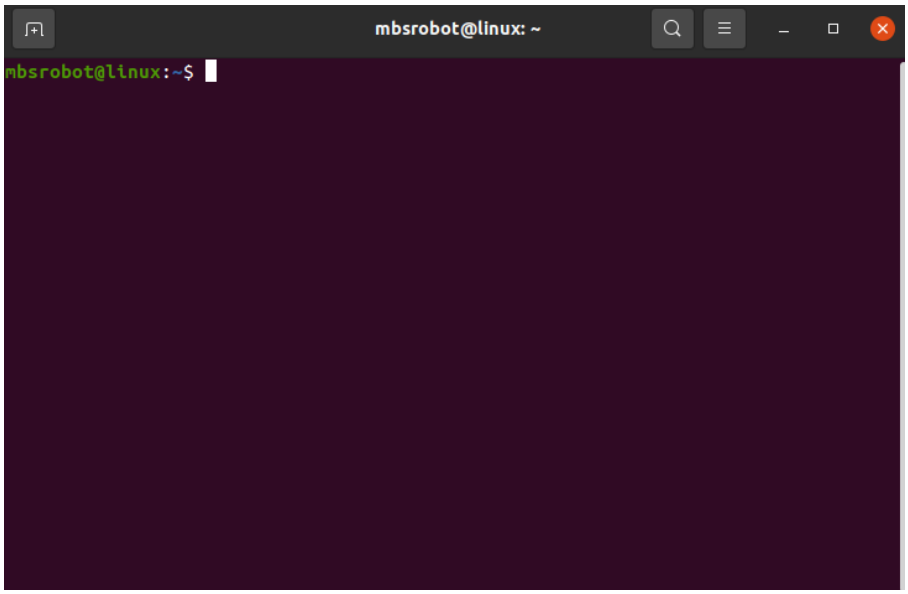
1. The "IMU" node **collects** the data from the different sensors (Accelerometer, Gyroscope, Magnetometer, etc …).
2. After the data have been collected, the "IMU" node **publish** the data through the **topic** "/cmd_vel". The type of **message** used is "geometry_msgs/Twist" and it contains 6 variables (linear.x, linear.y, linear.z, angular.x, angular.y, angular.z).
3. Once the data has been **published**, the "Fusion" node **subscribes** to the "/cmd_vel" topic and apply a fusion filter on the collected data.

Installing and configuring the ROS environment

Configure the ROS variables

As mentioned earlier, each type of robot (Moebius or Yahboom) comes with his **own pre-configured .img file**. In those files, ROS versions and the relevant packages **have already been installed**. However, when first powering up a new build robot, the user should **check** the **ROS variables**.

In the **Ubuntu environment**, we will need to navigate through files and directories in the **terminal**.




```
cd
```

The cd command allows to change directory (from current directory to subdirectories).

[2] We will further look onto actual scheme-like examples from the different robots.

```
ls
```

The ls command displays all the file located in the current directory.[3]

For example:

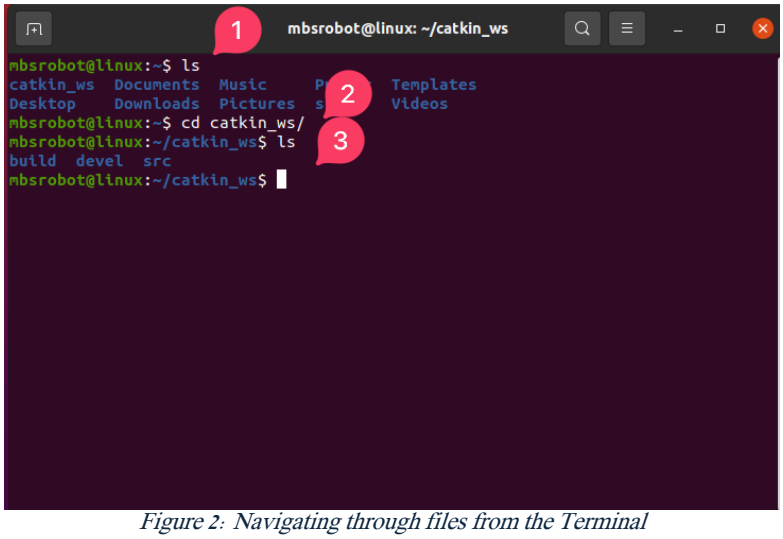


Figure 2: Navigating through files from the Terminal

1) We display all the files located in the home directory with **ls**
2) We change directory to the subdirectory **catkin_ws** (we will later explain what is this subdirectory)
3) We display all the files located in the **catkin_ws directory** with **ls**

As mentioned earlier, we need to ensure that the **environment variables** are properly set.
In a terminal shell, type:

```
printenv | grep ROS
```

[3] These two commands are essentials when working on the terminal in the Ubuntu environment.

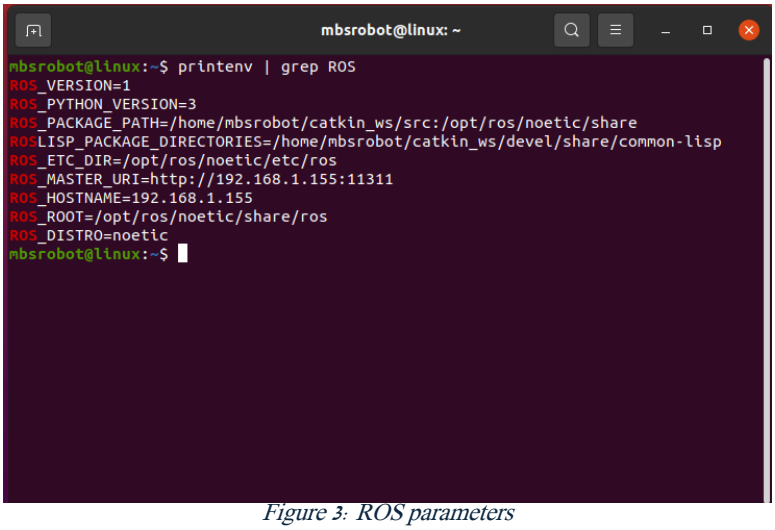
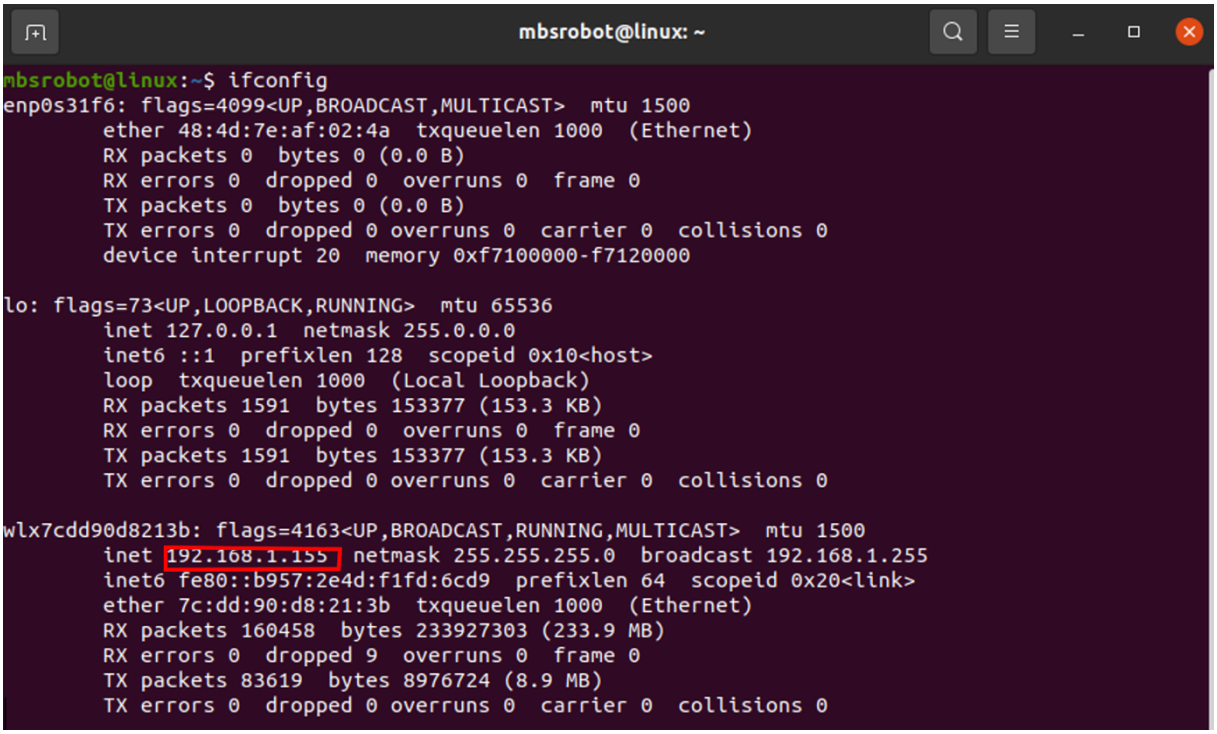


Figure 3: ROS parameters

The **ROS_ROOT** and **ROS_PACKAGE_PATH** should be set as below. Further, for Simulink control, we need to ensure that the **ROS_MASTER_URI** and **ROS_HOSTNAME** variables are also set correctly, where the address must be the IP address of the computer.[4]

This can be found using the following command:

```
Ifconfig
```



[4] The robots can be **controlled remotely** via SSH. However, the robots will need to be connected to the **same WiFi** as the host computer.

If the variables **ROS_MASTER_URI** and **ROS_HOSTNAME** are not set, the user needs to edit the **~/.bashrc** file:

```
nano ~/.bashrc
```

and add the following lines at the **end** of the file:

```
export ROS_MASTER_URI=http://<ip_address>:11311
export ROS_HOSTNAME=<ip_address>

source /opt/ros/noetic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

The last two lines are important. Indeed, every time we modify a file in a terminal shell, we need to "source" it to ensure that the modifications will be applied in the other terminals.

ROS filesystem architecture

A trivial **workspace** in ROS might look like this:

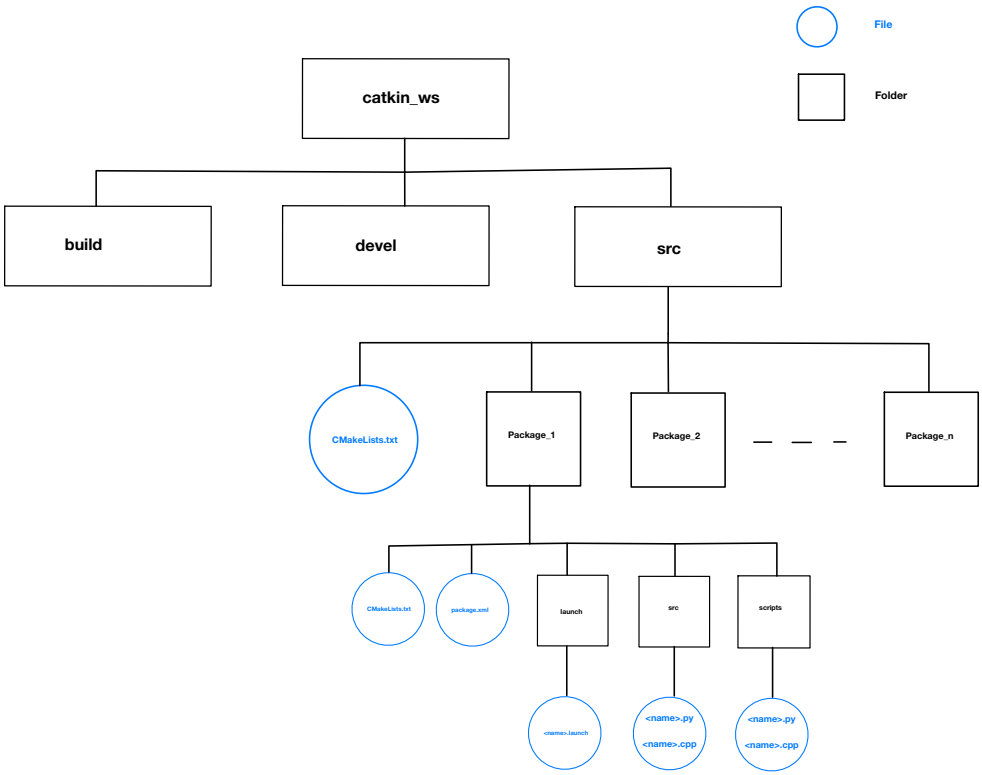


Figure 4: File system architecture in ROS

The workspace we are working in is the **catkin workspace**. In this workspace we have three folders:

- **build:** This is where CMake and catkin_make are called to configure and build packages
- **devel:** This is where the executables and libraries go before we install the packages
- **src:** This folder will contain all the packages created.

To get to the **package** from the terminal, two options (from the home directory):

```
cd catkin_ws/src/<package_name>
```

or

```
roscd <package_name>
```

In a **ROS package** we have:

- **launch folder**: contains launch files (<name>.launch)
- **src folder:** contains source files (cpp, python, …)
- **scripts folder:** contains scripts files (cpp, python, …). Those files can be launch as a standalone.
- **CMakelists.txt** and **package.xml :** not relevant at the moment (useful for advanced user)

Create a ROS workspace

To create or use ROS packages, we first need to **create** and **build a catkin workspace** (in the home directory):

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

***The catkin_make command is equivalent to "compile". Every time we create a new package in the catkin workspace, we need to "compile" it and source it:**

```
cd ~/catkin_ws/
catkin_make                                    //Use for compiling
source devel/setup.bash                        //Use for sourcing
```

To make sure the workspace is properly **overlayed by the setup script**, we need to make sure that the ROS_PACKAGE_PATH environment variable includes the directory we're in:

```
echo $ROS_PACKAGE_PATH
```

We should see something like this:

```
/home/youruser/catkin_ws/src:/opt/ros/kinetic/share
```

Once the catkin workspace is properly set, we can create our own package:

```
cd ~/catkin_ws/src
catkin_create_pkg <package_name> <package_dependency_1> < package_dependency_2>
```

***If our package use C++ source files we write "roscpp", python sources files we write "rospy"**

For example:

```
cd ~/catkin_ws/src
catkin_create_pkg robot_lab roscpp rospy
```

The robot_lab package will use C++ and python source files.

Now, we need to create the launch and scripts folder.

```
cd ~/catkin_ws/src/<package_name>
mkdir launch
mkdir scripts
```

Inside the launch folder, we need to insert our launch files (<name>.launch)

```
cd ~/catkin_ws/src/<package_name>/launch
touch <name>.launch                    //create an empty launch file
nano  <name>.launch                    //edit the new created launch file
```

Now, we need to edit this new created file and fill in with the appropriate format code.
Here is an example of a launch file:

```
<launch>
        <!-- My package launch file -->
        <node pkg="my_package" type="simple.py" name="Obiwan" output="screen">
        </node>
</launch>
```

- **pkg**: Name of the package that contains the code of the ROS program to execute
- **type**: Name of the program file that we want to execute (located in the src folder of the package)
- **name**: Name of the ROS node that will launch the python file
- **output**: Through which channel we will print the output of the python file

To create the python file, we need to do the same steps as we did to create the launch file:

```
cd ~/catkin_ws/src/<package_name>/src
touch <name>.py                          //create an empty source file
nano <name>.py                           //edit the new created source file
```

Note: The name of the python (or C++) file written in the launch file must be the same as the name of the file created in the src folder.

Once we created our folders and files, we must **"compile"** and **"source"**:

```
cd ~/catkin_ws/
catkin_make                                          //Use for compiling
source devel/setup.bash                             //Use for sourcing
```

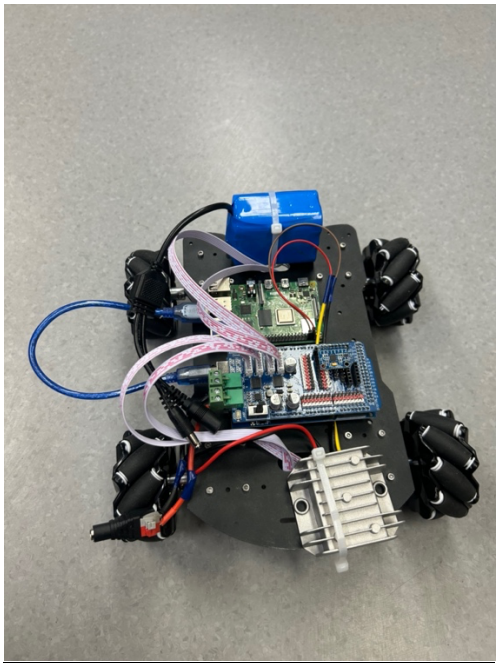
Import an existing ROS package from Github

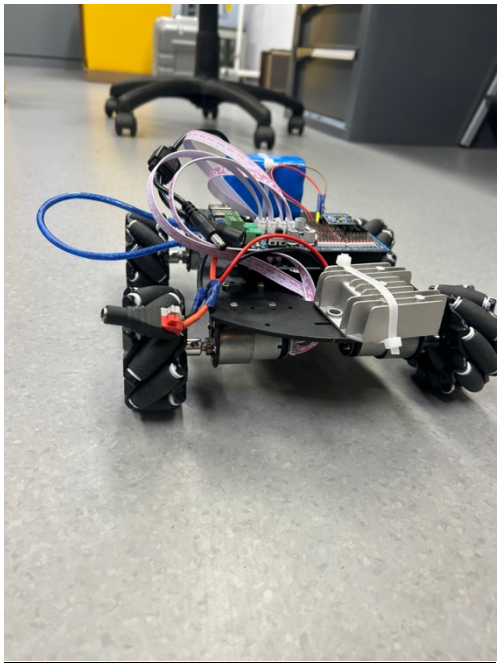
One way to use a ROS package is to import it from Github.
As an example, we are going to import the TurtleBot3 Simulation Package found on Github for the ROS kinetic version:

```
cd ~/catkin_ws/src
git clone -b kinetic-devel https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
cd ~/catkin_ws
catkin_make                                          //Use for compiling
source devel/setup.bash                             //Use for sourcing
```

The package is now **set** and **ready to use**!

Moebius mecanum wheel robot:





This robot is the **first one** we worked on in the lab.

Here the official Github page: https://github.com/MoebiusTech/ros_arduino-mega?spm=a2g0s.imconversation.0.0.13993e5fzdMXh9

The two following PDFs (from the above Github page) **need to be read** by the user for better understanding of the Software part and the Hardware part:

- **Mbsrobot ros smart car operating instructions.pdf** (Software: ROS + Linux)
- **ROS Car with STM32MEGA2560 Controller Installation Manual.pdf** (Hardware)

A newer version of the Github is available here: https://github.com/Andy-Jen/ros_install
This Github contains all the files that need to be downloaded into a new Raspberry pi for the ROS installation **ONLY**.

However, this Github page is NOT necessary after that all the ROS configurations have already been made on one robot. So, the .img file already exists and already contains all the necessary packages

In the lab, some **modifications** were made regarding the **Hardware**:

1. The Arduino MEGA2560 + Moebius motor driver shield is used as the main controller. This component sends the commands from the Raspberry pi 4 (ROS) to the motor via serial communication (Blue USB cable)

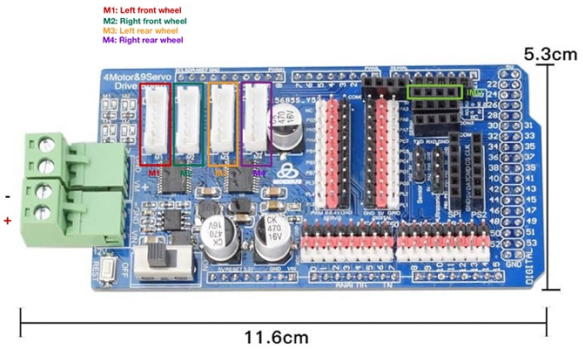


Figure 5: Moebius motor driver shield for Arduino MEGA 2560

2. Moebius 12 $[V]/5000$ $[mAh]$ battery

Figure 6: Moebius 12[V] battery

Notes:

- Every other 12 $[V]$ battery is suitable.
- Need to design and model a 3D part on the robot to put the battery on

3. CPT DC-DC converter

Figure 7: 12[V] DC to 5[V] DC converter

This part is indispensable as it allows to **convert** the 12 $[V]$ from the battery to 5$[V]$ that **powers up** the **Raspberry pi 4 independently** (see Figure 8). Indeed, as mentioned in **"Mbsrobot ros smart car operating instructions.pdf"**, the 5$[V]$ coming from the Arduino through the USB cable may **not be sufficient** to power up the Raspberry pi 4.

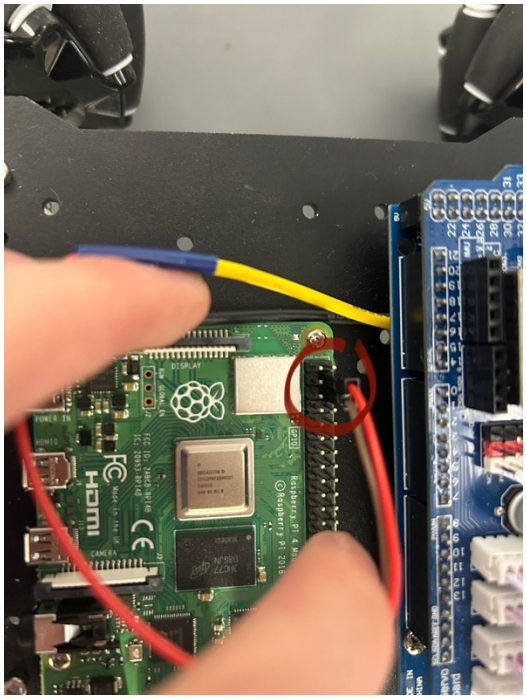


Figure 8: Connection from the DC-DC converter to the Raspberry pi 4

Notes:

- Need to design and model a 3D part on the robot to put the voltage converter on

How to power on the robot?

First, plug a **mouse** and **keyboard** to the **Raspberry pi 4** to navigate through the **Linux system onboard**. Then, plug the **HDMI cable** to monitor and display the files on screen. Then **connect the battery** to the alimentation port (see Figure 9).

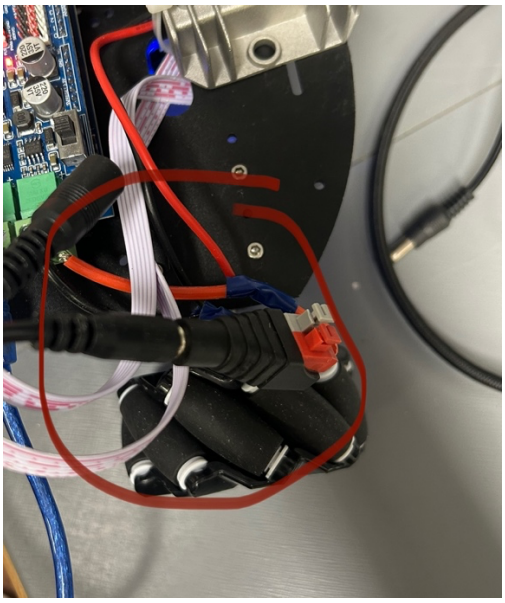


Figure 9: Connecting the battery to the robot

After the loading screen has disappeared (see Figure 10: Loading screen of the Raspberry pi), you may be prompted to **enter the password** of the robot.

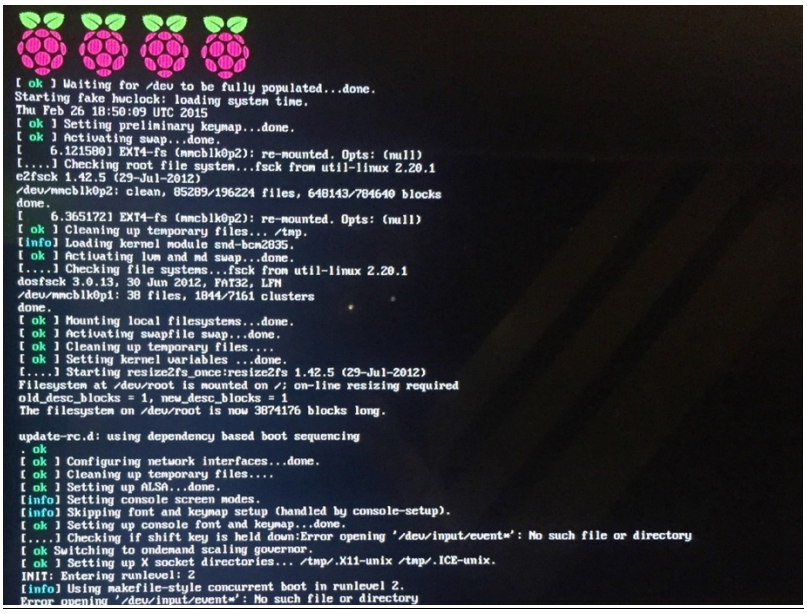


Figure 10: Loading screen of the Raspberry pi

Notes: If the battery is almost empty, a message will appear on the loading screen saying "under-voltage". The user must first charge up the battery to full before powering on the robot.

Here some **information** about the robot (including the password):

Username	mbsrobot
Password	mbsrobot
OS version	Ubuntu 20.04.6 LTS
ROS version	Noetic

Once the password is entered, the user may open a terminal and the following image should be displayed.

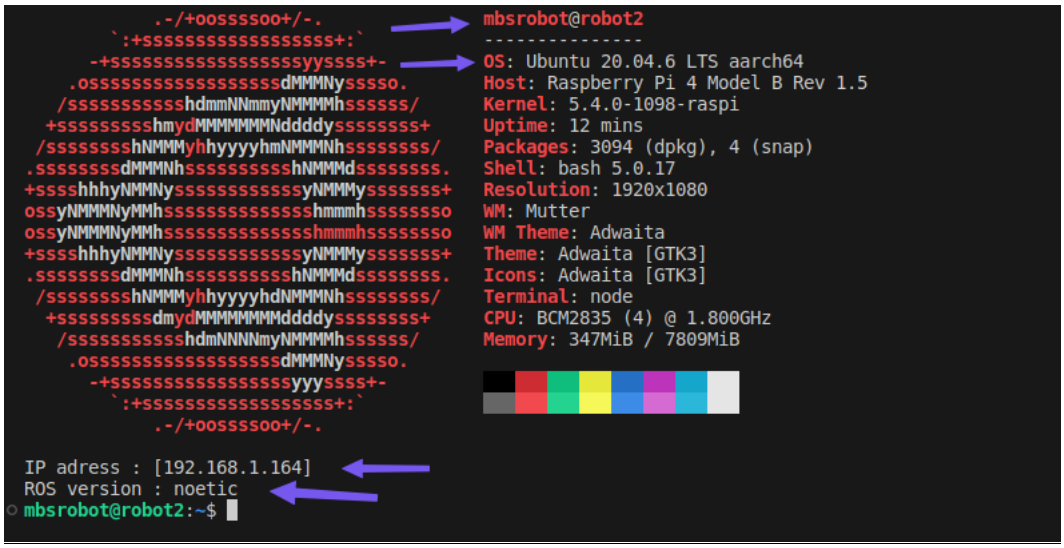


Figure 11: Terminal opening screen

The **information** of the robot should be displayed as above. If the boot is successful, the IP address should also be displayed. **If not**, the user should check for the **WiFi connection**[5] and reenter in a terminal until an IP address is displayed.

This step is **necessary** as now, we can control the robot remotely via **SSH.**

Recall: For controlling the robot via SSH, the robot and the lab computer must be connected to the same WIFI.

On the **lab computer**, the user needs to open a **terminal window** or **Visual Studio Code** (VSC is preferred as it offers some advantages while dealing with SSH. The functioning of VSC is explained in the **Appendix: Visual Studio Code**

Type

```
ssh -X mbsrobot@192.168.1.164
```

[5] See the **troubleshooting** section in the **Appendix: Visual Studio Code**

The user will be prompted to enter the password of the robot.

Figure 12: SSH connection to the robot

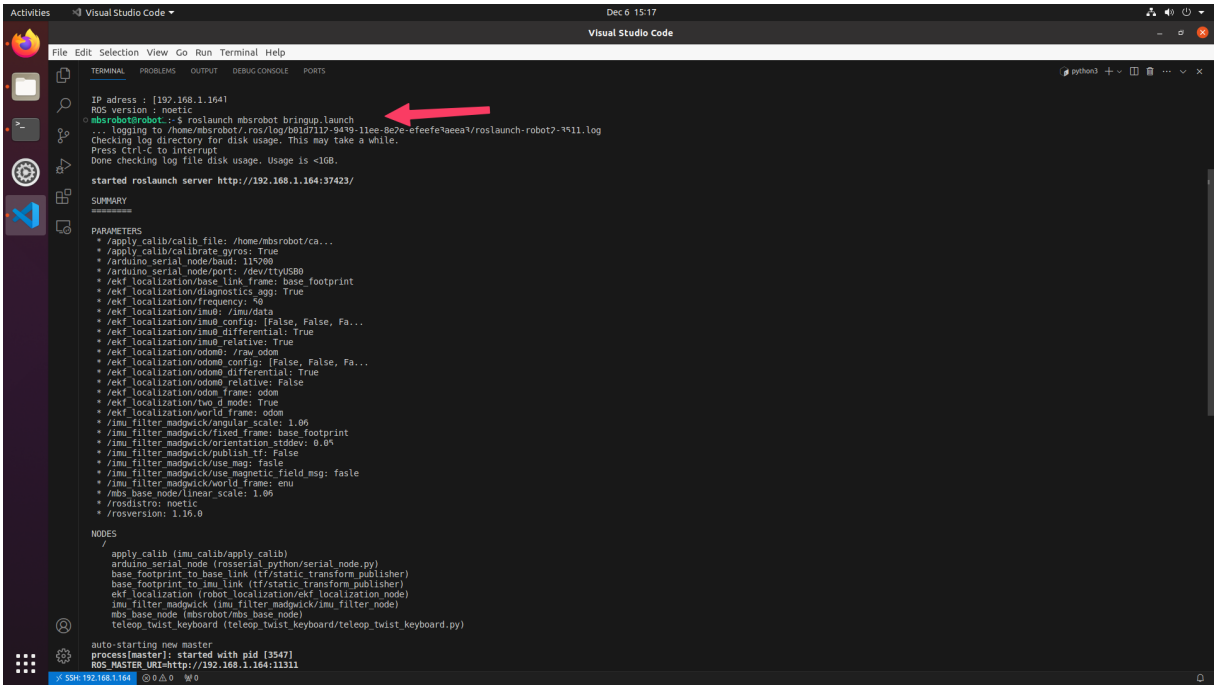
Once the connection is **established**, we can **disconnect** the mouse, the keyboard, and the monitor from the robot, and control it **remotely**.

Control the robot directly from the Raspberry Pi

As explained in **"Mbsrobot ros smart car operating instructions.pdf"**, for controlling the robot with a keyboard through Linux, once the connection is established via SSH, or directly from the robot,

Type

```
roslaunch mbsrobot bringup.launch
```



A **loading screen** should appear, indicating that **the ROS master** (the robot) has been launched:

Contrary to the steps written in the file **"Mbsrobot ros smart car operating instructions.pdf"**, the user **doesn't need** to open a new terminal and launch the keyboard control script as it was already included in the bringup.launch file[6].

Explanation of the launch file process

As explained earlier, once our bringup.launch file has been launched, several nodes has been called.

To **visualize** the **communication between nodes** and **topics**, type:

```
rqt_graph
```

This screen should appear:

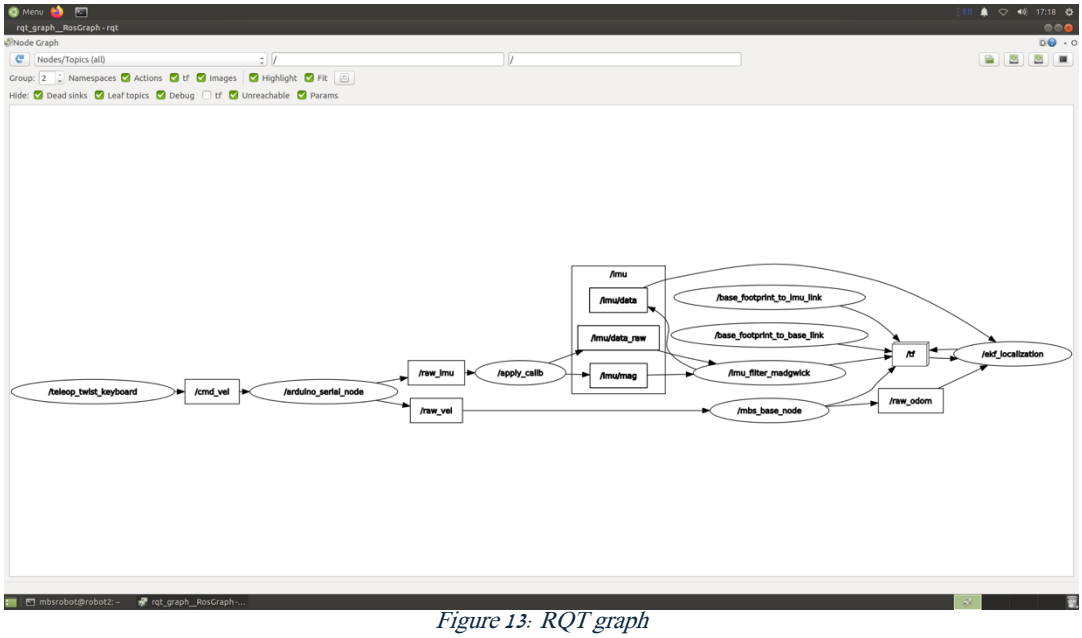


Figure 13: RQT graph

The **"circles"** represent the active **nodes** and the **rectangles** represent the active **topics**.
Here, the **/teleop_twist_keyboard** node takes the input from the keyboard.

[6] This section will later be explain.

Each **key** designates an action for the robot to execute.

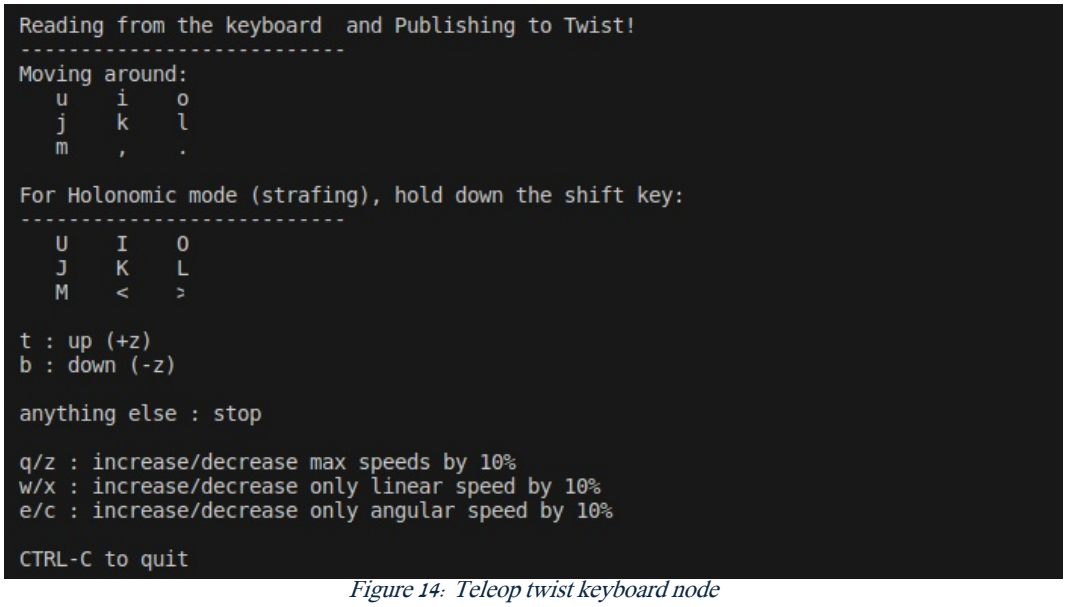


Figure 14: Teleop twist keyboard node

Then, it translates the **key pressing** to a **velocity command** to send to the robot and send those velocity commands to the **/arduino_serial_node** by publishing to the topic **/cmd_vel.**

The **/arduino_serial_node** subscribes to the **/cmd_vel** topic, takes the input velocity commands and send them to the Arduino via the USB cable. (The Arduino is responsible for the motor driving). The **/arduino_serial_node** also collects the data from the encoder on the wheels and the data from the IMU (MPU6050) and publish them to the topics **/raw_vel** and **/raw_imu** respectively, and so on …

Note: This visual step is important as it allows the user to fundamentally understand the communication between each nodes and its functioning.

Let's take a look at the bringup.launch file we previously launched.

Type

```
roscd mbsrobot/launch
```

Then[7]

```
nano bringup.launch
```

[7] The **nano** command allows to edit a file in Linux from the terminal.

This screen should appear:

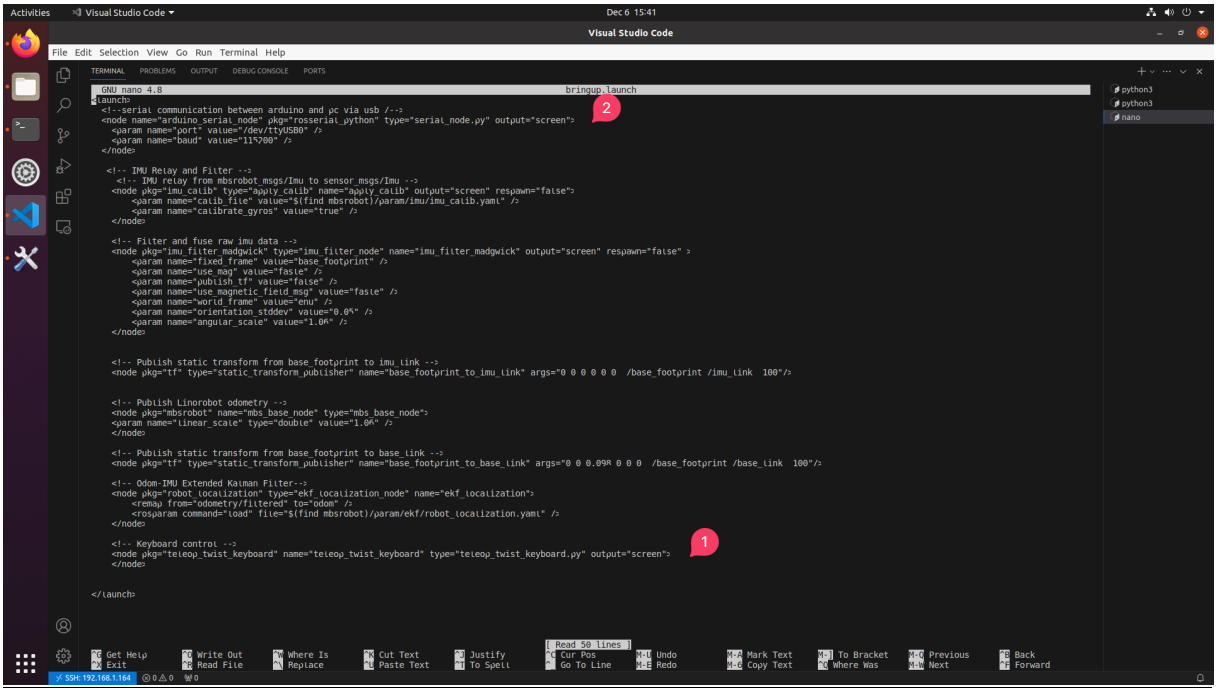


Figure 15: bringup.launch file

1) This line is responsible for the launching of the **/teleop_twist_keyboard** node
2) This line is responsible for the launching of the **/arduino_serial_node** node

Each node we have previously seen in Figure 13: RQT graph, appears here in this launch file.[8]

Notes: When building a new robot, it may happen that when launching the bringup.launch file, there is some error. Please refer to the troubleshooting section.

For **better precision** on the **IMU readings** and **robot control**, it may be preferable to **calibrate** the IMU and **tune** the correct PID constant values for the velocity of the robot.
All the process is explained in the **"Mbsrobot ros smart car operating instructions.pdf"** file.

If the user is interested in understanding the C++ or python code behind the node, he will need to **locate the package** from where the node is launched, and type

```
roscd mbsrobot/src
```

Then (For the /mbs_base_node for exemple)

```
nano mbs_base.cpp
```

[8] Type CTRL+X to exit this window

Control the Moebius robot with Matlab/Simulink

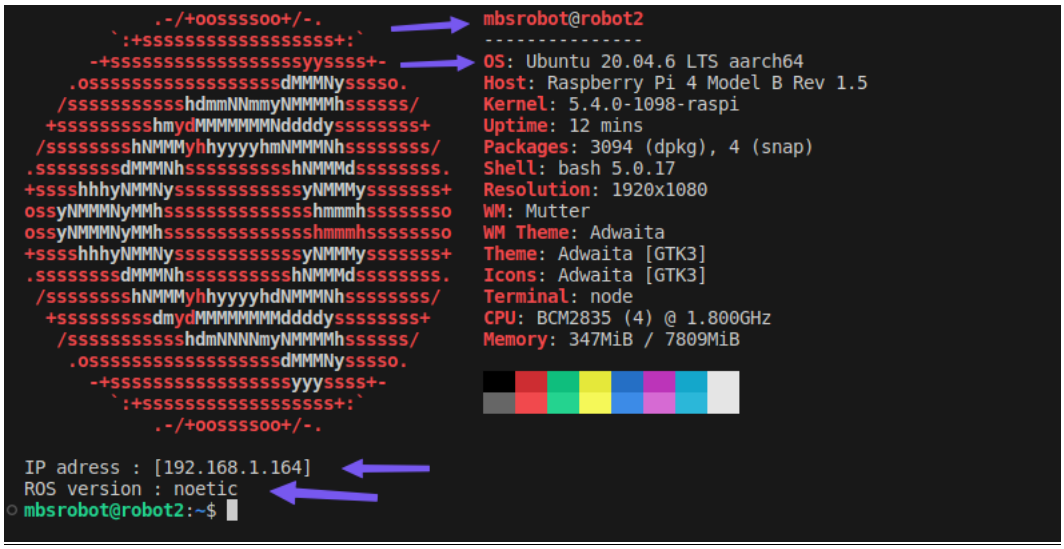
Prerequisites:

- ROS toolbox for MATLAB
- Matlab coder
- Simulink coder
- Python (The version needed can be different according to which MATLAB version we are working on)

The ROS toolbox is available from Matlab R2020b on Windows,Mac and Linux to Matlab R2023b.

First, we need to configure the Python environment in Matlab. All is detailed in this link:
https://www.mathworks.com/help/ros/gs/ros-system-requirements.html

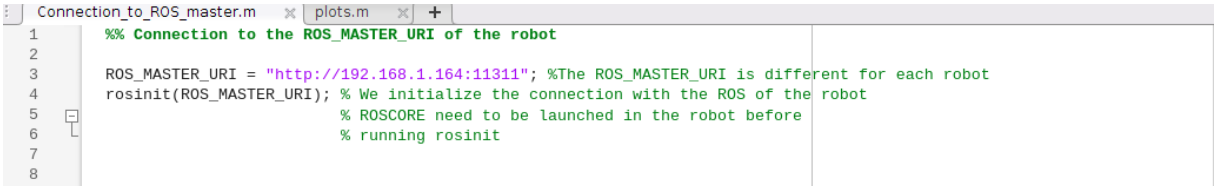
Then, for controlling the robot through Matlab, we need to go to the directory page from the robot in the terminal or from the remote computer (as explained in the previous section)



And type

```
roslaunch mbsrobot bringup.launch
```

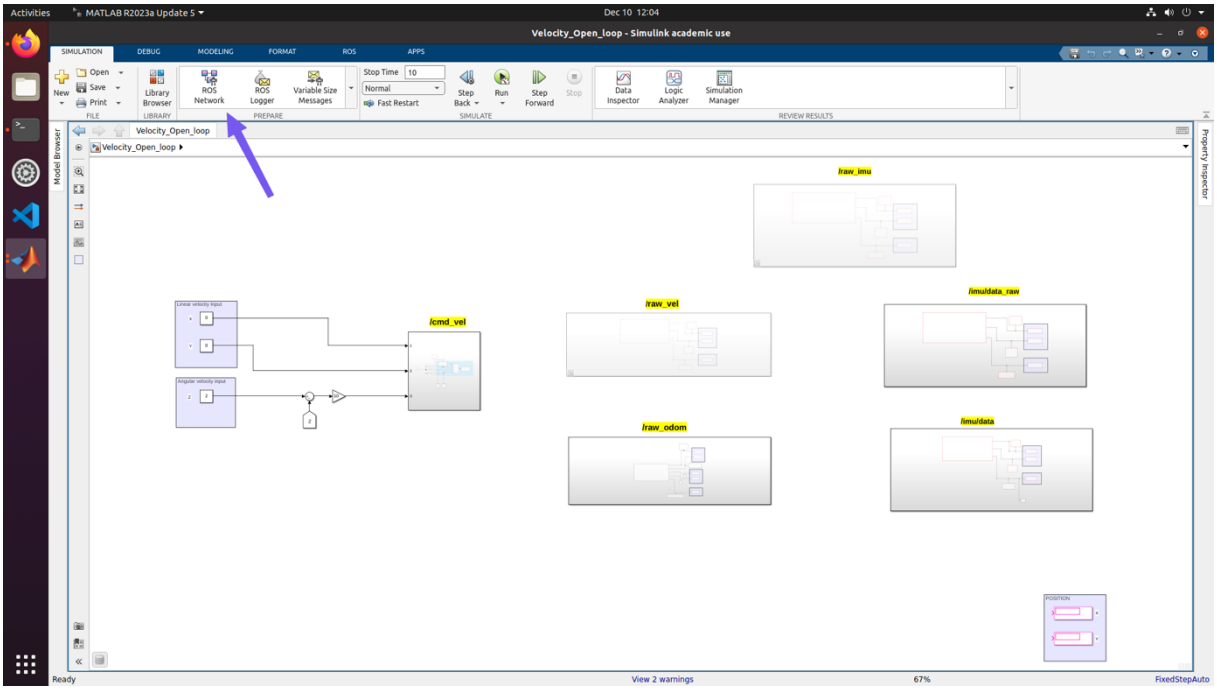
Now in Matlab, we enter this command:



We need to enter the IP address that appears in ROS_MASTER_URI
If the connection is successful, Matlab has now created a node and connected to the ROS master, and we can now control the robot through Matlab/Simulink rather than Ubuntu.

In the computer lab, open the **Velocity_Open_Loop.slx** file within the Robot cars/Moebius/Simulink/Velocity command open loop directory.

We first need to check the connection with the **ROS MASTER**.
Go to **ROS Network**



Then, we need to click on **Custom** and enter the IP address of the ROS MASTER

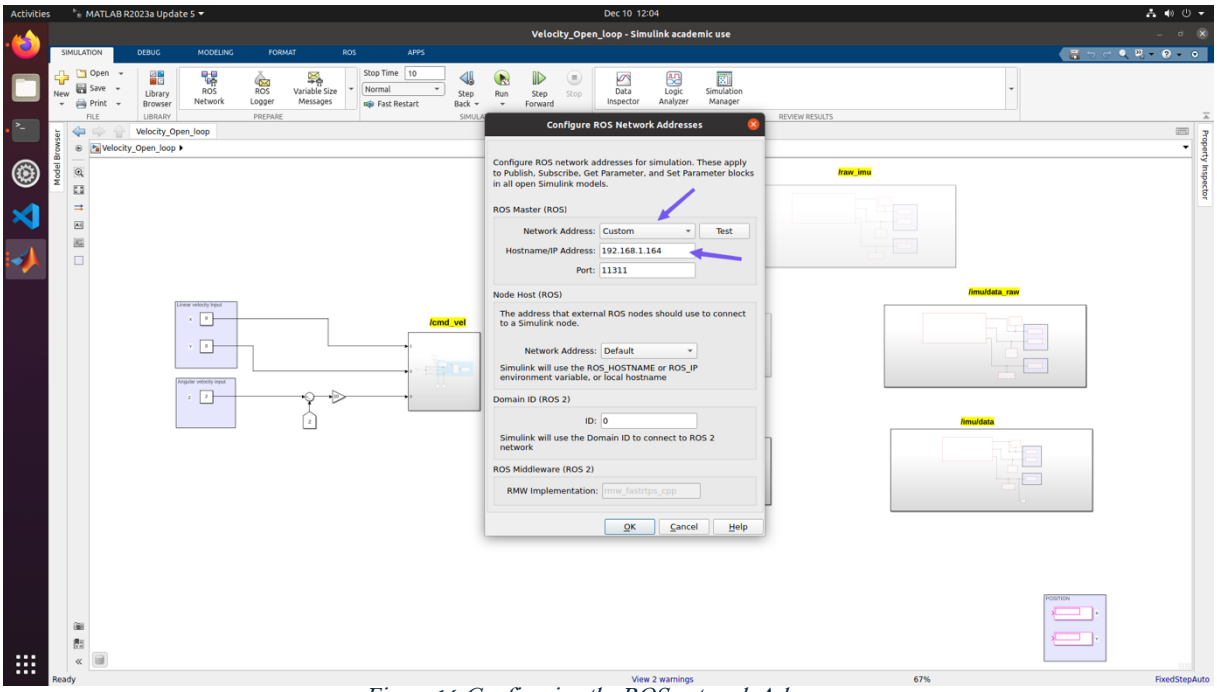


Figure 16: Configuring the ROS network Adressess

Then, press **TEST** and if the connection is successful, we can now control the robot from Simulink.

Explanation of the Simulink diagram

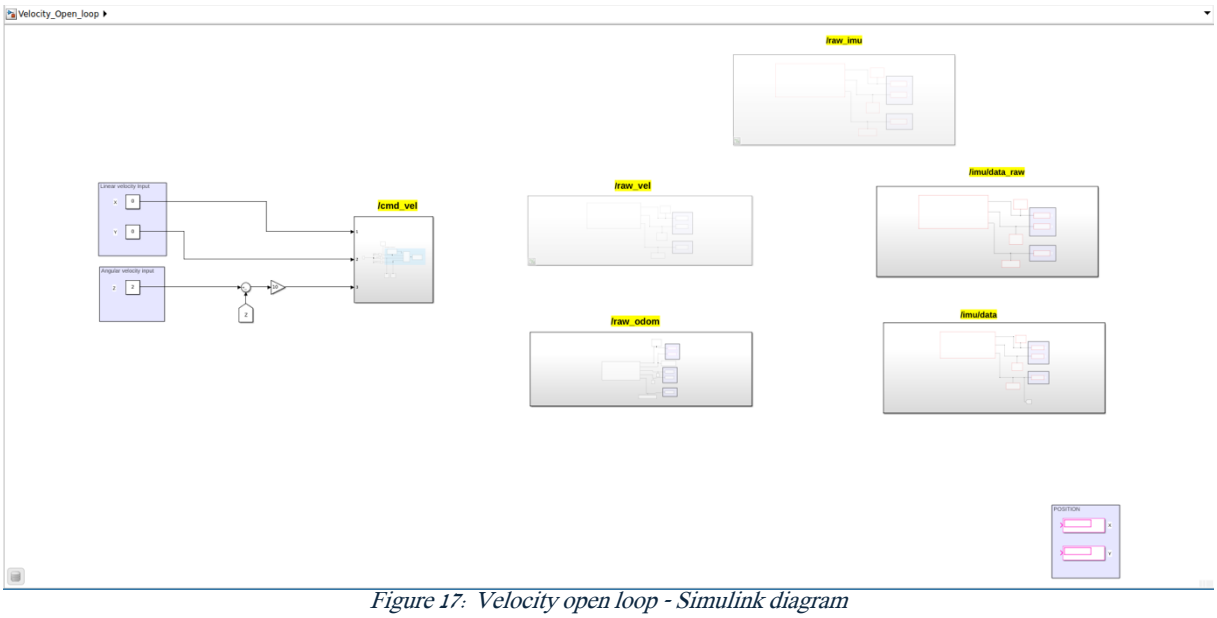


Figure 17: Velocity open loop - Simulink diagram

This Simulink diagram was built based on the **RQT_graph** we have previously seen (Figure 13: RQT graph)
Basically, we sent **velocity commands** to the motors (X and Y linear velocity, Z angular velocity) through the topic **/cmd_vel**.

Let's take a look into the **/cmd_vel** subsystem,

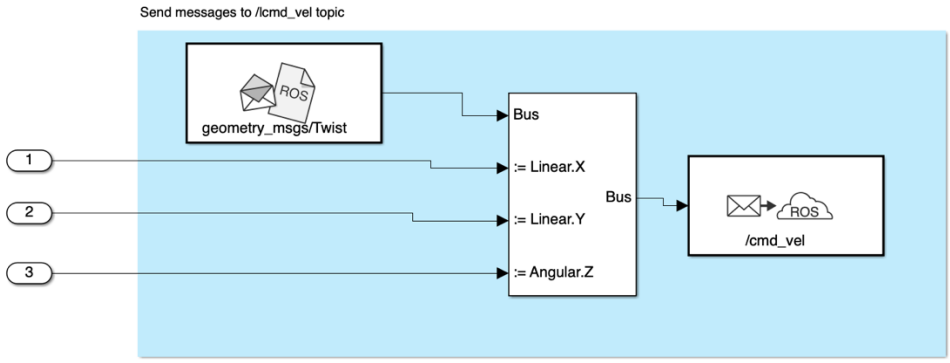


Figure 18: /cmd_vel subsytem

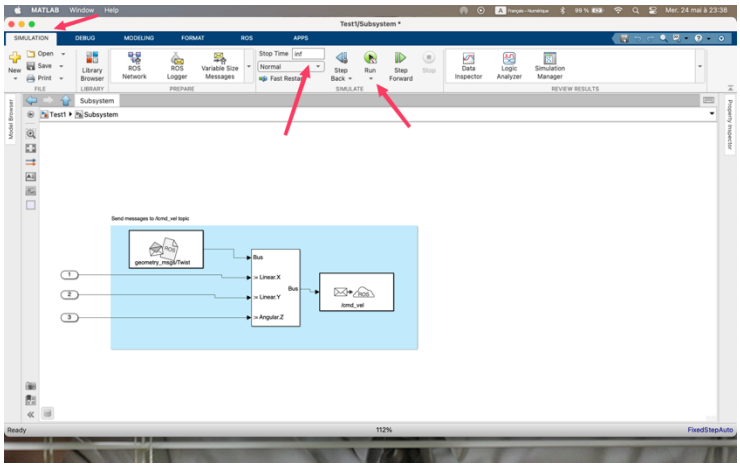
As explained before, **nodes** communicate each other through **topics** by transmitting **messages**.

1. In the example, we assign values to Linear.X , Linear.Y and Angular.Z (the type of message is **geometry_msgs/Twist**).

2. After assignment, the **message** is transmitted through the **topic** "/cdm_vel"

For easy use, we made a subsystem containing the above system. Now, we simply put step function blocks (with the desired amplitude = velocity) and we are ready to launch the robot.

To do so, we go to the **Simulation** tab, we put **inf** in the **Stop time** and we Run!



That's it! The robot is now supposed to move as he used to with the keyboard on Linux, but this time, with **Simulink** only!

In the **Simulink diagram**, we also have five other subsystems.
Basically, those blocks represents the topic also being used during the bringup.launch (see Figure 13: RQT graph).

When trying to plot the data outputted from the /raw_vel et /raw_imu topic, we get an ERROR from Simulink saying that these topics use custom messages. We need to find a way to import custom messages to Matlab/Simulink to exploit those data.

Each topic passes data after/before being processed by the nodes.

- **/raw_vel:**

Current velocities of the robot (data from the encoder)
When trying to plot the data, it seems that the response is not really smooth. Need to check why.

- **/raw_imu**

Raw data from the IMU (MPU6050)
When trying to plot the data, it seems that there is an offset with the reference values (values from /cmd_vel). Need to calibrate the IMU.

- **/imu/data**

Processed data from the IMU after the Extended Kalman Filter (EKF) filtering.
When trying to plot the data, it seems that there is an offset with the reference values (values from /cmd_vel). Need to calibrate the IMU.

Future Tasks

- For any reason, when powering up the robot, the wheels move, and we can hear a noise from the robot. If we look at the driver board, this is due to one of the wheels that is apparently "still moving". Check Arduino code to disable this.
- The /raw_vel and /raw_odom topics use custom messages created by Moebius. Therefore, we cannot access the messages from Simulink. Find a way to access custom messages in Simulink/Matlab or maybe change the code in ROS to use existing messages instead of the custom ones.
- At the end of each velocity command, there seems to be a little drift? why ?
- Retrieve data from the different topics/sensors

Troubleshooting

When launching the bringup.launch file, we may get an error saying "dev/ttyUSB0" not found This may be due to the fact that the Raspberry pi doesn't recognize the USB cable connected to the Arduino or It may be connected to a port with a different name. To fix this, we first check if there is a port connected (while the USB cable is connected)

```
ls /dev/ttyUSB0
```

If /dev/ttyUSB0 appears, that means that the port is detected.
If not, check

```
ls /dev/ttyACM0
```

Or

```
ls /dev/ttyACM1
```

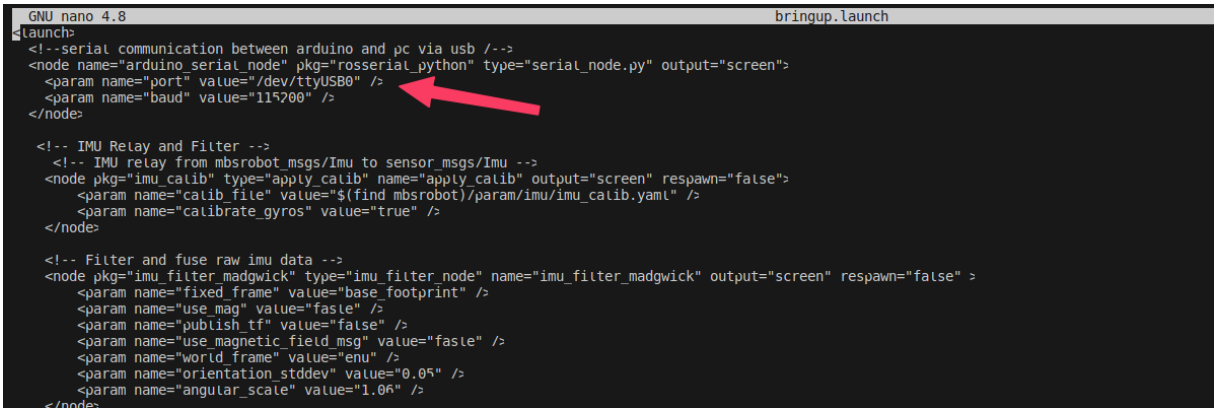
If one of this port is detected, go to the bringup.launch file

```
roscd mbsrobot/launch
```

Then

```
nano bringup.launch
```

and change this line with the corresponding port:



If an error occurred during the bringup saying “Device not sync, … , this may be due to Groovy Arduino”, that means that the Moebius shield is obsolete and need to be change.

Appendix

Build a new robot

Hardware Part:

For building a new robot, please refer to the file: **“ROS Car with STM32MEGA2560 Controller Installation Manual.pdf”**

For some reason, only DC motors from Moebius are compatible with the Arduino code uploaded. Indeed, the encoder values are not right when the motors are not from Moebius.

Software Part:

First, we need to **add** the required library to the Arduino IDE.
Please refer to this link: https://docs.arduino.cc/software/ide-v1/tutorials/installing-libraries#
In total, there are **four zip files** to add.

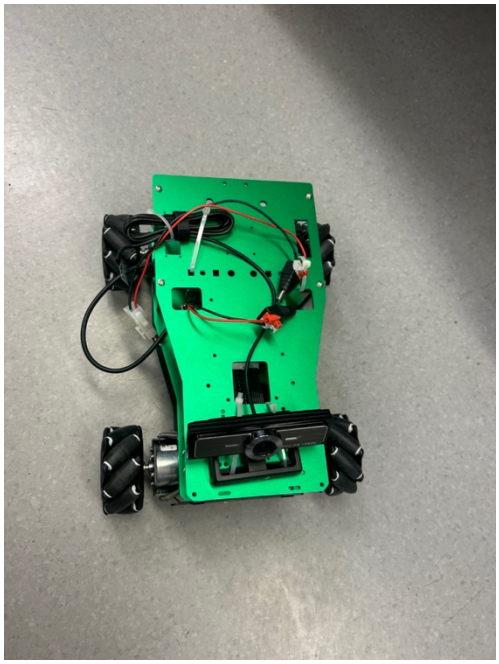
Then, we need to upload the Arduino code to the Arduino Mega 2560 + Driver board.
In the computer at the lab, open the Arduino code **“Mecanum.ino”** located within the Robot Cars/Moebius/Arduino packages directory.

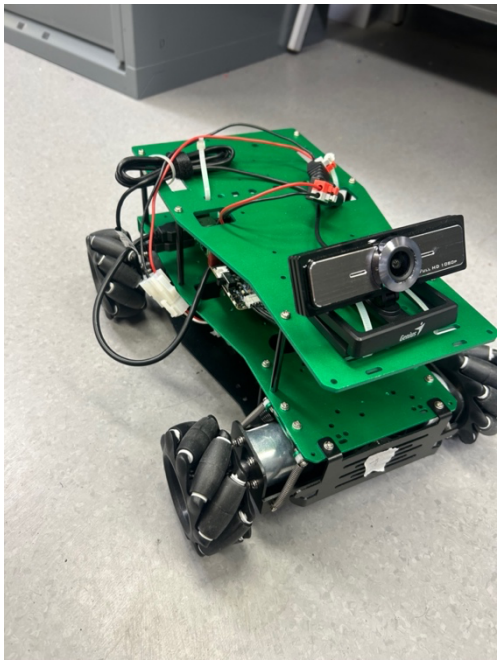
Then, we need to **write** the corresponding .img file on a new 64GB Micro SD card.
Please refer to **Appendix: Raspberry Pi imager**.

That’s it, the robot is ready to use ![9]

[9] **For any question regarding the ROS code or software, contact Andy:** 1260105099@qq.com

Yahboom X3:





This robot is the **second one** we worked on in the lab.
We started to work on this robot after the first one (Moebius) **caught on fire**. Moreover, this robot comes with more advanced ROS packages and launch files as line following algorithms, Lidar mapping, etc …

It is **highly recommended** for the user to read the Yahboom X3 repository (ROS 1 section only): http://www.yahboom.net/study/ROSMASTER-X3

Here is the official Github[10]: https://github.com/YahboomTechnology/ROSMASTERX3

Some of the packages used in the repository may be outdated.
For more information, send an email to: support@yahboom.com

Regarding the **Hardware**, no modifications were made except for the Webcam.

How to power on the robot?

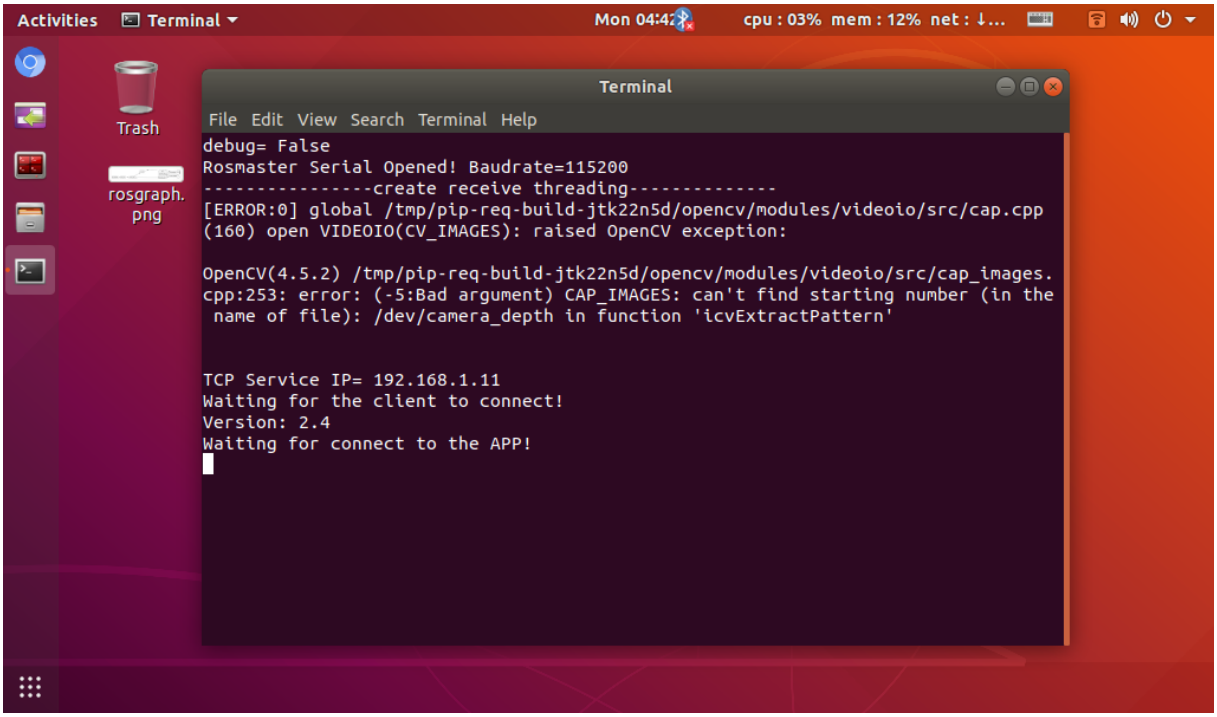
The steps are similar to the Moebius robot.

Here some **information** about the robot (including the password):

Username	pi
Password	yahboom
OS version	Ubuntu 18.04.6 LTS
ROS version	Melodic

[10] The Github page contains the same file as the ROSMASTER X3 repository.

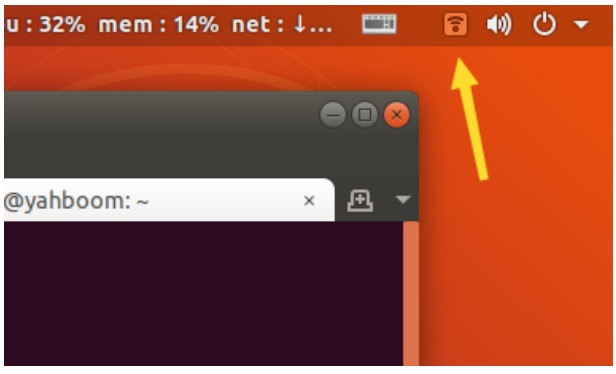
Once the **loading screen** is over, a terminal window is displayed.



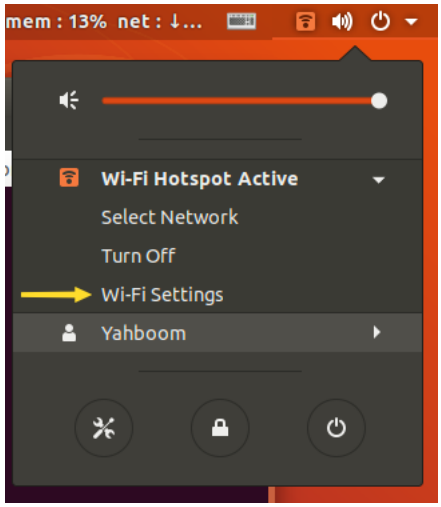
Recall: For SSH connection, the robot must be connected as the same WIFI as the host computer.

At the moment, the robot is in **Hotspot mode**. The user **must disable** the hotspot mode and connect the robot to the WIFI.

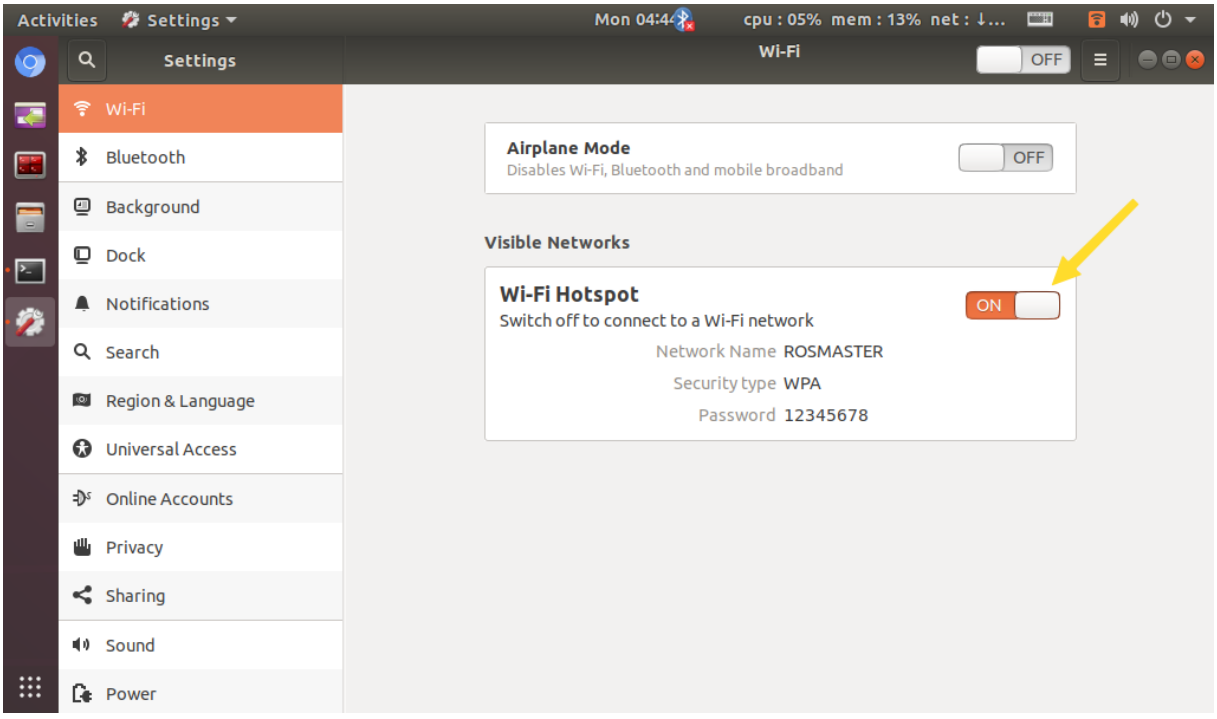
To do so, go to the top right corner.



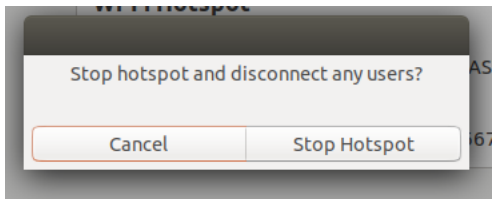
Click on **Wi-Fi Hotspot Active** and **Wi-Fi settings:**



Then, turn off **Wi-Fi Hotspot**

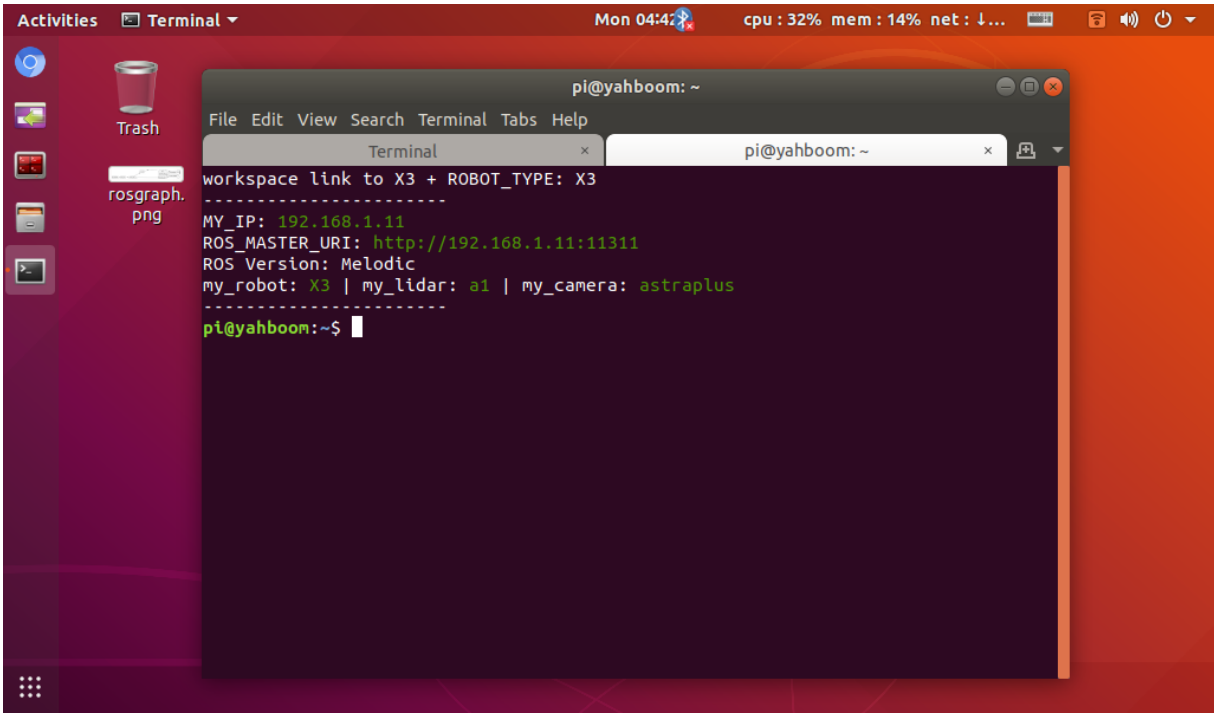


Finally, **Stop Hotspot**



A list of **Visible Networks** should appear.
Select the same Wi-Fi as the computer host and open a new window terminal.

This screen should appear:



The IP address on the terminal above is not the right IP address. The user must first execute the steps described above.

Control the robot directly from the Raspberry Pi

As explained for the Moebius robot, for controlling the robot with a keyboard through Linux, once the connection is established via SSH, or directly from the robot,

Type

```
roslaunch yahboomcar_bringup bringup.launch
```

Once the bringup launched, we have the same loading screen appearing as we had we the Moebius robot , indicating that **the ROS master** (the robot) has been launched.

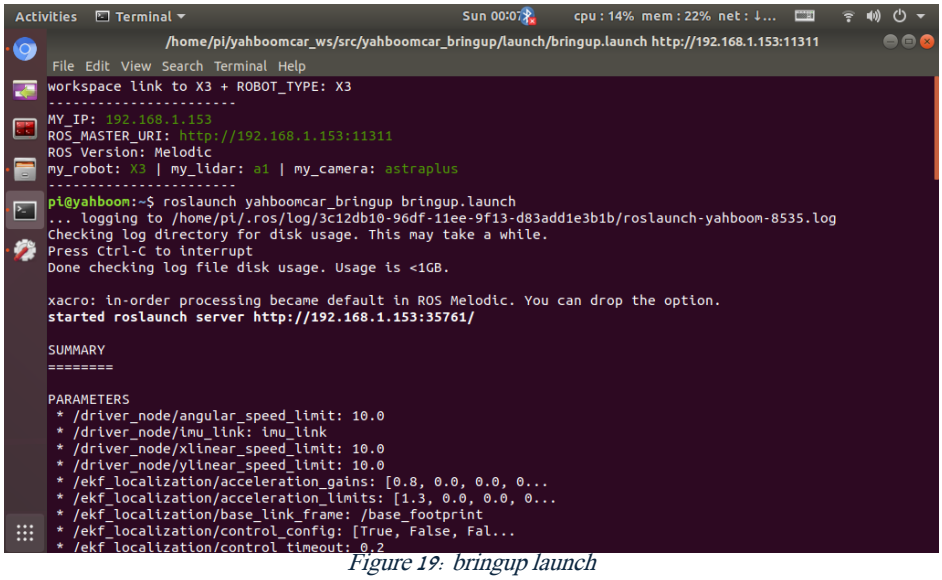


Figure 19: bringup launch

Explanation of the launch file process

As explained earlier, once our bringup.launch file has been launched, several nodes has been called.

Let's **visualize** the **communication between nodes** and **topics** with the rqt_graph tool.

```
rqt_graph
```

This screen should appear:

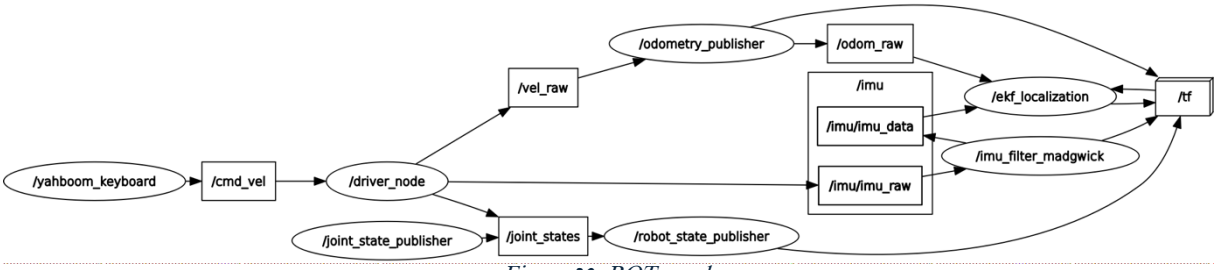


Figure 20: RQT graph

This graph is very similar to the one we had with the Moebius robot (except that the nodes and topics have different names). However, the process and the functioning are the same.

For **better precision** on the **IMU readings** and **robot control**, it may be preferable to **calibrate** the IMU and **tune** the correct PID constant values for the velocity of the robot.
Please refer to the ROSMASTER X3 repository [].

If the user is interested in understanding the C++ or python code behind the node, he will need to **locate the package** from where the node is launched, and type

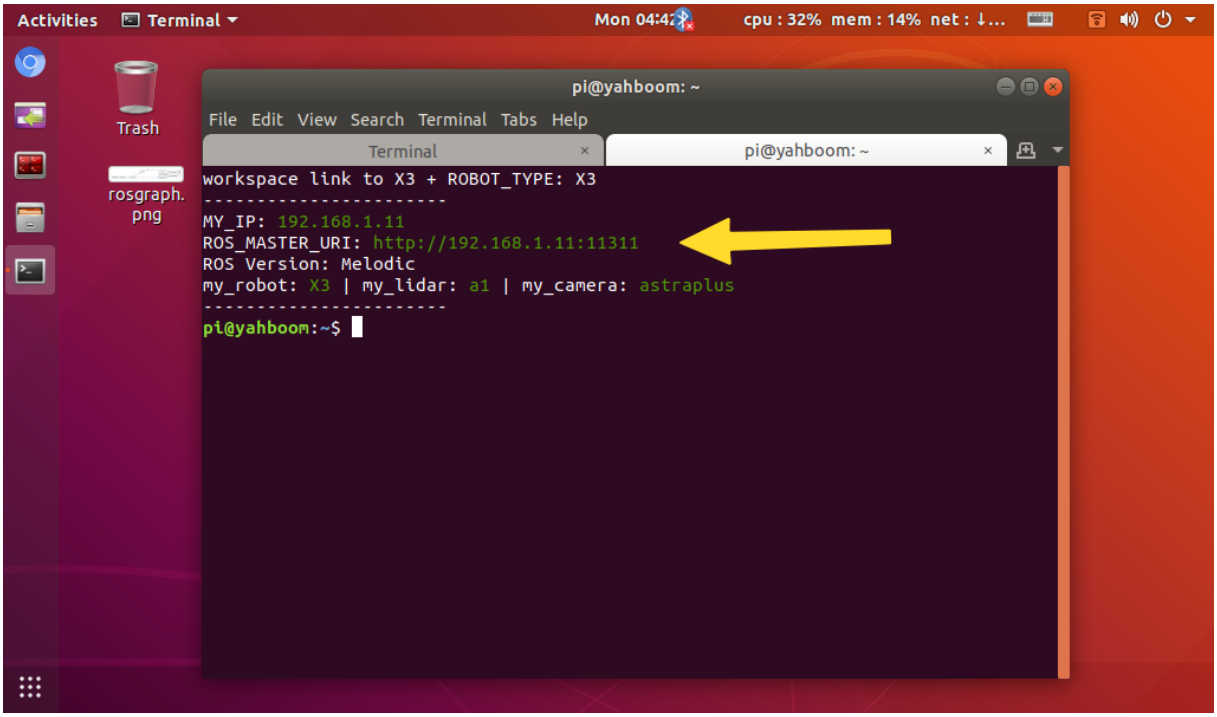
```
roscd yahboomcar_bringup/src[11]
```

[11] As explained in the ROS Basics sections, the src folder contains the .cpp or python file.

Control the Yahboom X3 robot with Matlab/Simulink

The procedure for controlling the robot with Matlab and Simulink is exactly the same as we did with the Moebius robot.

First, we need to go to the directory page from the robot in the terminal (or from the remote computer) and locate the **ROS_MASTER_URI:**



Then type

```
roslaunch yahboomcar_bringup bringup.launch
```

Now in Matlab, we enter this command:

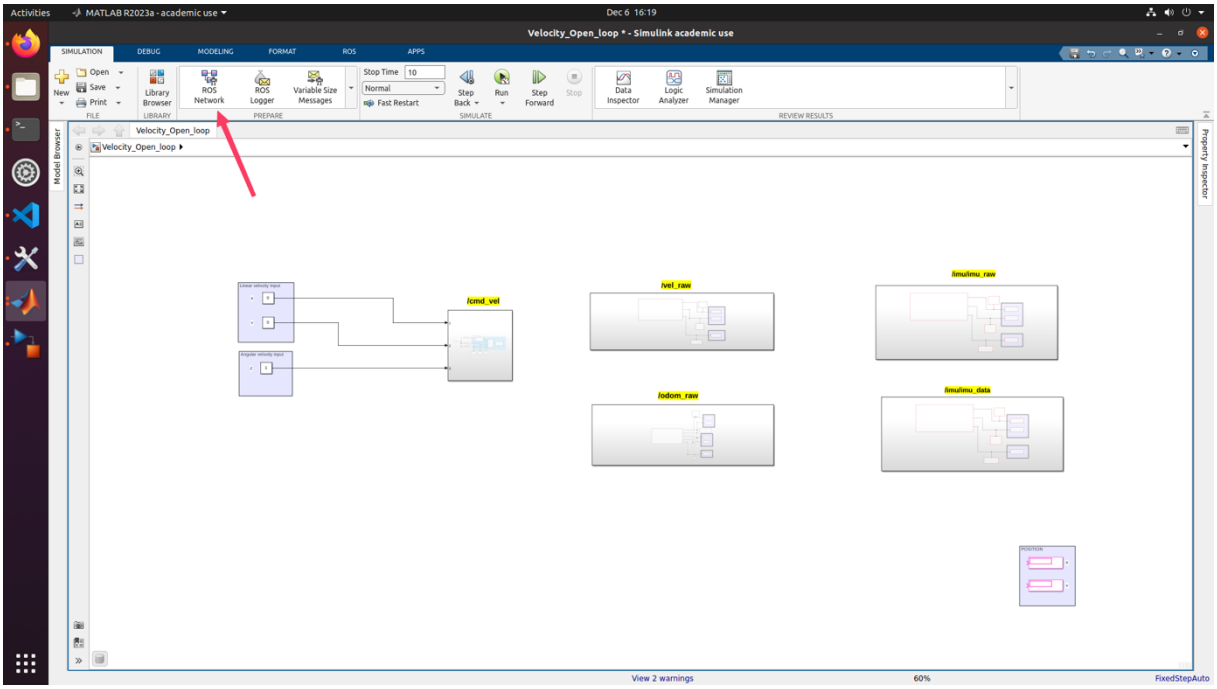
```
Connection_to_ROS_master.m | plots.m
1  %% Connection to the ROS_MASTER_URI of the robot
2
3  ROS_MASTER_URI = "http://192.168.1.164:11311"; %The ROS_MASTER_URI is different for each robot
4  rosinit(ROS_MASTER_URI); % We initialize the connection with the ROS of the robot
5                           % ROSCORE need to be launched in the robot before
6                           % running rosinit
7
8
```

We need to enter the IP address that appears in ROS_MASTER_URI
If the connection is successful, Matlab has now created a node and connected to the ROS master, and we can now control the robot through Matlab/Simulink rather than Ubuntu.

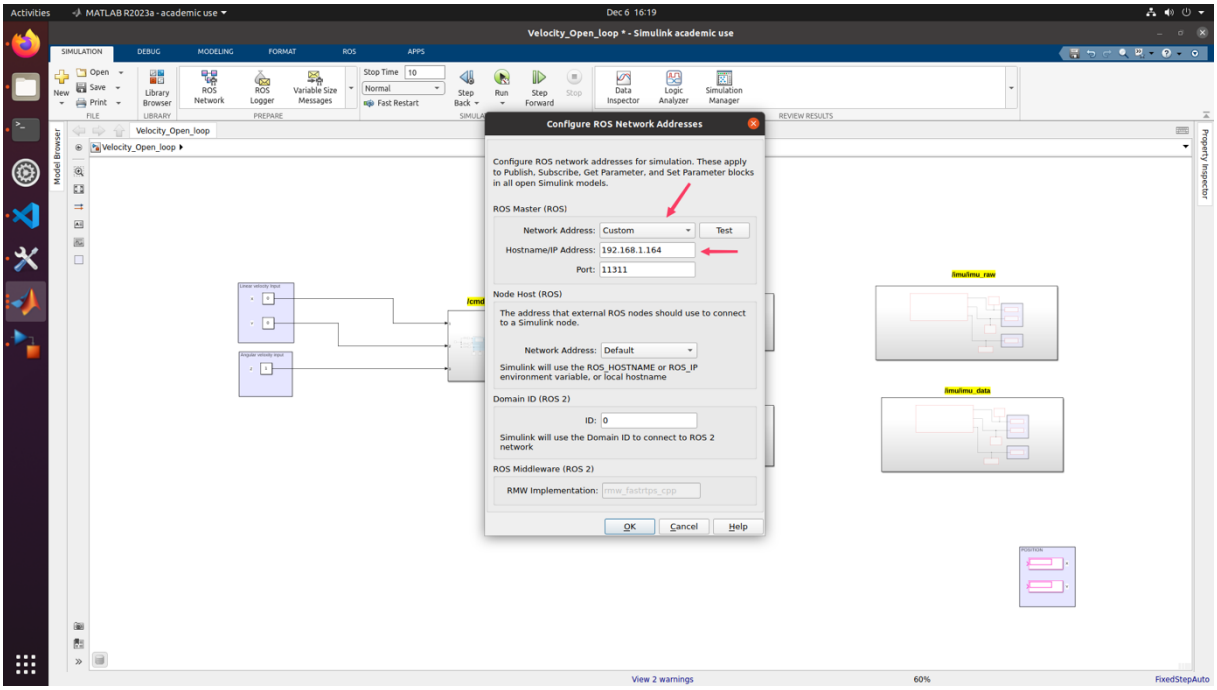
In the computer lab, open the Velocity_Open_Loop.slx file within the Robot cars/Yahboom/Simulink/Velocity command open loop directory

We first need to check the connection with the ROS MASTER.

Go to **ROS Network**



Then, we need to click on **custom** and enter the IP address of the **ROS MASTER**



Then, press **TEST** and if the connection is successful, we can now control the robot from Simulink.

Explanation of the Simulink diagram

Figure 21: Simulink Diagram for Open loop velocity commands

This **Simulink diagram** was built based on the **RQT graph** we have previously seen (Figure 20: RQT graph)
Basically, we sent velocity command to the motors (X and Y linear velocity, Z angular velocity) through the topic **/cmd_vel**. (see [] for more detailed explanation)

In the **Simulink diagram**, we also have **four other subsystems**.
Basically, those blocks represents the topic also being used during the bringup.launch (see Figure 19: bringup launch)

Each **topic** passes data after/before being processed by the nodes.

- **/vel_raw:**

Current velocities of the robot (data from the encoder)
When trying to plot the data, it seems that the response is not really smooth. Need to check why.

- **/imu/imu_raw**

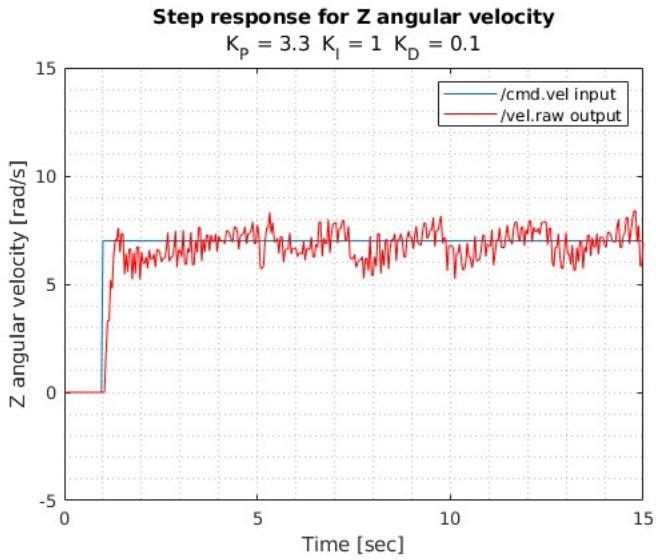
Raw data from the IMU (MPU6050)
When trying to plot the data, it seems that there is an offset with the reference values (values from /cmd_vel). Need to calibrate the IMU.

- **/imu/imu_data**

Processed data from the IMU after the Extended Kalman Filter (EKF) filtering.
When trying to plot the data, it seems that there is an offset with the reference values (values from /cmd_vel). Need to calibrate the IMU.

Results from the lab

While **tuning** the right **PID** values for the robot[12], the best response I could get was not smooth at all and had a low frequency tendency which was not converging.[13]



Future Tasks

- Currently at the lab, there is two already built Yahboom robots. For any reason, one of them don't work anymore when trying to power it on. This may be due to a default in the driver board (Yahboom board)
- Tune right PID values for the velocity commands.
- Model and 3D print a part to hold the webcam on the robot.

Appendix

Build a new robot

There are two more robots at the lab that need to be build.
Ask Ruslan for help.

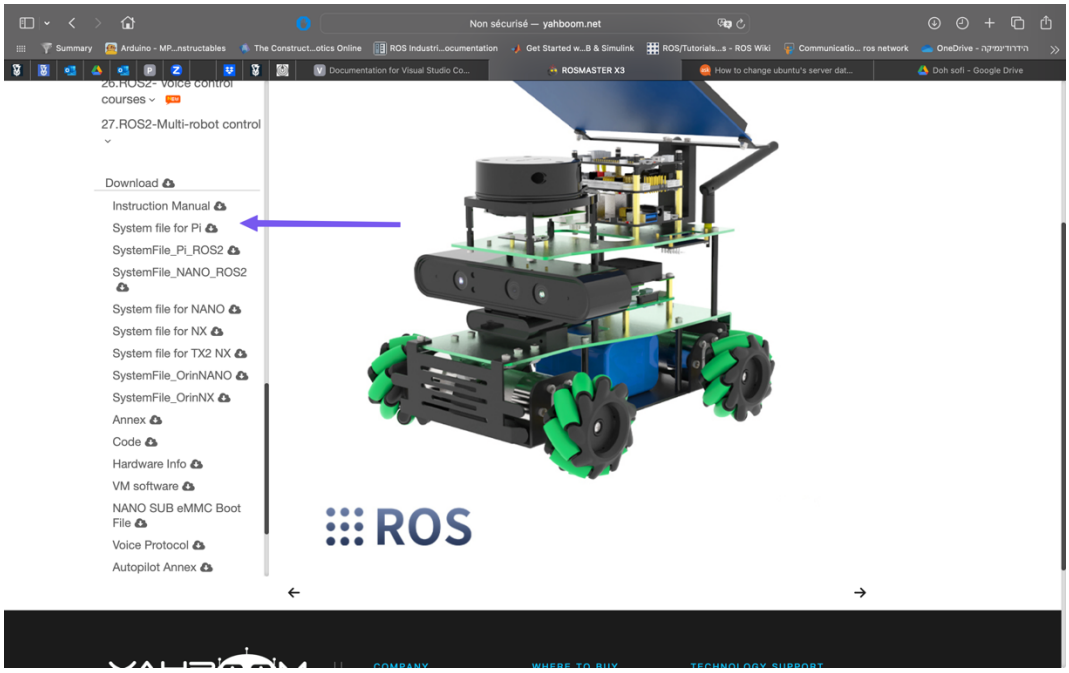
Hardware Part:
For building a new robot, please refer to the ROSMASTER X3 repository.

Software Part:

[12] See: http://www.yahboom.net/study/ROSMASTER-X3 (Section 9.2: Robot PID debugging)
[13] The scripts for the plots are located within the Robot Cars/Yahboom/Simulink directory in the computer in the lab.

We need to write the corresponding .img file on a new 64GB Micro SD card.

It can be found in the ROSMASTER X3 repository at the end of the page.



Or in the computer in the lab.
Please refer to **Appendix** for writing the .img file into the Raspberry pi.

Use the Webcam

To output the **webcam data stream** on the screen, type

```
roslaunch usb_cam usb_cam_test.launch
```

If the user wants to use the webcam as feedback output, follow this procedure.

First,

```
roscore
```

Then,

```
roslaunch usb_cam usb_cam_test.launch
```

Then,

```
rosnode list
```

The user should check what are the active nodes when launching the usb_cam_test.launch file.
Once the node is located, write a launch file that calls this launch file and use the webcam data stream as he pleases.

Make a simulated version of the mecanul wheel robot move with a Simulator

Unfortunately, we got some problem in the Lab during the year with the moebius robot during testing, and we had to find a way **to test algorithm** without real robots. So, we had to find a solution. The best option was to import a model of our robot in the **ROS simulator Gazebo** and make it move the same way as the real robot.

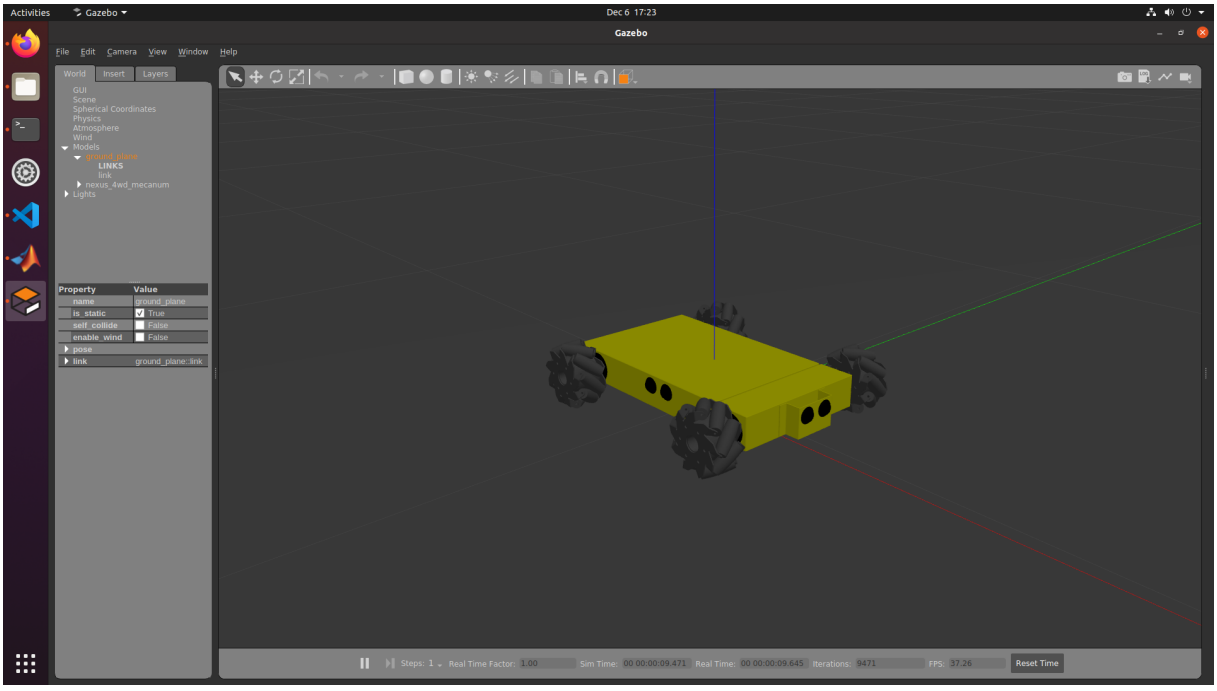
The **Gazebo simulator** has already been installed in the computer at the lab.
For more information and **learning about Gazebo** package and files structure, please refer to the following link: https://classic.gazebosim.org/tutorials?tut=ros_roslaunch&cat=connect_ros

We needed to find a model of our robot. A rough model (a car with four mecanum wheels) have been found on Github: https://github.com/RBinsonB/nexus_4wd_mecanum_simulator

To get familiar with the robot and the above package, one may launch the following launch file on the computer lab and get a preview of how it looks:

```
roslaunch nexus_4wd_mecanum_gazebo nexus_4wd_mecanum_world.launch
```

If we zoom in a little, we can see that the robot has **four mecanum wheels** and therefore dynamically behaves as our real robots.



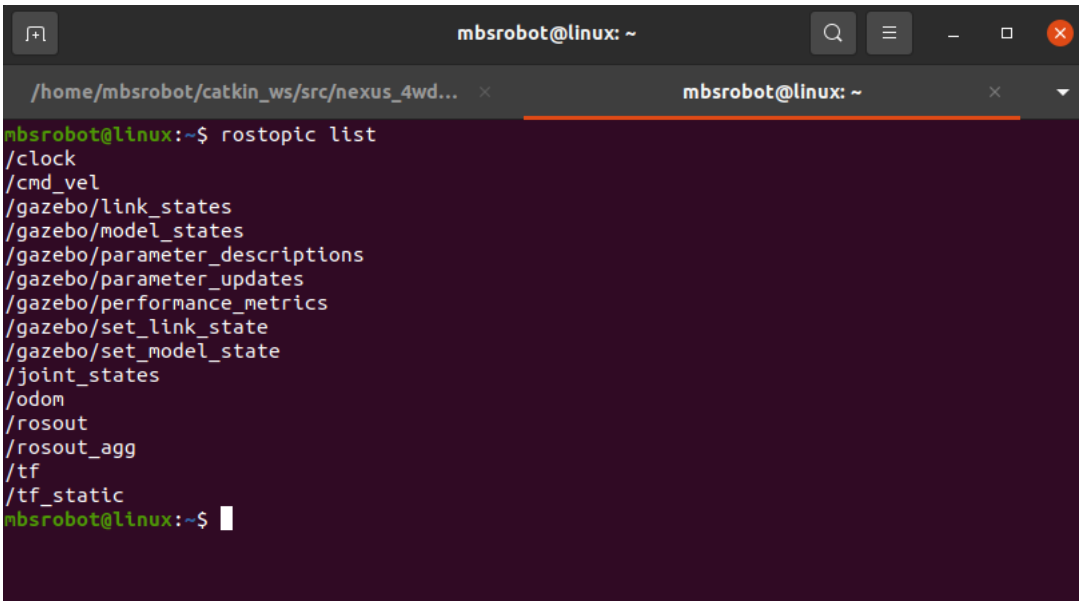
The **advantage** of using a **simulation** for our robot with Gazebo is that we can build **practically** every scenario possible.

We can add **"simulated" sensors** to the robot like laser, lidar, camera, etc … and make him move in any simulated environment we want. We can add walls, a white line on the ground that the robot needs to follow, a circuit, etc …

Also, during experiment labs, having a simulation allows to have **visualization** of the expected behavior of the real robot and **test Simulink models** before deploying them.
This setup is also **very modular** as we can add as many robots as we want and place them in coordinates we know.

In a new terminal

```
rostopic list
```



We can now see all the available topics for the simulated robot.

Recall: The /cmd_vel topic is responsible to control the linear and angular velocity of the robot. To do so, we need to publish desired values on this topic.

While the simulation is **still running** on Gazebo, on the new opened terminal:

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x:0.0 , y:1.0 , z:0.0} , angular: {
x:0.0 , y:0.0 , z:1.0}}'
```

This command means that we are **publishing** on the **/cmd_vel topic** with a rate of 10 $[Hz]$.
`geometry_msgs/Twist` is the type of **message** used when publishing on this topic.

Then we assign velocity values as we desire. Here, we assign $1[\frac{m}{s}]$ for the linear.y velocity and $1[\frac{m}{s}]$ for the angular.z velocity.

After typing this command, we should see the robot move accordingly.

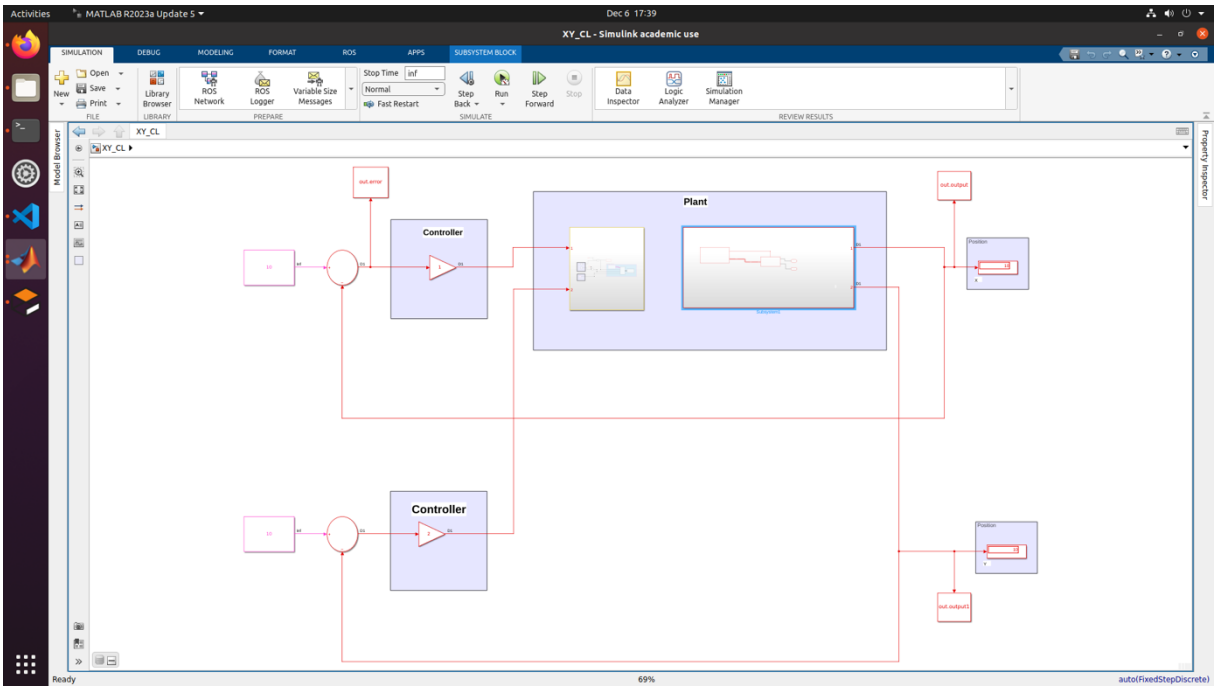
Make a simulated version of the mecanul wheel robot move with Matlab/Simulink

The steps **are similar to** what we did with the real robot in the previous section.
First, we need to launch the simulation as before:

```
roslaunch nexus_4wd_mecanum_gazebo nexus_4wd_mecanum_world.launch
```

In **Simulink**, the only setting we need to change is the **remote device** we want to connect to.

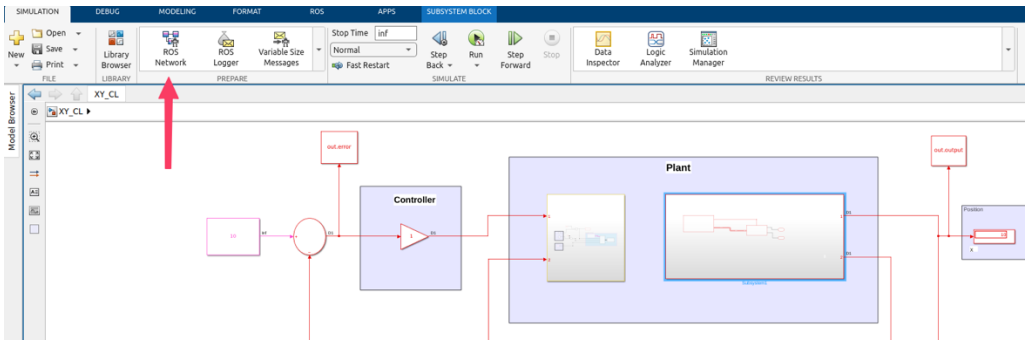
For the example, we are going to open the file called "XY_CL" located in the Robot Cars/Simulation/2D XY position CL one robot directory. This is a XY position closed loop on **one** simulated robot.[14]



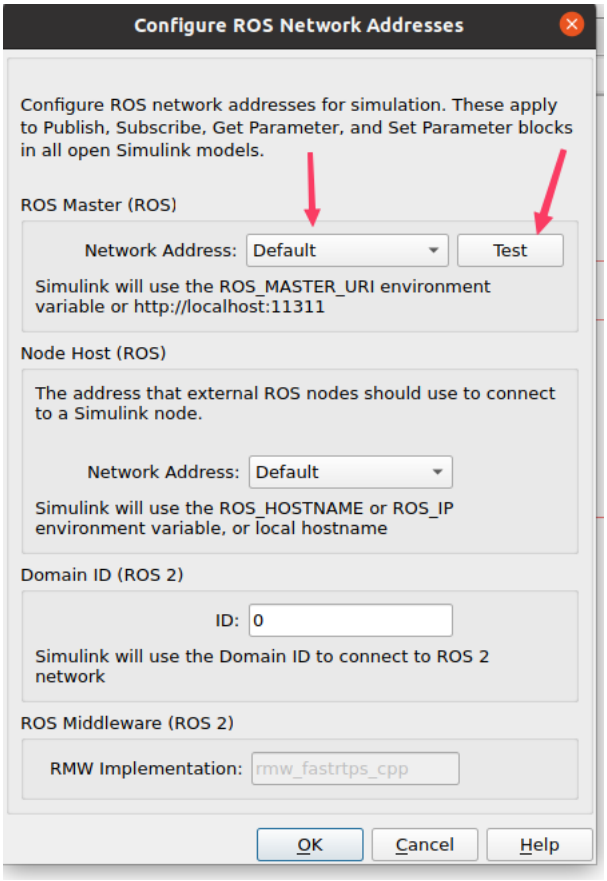
This **Simulink diagram** sends X and Y linear velocity commands to the robot. The **feedback data** is the X and Y position of the robot.

[14] We are later going to explaining the functioning of each Simulink diagram

Go to **ROS Network** in the **SIMULATION** tab,

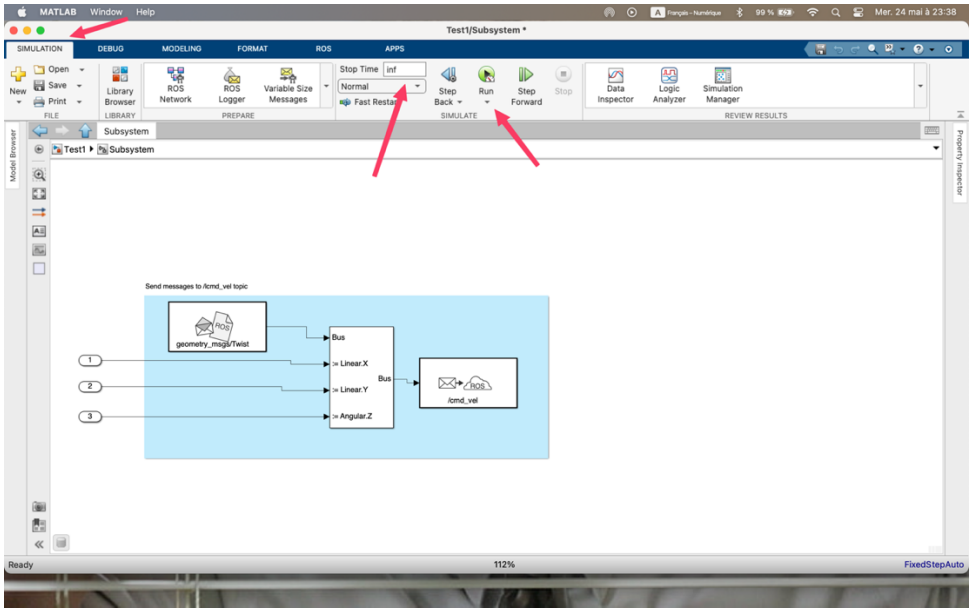


Then, choose **default** as the **Network Address** and press **Test**, to check the connection with the simulation.



If the Simulation has **successfully** been launched (so as the **ROSMASTER**), the connection should be established.

Now as we did with the real robots, in the **simulation** tab, choose the **Stop time**, and Run!



That's it! The robot is now supposed to move in the Simulation as the real robot.

Explanation of the launch file

Let's look at the **nexus_4wd_mecanum_world.launch** file we launched above.
For better understanding, edit the launch file:

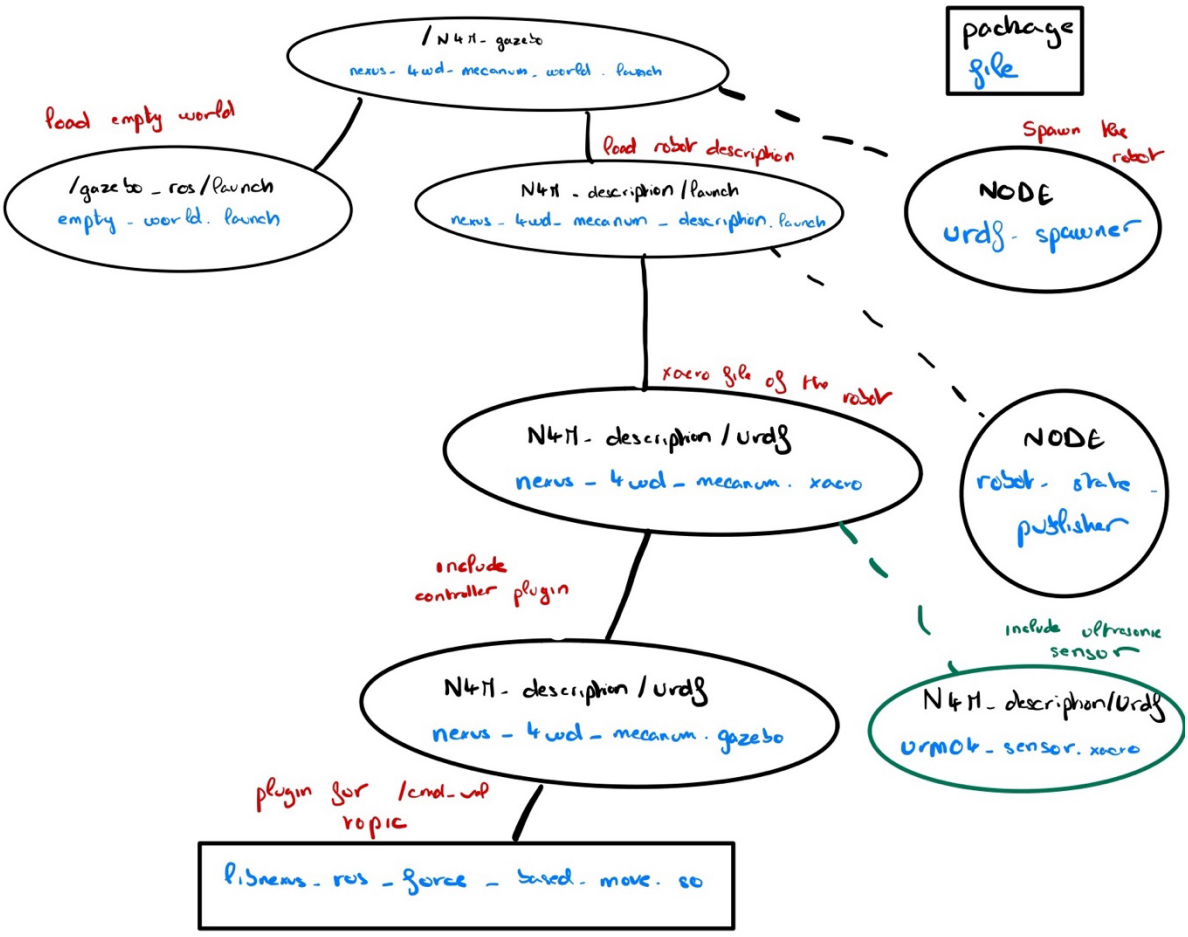
```
roscd nexus_4wd_mecanum_gazebo/launch
```

Then

```
nano nexus_4wd_mecanum_world.launch
```

```
  GNU nano 4.8                                                    nexus_4wd_mecanum_world.launch
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true" />
  <arg name="gui" default="true" />
  <arg name="headless" default="false" />
  <!-- <arg name="world_name" default="$(find nexus_gazebo)/worlds/nexus_4wd_mecanum.world" /> -->

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="0" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="use_sim_time" value="$(arg use_sim_time)" />
    <arg name="headless" value="$(arg headless)" />
    <!-- <arg name="world_name" value="$(arg world_name)" /> -->
    <arg name="paused" value="false"/>
  </include>

<!-- Load robot description -->
<include file="$(find nexus_4wd_mecanum_description)/launch/nexus_4wd_mecanum_description.launch" />

  <!-- Spawn the robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        args="-urdf -model nexus_4wd_mecanum -param robot_description -x 0 -y 0 -z 0.5" />

</launch>
```

Figure 22: Launch file of the simulated robot in an empty world



This scheme looks a litlle bit messy but It's not that hard to understand.

Everytime we launch the **nexus_4wd_mecanum_world.launch** file, we first **load an empty world**, **load the robot description,** create the node that spawns our robot and create the node that publish our robot states.

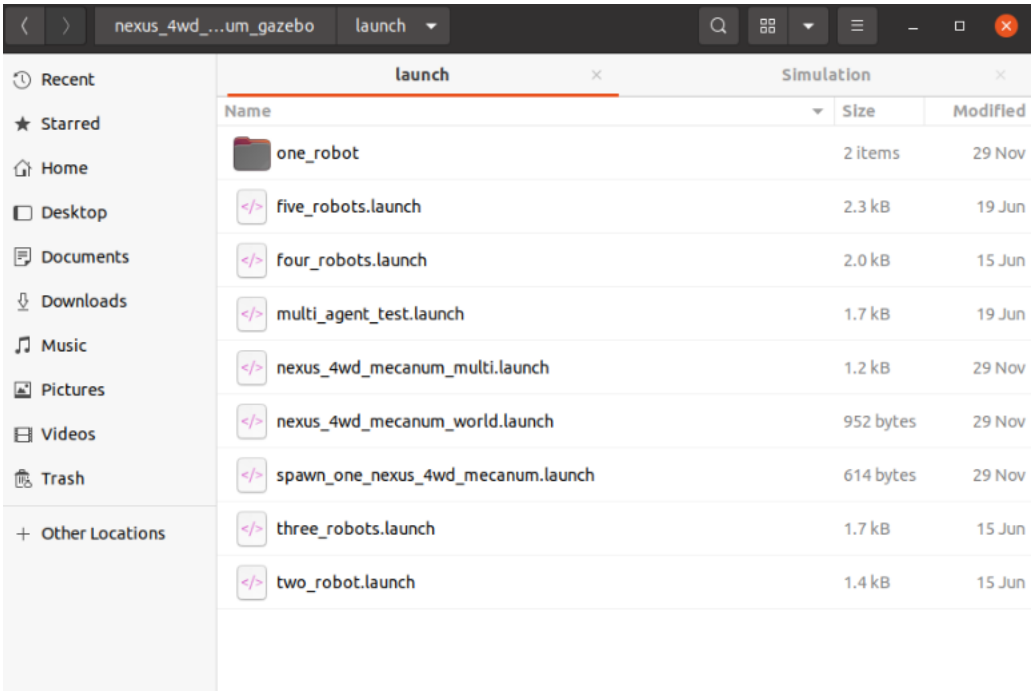
Inside the robot description launch file, we load the **xacro file** (file that describes the robot geometry) and optionnaly, we can add an ultrasonic sensor. This latter file, invokes the **nexus_4wd_mecanum.gazebo** file, that describes the dynamic of the robot.

This messy process can be reduced to two or three files but involve the user to have **experience** with ROS and Gazebo.

Let's look at the **other launch files** and **Simulink diagrams**.
In the lab computer,

```
roscd nexus_4wd_mecanum_gazebo/launch
```

We have multiple launch files:



The **files** called “two_robots.launch” , “three_robots.launch” , … , “five_robots.launch” spawn two robots, three robots, …, five robots respectively.

Five robots

Let's look at the **five_robots.launch** file:

```
roslaunch nexus_4wd_mecanum_gazebo five_robots.launch
```

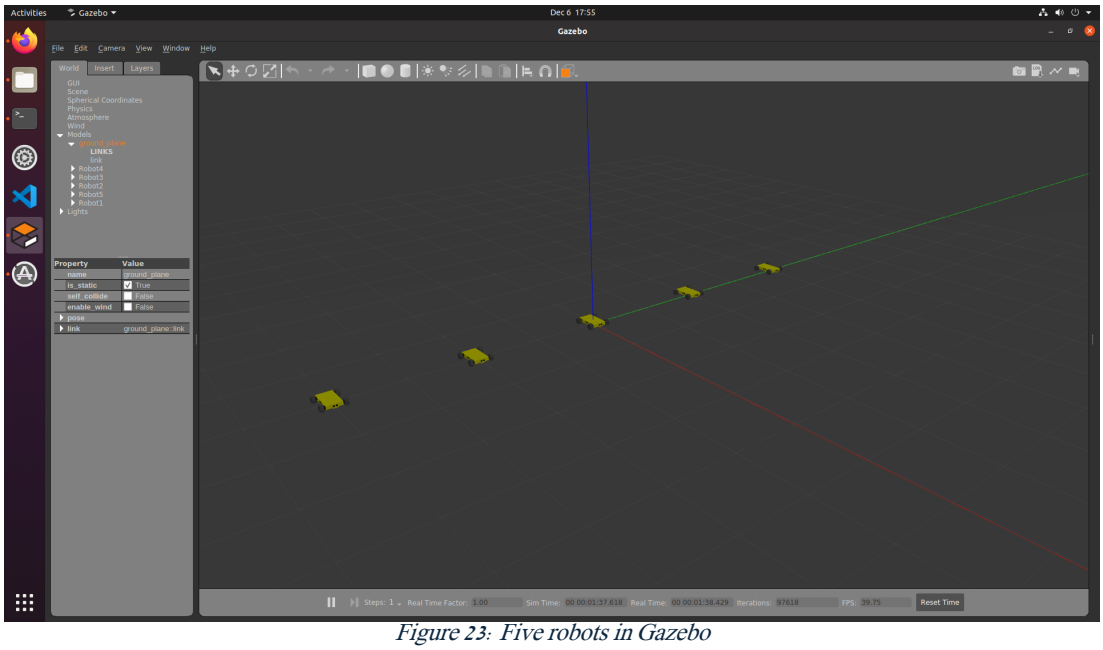
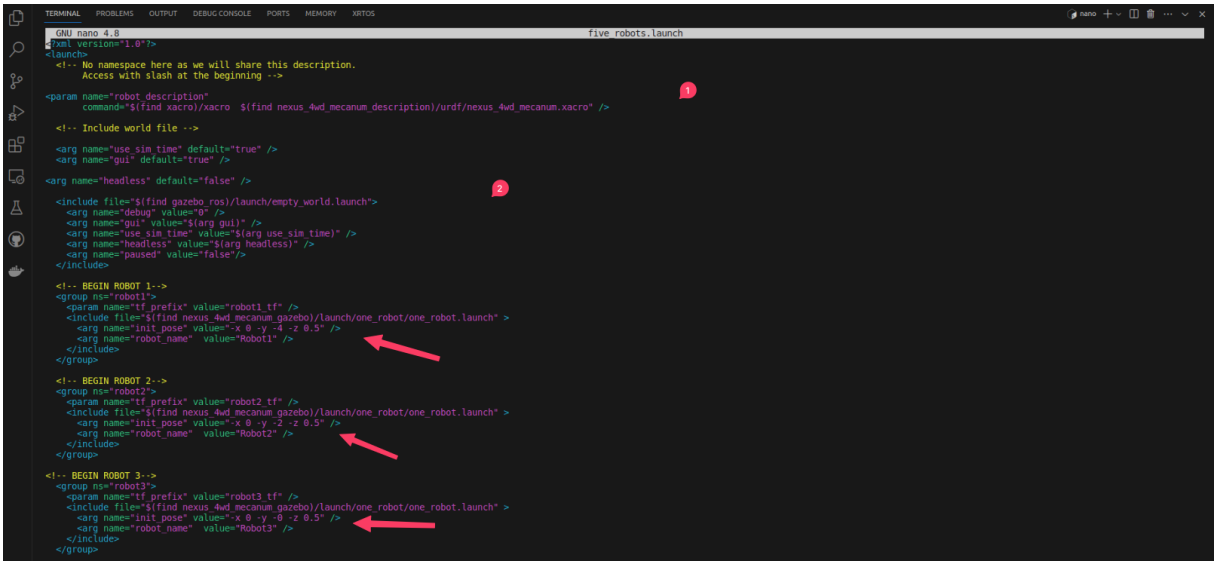


Figure 23: Five robots in Gazebo

As previously said, this file spawns five robots at **defined position**.
If we edit the launch file:

```
nano five_robots.launch
```



1) We first load the robot description file (.xacro)

The user can edit this above .xacro file and see how the robot was coded (not necessary)

2) We include the world file. As we can see, we summon an empty world, but this not mandatory. If the user wants to launch the five_robot.launch file in a world with walls for exemple, this can be done quite easily.

Then, for each summoned robot, we indicate his coordinates (pointing arrows).
For the Z coordinates, we enter 0.5. Otherwise, the robot will spawn inside the floor.

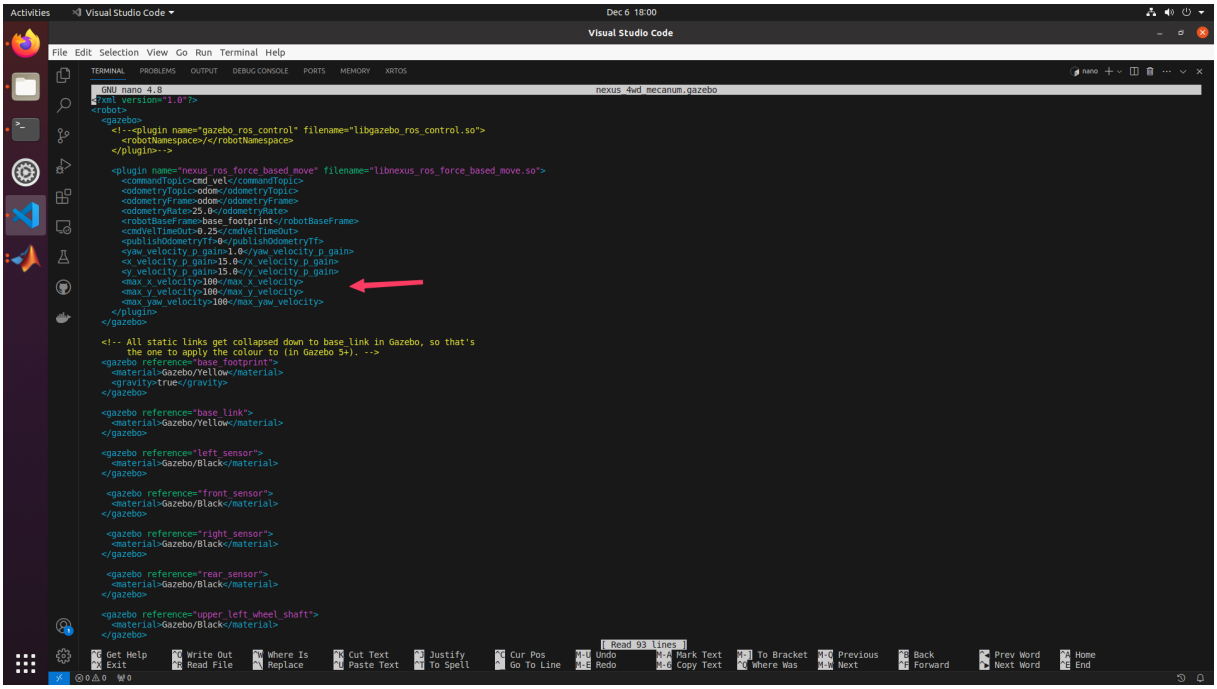
If the user wants to **change the maximum velocity** of the robot:

```
roscd nexus_4wd_mecanum_description/urdf
```

Then,

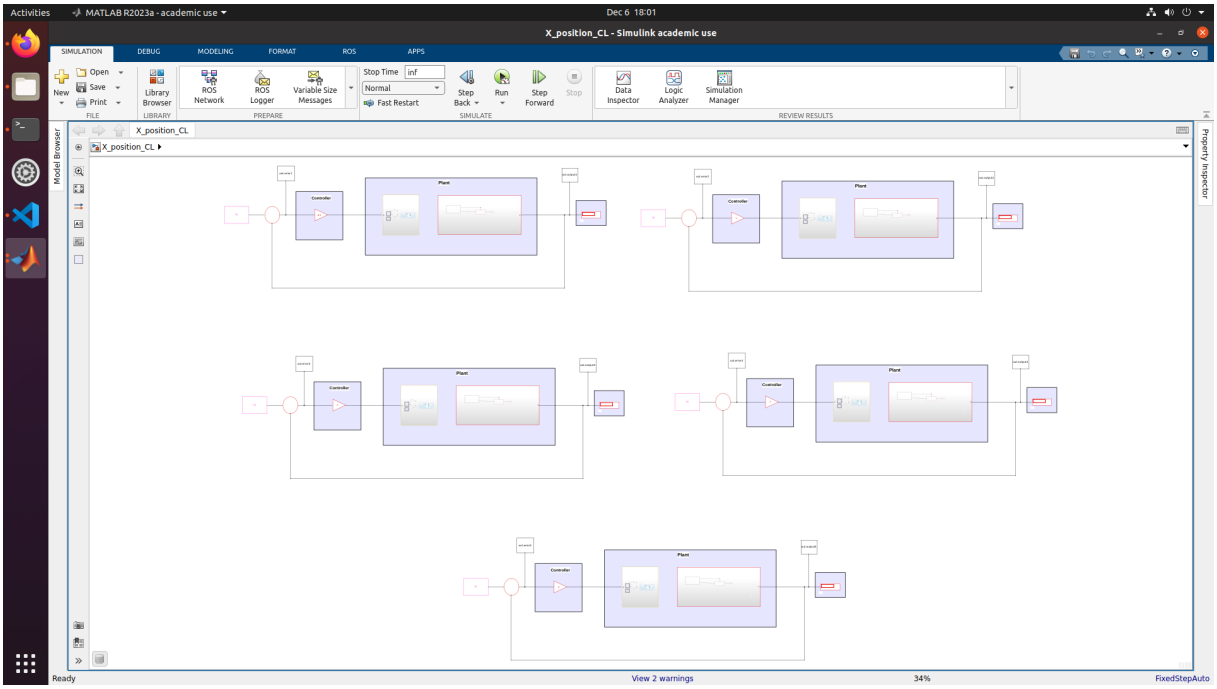
```
nano nexus_4wd_mecanum.gazebo
```

The user will need to change those values (they are in $\left[\frac{m}{s}\right]$)



One of the **tests** we did in the lab with this **five robots experiment**, was to **compare** the difference between various *P* gain. It allows to **visually understand** what we learnt in control theory with the simulation.

On the lab computer, open the Simulink diagram **X_position_CL.slx** within the Robot cars/Simulation/1D X position CL 5 robots directory.



For each robot, we made the same X position closed loop with different P gains. When the **simulation** is launched (as previously explained), we can plot the different reactions of each robot.

The **reference input** was a **step input block** of 10 $[m]$ forward.

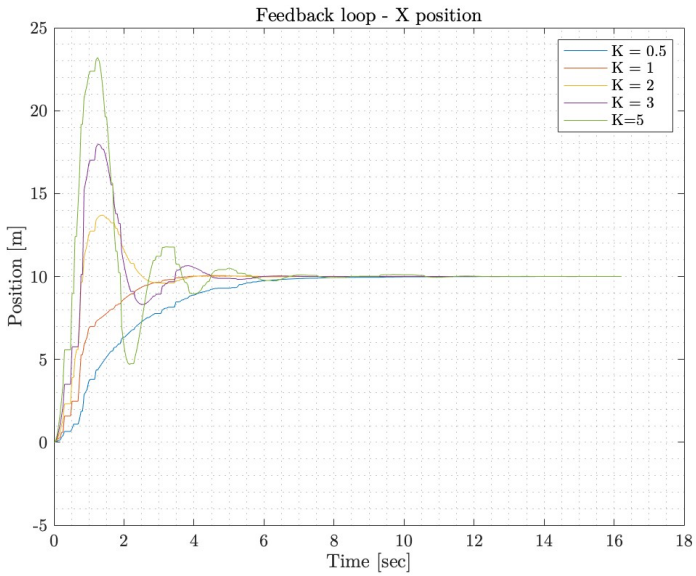


Figure 24: Step response for various P gain

This perfectly represents the fact that **too much *P* gain** causes a **bigger overshoot** reaction (as learnt).

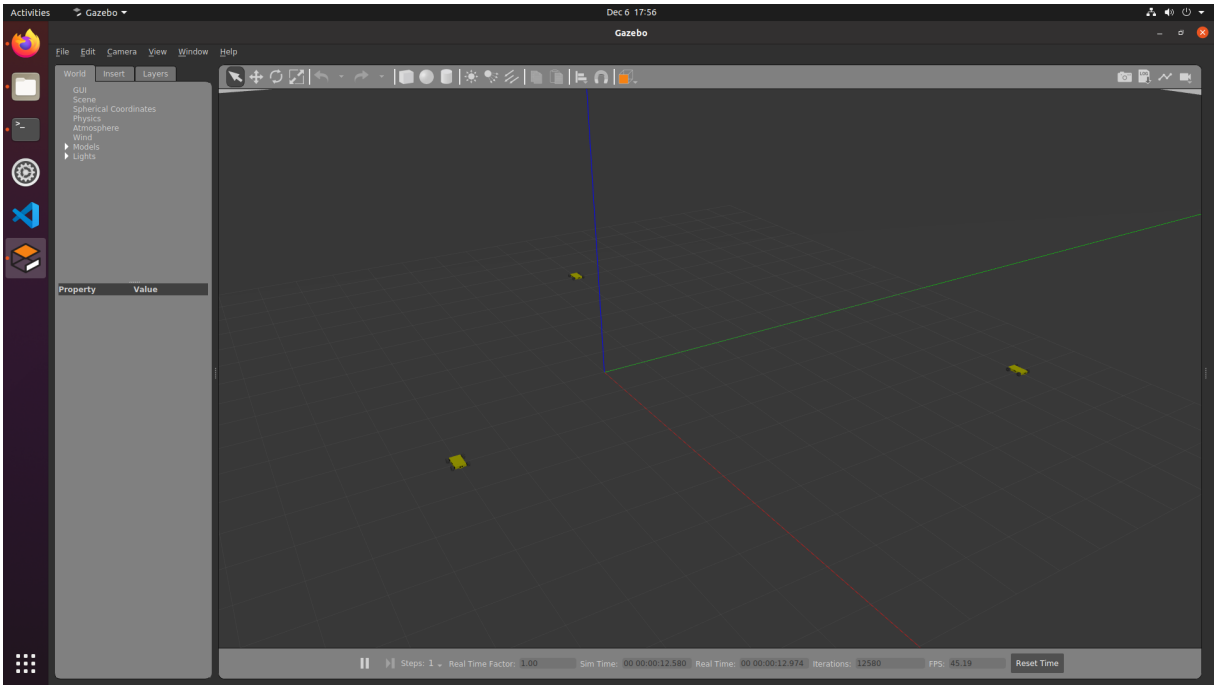
The possibilities are **unlimited.** We can test PID control loop, cascade loop, high level closed loop, etc…

Multi agent system

Another **simulated experiment** we worked on during this year was a multi agent system problem.

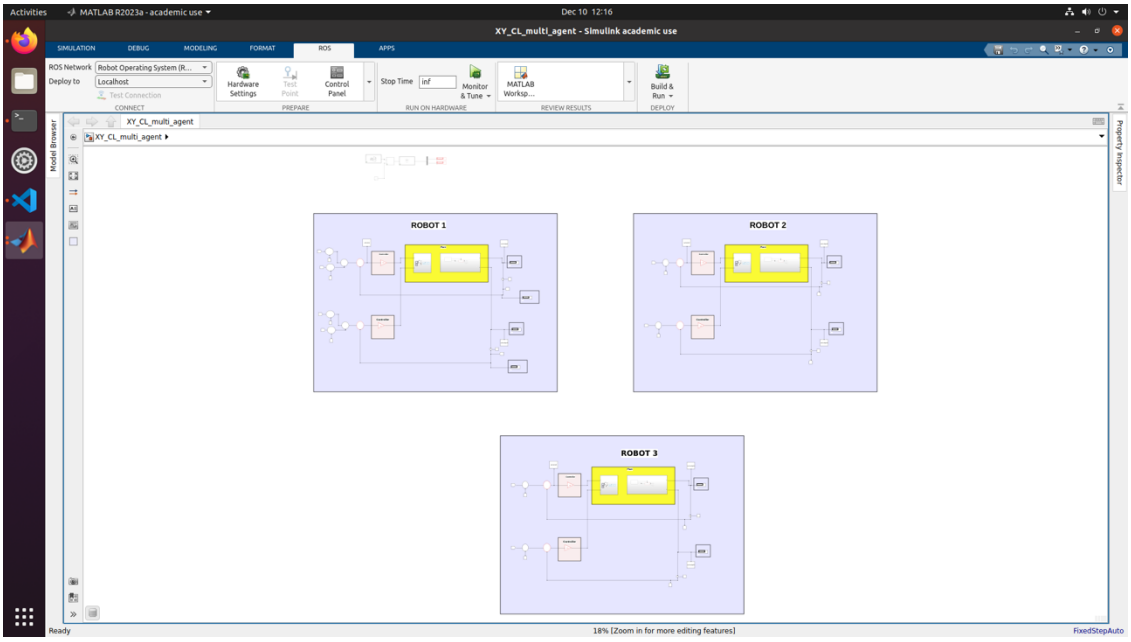
First, launch the simulation

```
roslaunch nexus_4wd_mecanum_gazebo multi_agent_test.launch
```



As previous, we can define the **initial positions** of each robot.

Then, open the Simulink diagram **XY_CL_multi_agent.slx** within the Robot cars/Simulation/Mutli agent directory.



The Simulink diagram is a little bit messy and need to be arrange.
After the simulation is launched from Simulink, the three robots are supposed to converge one to another.

Future Tasks

For **each Simulink diagram** file, make a matlab script, allowing to plot:

- the states of the robot during the simulation
- the velocity input
- the error
- etc …

Add **sensors** to the **robot description** file of the robot (lidar, laser, camera), to allow **feedback** for the different control loops.

See theses links:
https://articulatedrobotics.xyz/mobile-robot-8-lidar/
https://articulatedrobotics.xyz/mobile-robot-9-camera/

Add various worlds (walls, house, moving blocks …)

Organize the Simulink diagram (MIMO way)[15]
Instead of having a single control loop for each mission (X position, Y position, Heading), make a vector input with a singular transfer function that handle the three tasks.

Experiment:

Spawn two robots, one with a camera, another with a specific color or a QR code. The **goal** is that the robot equipped with the camera adjust his states, track the other robot, and approach/follow it.

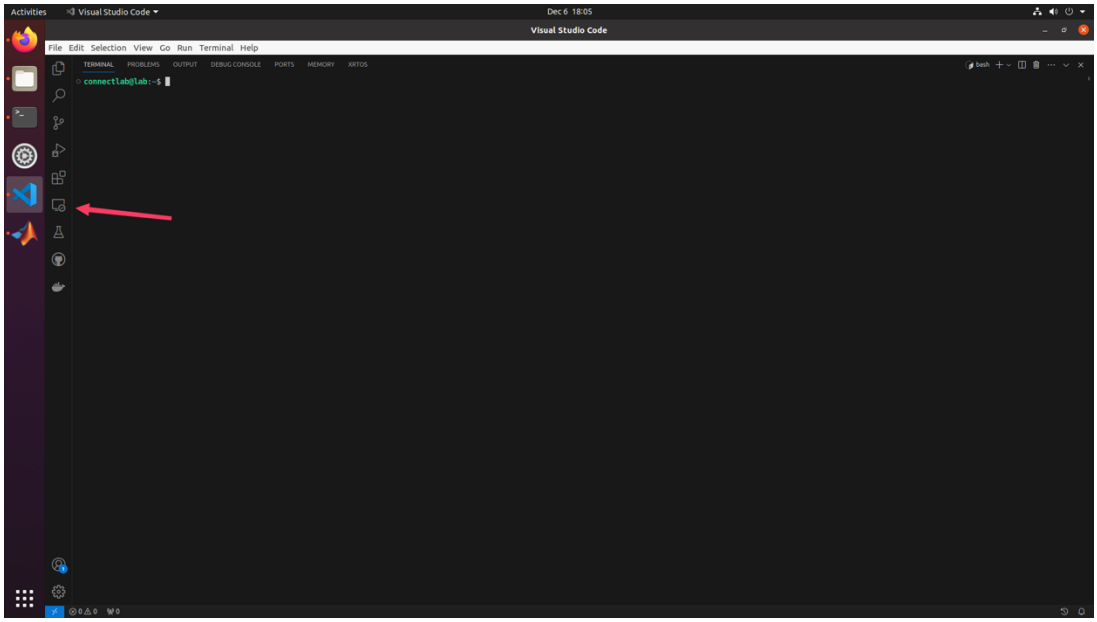
[15] Ask Professor Zelazo if the user didn't take the MIMO course.

Appendix: General

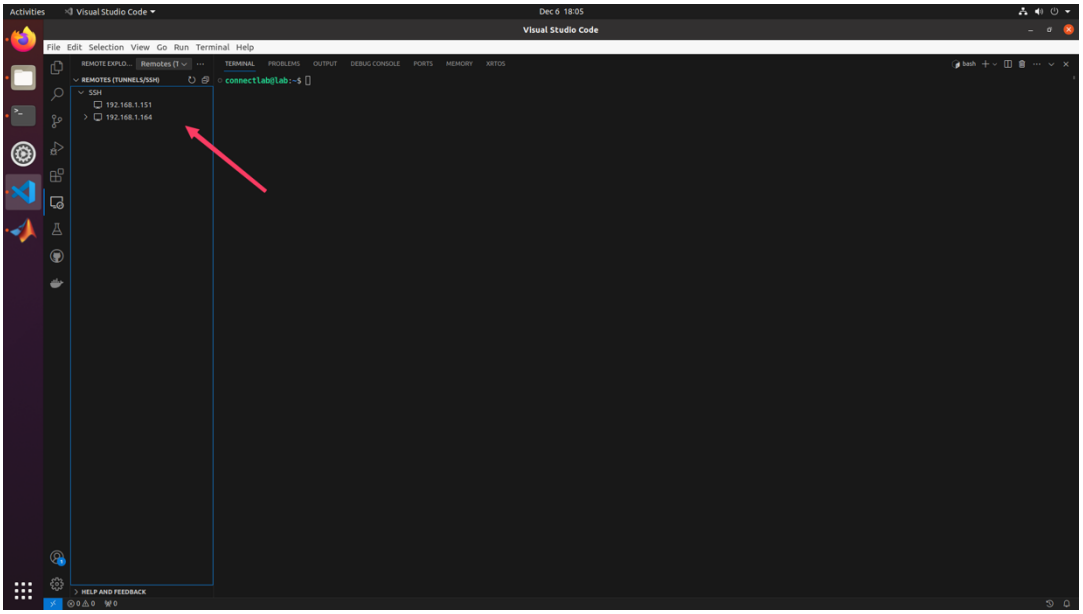
Visual Studio Code

The advantage of using **Visual Studio Code** (VSC) on the lab computer over the classic terminal, is that we can connect **automatically** to the robots via SSH without entering the command line at each terminal shell or the password. Plus, we can store the different IP addresses for easy access.

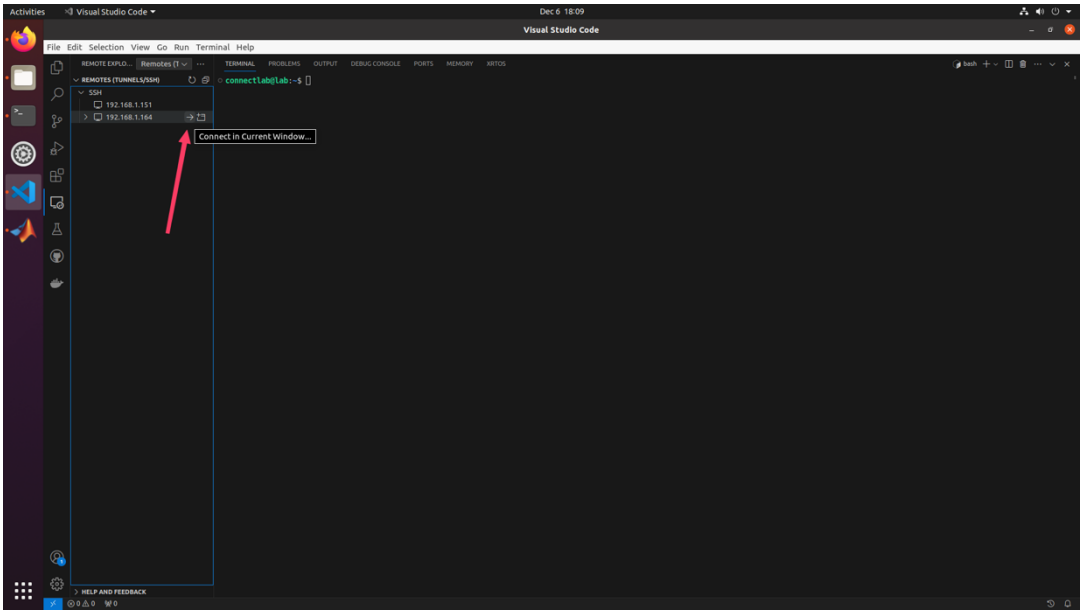
When opening VSC, go to the **Remote Explorer**:



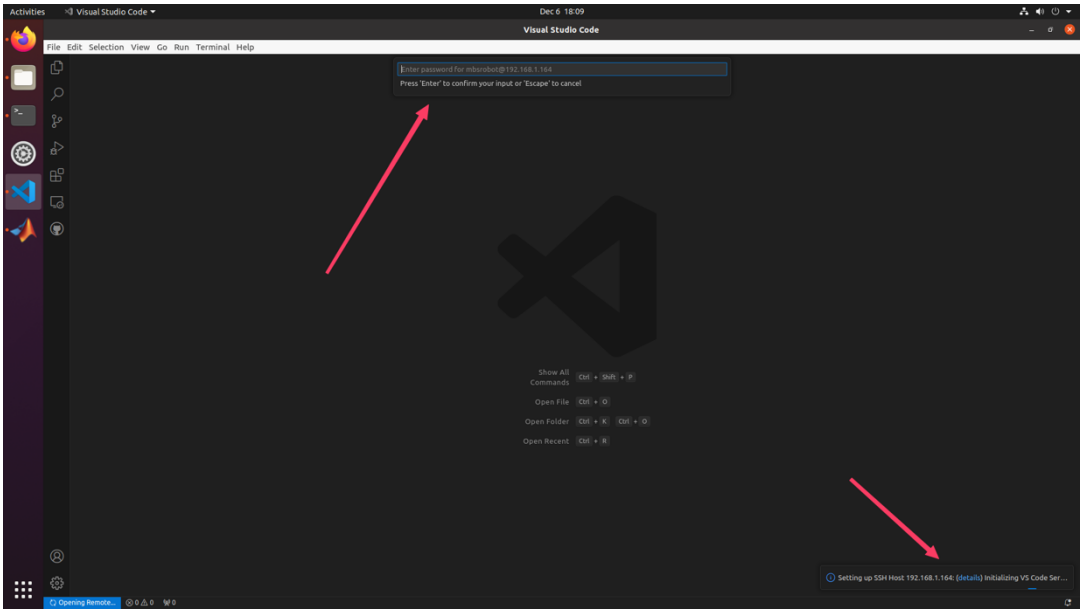
Then, the already entered **IP addresses** are shown:



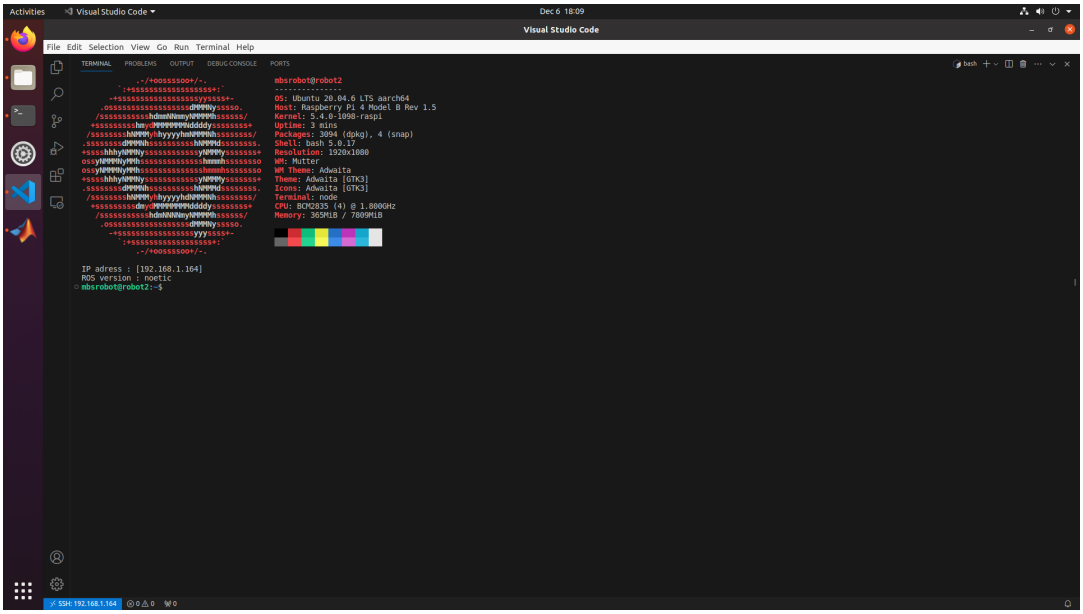
When highlighting one of the IP addresses, an arrow appears, and we can **directly connect** via SSH to this specific IP address.



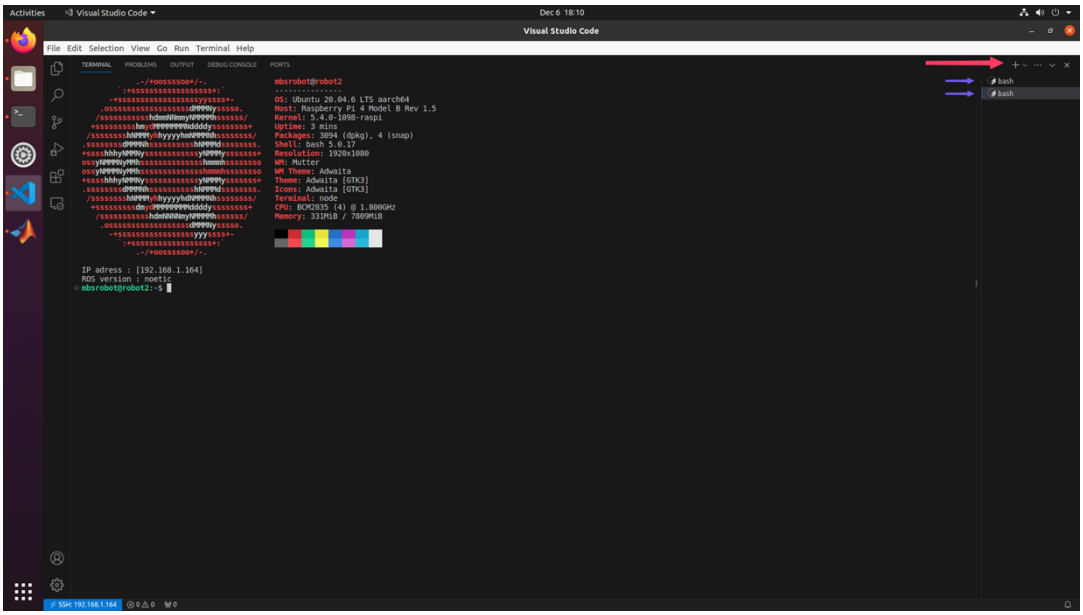
We are prompted to **enter the password** of the robot **twice**:



If the connection is successful, we are now **connected** to the robot via **SSH** !

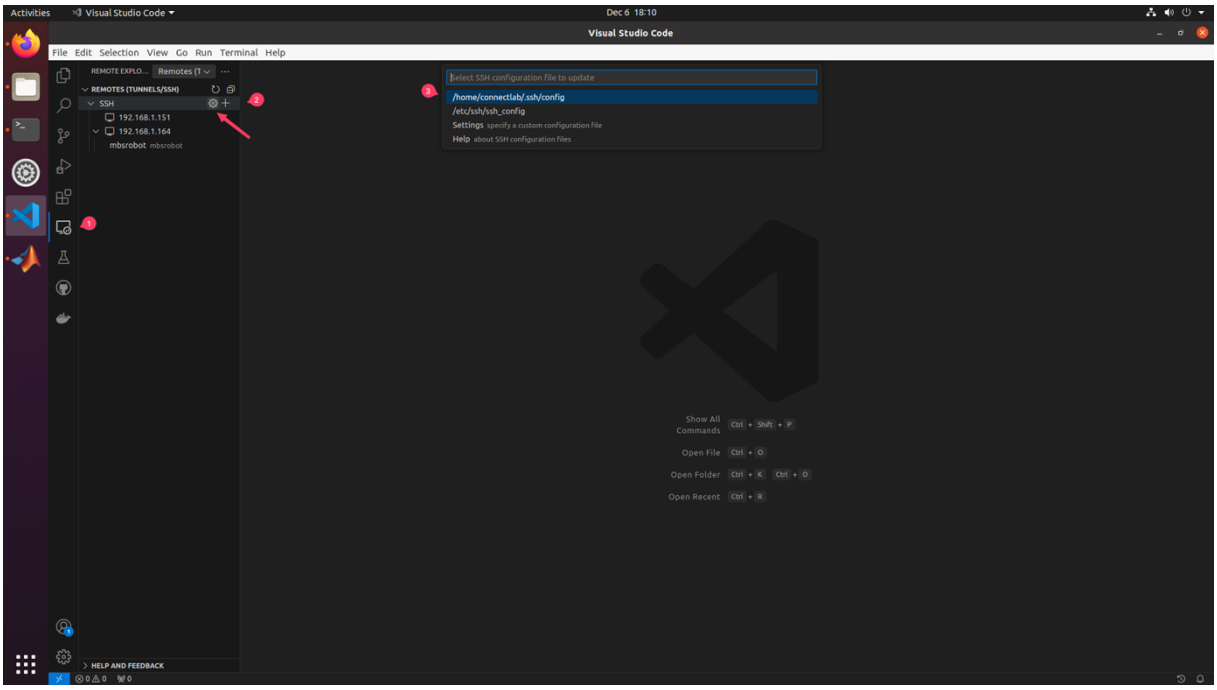


We can now **automatically** open a new shell in SSH **(without entering the password again)**

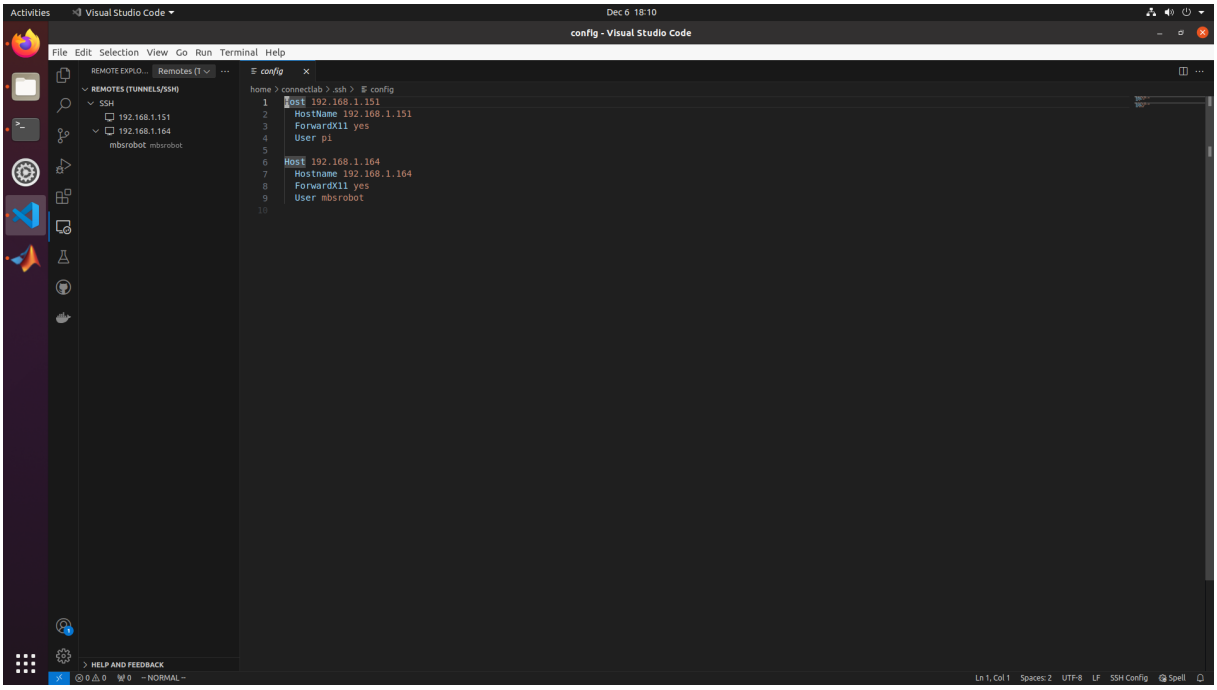


Or even split the screen for comfort.

If we want to **add another IP address** for SSH connection, go to **Remote explorer**, highlight the **"SSH" tab**, and click on the **gear**. Then enter the **/home/connectlab/.ssh/config** file:



Now the user can **add a new hostname** (IP address) and **user** of the robot:



Raspberry Pi imager

When building a **new robot**, we need to install the **.img file** on the SD card that goes into the **Raspberry pi**. The **Moebius** robot and **Yahboom** robot have their own image containing all the necessary **packages** for functioning.

The Raspberry Pi Imager should be already installed on the lab computer. If not, please refer to: https://www.raspberrypi.com/software/

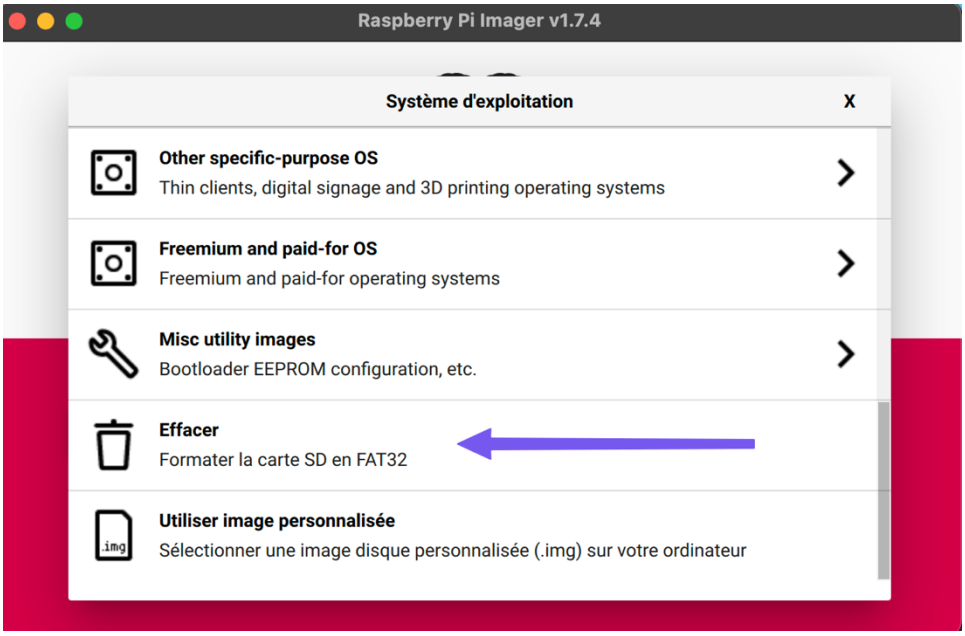
A 64GB SD card is necessary.

Once installed, it should look like this:

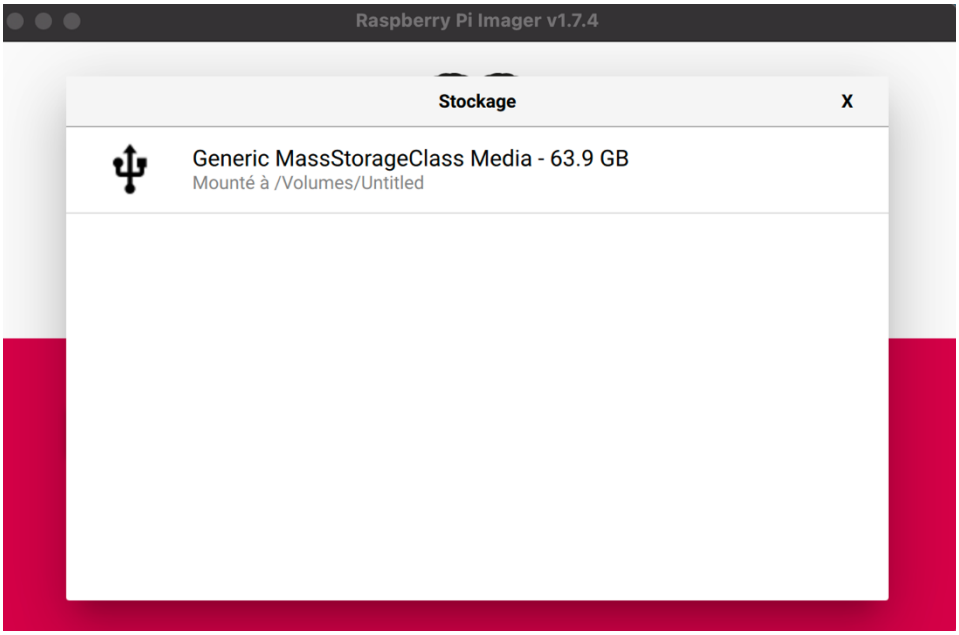
(My version is in French, but it looks the same in English)

Now, we need to **format** the SD card as **FAT32** for installing the new image.

In the **Choose OS** section, scroll down to **ERASE**:

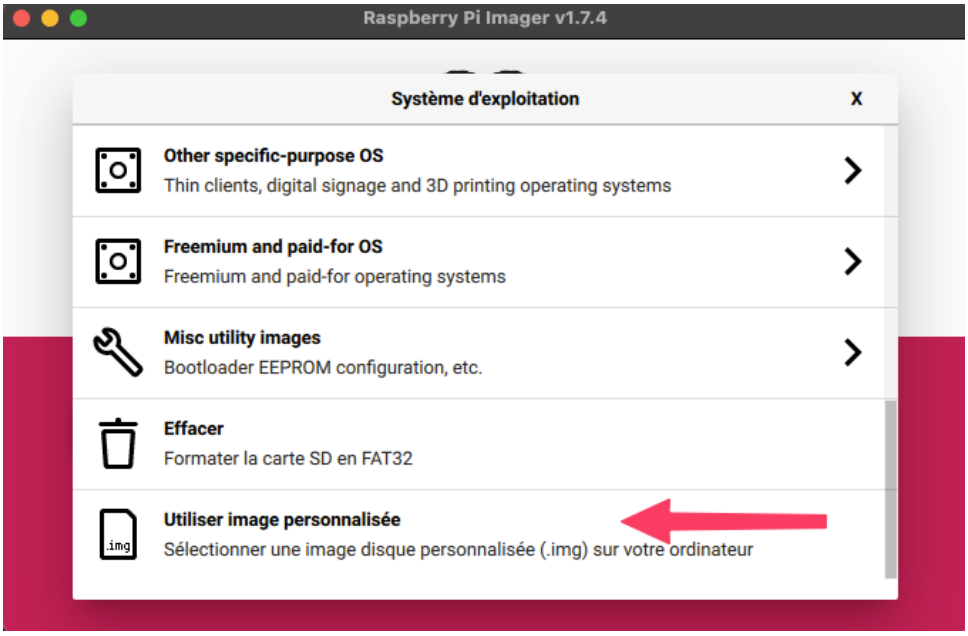


In the **Choose Storage** section, **select** the inserted SD card:

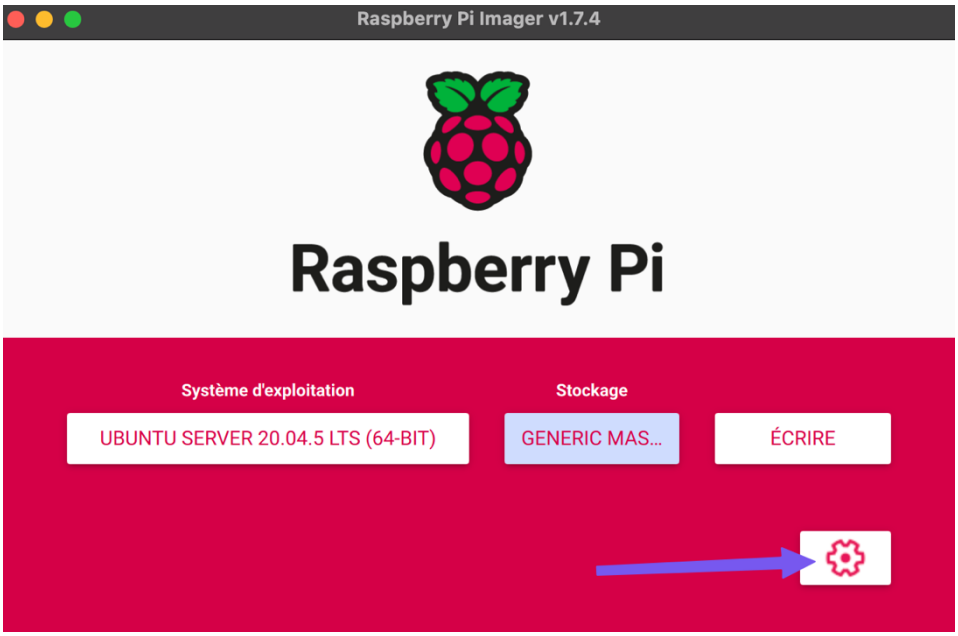


Then, click **NEXT**.

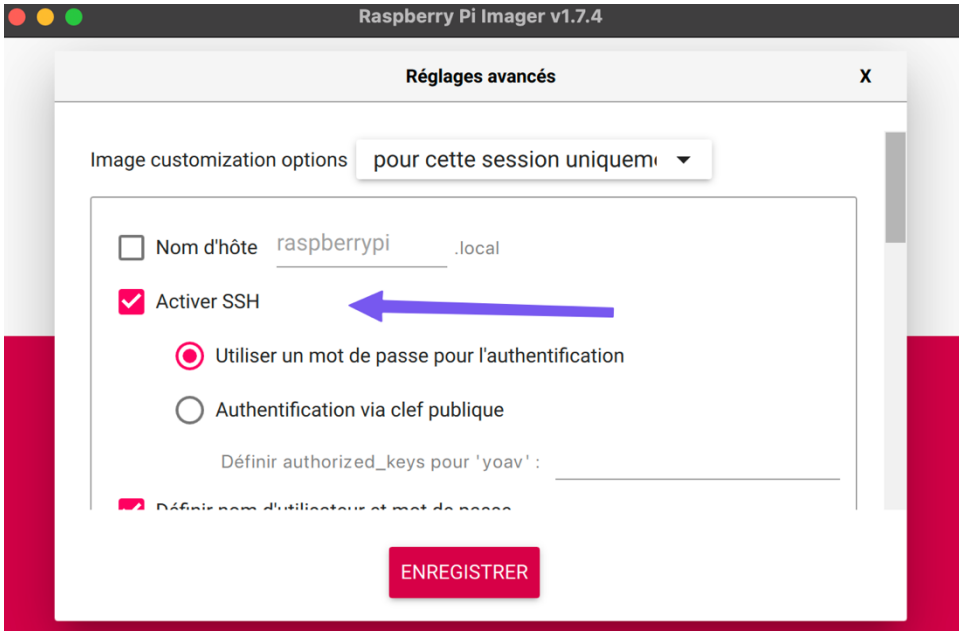
Once the SD card have been **formatted**, we go back to **choose OS**, scroll down to the end, and click on **Custom image.**



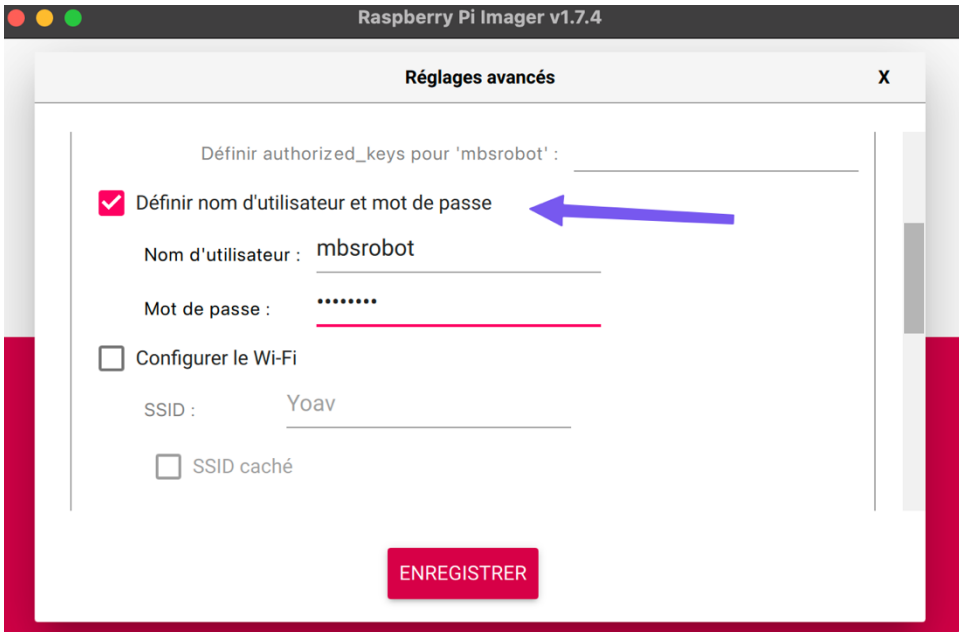
Then, we select the **desired image** (according to the robot).
Before writing on the SD card, we go back to the **main screen** and click on the **settings** button.



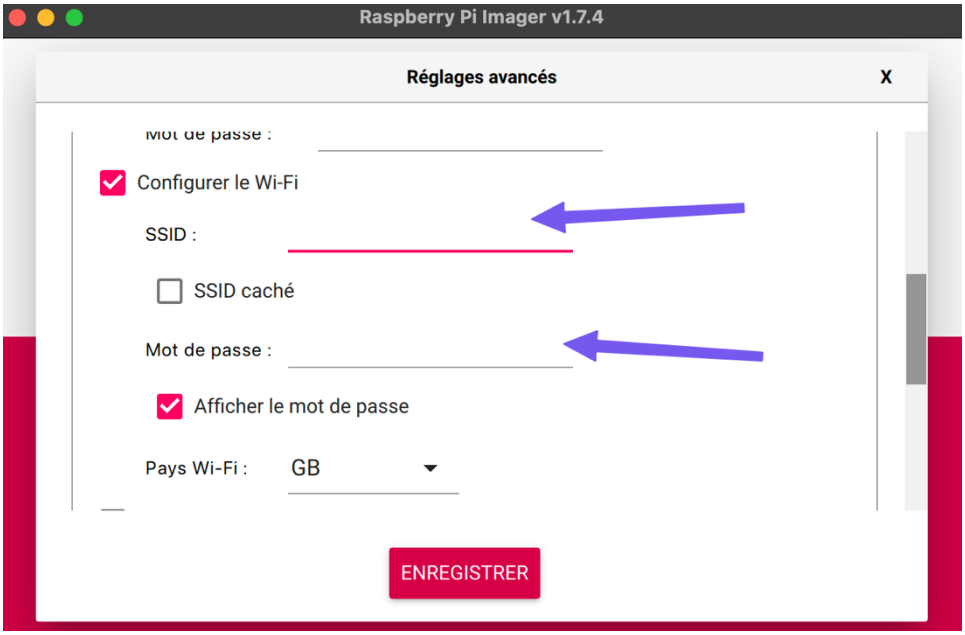
From there, we **enable SSH**,



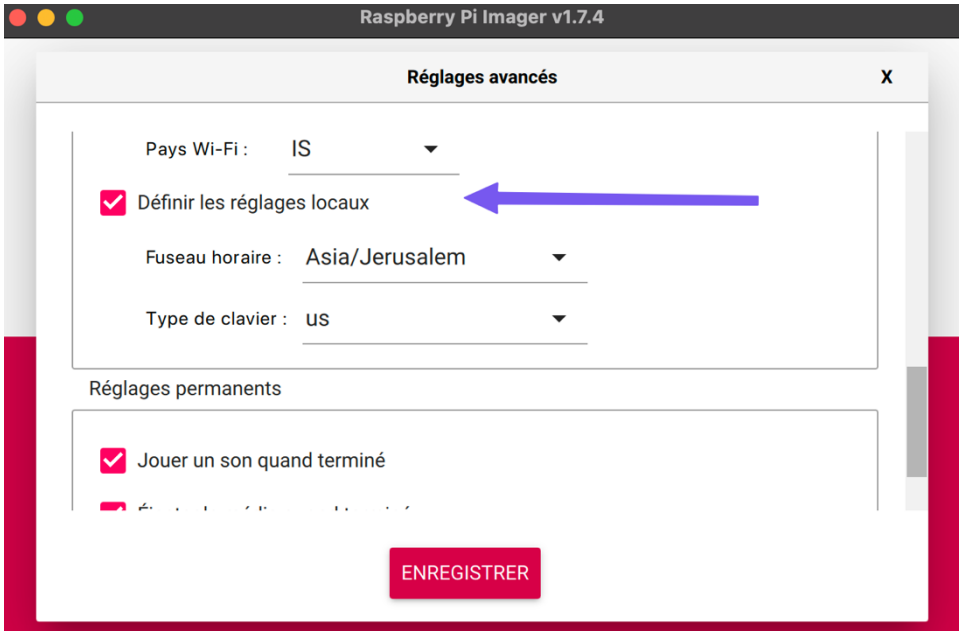
Choose a **username** and a **password**,



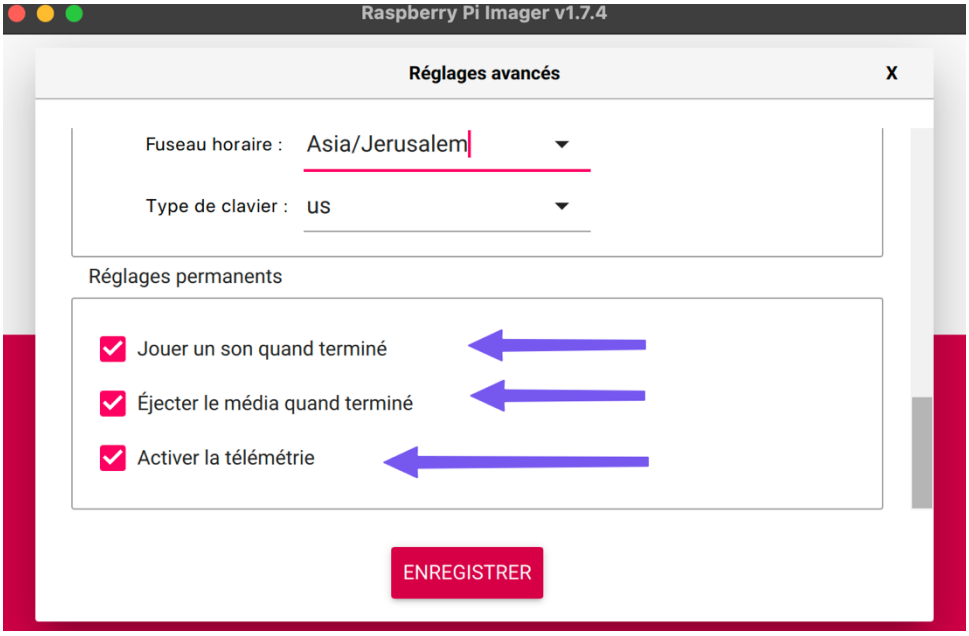
Configure the **Wi-Fi (SSID)**,



Set the correct **time zone**,



And finally, we **check** the **last three** parameters:



We are now **ready** to write!

Troubleshooting

Sometimes **when powering up the new Raspberry pi** for the first time, the **Wi-Fi connection** may not work and the **Wi-Fi icon** may be locked. We need to **manually** connect to the **Wi-Fi**:

In a terminal shell:

```
sudo nano /etc/netplan/50-cloud-init.yaml
```

The **password** has been **encrypted** for safety reason. **Change the already written password with the real one.**

(To exit the editing window, press CTRL+X, then ENTER)

In the same terminal, type:

```
sudo netplan -debug generate
sudo reboot
```

For more **details** or **troubleshooting**, check the video:
https://www.youtube.com/watch?v=s4ZDlV3tIuM

Basic ROS commands

General

`roscore`	Start the ROS master
`roscd`	Change directory to the ROS packages directory
`rospack list`	List all the ROS packages of the ROS system
`rospack list | grep <package_name>`	Filter and find the ROS packages with the name <package_name>
`printenv | grep ROS`	Print the ROS environment variables

Nodes

`rosnode list`	List all the running nodes
`rosnode ping <name_of_the_node>`	Check if a running node is responding
`rosnode info <name_of_the_node>`	Gives information about a specific node
`rosnode cleanup <name_of_the_node>`	Disactivate a running node

Topics

`rostopic list`	List all the running topics
`rostopic info <topic_name>`	Gives information about a running topic
`rostopic echo <topic_name>`	Read the information being published on a topic

`rostopic echo <topic_name> -n1`	Read the **last** information being published on a topic

Parameters

`rosparam list`	List all the running services
`rosparam get <parameter_name>`	Gives the values of a parameter
`rosparam set <parameter_name> <value>`	Set a new value for a parameter

Scripts

`rosrun <package_name> <python_file_name>.py`	Run python script
`rosrun <package_name> <node_name>`	Run a node from a given package

Messages

`rosmsg show <message_name>`	Show which type of message is used

RQT tools

`rqt_graph`	This tool allows to graphically visualizes which nodes talk to each other and on which topic they publish/subscribe
`rqt_plot`	This tool allows to live plot values from specific topic

Lenovo's laptop password

Username	TurtleBot-3
Password	ConecT-4

A list of all the components in the lab was made during the year. It can be found on the computer in the Lab or in the Github of the PFCL.