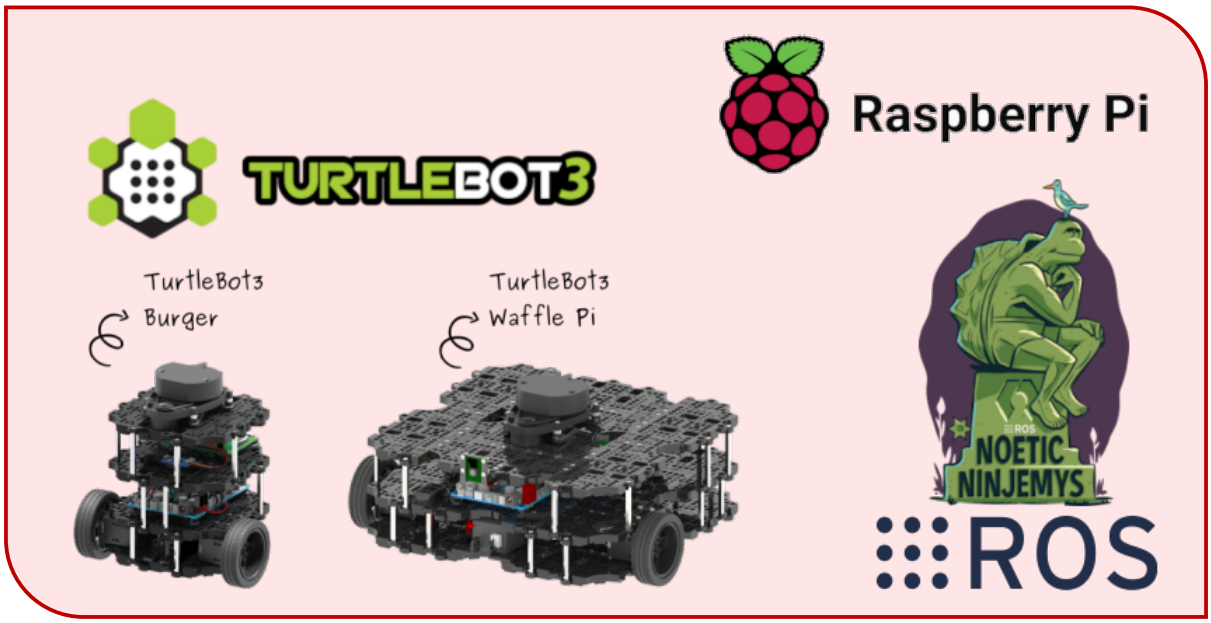
How to Control a Real TurtleBot with ROS
through a Remote Raspberry Pi as ROS Master and with an OptiTrack Motion Capture System

Sylvain Durand (sylvain.durand@insa-strasbourg.fr)

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES STRASBOURG

iCUBE

— Version 23.04 —



This document is still a work in progress...
Please feel free to send me any feedback that could help to improve it.

Acknowledgment: Many thanks to Bertrand Boyer for his tremendous help and contribution in learning, handling and implementing ROS and its tools.

Contents

Part I

Introduction to Raspberry Pi and ROS

Chapter 1

Basics of Raspberry Pi (RPi)

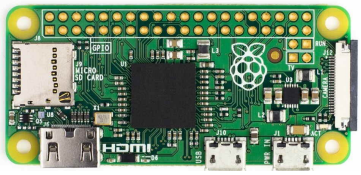
1.1 Presentation

1.1.1 The most popular Single-Board Computer for Embedded Systems

Raspberry Pi[1] (or RPi), developed by Raspberry Pi Foundation[2] in association with Broadcom, is a series of small single-board computers: it can provide all the expected features or abilities of a mini-computer, at low-cost and low power consumption. It has ARM-based Broadcom Processor SoC (system on chip) along with an on-chip GPU (graphics processing unit). The CPU (central processing unit) speed of Raspberry Pi varies from 700 MHz to 1.5 GHz. Also, it has onboard SDRAM that ranges from 256 MB to 8 GB as well as a micro SD card slot. Only the monitor, keyboard and mouse are missing to be connected to the dedicated interfaces (HDMI and USB). See Fig. 1.1 for examples of Raspberry Pi hardware.

RPi is more than a computer as it provides access to on-chip hardware GPIO (general-purpose input/output) for developing applications. It also provides on-chip communication modules: SPI, I2C, I2S, and UART.

(a) Raspberry Pi 3, model B+ (left) vs. A+ (right).



(b) Raspberry Pi Zero.

Figure 1.1: Raspberry Pi hardware.

Raspberry Pi is probably the most popular electronics prototyping platform for embedded systems. It is used for web servers, real-time image/video processing, IoT-based applications, sensor applications (like weather stations), robotics applications, automated control systems, or gaming devices...

1.1.2 Generations and Models

The current generations of regular Raspberry Pi boards are Zero (Fig. 1.1b), 1, 2, 3 (Fig. 1.1a), and 4 (Fig. 1.2). Generations 1, 2 and 3 have the following four models: A, A+, B and B+, where the B models are the original credit-card-sized format while the A models have a smaller and more compact footprint with reduced connectivity options (see Fig. 1.1a which compares both). The latest generation 4 is the most powerful. Zero models are the most compact of all the RPi boards.

See [1]/product for further details on the different models.

[1] Raspberry Pi support: https://www.raspberrypi.com/
[2] Raspberry Pi foundation: https://www.raspberrypi.org/

1.1.3 Hardware Components

The main hardware components are highlighted in Fig. 1.2 (for an RPi 4B) and listed below:

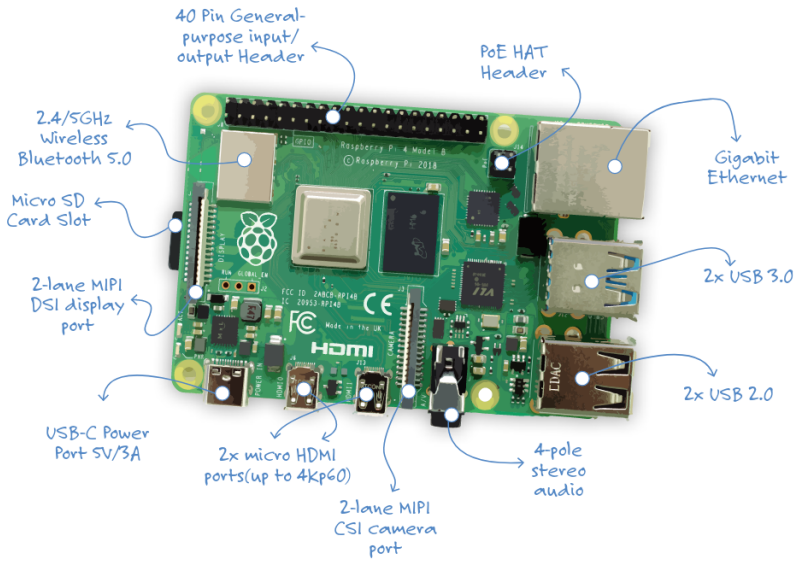


Figure 1.2: Raspberry Pi 4B hardware and its main components.

Power supply: RPi cards are powered by micro USB or USB-C for the latest version RPi 4.

USB ports: They allow connecting any USB-compatible device to the RPi, including keyboards, mice, digital cameras, or flash drives.

Ethernet: port Using an RJ45 cable, an RPi can be linked to a wired computer network.

PoE port: When available (from version 3B+), this 4-pin connector allows powering an RPi through the network connection (power supply via Ethernet cable).

Built-in WiFi and Bluetooth modules: When available, they allow the RPi to communicate wirelessly with other devices, like a remote computer and other nearby smart devices, sensors, or cellphones.

HDMI (High-Definition Multimedia Interface): It is used for transmitting uncompressed video or digital audio data to a monitor.

CSI (Camera Serial Interface): CSI provides a connection between the Broadcom processor and the Pi camera.

DSI (Display Serial Interface): DSI is used for connecting LCD to the Raspberry Pi using a 15-pin ribbon cable. DSI provides a fast high-resolution display interface specifically used for sending video data directly from GPU to the LCD display.

Composite video/audio output: A 3.5mm audio-visual jack allows carrying video along with audio signals to a composite audio/video system.

GPIO header: In two rows of 20 pins each, 40 metal GPIO pins are available along the RPi board (see section 1.1.4) to communicate with peripherals such as LEDs and buttons to sensors or joysticks.

SD card slot: The RPi has a microSD connector, placed on the other side of the circuit board. The SD card will contain the operating system that makes the Raspberry Pi work (see section 1.1.5) as well as all the user files.

1.1.4 GPIO Pinout

GPIO (general-purpose input/output) pins are depicted in Fig. 1.3. They can be used as input or output and allows RPi to connect with I/O devices. Some of the GPIO pins are multiplexed with alternate functions like I2C, SPI, UART, etc.

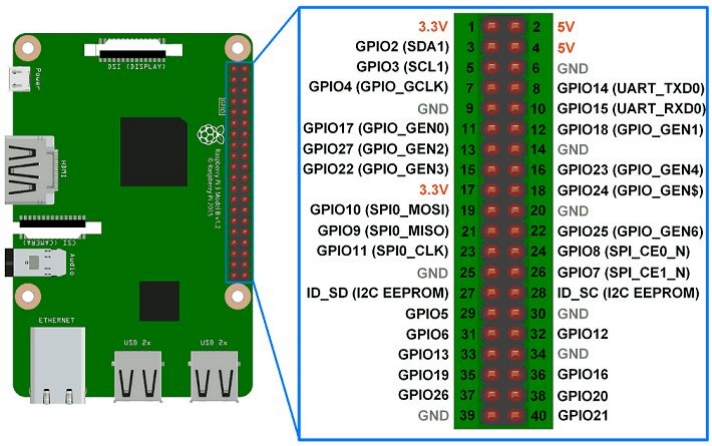


Figure 1.3: GPIO pin mapping of the Raspberry Pi 3B.

It is worth mentioning that Raspberry Pi has two ways of defining pin numbers: *GPIO numbering* (where a pin number refers to the number on Broadcom SoC, eg. GPIO12) and *physical numbering* (pin number refers to the pin of the 40-pin header on the RPi board, eg. pin 32).

See section "*Raspberry Pi hardware*" in [1]/documentation/computers/raspberry-pi.html for further details on GPIO pinout.

1.1.5 Raspberry Pi OS

Raspberry Pi OS (see [1]/software) is a free operating system based on Debian[3] (a Linux distribution composed exclusively of free and open-source software), optimized for the Raspberry Pi hardware. There are several different operating systems available for use on the Raspberry Pi, but the Raspberry Pi OS – formerly *Raspbian* – is the recommended one for normal use on an RPi. It is installed on the SD card.

Linux is great for critical applications because it focuses on security and stability instead of mainstream operating systems that focus on ease of use. Linux is also lightweight and less resource intensive than commercial operating systems. See section 1.2 for an overview of some useful Linux commands. The OS supports a wide range of programming languages, like Python, C/C++, Java, Scratch, etc.

The Raspberry Pi OS has two different versions – a version with a desktop and command line interface, or a lightweight version with just a command line interface (called *Lite* version). The present document is built on the latter, i.e. **Raspberry Pi OS Lite**.

1.1.6 Raspberry Pi Community

Besides the official RPi website[1] to find information, you should also find many tutorials, tips and advice on the many discussion forums or video channels. The Linux and Raspberry Pi community is huge. For example, see the video from Elektor[4], to name only one, or some selected RPi overview websites[5].

[3]Debian operating system: https://www.debian.org/

[4]Video from Elektor: Raspberry Pi – Overview and Getting Started

[5]Selection of RPi tutorials: Simply easy learning of Raspberry Pi (Tutorials Point), Introduction to Raspberry Pi (The Engineering Projects), Raspberry Pi basics (Electronic Wings)

1.1.7 How about other Single-Board Computers (not covered here)

The Raspberry Pi is not the only single-board computer series. To name just one other, the *NVIDIA Jetson Nano*[6] is a serious alternative, particularly for Artificial Intelligence applications. In addition to the standard hardware components that feature (see section 1.1.3), the Jetson Nano has a quad-core ARM processor (as in the RPi 4B) but also a 128-core Maxwell GPU that allows multiple neural networks to run in parallel for applications such as image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that consumes only 5 watts of power. The Jetson Nano platform is depicted in Fig. 1.4 and compared to an RPi 4B (in size).

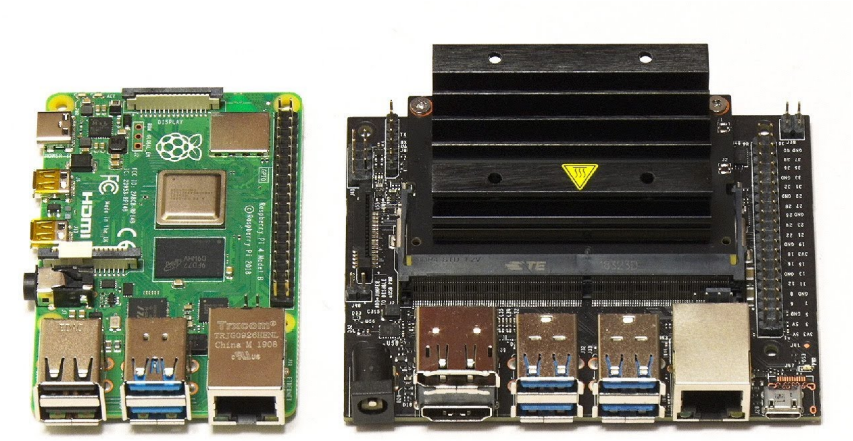


Figure 1.4: NVIDIA Jetson Nano vs. Raspberry Pi 4B.

It is worth mentioning that Jetson is a series of which the Nano is the most compact version, designed to compete with the Raspberry Pi, but the NVIDIA brand has even more powerful embedded boards.

[6]NVIDIA support: Jetson Nano Developer Kit

1.2 Useful Unix Commands

General tricks

- `TABS` for *auto-completion*: after entering the first letters of a command or a file/directory path, click the `TABS` key for an auto-completion (no need to type complete file names or even complete commands). Double `TABS` will display all the possibilities.
- *Repeat previous commands*: use the `UP ARROW` to view the previous commands and press `Enter` to execute it.
- *Search the history*: press `Ctrl+R` and type a keyword from a previous command in the history to re-execute it (without having to type it fully or to use the `UP ARROW` too much).
- The *pipe* symbol `|` allows the use of several commands such that the output of one command serves as input to the next one, like a pipeline.

Terminal

- `Ctrl+Alt+T` is the shortcut to open a new terminal in Linux.
- `$` indicates a line in the terminal (the character `$` is not to enter in the terminal).
- `Ctrl+C` stops any command in the terminal safely.
- `Ctrl+Z` forces a stop (when `Ctrl+C` does not work).
- `&` when entered at the end of a command, execute the command as a background activity (to use the terminal for another command to enter without needing to stop it).
- `$ clear` clears the terminal (when it gets filled up with too many commands).
- `$ exit` exits a terminal (or an SSH session).

Manual page

- `$ man COMMAND` opens the manual/help page for the command that follows.
  eg. `$ man ls`

The *sudo* command

- `$ sudo COMMAND` gives administrator privileges for the command that follows (*super user*).

Files management

- `$ cd PATH` changes directory, go to the specified folder in the files tree.
  eg. `$ cd ~/pi` (`~` is the `home` directory path) or equivalently `$ cd /home/pi`,
  `$ cd /` (change to the root directory),
  `$ cd ./` (change to the current directory),
  `$ cd ../` (change to the parent directory)
- `$ ls` lists the files and directories in the current or specified folder.
- `$ pwd` checks the current directory.
- `$ nano FILE` is a text editor.
  eg. `$ nano ~/pi/test.py` (save changes with `Ctrl+S`, `Enter`, `Ctrl+X`, `Enter`).
- `$ cat FILE` displays all the content of the specified file in the terminal.
  eg. `$ cat ~/pi/test.py`
- `$ mkdir FOLDER` creates a new sub-folder in the current or specified folder.
  eg. `$ mkdir myfolder` or `$ mkdir ~/pi/myfolder`
- `$ cp FILE DESTINATION` copies a file or a directory to another location (to copy a complete directory, add the `-r` parameter for "recursive").
  eg. `$ cp test.py ~/pi/myfolder/` or `$ cp ~/pi/test.py ~/pi/myfolder/`
  `$ cp -r test/ ~/pi/myfolder/`
- `$ mv SOURCE DESTINATION` moves a file or a directory to another location.
  eg. `$ mv test.py ~/pi/myfolder/` or `$ mv test/ ~/pi/myfolder/`
- `$ rm FILE` deletes a file or a folder (add option `-rf` for recursive and force).
  eg. `$ rm test.py` or `$ rm -rf ~/pi/myfolder/`

Rmk: Be careful when using `sudo` with the `rm` command, system folders can be deleted without any warning message, which can break the system.
- `$ chmod` modifies the read (`r`), write (`w`), and execute (`x`) permissions of a file or directory (execute `$ man chmod` for further details).
- `$ grep STRING` is a powerful tool to search string in a text, in a file, or to filter the output of another command or script (execute `$ man grep` for further details).

Network commands

Configuration:

- `$ ifconfig` displays the current network configuration, mainly the IP/MAC addresses if connected (see section 1.3.1).
- `$ iwconfig` checks which network the wireless adapter is using.
- `$ hostname -I` finds the IP address of the system.
- `$ ping IP` sends a ping packet to another IP on the network to check if the host is alive.
  eg. `$ ping 192.168.1.1`
- `$ ifup INTERFACE` enables the specified interface.
  eg. `$ sudo ifup eth0`
- `$ ifdown INTERFACE` disables the specified interface.
  eg. `$ sudo ifdown wlan0`

File transfer and remote connection:

- `$ wget URL` allows downloading a file from the Internet.
- `$ ssh USER@IP` SSH is a network protocol that provides a way to connect securely to a remote computer (see section 1.3.1).
  eg. `$ ssh pi@192.168.1.201`
- `$ scp FILE USER@IP:PATH` to transfer a file to a remote computer over SSH (see section 1.3.1).
  eg. `$ scp test.py pi@192.168.1.201:~/pi/myfolder/`

System updates and packages management

System updates and packages management:

- `$ apt-get update` downloads the last repository version for each one in the computer configuration (in `/etc/apt/sources.list`).
  eg. `$ sudo apt-get update`
- `$ apt-get upgrade` updates all installed packages if needed.
  eg. `$ sudo apt-get upgrade`
- `$ apt-get install PACKAGE` installs the specified package(s).
  eg. `$ sudo apt-get install phpmyadmin`
- `$ apt-get remove PACKAGE` removes a previously selected package.
  eg. `$ sudo apt-get remove phpmyadmin`

Rmk: you can use either `apt-get` or `apt` (only).

System management

- `$ reboot` or equivalently `$ shutdown -h now` restarts the system immediately.
  eg. `$ sudo reboot`, `$ sudo shutdown -h now`
- `$ halt` shutdowns the system.
  eg. `$ sudo halt`
- `$ ps` displays all running processes on the computer.
  eg. `$ ps aux` (to display everything) or `$ ps -u pi` (to display process started by a specific user)
- `$ who` allows knowing the running users.
- `$ htop` is a tool to monitor the system resources.
- `$ kill PID` allows terminating a process in which ID is `PID` (see the previous command to know the ID of running processes).

eg. `$ kill 12345`, `$ kill -9 12345` (to force all related commands to stop), `$ killall php` (to stop all `php` scripts).
- `$ df` displays the partition list, with the disk space used and available for each one.
  eg. `$ df -h` (`-h` option for a more readable format).
- `$ vcgencmd measure_temp` displays the current CPU temperature.

Raspberry Pi OS commands

Configuration:

- `$ raspi-config` is a tool that allows managing any configuration from a terminal or an SSH connection.
  eg. `$ sudo raspi-config`
- `$ passwd` changes the password of the Raspberry Pi.

Use of a camera Pi (when a camera is plugged into the camera module):

- `$ libcamera-still` takes a shot and saves it as an image file.
  eg. `$ libcamera-still -o image.jpg`
- `$ libcamera-vid` captures video from the camera.
  eg. `$ libcamera-vid -o video.h264 -t 10000` (with `-t` parameter the capture time in ms).

1.3 Practical Work

This practical work is partially inspired by the Elektor's handbook "Getting Started with Raspberry Pi"[7].

1.3.1 Getting Started

1.3.1.1 Raspberry Pi setup

- Install *Raspberry Pi Imager* ([1]/software) that will be needed to install an operating system to a microSD card (eventually see the video on how to install an OS with Raspberry Pi Imager).
- Install the **Raspberry Pi OS Lite** (without desktop) version ([1]/operating-systems), in 32 or 64-bit regarding your RPi board.

  Rmk: The *Lite* version will only be accessible in command mode and not in desktop mode (no Graphical User Interface installed). To have a GUI, install the version *RPi OS with desktop*.

  The `advanced options` of RPi Imager (see Fig. 1.5) allow several settings:
  - `Enable SSH` (that will be useful after accessing the RPi);
  - `Set username and password` (choose **insa** and **insa** for the `RPi_USERNAME` and `RPi_PASSWORD` respectively);
  - `Configure wireless LAN` (select the `WIFI_SSID` and `WIFI_PASSWORD` of your WiFi hotspot);
  - `Set locale settings` (to configure a French keyboard for instance).

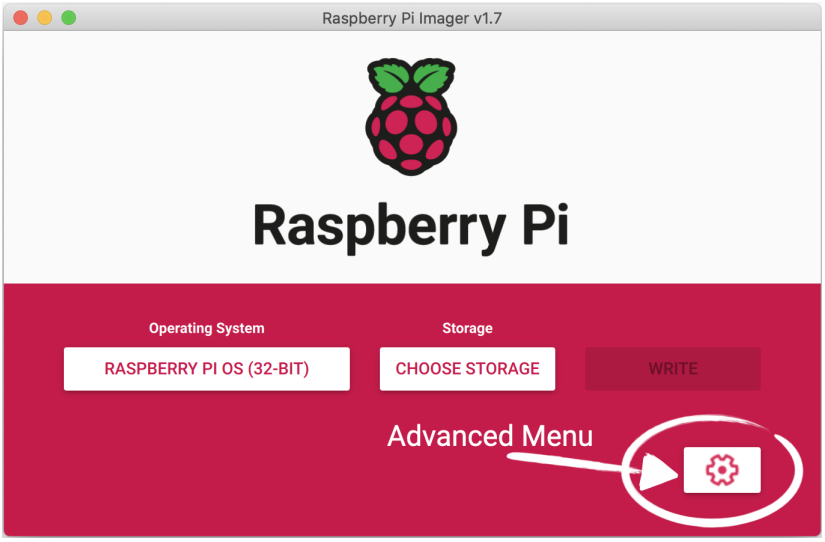


Figure 1.5: *Advanced menu* of the Raspberry Pi Imager used to install RPi OS.

- Connect the Raspberry Pi with its microSD card: keyboard, mouse, monitor, Ethernet cable, and finish with its power supply.

  Rmk: A mouse is not really needed here with the *Lite* OS version.

  The default `RPi_USERNAME` and `RPi_PASSWORD` are **pi** and **raspberry** respectively (if not configured).

→ Try to install **Raspberry Pi OS Lite** on the SD card of your Raspberry Pi.

[7] Elektor's handbook (in French): "Prise en main du Raspberry Pi (2021)"

1.3.1.2 Network configuration

- In case the network was not configured in the `advanced options` of RPi Imager (see above), you have two main options for a headless setup (without desktop):
  1. Use the `raspi-config` graphical tool (even with a Lite OS version or through SSH), using the command:

     ```
     $ sudo raspi-config
     ```

     Select `System Options` and `Wireless LAN` to configure the WiFi hotspot.

     Rmk: There are plenty of other options for the RPi settings.

  2. The `raspi-config` tool is the easiest way to configure the RPi, but it requires to have a monitor. If not, the solution is to create a file on the SD card (from a remote computer), allowing Raspberry Pi OS to read the file at the next startup and apply the configuration directly. Copy and paste these lines into your editor:

     ```
     country=FR
     ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
     update_config=1
     network={
       ssid="WIFI_SSID"
       scan_ssid=1
       psk="WIFI_PASSWORD"
       key_mgmt=WPA-PSK
     }
     ```

     Then save the file with name `wpa_supplicant.conf` and place it in the boot partition of the SD card.

     Rmk: To enable SSH (if not configured in the RPi Imager's `advanced options`), just create an empty file named `ssh` in the same boot partition.

- Open a terminal (`Ctrl+Alt+T`) and read the RPi IP addresses (this will be useful for the SSH connection), using the command:

  ```
  $ ifconfig
  ```

  – `eth0` gives the Ethernet interface settings;
  – `wlan0` gives the Wireless interface settings;
  – `lo` (for *loopback*) gives the local interface settings.

  Note the `wlan0` IP address (`inet`), hereafter denoted `RPi_IP`.

  Rmk: The `ifconfig` settings also give the MAC addresses of the RPi board.

→ Verify the IP address of your Raspberry Pi.

1.3.1.3 Update (could take a while)

- Update the list of packages:

  ```
  $ sudo apt-get update
  ```

  Rmk: In case of a synchronization error message "*Updates for this repository will not be applied*", you might need to update the system date::

  ```
  $ sudo date -s "MM/DD/YY HH:MM:SS"
  ```

- Update the packages installed on the RPi:

  ```
  $ sudo apt-get upgrade
  ```

- If you want to install one package in particular, use the following command:

  ```
  $ sudo apt-get install PACKAGE
  ```

1.3.1.4 SSH remote access

SSH (*Secure Shell*) is a secure communication protocol that allows connecting remotely from a computer to an embedded device, to get a shell or command line.

- Disconnect the keyboard, mouse, and monitor. Now you will be able to connect remotely to the Raspberry Pi by SSH, with another computer connected to the same network.
- From the remote computer, open a terminal and type:

  ```
  $ ssh RPi_USERNAME@RPi_IP
  ```

  Then you will be asked for the `password`.
- Then you can execute commands as you would on an RPi terminal.
- To end the SSH session and disconnect from the RPi, use the command:

  ```
  $ exit
  ```

  Rmk: For a remote computer on Windows, you can also use a graphical tool like *PuTTY*, but the last Windows' versions now support natively the `ssh` command.

SCP (*Secure Copy*) is an SSH command that allows files and directories (`-r`) to be exchanged between two machines (client and server), typically between a remote computer and a Raspberry Pi.

- Commands to copy files:
  - To copy a local file on the remote computer (`PATH_LOCAL/FILE`) to a directory on the RPi (`PATH_RPi`):

    ```
    $ scp PATH_LOCAL/FILE RPi_USERNAME@RPi_IP:PATH_RPi
    ```
  - To copy a file from the RPi to the remote computer:

    ```
    $ scp RPi_USERNAME@RPi_IP:PATH_RPi/FILE PATH_LOCAL
    ```
  - To copy a file from an RPi1 to another RPi2:

    ```
    $ scp RPi1_username@RPi1_IP:PATH_RPi1/FILE RPi2_username@RPi2_IP:PATH_RPi2
    ```
- Useful options:
  - `-r` is for a recursive copy, which will include all files and sub-directories (to copy a directory and not only a file);
  - `-p` will preserve the original times and attributes of the file;
  - `-u` will delete the source file after the transfer is complete.

  Rmk: For a remote computer on Windows, you can use either the *PuTTY* application (as for SSH) or *WinSCP*.

Rmk: For a *RPi OS with desktop* version (not like the *Lite* version used here), it is also possible to set a VNC (*Virtual Network Computing*) remote access to control the whole desktop of the RPi (and not only a terminal). See [1]/documentation/computers/remote-access.html for further details.

→ Try to connect to your Raspberry Pi by SSH from a local computer (on the same network), either on Windows or Linux.
→ Try to send a file from your local computer to your Raspberry Pi using SCP.

1.3.2 Grasp the Linux commands

- Useful Linux commands were presented above in section 1.2.

→ Try to handle the commands introduced in section 1.2.

1.3.3 Use of GPIO and Python Programming

Raspberry Pi boards offer I/O possibilities using their GPIO pins. One way to access these I/O is using the `RPi.GPIO` library. We will see here how to use the GPIO through an example in Python to make a LED blink in response to a push button.

- Connect a LED to GPIO4 of the RPi (through a current limiting resistor) and a push button to GPIO17, as shown in Fig. 1.6.

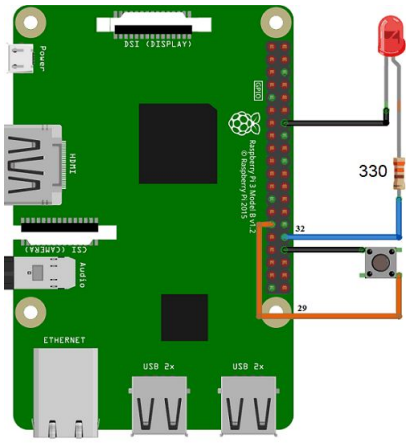


Figure 1.6: Wiring of the RPi to make a LED blink according to a push button.

- Install the package `RPi.GPIO`:

```
$ sudo apt-get install rpi.gpio
```

- Create a file `test_GPIO.py` in the `home` directory (for instance) using the `nano` editor:

```
$ nano ~/test_GPIO.py
```

- Copy and paste these lines into the editor:

```
1   import RPi.GPIO as GPIO
2   import time
3
4   PIN_LED=12    #GPIO12 or pin 32
5   PIN_BUTTON=5  #GPIO5 or pin 29
6   GPIO.setmode(GPIO.BCM)             #GPIO numbering (or GPIO.BOARD for pin numbering)
7   GPIO.setup(PIN_LED, GPIO.OUT)      #GPIO4 configured as output
8   GPIO.setup(PIN_BUTTON, GPIO.IN)    #GPIO17 configured as input
9
10  while True:
11    if (GPIO.input(PIN_BUTTON)):     #Read the push button
12      GPIO.output(PIN_LED,1)         #Turn on the LED
13      print('Switch ON')             #Print a message
14      time.sleep(1)                  #Wait 1s
15    else:
16      GPIO.output(PIN_LED,0)         #Turn off the LED
17      print('Switch OFF')            #Print a message
18      time.sleep(1)                  #Wait 1s
19
20  GPIO.cleanup()
```

- Save and exit the file: `Ctrl+s`, `Ctrl+x`
- Execute the file:

```
$ sudo python ~/test_GPIO.py
```

The LED should blink.

Rmk: The LED intensity can be changed by acting on the pin PWM:

```
GPIO.setup(PIN_LED, GPIO.OUT)      #GPIO4 configured as output
led = GPIO.PWM(PIN_LED,50)         #Set PWM frequency to 50Hz
led.start(25)                      #Turn on the LED with a 25% duty cycle
```

Rmk: The file could also be created on the local computer and sent to the RPi through an SCP command (see section 1.3.1).

→ Try to make a LED blink on your Raspberry Pi with a Python file.

1.3.4 Use of Matlab/Simulink and its Raspberry Pi Support Package

Matlab Support Package for Raspberry Pi Hardware[8] enables to communicate with an RPi remotely from a computer running Matlab (or through a web browser with Matlab Online). This feature allows the acquisition of data from sensors and imaging devices connected to the Raspberry Pi and processing them with Matlab functions. It is also possible to communicate with other hardware through the GPIO, serial, I2C, and SPI pins.

Analogously, *Simulink support package for Raspberry Pi hardware*[9] offers Simulink blocks and enables to create and run Simulink models on RPi hardware, to configure and access I/O peripherals and communication interfaces.

→ Try to make a LED blink on your Raspberry Pi (as above in section 1.3.3) with the Matlab and/or Simulink support packages.

[8] Mathworks support: Matlab support package for Raspberry Pi hardware
[9] Mathworks support: Simulink support package for Raspberry Pi hardware

Chapter 2

Basics of ROS

2.1 Presentation

2.1.1 ROS Framework

Robot Operating System (ROS)[10] [11] is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers or computing platforms.

ROS is an operating system by name but is not an operating system as you probably know. Instead, it is a flexible and collaborative open-source framework for developing robot software (also called software development kit), i.e. a collection of tools, libraries, and standards that are all focused on robotics development.

The primary goal of ROS is to support code reuse in robotics research and development, in a language-independent framework. In ROS, the robot's software is not made of one single routine, but of multiple autonomous programs that share data according to standardized protocols and formats. ROS is designed as a system of systems, i.e. a collection of individual programs that communicate and collaborate together to perform a common task. Thanks to such a structure, the designer can assemble elementary bricks to build his application according to his needs. He can exchange bricks, when he replaces a sensor for example, without having to rebuild everything.

2.1.2 The ROS Ecosystem

The key of ROS is the implementation of 4 main types of mechanisms allowing to build of a robotic application (see Fig. 2.1):

Figure 2.1: Illustration of the ROS Ecosystem.

Plumbing: ROS provides a message-passing system, often called "middleware" or "plumbing", that sets the connection of the software components between them whatever the distribution of the computing nodes on a network. This will be detailed in section 2.2.

Tools: a set of software allowing to analyze, display and debug a complex and distributed application.

Capabilities: libraries that implement functionalities such as perception, mapping, planning, control, or manipulation. From drivers to algorithms, to user interfaces, ROS provides the building blocks that allow you to focus on your application.

Community: a sufficient number of users who have more interest in collaborating than in rebuilding the same tools individually[12]. This support community includes documentation, tutorials, package organization, and even software distribution.

2.1.3 ROS Distributions and related Operating Systems

As mentioned above, ROS is not an operating system in the strict sense, but rather a software development kit, and it needs a genuine OS to run. A ROS distribution is a versioned set of ROS packages, similar to Linux distributions (e.g. *Ubuntu*), that allow developers to work against a relatively stable code base until they are ready to roll everything forward. A ROS distribution is generally associated

[10] ROS support: https://www.ros.org/
[11] ROS wiki: https://wiki.ros.org/
[12] ROS community: https://www.ros.org/blog/community/

with an OS release. The recommended ROS distribution, for now, is called **ROS Noetic (Ninjemys)** which is designed for use with **Ubuntu 20.04 (Focal)** release. The present document is built on these.

Although other operating systems are supported to varying degrees (such as *Windows* since ROS 2, see section 2.1.5), Ubuntu has been the main platform for ROS since the very beginning, due to its flexibility and user-friendliness. ROS is led by Open Robotics[13], similar to how Canonical[14] supports Ubuntu: Open Robotics steers the ship but it is driven by the community.

2.1.4 ROS explained in Video

Apart from the official ROS website[10] and wiki[11] to find information and tutorials, you should also find a lot of tips and advice on discussion forums or video channels... For example, see the interesting collection of videos from RoboJackets[15], to name only one, where concepts are very simply explained.

2.1.5 How about ROS 2 (not covered here)

ROS 2 is a new version of the middleware, completely redesigned from scratch with new capabilities, like real-time support. ROS 2 is also supported on Windows and not only Ubuntu anymore. However, it is worth mentioning that ROS 2 is still under development. Some packages have already been released but most of them are not yet compatible. Moreover, some features of ROS 2 are not yet available.

This is why we have chosen ROS here (and not ROS 2), with its **Noetic** distribution in particular.

[13] Open Robotics
[14] Canonical: publisher of Ubuntu
[15] ROS training, video collection from RoboJackets

2.2 ROS Concepts

2.2.1 Nodes

ROS is designed to be modular at a fine-grained scale. Instead of having one big program that handles all the behavior of a robot, the problem is divided into many smaller processes, each of which only handles a piece of the problem. In ROS, these processes are called *nodes*, and any robotics application usually comprises many nodes. Nodes are processes that perform computation.

Each of the nodes is an individual and independent program running on one or several computing platforms, and they are all working together to get a larger task. Furthermore, different implementations can be swapped out for each of these nodes, without having to recompile the complete code and without the other nodes even knowing about the change.

2.2.2 Topics and Messages

Topics: The simplest and most common way to interconnect nodes to get data between them is called a *topic*. Topics are the connection or channels across which nodes can communicate, see Fig. 2.2.

The first key property of a topic is that it is a *one-way* communication channel: data only goes in one direction from one topic to another one. Any node that is sending information is called a *publisher*, and any node that is receiving information from a topic is called a *subscriber* for that topic. Any node can have multiple topic connections: it might be a publisher for some topics and a subscriber for other topics. Finally, for two-way communication between two nodes, two separate topics are needed to set up such a configuration: the first node will be a publisher for one topic and a subscriber for the other, and vice versa for the second node.

Another important property about topics is that they are said *many-to-many*: any individual topic can have as many publishers and as many subscribers as required for the application. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

Messages: Nodes communicate with each other through topics by passing *messages*. A message is simply a data structure, with an associated type that explains its layout, see Fig. 2.2. That type is broken down into fields, with name and associated data type. Standard primitive types (integer, floating point, boolean, string, etc.) are supported, as are arrays of primitive types. And messages can include nested structures: fields can use message types as their field type.

Topics are associated with one message type: every message that goes across a given topic has to be of the same type. As a consequence, a node that needs to send out multiple kinds of information requires publishing multiple and appropriate topics, where each topic has a different message type associated.

Having these predefined structures and having messages associated with a given topic implies having an interface mechanism for decoupling nodes. Publishers and subscribers are not aware of each other's existence. But when a node subscribes to a given topic, it knows what kind of information it is getting without having to identify the content in every single message. The message does not have to describe its own structure because both endpoints already know which structure they are expecting.

2.2.3 ROS Computation Graph

Nodes are connected by *topics* across which *messages* are sent: this is the *ROS Computation Graph*. The ROS Graph also specifies the node and topic names (see Fig. 2.2).

2.2.4 ROS Master and ROS Parameter Server

ROS Master: ROS is definitely a distributed system, where code is broken up into different programs (or nodes) that are talking to each other (through topics) with a predefined structure (message). However, there is one part of ROS that is centralized, called the *ROS Master*.

The ROS Master provides name registration and lookup to the rest of the ROS computation graph. Without the ROS Master, nodes would not be able to find each other and exchange. The job of the ROS

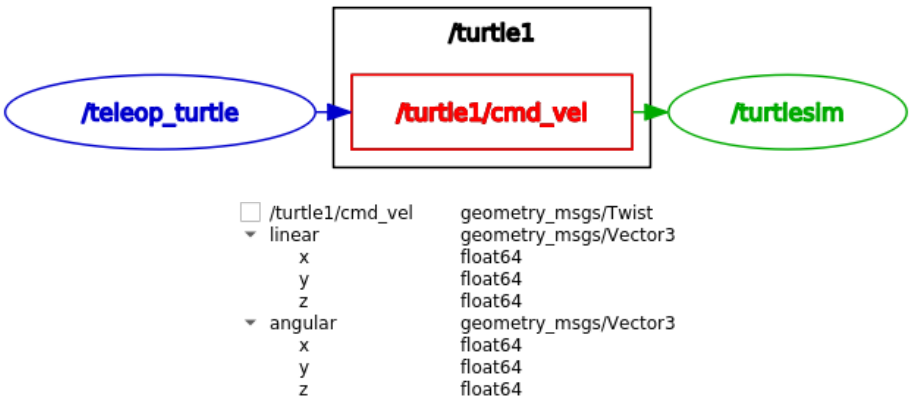


Figure 2.2: ROS nodes, topics, and message example: a first node (*/turtle1/teleop_turtle* for teleoperation) can publish messages on a given topic (*/turtle1/cmd_vel* for the robot velocity) to which is subscribed another node (*/turtlesim*, i.e. a robot simulator) in order to make the robot move in the simulator. The message structure is of a defined type (*geometry_msg/Twist*, that is two vectors of *floats* for linear and angular velocities of the robot, both in 3 dimensions).

Master is to keep track of the state of the ROS Graph, to know which nodes exist, with which topics each node can communicate, and how.

The ROS Master is launched using the command:

```
$ roscore
```

that has to be executed before doing anything else.

ROS Parameter Server: The *ROS Parameter Server* is a dictionary containing global variables which are accessible from anywhere in the current ROS environment. The Parameter Server is automatically created inside the ROS Master just after launching this latter one. `rosparam` is a command-line tool for getting and setting parameters on the ROS Parameter Server.

2.2.5 Services and Actions

Services: The publish/subscribe model, through topics and messages, is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request/reply interactions, which are often required in a distributed system. Request/reply is done via *services*, a *two-way* communication which is defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. Services are sometimes referred to as *remote procedure calls*.

Actions: In some cases, if the service takes a long time to execute, the user might want the ability to get periodic feedback about how the request is progressing or cancel the request during execution. This is possible via *actions*, which provide tools to create servers that execute long-running goals that can be preempted. It also provides a client interface to send requests to the server.

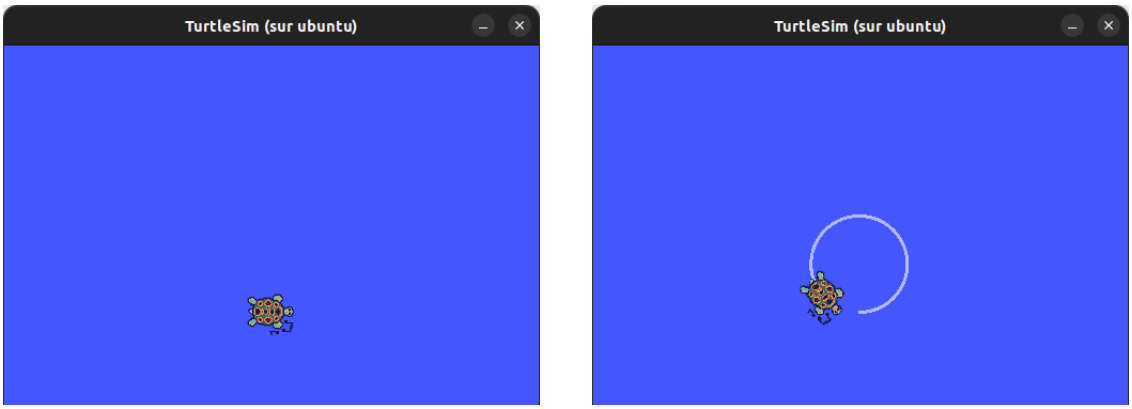
2.2.6 RQT

`rqt`[16] is a software framework of ROS that implements the various graphical user interface (GUI) tools in the form of plugins (see [16]/Plugins). `rqt` is a powerful interface to visualize how ROS works, in particular the ROS Graph with nodes (plugin group *Introspection*), topics and messages (plugin group *Topics*), services and actions (plugin groups *Services* and *Actions*)...

[16]RQT package summary: http://wiki.ros.org/rqt

2.2.7 TurtleSim

`turtlesim`[17] is a lightweight simulator for learning ROS. This package illustrates what ROS does at the most basic level, to give an idea of what will be possible to do with a real robot or robot simulation later on. The simulator consists of a graphical window that shows a turtle-shaped mobile robot (see Fig. 2.3). The turtle can be moved around on the screen by ROS commands or using the keyboard.



(a) Random turtle-shape robot in TurtleSim.

(b) Example of movement of the turtle in TurtleSim.

Figure 2.3: TurtleSim simulator interface.

2.2.8 ROS Bags

A `bag` is a file format in ROS for saving and playing back ROS message data. Bags (so named because of their `.bag` extension) are an important mechanism for storing data, processing, analyzing, and visualizing them. Data, such as sensor data, can be difficult to collect but is necessary for developing and testing algorithms; bags allow focusing on the development by putting aside the experimental aspect.

`rosbag` is a console tool for recording, playback, and other operations on ROS bags. `rqt_bag` is a graphical tool for visualizing bag file data in `rqt` (see section 2.2.6).

[17] TurtleSim package summary: http://wiki.ros.org/turtlesim

2.3 Useful ROS Commands

Help

- `$ COMMAND -h` opens the help page for the command.
  eg. `$ rosnode -h`

ROS install

- `$ sudo apt update` to first make sure the Ubuntu package index is up-to-date (see section 1.2).
- `$ sudo apt install ros-DISTRO-INSTALL` installs ROS packages, where `INSTALL` has to be picked according to the amount of ROS packages you would like to install:
  - `$ sudo apt install ros-DISTRO-desktop-full` (recommended) installs everything in the `desktop` install plus 2D/3D simulators and 2D/3D perception packages.
    eg. `$ sudo apt install ros-noetic-desktop-full`
  - `$ sudo apt install ros-DISTRO-desktop` installs everything in the `ros-base` install plus tools like `rqt` and `rviz`.
  - `$ sudo apt install ros-DISTRO-ros-base` installs ROS packaging, builds, and communication libraries (no GUI tools).
- `$ sudo apt install ros-DISTRO-PACKAGE` installs a specific package directly.
  eg. `$ sudo apt install ros-noetic-turtlebot3`

ROS environment source and .bashrc file

- After installing ROS, you will need to run a `source` command on every new terminal to set the environment and have access to the ROS commands and paths:
  `$ source /opt/ros/DISTRO/setup.bash`
  This process allows the installation of several ROS distributions (eg. Melodic and Noetic) on the same computer and switching between them.
  eg. `$ source /opt/ros/noetic/setup.bash`
- In order not to do that on each terminal, you could add the ROS sourcing in the `.bashrc` configuration file:
  `$ echo "source /opt/ros/DISTRO/setup.bash" >> ~/.bashrc`
  eg. `$ echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc`
  Then, when a terminal is opened the `source` command is automatic. Note that for the first time, you add the command line in the `.bashrc` file, consider the modification by doing:
  `$ source ~/.bashrc`

Rmk: The `.bashrc` file is a file that configures all the terminal windows, it is also used to configure the ROS Master's IP address when `roscore` is running on a remote computer for instance.

ROS packages, rosrun, roslaunch

- `rospack` allow retrieving information about ROS packages available on the filesystem.
  - `rospack list` lists all installed packages.
  - `rospack find PACKAGE` allows knowing where is a package.
- `roscd PACKAGE` allows a `cd` directly to a package or common location by name rather than having to know the package path.
- `rosls PACKAGE` lists files of a ROS package.
- `rosrun PACKAGE EXECUTABLE` allows running an executable in an arbitrary package without having to `cd` (or `roscd`) there first.
  eg. `$ rosrun turtlesim turtle_teleop_key`
- `rosrun PACKAGE EXECUTABLE __name:=NEW_NAME` allows the launch of a node but changes its name.
  eg. `$ rosrun turtlesim turtle_teleop_key __name:=tele2`
- `roslaunch PACKAGE FILE.launch` launches a set of nodes from a configuration file.
  eg. `$ roslaunch turtlebot3_bringup turtlebot3_robot.launch`

roscore, rosparam

- `roscore` runs the ROS core stack: ROS Master, ROS Parameter Server, `rosout` (logging node), etc... You must have a `roscore` running for ROS nodes to communicate.
  eg. `$ roscore &` (with `&` at the end to execute the command in the background (see section 1.2)
- `rosparam` enables getting and setting parameter server values using YAML-encoded text.
  - `rosparam list` lists all parameter names.
  - `rosparam get PARAMETER` gets a parameter value.
  - `rosparam set PARAMETER VALUE` sets a parameter value.
  - `rosparam delete PARAMETER` deletes a parameter value.

rosnode, rostopic, rosmsg, rosservice/rossrv

- `rosnode` displays runtime node information.
  - `rosnode list` lists all current nodes.
  - `rosnode info /NODE` displays information about a node, including publications and subscriptions.
    eg. `$ rosnode info /teleop_turtle`
- `rostopic` displays run-time information about topics and allows printing out messages being sent to a topic.
  - `rostopic list` lists all active topics.
  - `rostopic info /TOPIC` displays information about a topic.
    eg. `$ rostopic info /turtle1/cmd_vel`
  - `rostopic type /TOPIC` displays topic type of a topic.
    eg. `$ rostopic type /turtle1/cmd_vel`
  - `rostopic pub /TOPIC` publishes data to a topic.
    eg. `$ rostopic pub /turtle1/cmd_vel`
  - `rostopic echo /TOPIC` displays messages published to a topic.
    eg. `$ rostopic echo /turtle1/pose`
- `rosmsg` displays message data structure definitions.
  - `rosmsg list` lists all messages.
  - `rosmsg show MESSAGE` display the fields in a ROS message type.
    eg. `$ rosmsg show geometry_msgs/Twist`
- `rosservice` displays run-time information about services and allows printing out messages being sent to a service.
  - `rosservice list` lists all active services.
  - `rosservice info /SERVICE` displays information about a service.
  - `rosservice type /SERVICE` displays service type.
  - `rosservice call /SERVICE ARGS` calls the service with the provided arguments `ARGS`.
    eg. `$ rosservice call /add_two_ints 1 2`
- `rossrv` displays service data structure definitions.
  - `rossrv list` lists all services.
  - `rossrv show MESSAGE` display the fields in a ROS service type.

rosbag

- `rosbag` performs various operations on ROS bag files, including playing, recording, and validating.

2.4 Practical Work

This section is partially based on [11]/Tutorials.

2.4.1 Understanding ROS through TurtleSim

A few steps to use the `turtlesim` simulator together with the `rqt` tool (see section 2.2), is proposed here to better understand the ROS Graph concepts.

It is assumed to have ROS correctly installed (**Noetic** distribution here) on a **local computer** running **Ubuntu 20.04** (eventually through a virtual machine).

2.4.1.1 Preliminaries

- Each time of opening a new terminal, you first have to set the ROS environment (if not done in the `.bashrc` file), as explained in section 2.3, to access the ROS commands and paths. For the **Noetic** distribution:

  ```
  $ source /opt/ros/noetic/setup.bash
  ```

- As a reminder, the `-h` option opens the help page for a command (see section 2.3).

2.4.1.2 Roscore

- In a first terminal `[TERMNAL 1]`, execute `roscore`:

  ```
  $ roscore
  ```

  Rmk: The `roscore` can be executed in the background (using `&` at the end of the command line).

2.4.1.3 TurtleSim

- In a second terminal `[TERMNAL 2]`, run in background the node `turtlesim_node` from the package `turtlesim` to visualize the turtle:

  ```
  $ rosrun turtlesim turtlesim_node &
  ```

  The simulator window appears as in Fig. 2.3a, with a random turtle in its center.
- In the same terminal, then run `turtle_teleop_key` to move the turtle using the keyboard:

  ```
  $ rosrun turtlesim turtle_teleop_key
  ```

  Use arrow keys to move the turtle, visualize the robot moving in the simulator window.
  Use 'q' to quit the `turtle_teleop_key` node.

→ Make the turtle move using the `teleop`.

2.4.1.4 Understanding nodes, topics and messages with command lines

- The turtle in `turtlesim` can also be moved with command lines.
- In a new terminal `[TERMNAL 3]`, try the tools `rosnode`, `rostopic` or `rosmsg` for printing information about nodes, topics, and messages respectively. For instance:
  - `$ rosnode list` lists the active nodes (`rosout` node is for `roscore`):

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

  - `$ rostopic list` shows the active topics:

```
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

For instance, `/turtle1/cmd_vel` and `/turtle1/pose` allow publishing/subscribing the velocity of the turtle or its pose respectively.

- `$ rostopic info /turtle1/cmd_vel` gets the message type and lists which nodes are publishers/subscribers for the topic `/turtle1/cmd_vel`:

```
Type: geometry_msgs/Twist
Publishers: None
Subscribers:
 * /turtlesim
```

- `$ rosmsg info geometry_msgs/Twist` prints the message structure (i.e. linear and angular velocities of the robot in all directions):

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

- `$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -- 'value'` allows publishing a message on the topic (i.e. velocity of the turtle), where `value` has to respect the structure of the message (i.e. `geometry_msgs/Twist`).
  For instance, set `'[2,0,0]' '[0,0,2]'` to move the turtle with a linear velocity of $2\,m.s^{-1}$ on the x-axis (longitudinal movement of the robot) and an angular velocity of $2\,rad$ on the z-axis (yaw), that draws a circle in `turtlesim` as depicted in Fig. 2.3b.
  Rmk: Use the `-r` option to periodically publish the velocity commands on the velocity topic: `$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- 'value'` for a rate of 1 Hz.
- `$ rostopic echo /turtle1/pose` allows subscribing/printing the topic information for the pose of the turtle (that is periodically published):

```
x: 5.544444561004639
y: 5.544444561004639
theta: 0.0
linear_velocity: 0.0
angular_velocity: 0.0
```

- `$ rostopic echo /turtle1/cmd_vel` allows subscribing/printing the topic information for the velocity of the turtle (but only when a message is published on this topic, for instance when listening at the previous `pub` command):

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0
```

→ Try to make the turtle draw a square using ROS commands.
→ Try to print the position `(x,y)` of the square corners.

2.4.1.5 Understanding services with command lines

- Some services are also available in `turtlesim`. Try the tools `rosservice` and `rossrv` for printing information about services (still in `[TERMNAL 3]`). For instance:
  - `$ rosservice list` shows the active services:

    ```
    /clear
    /kill
    /reset
    /spawn
    ...
    ```

  - `$ rosservice type /clear` prints the `/clear` service type:

    ```
    std_srvs/Empty
    ```

    Rmk: The service type is `empty` for `/clear`, this means when the service call is made it takes no argument (i.e. it sends no data when making a request and receives no data when receiving a response).
  - `$ rosservice call /clear` clears the background of the `turtlesim` node.
  - `$ rosservice type /spawn` prints the `/spawn` service type:

    ```
    turtlesim/Spawn
    ```

  - `$ rossrv show turtlesim/Spawn` prints the `/spawn` service structure:

    ```
    float32 x
    float32 y
    float32 theta
    string name
    ---
    string name
    ```

    Rmk: The two latter commands can be combined using a pipe (see section 1.2):

    ```
    $ rosservice type /spawn | rossrv show
    ```

  - `$ rosservice call /spawn value` allows spawning a new turtle at a given location and orientation, where `value` has to respect the structure of the service (the name field is optional). For instance, set `2 2 0.2 "turtle2"` to spawn a new turtle called `turtle2`.
    Executing `$ rostopic list` will show the new turtle topics related to `/turtle2`.

→ Try to make the second turtle move.
→ Try to teleport a turtle, erase its trace and reset the `turtlesim` window.

2.4.1.6 Understanding ROS parameters with command lines

- The tool `rosparam` allows the handle of data on the ROS Parameter Server (in `[TERMNAL 3]`). For instance:
  - `$ rosparam list` shows the `turtlesim` parameters (among three of them are for background color):

    ```
    /rosdistro
    /roslaunch/uris/host_nxt__43407
    /rosversion
    /run_id
    /turtlesim/background_b
    /turtlesim/background_g
    /turtlesim/background_r
    ```

  - `$ rosparam get /turtlesim/background_g` prints the parameter value for the green channel of the `turtlesim` background color.

Rmk: `$ rosparam get /` prints the contents of the entire Parameter Server:

```
rosdistro: 'noetic'
roslaunch:
  uris:
    host_nxt__43407: http://nxt:43407/
rosversion: '1.15.5'
run_id: 7ef687d8-9ab7-11ea-b692-fcaa1494dbf9
turtlesim:
  background_b: 255
  background_g: 86
  background_r: 69
```

– `$ rosparam set value` will change the red channel of the background color.
Rmk: this changes the parameter value but the `/clear` service as to be called for the parameter change to take effect.

→ Try to change the background color of the `turtlesim` window.

2.4.1.7 Understanding the ROS Computation Graph with RQT

- In another terminal `[TERMNAL 4]`, run the GUI tool `rqt` to visualize the ROS graph:

```
$ rqt &
```

- The results obtained in command lines above can now be visualized thanks to the different `rqt` plugins. For instance:
  – `Plugins/Introspection/Node Graph` to visualize either the active nodes (`Nodes only`) or the nodes with the active topics (`Nodes/Topics`), see Fig. 2.4a.
  – `Plugins/Topics/Topics Monitor` to list the active topics, and the structure of the associated messages (check one of the topics to visualize the message values online), see Fig. 2.4b.
  – `Plugins/Topics/Message Publisher` to send messages (after adding a message, change the values and check the topic to send the message online).
  For instance, see in Fig. 2.4c how to draw a circle with the turtle (as previously in Fig. 2.3b, with `linear.x = 2` $[m.s^{-1}]$ and `angular.z = 2` $[rad]$).
  Rmk: The `rate` value is the repeat publish rate (in Hz).

→ Try to make the turtle move and reproduce the behaviors tested before with ROS commands.

2.4.1.8 RQT plot

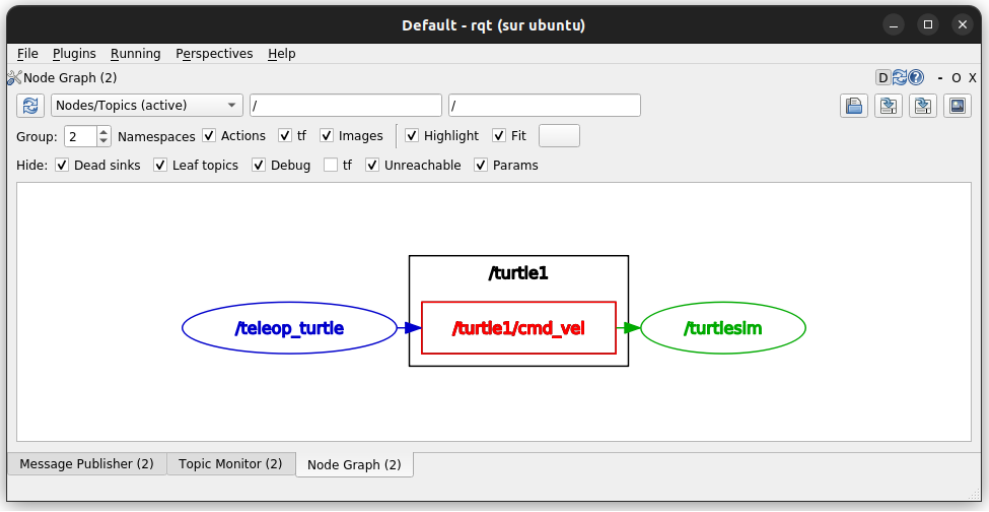
- In `[TERMNAL 4]`, run the `rqt_plot` tool to display a scrolling time plot of the data published on topics:

```
$ rosrun rqt_plot rqt_plot
```

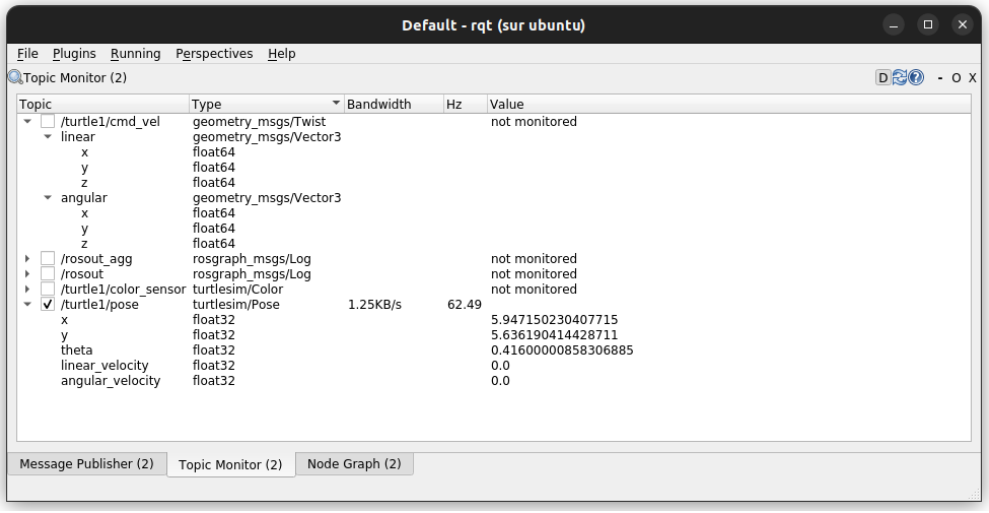
→ Try to plot the data being published on the `/turtle1/pose` topic (as in Fig. 2.5).

2.4.1.9 .bashrc file

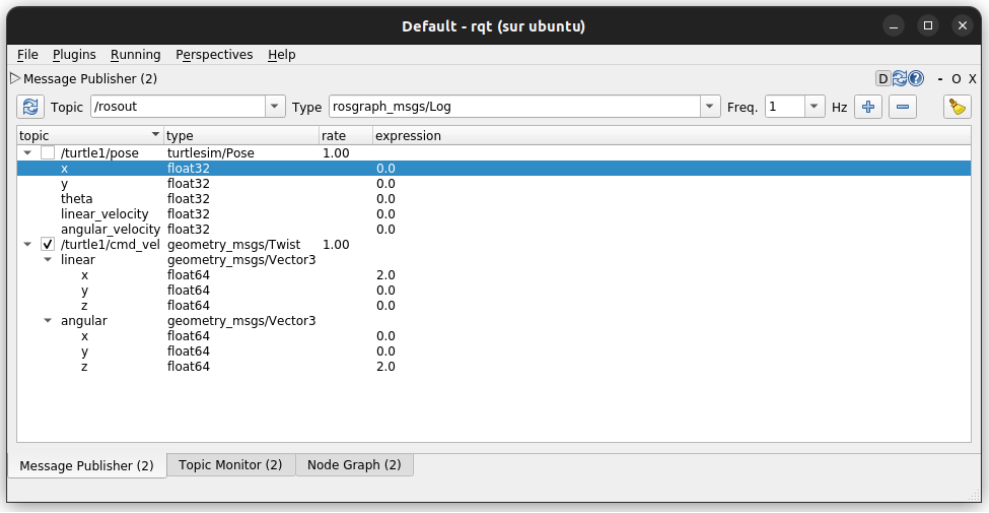
- In order not to set the ROS environment for every new terminal, add the ROS sourcing in the `.bashrc` configuration file (see section 2.3).
  – Edit the `.bashrc` file:



(a) ROS Graph example.



(b) ROS topics monitor example.



(c) ROS message publisher example to send messages to the turtle-shaped robot.

Figure 2.4: RQT interface.

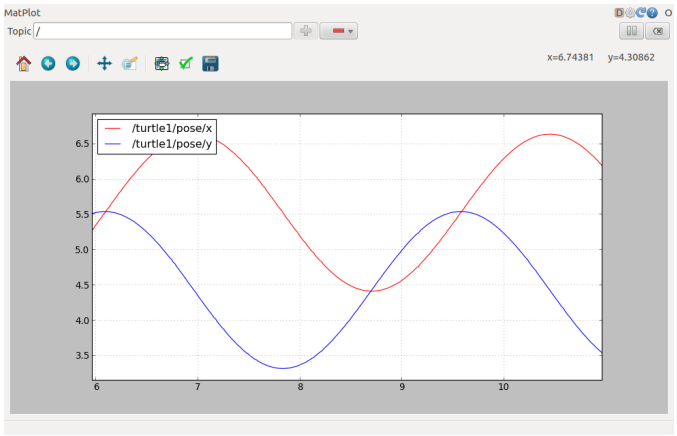


Figure 2.5: RQT plot interface.

```
$ sudo nano ~/.bashrc
```

  - At the end of the file, enter the following line:

    ```
    source /opt/ros/noetic/setup.bash
    ```

    Then, when a terminal is opened the `source` command is automatic.
  - For the first time you add the command line in the `.bashrc` file, consider the modification by doing this will not be required anymore when opening new terminals:

    ```
    $ source ~/.bashrc
    ```

- This configuration file also allows connecting to a remote ROS Master.
  - Try to connect to a remote ROS Master (`roscore` and `turtlesim` have to be running on the remote machine) by adding these lines at the end of the `.bashrc` file of the local computer:

    ```
    export ROS_MASTER_URI=http://ROSMASTER_IP:11311
    export HOSTNAME=LOCAL_IP
    ```

    where ROSMASTER_IP and LOCAL_IP are the ROS Master IP address (where `roscore` is running) and the local machine IP address respectively.
  - Source the `.bashrc` file.
  - Use the ROS commands (see above) to move the turtles in the remote computer's `turtlesim`.

→ Try to make the turtle of a remote computer move (using ROS commands or `rqt`).

2.4.1.10 ROS bags

- We will first record topic data from a running ROS system (in `[TERMNAL 4]`):
  - First, we assume the `roscore`, `trutlesim` and `turtle_teleop_key` are running.
  - We can examine the full list of topics that are currently being published in the running system:

    ```
    $ rostopic list -v
    ```

    that should yield the following output:

    ```
    Published topics:
     * /turtle1/color_sensor [turtlesim/Color] 1 publisher
     * /turtle1/cmd_vel [geometry_msgs/Twist] 1 publisher
     * /rosout [rosgraph_msgs/Log] 2 publishers
     * /rosout_agg [rosgraph_msgs/Log] 1 publisher
     * /turtle1/pose [turtlesim/Pose] 1 publisher

    Subscribed topics:
    ```

```
* /turtle1/cmd_vel [geometry_msgs/Twist] 1 subscriber
* /rosout [rosgraph_msgs/Log] 1 subscriber
```

The list of published topics gives the only message types that could potentially be recorded in the data log file, as only published messages are recorded. For instance, the topic `/turtle1/cmd_vel` is the command message published by `teleop_turtle` that is taken as input by the `turtlesim` process (see Fig. 2.4a). The messages `/turtle1/color_sensor` and `/turtle1/pose` are output messages published by `turtlesim`.

- To record the published data, run the following commands:

```
$ mkdir ~/bagfiles
$ cd ~/bagfiles
$ rosbag record -a
```

The `-a` option is to indicate that all published topics should be accumulated in a bag file, stored in the temporary directory `~/bagfile`.

- In the terminal with `turtle_teleop` (`[TERMNAL 2]`), move the turtle around for a few seconds.
- In the terminal running `rosbag` record (`[TERMNAL 4]`), exit with a `Ctrl+C`. Now examine the contents of the directory `~/bagfiles`, you should see a file with a name that begins with the year, date, and time and the suffix `.bag` (called BAGFILE.bag hereafter). This is the bag file that contains all topics published by any node in the time that the `rosbag` record was running.

- Now that a bag file was recorded, it can be played back using the commands `rosbag info` and `rosbag play`:
  - First check the contents of the bag file (without playing it back) by executing the following command from the `bagfiles` directory:

```
$ rosbag info BAGFILE.bag
```

that should give something like:

```
path:        2014-12-10-20-08-34.bag
version:     2.0
duration:    1:38s (98s)
start:       Dec 10 2014 20:08:35.83 (1418270915.83)
end:         Dec 10 2014 20:10:14.38 (1418271014.38)
size:        865.0 KB
messages:    12471
compression: none [1/1 chunks]
types:       geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
             rosgraph_msgs/Log   [acffd30cd6b6de30f120938c17c593fb]
             turtlesim/Color     [353891e354491c51aabe32df673fb446]
             turtlesim/Pose      [863b248d5016ca62ea2e895ae5265cf9]
topics:      /rosout                 4 msgs    : rosgraph_msgs/Log   (2 connections)
             /turtle1/cmd_vel      169 msgs    : geometry_msgs/Twist
             /turtle1/color_sensor 6149 msgs   : turtlesim/Color
             /turtle1/pose         6149 msgs   : turtlesim/Pose
```

which tells about topic names and types as well as the number of each message topic contained in the bag file.

  - To replay the bag file, first quit the `turtle_teleop_key` program (if still running) but leave `turtlesim` running. Then run the following command from the `bagfiles` directory:

```
$ rosbag play BAGFILE.bag
```

This will reproduce the recorded behavior in the running system.
Different options that can be tested:

* `-d` to specify the time after advertising each message (default is 0.2s).
* `-s` start the recorded data some duration past the beginning of the bag file.
* `-r` to change the rate of publishing by a specified factor (like changing the sampling time).

→ Try to record and play a scenario where the turtle is drawing a square.

2.4.2 Use of Matlab/Simulink and its ROS Toolbox

The *ROS toolbox*[18] includes Matlab functions and Simulink blocks that provide an interface connecting Matlab/Simulink with ROS, and enabling to create a network of ROS nodes.

It is assumed to have both i) ROS correctly installed (**Noetic** distribution here) on **one computer** running **Ubuntu 20.4** (or through a *Virtual Machine*), called `PC_ROS` and which IP address is `IP_ROS` hereafter, and ii) Matlab/Simulink with its ROS toolbox installed on **another computer** running Windows (for instance), called `PC_MATLAB`.

2.4.2.1 Matlab

First, start with the ROS-dedicated Matlab functions (instructions are adapted from the Mathworks support[19]):

- In a terminal of `PC_ROS`, start the ROS Master (with the command `roscore`) and `turtlesim` (with `rosrun turtlesim turtlesim_node`). See previous sections 2.4.1.2 and 2.4.1.3 for details if needed.
- In a Matlab script (on `PC_MATLAB`):
  - Connect to the ROS Master (that is installed on `PC_ROS`):

    ```
    rosinit('IP_ROS')
    ```

    A new node has been created (with a name like `/matlab_global_node_NB`), verify with the `rosnode list` command (on `PC_ROS`).
- Verify the active topics (`rostopic list` or using `rqt`), and get the message type for `/turtle1/pose` and `/turtle1/cmd_vel` in particular (using `rostopic info`).
  You should verify that `/turtle1/cmd_vel` has neither publisher nor subscriber, `/turtle1/pose` has one publisher (`/turtlesim`) and no subscriber.
- In the Matlab script:
  - To move the turtle in `turtlesim`, you can control its motion by publishing a message to the `/turtle1/cmd_vel` topic. The message has to be of type `geometry_msgs/Twist`, which contains data specifying desired linear and angular velocities. The turtle's movements can be controlled through two different values: the linear velocity along the X-axis controls forward and backward motion and the angular velocity around the Z-axis controls the rotation speed of the turtle.
    Example to draw a circle as in Fig. 2.3b. :

    ```
    cmdPub = rospublisher('/turtle1/cmd_vel','geometry_msgs/Twist')
    cmdMsg = rosmessage(cmdPub)
    cmdMsg.Linear.X = 2;            % in meters per second
    cmdMsg.Angular.Z = 2;           % in radians
    send(cmdPub,cmdMsg)
    ```
  - The turtle also uses the `/turtle1/pose` topic to publish its current pose (position and orientation). The message has to be of type `turtlesim/Pose`, which contains data specifying the position (x, y), the orientation (θ), the linear and angular velocities (v and w respectively).

    ```
    poseSub = rossubscriber('/turtle1/pose','turtlesim/Pose')
    poseMsg = receive(poseSub,3)   % timeout period, in seconds
    x = poseMsg.X
    y = poseMsg.Y
    theta = poseMsg.Theta
    v = poseMsg.LinearVelocity
    w = poseMsg.AngularVelocity
    ```
  - At the end of the Matlab script, eventually quit ROS:

    ```
    rosshutdown
    ```

[18] Mathworks support: ROS toolbox
[19] Mathworks support: Communicate with the TurtleBot

→ Verify the active topics when publishing or subscribing with Matlab. In particular, the number of publishers and subscribers should vary for `/turtle1/cmd_vel` and `/turtle1/pose`.
→ Try to make the turtle draw a square using Matlab commands.
→ Try to print the position `(x,y)` of the square corners.
→ Try to control two turtles in `turtlesim`, where the first one is controlled by `turtle_teleop_key` (see section 2.4.1.3) while the second one follows it.

2.4.2.2 Simulink

Then continue with the ROS-dedicated Simulink blocks (instructions are adapted from the Mathworks support[20] [21]):

- In Simulink (on `PC_MATLAB`):
  - Connect to the ROS Master in Matlab with `rosinit` (see above, `roscore` and `turtlesim` have to be running).
    Rmk: This initialization step can eventually be done in the `Model Properties` of the Simulink, within the `InitFcn` function in the `Callbacks` tab. Executed commands can be watched in the `Diagnostic Viever`.
  - As in Matlab, you can control the motion of the turtle by publishing a message to the `/turtle1/cmd_vel` topic. The message has to be of type `geometry_msgs/Twist`, which contains data specifying desired linear and angular velocities.
    Use a *Blank Message* block (from ROS toolbox[18]), select the *Class* `Message` and specify the *Type* as `geometry_msgs/Twist`, to define the structure of the message you want to publish.
    Use a *Bus Assignment* block to select the elements of the message structure you want to publish: here the linear velocity of the turtle along the X-axis (i.e. `Linear.X`) and its angular velocity around the Z-axis (`Angular.Z`).
    Then set the new inputs of this block as desired (for instance using *Constant* blocks with respectively `2` and `2` values respectively to make the turtle draw a circle, or using a *Matlab Function* block).
    Use a *Publish* block, set `Select from ROS network` for the *Topic source* and select the `/turtle1/cmd_vel` *Topic* with `geometry_msgs/Twist` as *Message type*.
    See the *Publisher* area in Figure 2.6.
  - The turtle can also use the `/turtle1/pose` topic to publish its current pose (position and orientation). The message has to be of type `turtlesim/Pose`.
    Use a *Subscribe* block, `Specify your own` *Topic source* and select the `/turtle1/pose` *Topic* with `turtlesim/Pose` as *Message type*.
    Use a *Bus Selector* block to select the elements of the message structure you want to subscribe: here the position of the turtle (`X`, `Y`) and its orientation (`Theta`).
    The different values can be visualized in *Scope* blocks.
    See the *Subscriber* area in Figure 2.6.
  - At the end of the script, eventually quit ROS (see above).
    Rmk: This final step can eventually be done in the `Model Properties` of the Simulink, within the `StopFcn` function in the `Callbacks` tab.

→ Try to make the turtle draw a square using Simulink blocks.
→ Try to print the position `(x,y)` of the square corners.
→ Try to control two turtles in `turtlesim`, where the first one is controlled by `turtle_teleop_key` (see section 2.4.1.3) while the second one follows it.

Rmk: In Simulink use a *Matlab function* block to use Matlab code in a Simulink model (in order not to make all the following strategy with blocks for instance).

[20] Mathworks support: Get started with ROS in Simulink
[21] Mathworks student lounge: Getting started with Matlab, Simulink, and ROS

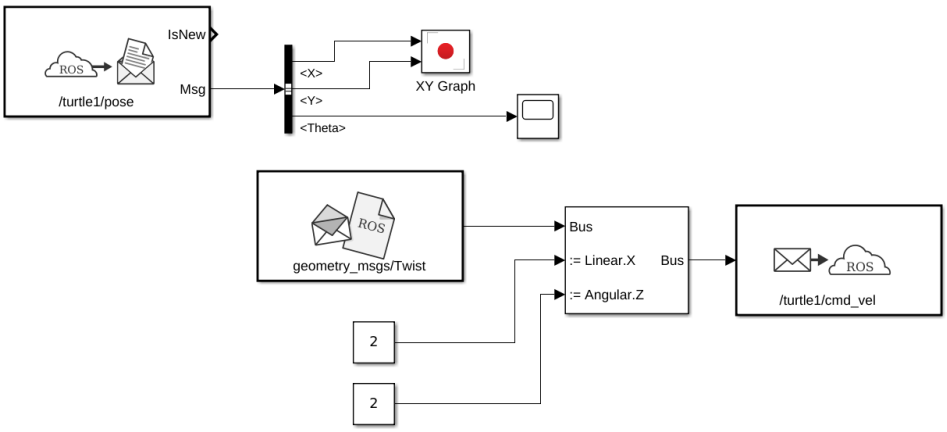


Figure 2.6: Publisher/Subscriber example in Simulink.

2.4.2.3 Standalone ROS nodes

After the prototyping process, an algorithm developed in Matlab and/or Simulink can eventually be implemented as a stand-alone node on a remote computer or robot (see[21] for details).

2.4.3 Create a ROS package using catkin

A program in ROS is called a *node* (see section 2.2) and a set of nodes is called a *package*. A package is the distribution brick of ROS, it is a set of programs that will contribute to the realization of a task.

It is assumed to have ROS correctly installed (**Noetic** distribution here) on a **local computer** running **Ubuntu 20.04** (eventually through a virtual machine).

2.4.3.1 Create a catkin workspace

- For the first time (if the catkin workspace directory `~/catkin_ws` does not exist yet), you need to create and build a catkin workspace in your home directory:

  ```
  $ mkdir -p ~/catkin_ws/src
  $ cd ~/catkin_ws/
  $ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3
  ```

  The `catkin_make` command is a convenient tool for working with catkin workspace. Running it the first time in your workspace will create a `CMakeLists.txt` link in your `src` folder.
  Rmk: Python 3 is recommended for ROS **Noetic**: the first `catkin_make` command in a clean catkin workspace with option `-DPYTHON_EXECUTABLE` will configure `catkin_make` with Python 3. You may then proceed to use just `catkin_make` for subsequent builds.
- Looking in the current directory you should now have a `build` and `devel` folder.
  Rmk: Inside the `devel` folder you can see that there are now several `setup.*sh` files. Sourcing any of these files will overlay this workspace on top of your environment.
- Source the new `setup.bash` file in the `devel` folder to overlay your catkin workspace on top of your ROS environment:

  ```
  $ source devel/setup.bash
  ```

- To make sure your workspace is properly overlayed by the setup script, verify that `ROS_PACKAGE_PATH` environment variable includes your catkin workspace directory:

  ```
  $ echo $ROS_PACKAGE_PATH
  ```

  This should print something like `/home/USER/catkin_ws/src:/opt/ros/noetic/share`

2.4.3.2 Create a first *Hello World* ROS package in the catkin workspace

- For a package to be considered a catkin package it must meet a few requirements:
  1. The package must contain a catkin compliant `package.xml` file (which provides meta-information about the package, see [11]/Tutorials/CreatingPackage for further details);
  2. The package must contain a `CMakeLists.txt` (used by catkin);
  3. Each package must have its own folder (this means no nested packages nor multiple packages sharing the same directory).
- The packages tree in a catkin workspace should look as follows:

```
catkin_ws/                -- CATKIN WORKSPACE
  src/                    -- SOURCE SPACE
    CMakeLists.txt        -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt      -- CMakeLists.txt file for package_1
      package.xml         -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt      -- CMakeLists.txt file for package_n
      package.xml         -- Package manifest for package_n
```

- Create a new catkin package:
  - First change to the source space directory of the catkin workspace:

    ```
    $ cd ~/catkin_ws/src
    ```

  - Then use the `catkin_create_pkg` command to create a new package called `beginner_tutorials` which depends on `std_msgs`, `roscpp`, and `rospy`:

    ```
    $ catkin_create_pkg PACKAGE_NAME DEPENDENCIES
    ```

    This will create a `PACKAGE_NAME` folder which contains a `package.xml` and a `CMakeLists.txt`, which have been partially filled out with the optional list of `DEPENDENCIES` on which that package depends.
  - For instance, we propose to create a package `hello_world` in Python (which depends on `rospy`, see [11]/rospy/Overview):

    ```
    $ catkin_create_pkg hello_world rospy
    ```

    A new directory `hello_world` was created (use the `ls` or `rosls` command to verify).
- Build the catkin workspace and source the setup file:

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

- Write a *Hello World* node in Python:
  - Change directory into the `hello_world` package you created before:

    ```
    $ roscd hello_world
    ```

  - Create a `scripts` folder to store the Python scripts:

    ```
    $ mkdir scripts
    $ cd scripts
    ```

  - Create and edit the file `helloworld.py`:

    ```
    $ nano helloworld.py
    ```

    The file will contain the following lines (be careful about indentation in Python):

```
1  #!/usr/bin/env python3
2  import rospy
3
```

```
4    if __name__ == "__main__":
5      rospy.init_node("HelloWorld_node")
6      rospy.loginfo("HelloWorld node has been started.")
7
8      rate = rospy.Rate(10)
9      while not rospy.is_shutdown():
10       print("Hello World")
11       rate.sleep()
```

Explanations:

* The first line 1 allows choosing the Python interpreter (for the **Noetic** distribution it is advised to use Python3 and not Python).
* Line 2 is to import all libraries for using Python with ROS.
* Line 4 is to declare the main program.
* Line 5 is to initialize a node which will have the name `HelloWorld_node` in the ROS graph. Be careful the names must be unique (if a node with the same name comes up, it bumps the previous one).
* Line 6 is to log messages from nodes. `loginfo()`, `logwarn()` and `logerr()` change according to the importance of the information.
* Line 8 is to set the frequency of the loop that follows.
* Line 9 is a while loop until the program ends (e.g. if there is a `Ctrl+C` or otherwise).
* Line 10 is for printing the "Hello world" message.
* Line 11 pauses the program for the time necessary to meet the frequency defined in line 8.

- Add the `helloworld.py` script path to the `CMakeLists.txt` of the package `hello_world`:

```
$ roscd hello_world
$ nano CMakeLists.txt
```

and add the following lines in the `install` section:

```
catkin_install_python(PROGRAMS
  scripts/helloworld.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

This makes sure the Python script gets installed properly and uses the right python interpreter.

- Build the node in the catkin workspace and source the setup file:

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

- Finally, run the node `helloworld.py` of the package `hello_world`:

```
$ rosrun hello_world helloworld.py
```

The "Hello World" message should be printed in the terminal.

2.4.3.3 Create a ROS package with Publisher/Subscriber nodes

- Here, we propose to sum up the steps to create a package `turtle_circle` with 2 nodes in Python: one to make a turtle in `turtlesim` draw a circle and another one to print the turtle pose:
  - Create a new catkin package `turtle_circle` (which depends on `rospy` and `turtlesim`):

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg turtle_circle rospy turtlesim
```

  - Write a *Publisher* node in Python (called `turtle_publisher.py`):

```
$ roscd turtle_circle
$ mkdir scripts
$ cd scripts
$ nano turtle_publisher.py
```

The file will contain the following lines:

```
1   #!/usr/bin/env python3
2   import rospy
3   from geometry_msgs.msg import Twist
4
5   if __name__ == "__main__":
6     rospy.init_node("Publisher_node")
7     rospy.loginfo("Publisher node has been started.")
8
9     pub = rospy.Publisher("/turtle1/cmd_vel", Twist)
10
11    rate = rospy.Rate(10)
12    while not rospy.is_shutdown():
13      print("Make the turtle draw a circle")
14      msg = Twist()
15      msg.linear.x = 2
16      msg.angular.z = 2
17      pub.publish(msg)
18      rate.sleep()
```

Explanations:

* See the `helloworld.py` file above for the general explanations.
* Line 9 is to register the node as a publisher on topic `/turtle1/cmd_vel`, with messages of type `Twist`.
* Line 3 is to import the structure of message type `Twist`, that is `geometry_msgs/Twist` for the topic `/turtle1/cmd_vel` (remember to use `rostopic info /turtle1/cmd_vel`, see section 2.4.1.4). The message type `Twist` consists of two vectors for the linear and angular velocities of the turtle.
* Lines 14 to 16 is to initialize the message to send (to make the turtle draw a circle).
* Line 17 is to send/publish the message on topic `/turtle1/cmd_vel`.

– Write a *Subscriber* node in Python (called `turtle_subscriber.py`):

```
$ nano turtle_subscriber.py
```

The file will contain the following lines:

```
1   #!/usr/bin/env python3
2   import rospy
3   from turlesim.msg import Pose
4
5   pos = Pose()
6   def pose_callback(data)
7     global pos
8     pos = data
9
10  if __name__ == "__main__":
11    rospy.init_node("Subscriber_node")
12    rospy.loginfo("Subscriber node has been started.")
13
14    sub = rospy.Subscriber("/turtle1/pose", Pose, callback = pose_callback)
15
16    rate = rospy.Rate(10)
17    while not rospy.is_shutdown():
18      print('Print the turtle pose:)
19      print('   x=',pos.x,' y=',pos.y,' theta=',pos.theta)
20      rate.sleep()
```

Explanations:

* See the `helloworld.py` file above for the general explanations.

* Line 14 is to register the node as a subscriber on topic `/turtle1/pose`, with messages of type `Pose`.
* Line 3 is to import the structure of message type `Pose`, that is `turtlesim/Pose` for the topic `/turtle1/pose` (remember to use `rostopic info /turtle1/pose`, see section 2.4.1.4). The message type `Pose` consists of 5 real numbers among the position (x, y) and orientation (θ) of the turtle.
* When a message is published on topic `turtlesim/Pose`, the callback subroutine `pose_callback` (defined in line 14) is run with the received message passed as a parameter. This subroutine is defined in lines 5 to 8. Line 5 is to initialize the message to receive of type `Pose` (as defined in line 14).
* Line 19 is to print the pose of the turtle, received from the messages of the topic `/turtle1/pose`.

– Add the path of `turtle_publisher.py` and `turtle_subscriber.py` scripts to the `CMakeLists.txt` of the package `turtle_circle`:

```
$ roscd turtle_circle
$ nano CMakeLists.txt
```

and add the following lines in the `install` section:

```
catkin_install_python(PROGRAMS
  scripts/turtle_publisher.py
  scripts/turtle_subscriber.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

– Build the node in the catkin workspace and source the setup file:

```
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

– Run the node `turtle_publisher.py` in a first terminal:

```
$ rosrun turtle_circle turtle_publisher.py
```

The turtle should draw a circle in `turtlesim` (assuming `roscore` and `turtlesim` are running).

– Run the node `turtle_subscriber.py` in a second terminal:

```
$ rosrun turtle_circle turtle_subscriber.py
```

The pose of the turtle should be printed.

→ Try to make the turtle draw a square using a new package in Python.
→ Try to print the position `(x,y)` of the square corners.
→ Try to control two turtles in `turtlesim`, where the first one is controlled by `turtle_teleop_key` (see section 2.4.1.3) while the second one follows it.

2.4.3.4 Create a roslaunch file

- `roslaunch` allow launching several nodes in a single command, as defined in a launch file. Here, we propose to create a launch file in order to run the two nodes of the package `turtle_circle` defined above.

  – First go to the `turtle_circle` package and make a `launch` directory:

  ```
  $ roscd turtle_circle
  $ mkdir launch
  $ cd launch
  ```

  – Write a launch file (called `turtle_circle.launch`):

  ```
  $ nano turtlecircle.launch
  ```

  The file will contain the following lines:

```
1  <launch>
2    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
3    <node pkg="turtle_circle" name="pub" type="turtle_publisher.py"/>
4    <node pkg="turtle_circle" name="sub" type="turtle_subscriber.py"/>
5  </launch>
```

Explanations:

* Lines 1 and 5 are the start and close tags for the launch file.
* Lines 2 to 4 allow running a node (`type`) of a package (`pkg`) with a given name (`name`).

– Use `roslaunch` to launch the file:

```
$ roslaunch turtle_circle turtlecircle.launch
```

Rmk: With a launch file, it is not necessary to launch the `roscore` since the `roslaunch` command automatically opens the ROS Master if it does not exist.

→ Try to create a launch file to start `turtlesim` and `turtle_teleop_key` (see section 2.4.1.3).
→ Try to create another launch file to control two turtles in `turtlesim`, where the first one is controlled by `turtle_teleop_key` while the second one follows it.

Rmk: It is also possible to launch services in a launch file. For instance, to spawn a second turtle add the following line in the launch file:

```
<node pkg="rosservice" name="spawn" type="rosservice"
                  args="call /spawn '2' '2' '0' 'turtle2'"/>
```

Part II

Control of a real Robot

Chapter 3

Introduction

3.1 General Setup

The general setup – presented in Fig. 3.1 – is dedicated to controlling a real TurtleBot mobile robot (see section 3.1.2) with ROS, using an OptiTrack motion capture system for its localization (section 3.1.3).

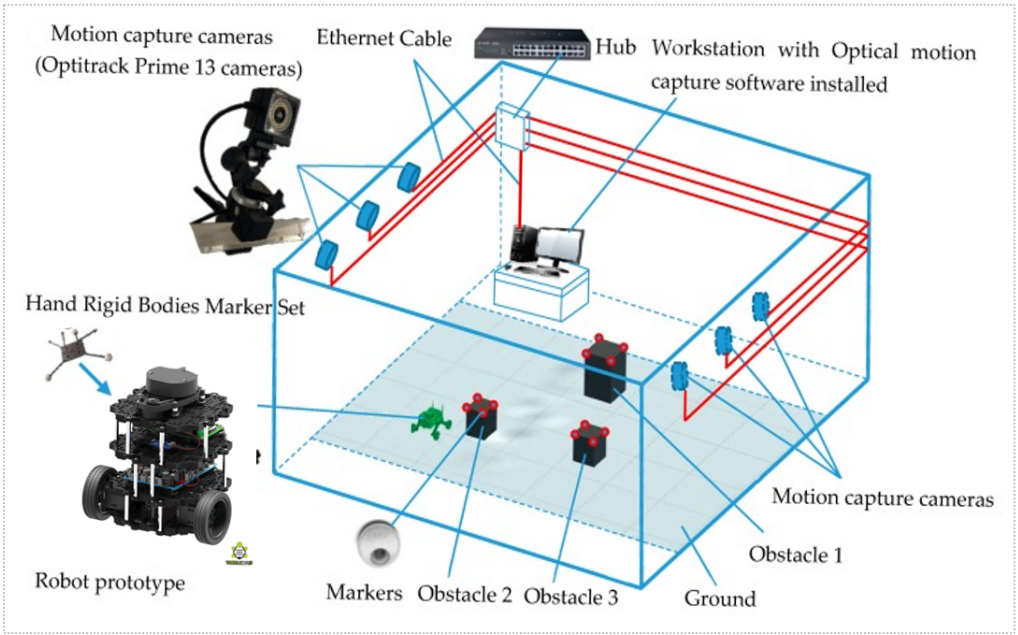


Figure 3.1: General setup.

The architecture decided here is to only have ROS installed on Raspberry Pi single-board computers (section 3.1.1) and not on any computer, in order not to need a dedicated computer and for students not to need to install ROS on their personal computer. A computer (workstation) is only needed here for the motion capture software.

One Raspberry Pi is dedicated to the ROS Master (see section 2.2 for details). It will also be used to collect data from the motion capture system and could be useful if several robots have to be controlled (not discussed here). A Raspberry Pi is also embedded in each robot to control. All these devices are connected to a wireless network through a WiFi hotspot.

This setup is partially inspired by the *RobotMe* educational robotic platform [1] implemented at GIPSA-lab, Grenoble, France.

3.1.1 Raspberry Pi

The small single-board computer Raspberry Pi was introduced in section 1.

Here, one Raspberry Pi is dedicated to the ROS Master and another Raspberry Pi is embedded in each TurtleBot robot to control, all connected through a WiFi network. ROS Master or TurtleBot's Raspberry Pi can be accessed remotely with any remote computer connected to the same network:

i) either through SSH (Secure Shell), typically for installation. To remotely access a Raspberry Pi with SSH, use the command line:

```
$ ssh RPi_USERNAME@IP_TURTLEBOT1
```

where `RPi_USERNAME` and `RPi_PASSWORD` are the login and password of the device respectively, `IP_TURTLEBOT1` is its IP address (see section 3.2).

ii) or through Matlab/Simulink, typically for development (see section 4.3.2).

3.1.2 The TurtleBot3 Mobile Robot

TurtleBot is a lightweight, compact, cost-effective, and customizable mobile robotics educational platform.

The TurtleBot3 versions that are concerned here, the Burger or Waffle models (see Fig. 3.2), were created in a partnership between Open Robotics[22] (who is developing the open-source ROS software) and ROBOTIS[23] (who is developing open-source hardware). They include powerful sensors required for navigation, like an inertial measurement unit (IMU), LIDAR or camera, and can be connected to various other sensors. TurtleBot is also an open hardware platform and its mechanical structure can be easily modified and/or 3D printed.

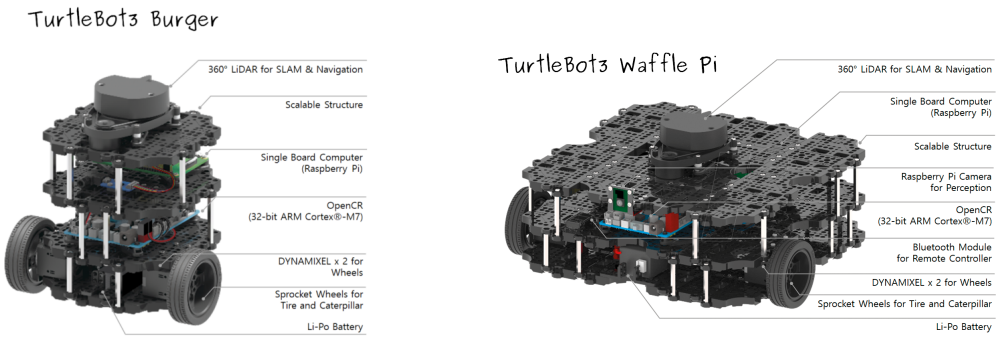


Figure 3.2: TurtleBot3 mobile robot, Burger and Waffle Pi versions.

3.1.3 The OptiTrack Motion Capture System and its Motive Software

The OptiTrack[24] motion capture system, see Fig. 3.3a, is based on infrared cameras and passive reflective markers to place on the robots to be controlled. Knowing the a priori configuration of the markers on each robot, the pose (positions and orientations) is calculated by triangulation. The 6-DoF (degrees of freedom) pose can then be stored for a posteriori ground truth verification (in order to validate localization methods for example), or transmitted in real-time (via a TCP/IP communication channel) for the closed-loop control of the robot.

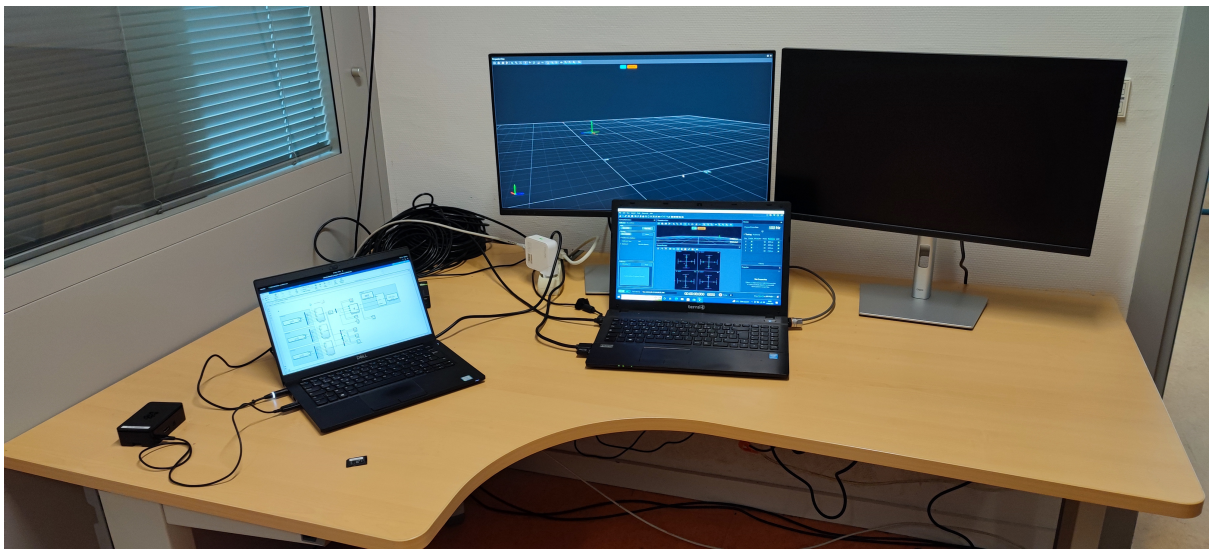
The 4 motion capture cameras (Prime^{X} 13[25]) are installed in each corner of the room through adjustable wall mounts, see Fig. 3.3b. Cameras are connected through and powered by a PoE switch, via Ethernet cables, into the workstation (with Motive installed), see Fig. 3.3c. Motive is a software platform designed to control OptiTrack motion capture systems for various tracking applications. Motive not only allows the user to calibrate and configure the system, but it also provides interfaces for both capturing and processing of 3D data. Motive will be detailed in section 5.2.1.

Motion capture data can be recorded or live-streamed into other pipelines. Here, a second network is implemented through a WiFi hotspot, used to stream the OptiTrack information to the ROS Master. The WiFi hotspot is also used to connect the ROS Master and the TurtleBot's Raspberry Pis, see Fig. 3.3c.

[22] https://www.openrobotics.org/
[23] http://en.robotis.com/
[24] https://www.optitrack.com/
[25] Prime^{X} 13 OptiTrack cameras

(a) OptiTrack motion capture room.

(b) Prime^{X} 13 camera and its adjustable wall mount.



(c) OptiTrack's switch, workstation with Motive software, and ROS Master's Raspberry Pi.

Figure 3.3: OptiTrack motion capture room setup.

3.2 Definition of Variables

WiFi network configuration:

- `WIFI_SSID` and `WIFI_PASSWORD` are **INSA-IoT** and ***** respectively.
- The WiFi hotspot's IP address is called `IP_GATEWAY` hereafter.
- The ROS Master's Raspberry Pi IP address `IP_ROSMASTER` is **10.173.20.101**.
- The TurtleBots' Raspberry Pi IP addresses `IP_TURTLEBOTi` go from **10.173.20.120** to **10.173.20.129** (120 + `TURTLEBOTi` the TurtleBot number).
- The OptiTrack motion capture system (with Motive software)'s IP address `IP_MOCAP` is **10.173.20.130**.
- The mask for this local network `IP_MASK` is **10.173.20.***.

Raspberry Pi configuration:

- `RPi_USERNAME` is **ubuntu** and the default `RPi_PASSWORD` is **turtlebot** for all Raspberry Pis (ROS Master and TurtleBots).

OptiTrack configuration:

- The name of each TurtleBot to track (i.e. a rigid body for the OptiTrack System) `TURTLEBOTi` is denoted **TurtleBoti** in the Motive software, with `i` the number of TurtleBot.

Chapter 4

ROS for TurtleBot Robots

4.1 Quick Start Guide

1. **Turn on the devices:**

   (a) Turn on the ROS Master and TurtleBot's Raspberry Pis (with IP addresses `IP_ROSMASTER` and `IP_TURTLEBOTi` respectively), with `i` the TurtleBot number.

   Rmk: Refer to section 3.2 for the variable values.

2. **Connect to the devices**

   Option A **using a monitor+keyboard on the ROS Master's Raspberry Pi:**

   (this option will slow down the operation)

   (a) Log on to the ROS Master's Raspberry Pi, using `RPi_USERNAME` and `RPi_PASSWORD`.

   Rmk: in case the `azerty` keyboard is not configured:

   ```
   $ sudo setxkbmap fr
   ```

   (b) Optional: open a terminal to verify that:

   - The IP addresses of all devices are connected to the local hotspot:

     ```
     $ nmap -sP IP_MASK
     ```

     You should see `3 hosts up`: the WiFi hotspot, the ROS Master, and the robot.

   (c) From the ROS Master's Raspberry Pi:

   i. Open a new terminal `[TERMNAL 1]` and run `roscore` (in background, using `$` at the end of the command line):

   ```
   $ roscore &
   ```

   ii. Open a new terminal `[TERMNAL 2]`, connect to the TurtleBot's Raspberry Pi via SSH and launch `turtlebot3_bringup`:

   ```
   $ ssh RPi_USERNAME@IP_TURTLEBOTi
   $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
   ```

   Option B (preferred) **using a remote computer to control the different Raspberry Pis:**

   (a) Connect the remote computer to the WiFi hotspot, using `WIFI_SSID` and `WIFI_PASSWORD`.

   (b) Optionnal: open a terminal to verify that:

   - The remote computer's IP address is in the local network range `IP_MASK`:

     ```
     $ ifconfig
     ```

   - The IP addresses of all devices are connected to the local hotspot:

     ```
     $ nmap -sP IP_MASK
     ```

     You should see at least `3 hosts up`: the ROS Master, the robot, and the remote computer.

   (c) From the remote computer:

   i. Open a new terminal `[TERMNAL 1]`, connect to the ROS Master's Raspberry Pi via SSH and run `roscore` (in the background, using `&` at the end of the command line):

   ```
   $ ssh RPi_USERNAME@IP_ROSMASTER
   $ roscore &
   ```

   ii. Open a new terminal `[TERMNAL 2]`, connect to the TurtleBot's Raspberry Pi via SSH and launch `turtlebot3_bringup`:

   ```
   $ ssh RPi_USERNAME@IP_TURTLEBOTi
   $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
   ```

3. **Try to move the robot:**

   Manually **using the keyboard from the ROS Master:**

- Launch the teleoperation node in `[TERMNAL 1]`:

  ```
  $ export TURTLEBOT3_MODEL=TB3_MODEL
  $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
  ```

  where `TB3_MODEL` is either **burger**, **waffle**, or **waffle_pi** (depending on the robot to teleoperate), and follow the instructions.

In an automatized way **through Matlab/Simulink** (see section 4.3.2).

4. **Turn off the devices:**

   (a) Quit the running nodes using `Ctrl+C` (in both `[TERMNAL 1]` and `[TERMNAL 2]`).

   (b) Shutdown the Raspberry Pis, first in `[TERMNAL 2]` to turn off the TurtleBot's Raspberry Pi and then in `[TERMNAL 1]` for the ROS Master's one:

   ```
   $ sudo halt
   ```

4.2 *ROS Install and Configuration

This tutorial is adapted from the *Quick Start Guide* in the e-manual of Robotis [26]/quick-start.

Rmk: The PC Setup step is not needed since we won't use a remote PC here but a remote Raspberry Pi (RPi) instead for the ROS Master (the next RPi Setup steps are a merge of the *SBC Setup* and the *PC Setup* from this *Quick Start Guide* while adding some extra steps for the motion capture room). Don't forget to select the right ROS distribution in the top tabs (**Noetic** here) for each step!

4.2.1 RPi Setup from SD Card Images

This first stage allows burning an image to SD card (to avoid doing all the installation steps for each Raspberry Pi). In particular, two images are available here, that can be directly installed on the Raspberry Pi's SD card:

- `setup_ROSmaster.img` for the ROS Master's Raspberry Pi;
- `setup_ROSturtle.img` for the TurtleBot's Raspberry Pi.

The detailed steps to obtain these images are detailed in section 4.2.2.

To write an image file to the SD card, use either the `dd` command (reversed to section 4.2.2.4) or graphical tools (preferred here, see [26]/sbc_setup):

1. Burn the image file to SD card with *Raspberry Pi Imager*.
2. Resize the partition of the SD card to use the unallocated space with *GParted GUI Tool*.

4.2.2 RPi Setup (optional, details of all steps to obtain the SD card images used in section 4.2.1)

4.2.2.1 Preliminary Setup for both RPi (ROS Master and TurtleBots)

Instructions are adapted from [26]/sbc_setup, select the right ROS distribution (**Noetic** here) with small changes as below.

These preliminary steps have to be done twice, on one hand on the ROS Master's Raspberry Pi and on the other hand on each TurtleBot's Raspberry Pi:

1. Download TurtleBot3 SBC image, unzip, burn the image file (with *Raspberry Pi Imager* or *Disks Utility*), and resize the partition (with *GParted GUI Tool*).
2. Configure the WiFi network setting. Go to the netplan directory in the microSD card and start editing the `50-cloud-init.yaml` file with superuser permission:

   ```
   $ sudo nano /media/$USER/writable/etc/netplan/50-cloud-init.yaml
   ```

   When the editor is opened, replace the `WIFI_SSID` and `WIFI_PASSWORD` with yours.

   IP address of each Raspberry Pi is reserved in such a way the DHCP server always gives the same address for a given robot (based on its MAC address). Therefore, it is only required here to enable the DHCP server (`dhcp4` and `dhcp6`):

   ```
   network:
     version: 2
     rendered: networkd
     ethernets:
       eth0:
         dhcp4: yes
         dhcp6: yes
         optional: true
     wifi:
   ```

[26] Robotis e-manual: https://emanual.robotis.com/docs/en/platform/turtlebot3/

```
    wlan0:
      dhcp4: yes
      dhcp6: yes
      access-points:
        WIFI_SSID:
          password: WIFI_PASSWORD
```

Save the file with *Ctrl+S* and exit with *Ctrl+X*.

Rmk: In case of need to configure a static IP address for a device[27], specify the static IP address of the Raspberry Pi and of its gateway:

```
network:
  version: 2
  rendered: networkd
  ethernets:
    eth0:
      dhcp4: yes
      dhcp6: yes
      optional: true
  wifi:
    wlan0:
      dhcp4: no
      dhcp6: no
      addresses:
        - IP_ROSMASTER/24
      gateway4: IP_GATEWAY
      nameservers:
        addresses: [8.8.8.8, 1.1.1.1]
      access-points:
        WIFI_SSID
          password: WIFI_PASSWORD
```

3. Configure the ROS network. Here the ROS master is installed on a remote Raspberry Pi, then select the ROS Master's IP address (called IP_ROSMASTER above) for both the ROS_MASTER_URI and the ROS_HOSTNAME when editing the ~/.bashrc file:

   ```
   $ sudo nano /media/$USER/writable/home/RPi_USERNAME/.bashrc
   ```

   When the editor is opened, modify the end of the file:

   ```
   IP_ADDRESS_OF_MASTER_RPi=IP_ROSMASTER
   export ROS_MASTER_URI=http://$IP_ADDRESS_OF_MASTER_RPi:11311
   export ROS_HOSTNAME=$IP_ADDRESS_OF_MASTER_RPi
   ```

   Save the file with *Ctrl+S* and exit with *Ctrl+X*.

   Rmk: the IP address for ROS_HOSTNAME will be changed for each TurtleBot in the next section.

4. Boot up the Raspberry Pi. Use SSH command to connect remotely to the robot:

   ```
   $ ssh RPi_USERNAME@IP_ROSMASTER
   ```

   where RPi_USERNAME is **ubuntu** and the default RPi_PASSWORD is **turtlebot** (if SSH is not installed, see [26]/faq *section 1.1. enable SSH server in Raspberry Pi*).

   Rmk: if you want to check the IP addresses connected to your local hotspot:

   ```
   $ nmap -sP IP_MASK
   ```

[27] How to configure a static IP address on Ubuntu 20.04

where IP_MASK is the mask for your local network. You should see IP_ROSMASTER is connected (as configured right above).

5. Update ubuntu:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

(in case of '*could not get lock /var/lib/dpkg/lock*' error, you probably just have to wait for a few minutes[28]).

4.2.2.2 Final Setup for the ROS Master only

Instructions are adapted from [26]/quick-start, select the right ROS distribution (**Noetic** here) with small changes as below:

7. Boot up the ROS Master's Raspberry Pi (with preliminary setup from section 4.2.2.1).
8. In case of using a monitor+keybord for the ROS Master (see the *quick start guide* in section 4.1) or to debug a robot, install some useful tools and change for an `azerty` keyboard:

```
$ sudo apt install gnome-terminal
$ sudo apt-get install nmap
$ sudo setxkbmap fr
```

9. Install dependent ROS packages (needed by the ROS master):

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
ros-noetic-rosserial-python ros-noetic-rosserial-client \
ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers
```

10. Install TurtleBot3 packages:

```
$ sudo apt install ros-noetic-dynamixel-sdk
$ sudo apt install ros-noetic-turtlebot3-msgs
$ sudo apt install ros-noetic-turtlebot3
```

11. Install RQT/TurtleSim if needed (see section 2.2):

```
$ sudo apt-get install ros-noetic-rqt ros-noetic-rqt-common-plugins
$ sudo apt-get install ros-noetic-turtlesim
```

→ All these steps give the SD card image `setup_ROSmaster.img` (see section 4.2.1).

4.2.2.3 Final Setup for each TurtleBot

Instructions are adapted from [26]/quick-start, select the right ROS distribution (**Noetic** here) with small changes as below:

7. Boot up the TurtleBot's Raspberry Pi (with preliminary setup from section 4.2.2.1).
8. In case of using static IP addresses (and not DHCP), replace the IP address IP_ROSMASTER in the netplan file (see item 2 in section 4.2.2.1) by IP_TURTLEBOTi, with i the TurtleBot number.

[28] Could not get lock error

9. Configure the ROS network. Select here differently the ROS Master's IP address IP_ROSMASTER for the ROS_MASTER_URI and the TurtleBot's IP address IP_TURTLEBOTi for the ROS_HOSTNAME when editing the ~/.bashrc file:

   ```
   $ sudo nano ~/.bashrc
   ```

   When the editor is opened, modify the end of the file:

   ```
   IP_ADDRESS_OF_MASTER_RPi=IP_ROSMASTER
   export ROS_MASTER_URI=http://$IP_ADDRESS_OF_MASTER_RPi:11311
   IP_ADDRESS_OF_TURTLEBOT_RPi=IP_TURTLEBOTi
   export ROS_HOSTNAME=$IP_ADDRESS_OF_TURTLEBOT_RPi
   ```

   In addition, define the type of TurtleBot3 and give a name for the robot (mandatory in a multi-robot framework) at the end of the ~/.bashrc file:

   ```
   export TURTLEBOT3_MODEL=TB3_MODEL
   export ROS_NAMESPACE=TURTLEBOTi
   ```

   where TB3_MODEL is either **burger**, **waffle** or **waffle_pi** and TURTLEBOTi is **TurtleBoti**.

   Save the file with *Ctrl+S* and exit with *Ctrl+X*.

   Don't forget to apply changes with the command:

   ```
   $ source ~/.bashrc
   ```

→ All these steps give the SD card image setup_ROSturtle.img (see section 4.2.1).

4.2.2.4 Clone a Raspberry Pi SD Card and Shrink the Cloned Image

This last stage allows cloning a Raspberry Pi SD card to an image (that can be useful to duplicate a stable install). Instructions are adapted from [29] and [30] to clone/shrink an SD card.

1. Insert the SD card and enter the next command to list all the filesystems present on your system:

   ```
   $ sudo fdisk -l
   ```

   Try to find out the device name of your SD card. It should be /dev/mmcblk0.

2. Use the dd command to write the image to your hard disk (be careful and double check the parameters before executing the dd command, as entering the wrong parameters can potentially destroy the data on your drives):

   ```
   $ sudo dd if=/dev/mmcblk0 of=~/NAME_IMAGE.img
   ```

   Rmk: The if parameter (input file) specifies the file to clone (here the SD card); the of parameter (output file) specifies the file name to write (here NAME_IMAGE is either setup_ROSmaster or setup_ROSturtle in the home directory ~, see section 4.2.1).

   You will not see any output from the command until after the cloning is complete, and that might take a while, depending on the size of your SD card.

3. *PiShrink* is a script that automatically shrinks a Raspberry Pi image. Download the script and make it executable:

   ```
   $ wget https://raw.githubusercontent.com/Drewsif/PiShrink/master/pishrink.sh
   $ chmod +x ./pishrink.sh
   ```

4. Run the script, followed by the name of the image that you want to shrink:

   ```
   $ sudo ./pishrink.sh ~/NAME_IMAGE.img
   ```

[29] How to clone Raspberry Pi SD card on Windows, Linux, and macOS
[30] PiShrink: make Raspberry Pi images smaller

4.2.3 Open CR Setup (optional, only for newly assembled robot or newly installed ROS version)

Follow instructions on [26]/opencr_setup, select the right ROS distribution (**Noetic** here):

1. Install the required packages on the Raspberry Pi to upload the OpenCR firmware.
2. Download the firmware and loader (select the right Open CR model), extract the file and upload firmware to the OpenCR.
3. Test the OpenCR installation with `PUSH SW 1` and `PUSH SW 2` buttons (see Figure) to see whether the TurtleBot3 robot has been properly assembled.

4.3 Practical Work

4.3.1 Bringup and Basic Operation

Instructions are adapted from [26]/bringup, select the right ROS distribution (**Noetic** here):

1. Install both ROS Master's and TurtleBot's Raspberry Pi, directly from the SD car images (see section 4.2.1) or from the detailed steps (see section 4.2.2).
2. Run `roscore` (in the background, using `&` at the end of the command line) on the ROS Master's Raspberry Pi via SSH:

   ```
   $ ssh RPi_USERNAME@IP_ROSMASTER
   $ roscore &
   ```

3. Bringup basic packages to start TurtleBot3 applications on the TurtleBot's Raspberry Pi:

   ```
   $ ssh RPi_USERNAME@IP_TURTLEBOTi
   $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
   ```

   Rmk: In case of error(s), you might need to reinstall Open CR with the right TurtleBot model (see section 4.2.3).

4. In a new terminal, test a basic operation (see [26]/basic_operation), like teleoperation of the robot with a keyboard from the ROS Master (while `turtlebot3_bringup` is launched in the TurtleBot):

   ```
   $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
   ```

   Follow the instructions.

   Rmk: You should need to define before the type of robot to teleoperate:

   ```
   $ export TURTLEBOT3_MODEL=TB3_MODEL
   ```

   where `TB3_MODEL` is either **burger**, **waffle** or **waffle_pi**.

→ Try to connect to the TurtleBot robot from a remote computer (in the same network), modifying the IP address of the `ROS_MASTER_URI` in its `.bashrc` file (see section 2.4.1.9).
→ Try to understand the ROS architecture of the TurtleBot using either ROS commands or `rqt` (extending the practical work in section 2.4.1).

Rmk: In particular, with the command `rostopic list` you should see that:

- The topic to publish the robot's velocity is `/cmd_vel` (or `/TURTLEBOTi/cmd_vel` if a robot name has been defined in section 4.2.2.3) instead of `/turtle1/cmd_vel` in `turtlesim` (section 2.4.1.4);
- The one to subscribe its pose is `/odom` (or `/TURTLEBOTi/odom` resp.) instead of `/turtle1/pose` in `turtlesim`, because the pose of the robot is estimated by odometry (see [2] for further details on odometry).

→ Try to make a real TurtleBot draw a square using either ROS commands (see section 2.4.1) or a new package in Python (section 2.4.3).
→ Try to print the position `(x,y)` of the square corners (using odometry feedback).

Rmk: If you create a ROS package on a remote computer, you have to install the `turtlebot3` packages (because your node will need to know the message types for instance):
`$ sudo apt install ros-noetic-turtlebot3` (see section 4.2.2)

4.3.2 Control a TurtleBot from Matlab/Simulink with Odometry Feedback

The following instructions are adapted from section 2.4.2.

4.3.2.1 Matlab

1. Connect to the ROS Master (assuming `roscore` is running on the ROS Master and `turtlebot3_bringup` launched on the TurtleBot, see section 4.3.1):

   ```
   rosinit('IP_ROSMASTER')
   ```

   where `IP_ROSMASTER` is the ROS Master's IP address.

2. To move the TurtleBot3 robot, you can control the motion of the TurtleBot by publishing a message to the `/cmd_vel` topic. The message has to be of type `geometry_msgs/Twist`, which contains data specifying desired linear and angular velocities. The TurtleBot3's movements can be controlled through two different values: the linear velocity along the X-axis controls forward and backward motion and the angular velocity around the Z-axis controls the rotation speed of the robot base.

   Example to move the robot ahead and then rotate:

   ```
   cmdPub = rospublisher('/cmd_vel','geometry_msgs/Twist');
   cmdMsg = rosmessage(cmdPub);
   % translational motion
   cmdMsg.Linear.X = 0.1;        % in meters per second
   cmdMsg.Angular.Z = 0;
   send(cmdPub,cmdMsg)
   pause(4)
   % rotational motion
   cmdMsg.Linear.X = 0;
   cmdMsg.Angular.Z = pi/2;      % in radians
   send(cmdPub,cmdMsg)
   ```

3. The TurtleBot3 also uses the `/odom` topic to publish its current pose (position and orientation). The message has to be of type `nav_msgs/Odometry`, which contains data specifying the position (x, y, z) and orientation (in quaternion form). Since the robot is not equipped with exteroceptive sensors, the pose will be relative to the pose that the robot had when it was first turned on.

   ```
   odomSub = rossubscriber('/odom','nav_msgs/Odometry');
   odomMsg = receive(odomSub,3);  % timeout period, in seconds
   pose = odomMsg.Pose.Pose;
   t = odomMsg.Header.Stamp.Sec + 1e-9*odomMsg.Header.Stamp.Nsec; % in [s]
   x = pose.Pose.Pose.Position.X
   y = pose.Pose.Pose.Position.Y
   quaternion = [pose.Pose.Pose.Orientation.W, pose.Pose.Pose.Orientation.X, ...
           pose.Pose.Pose.Orientation.Y, pose.Pose.Pose.Orientation.Z];
   eularAngles = quat2eul(quaternion,'XYZ');
   theta = eularAngles(3)
   ```

   Rmk: Time t is the time elapsed since the `turtlebot3_bringup` was launched on the TurtleBot. Similarly, the origin for X, Y and θ depends on the pose of the robot at launch.

4. At the end of the script, eventually quit ROS:

   ```
   rosshutdown
   ```

4.3.2.2 Simulink

- A block diagram of a Simulink Publisher/Subscriber example is provided in Fig. 4.1.

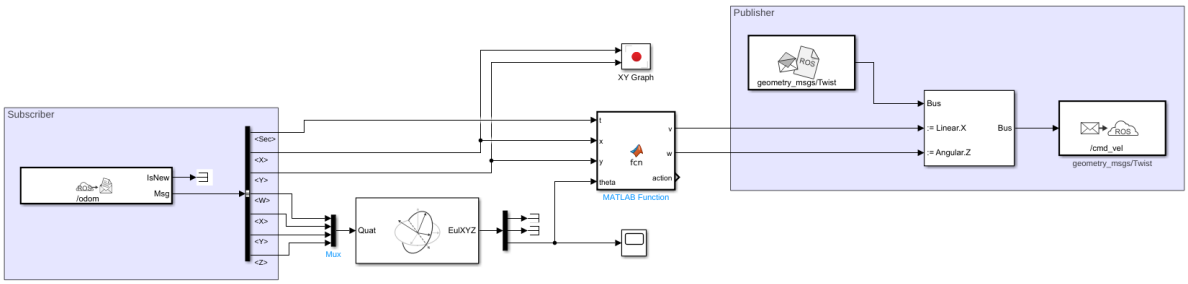


Figure 4.1: Publisher/Subscriber example in Simulink.

Rmk: The quaternion for the TurtleBot's orientation can be transformed in Euler angles thanks to a *Coordinate Transformation Conversion* block (from Robotics System toolbox[31]). The time stamp can also be obtained (with `Header.Stamp.Sec` and `Header.Stamp.Nsec` as in item 3 of the section above).

→ Try to make a real TurtleBot draw a square using either Matlab or Simulink (extending the practical work in section 2.4.2).
→ Try to print the pose `(x,y,theta)` of the square corners (using odometry).
→ Try to control two TurtleBots, where the first one is controlled by `turtlebot3_teleop_key` while the second one follows it with odometry feedback.

4.3.3 *Advanced Feedback Control

To control a TurtleBot robot with advanced feedback control strategies, the reader can refer to [2]: an *introduction to mobile robotics*, which gives the model of a nonholonomic wheeled mobile robot (such as TurtleBot robots) and different path following and trajectory tracking techniques.

[31] Mathworks support: Robotics System toolbox

Chapter 5

Motion Capture Data Streaming to the ROS Master

5.1 Quick Start Guide

1. **Turn on the devices:**
   (a) Turn on the ROS Master and TurtleBot's Raspberry Pis (with IP addresses `IP_ROSMASTER` and `IP_TURTLEBOTi` respectively), with `i` the TurtleBot number.
   (b) Turn on the Optitrack's workstation and open the Motive software. Select the rigid body to track (`TURTLEBOTi` here).
   Rmk: Refer to section 3.2 for the variable values.

2. **Connect to the devices (using a remote computer):**
   (a) Connect the remote computer to the WiFi hotspot, using `WIFI_SSID` and `WIFI_PASSWORD`.
   (b) From the remote computer:
      i. Open a new terminal `[TERMNAL 1]`, connect to the ROS Master's Raspberry Pi via SSH, run `roscore` and launch `vrpn_client_ros` (in background, using `$` at the end of the command line):
      ```
      $ ssh RPi_USERNAME@IP_ROSMASTER
      $ roscore &
      $ roslaunch vrpn_client_ros vrpn_turtleboti.launch server:=IP_MOCAP &
      ```
      Rmk: Eventually verify the OptiTrack data streaming for `TURTLEBOTi` is active:
      ```
      $ rostopic echo /vrpn_client_node/TURTLEBOTi/pose
      ```
      ii. Open a new terminal `[TERMNAL 2]`, connect to the TurtleBot's Raspberry Pi via SSH and launch `turtlebot3_bringup`:
      ```
      $ ssh RPi_USERNAME@IP_TURTLEBOTi
      $ roslaunch turtlebot3_bringup turtlebot3_robot.launch
      ```

3. **Try to move the robot through Matlab/Simulink** (as in section 4.3.2 but subscribing to the topic `/vrpn_client_node/TURTLEBOTi/pose` instead of `/odom`, see section 5.3.2).

4. **Turn off the devices:**
   (a) Quit the running nodes using `Ctrl+C` (in both `[TERMNAL 1]` and `[TERMNAL 2]`).
   (b) Shutdown the Raspberry Pis, first in `[TERMNAL 2]` to turn off the TurtleBot's Raspberry Pi and then in `[TERMNAL 1]` for the ROS Master's one:
   ```
   $ sudo halt
   ```
   (c) Turn off the OptiTrack workstation with Motive software.

5.2 *Install and Calibration

5.2.1 Motive Software

Follow instructions from OptiTrack Support[32]:

1. Install the Motive software and activate its license (only once).

   Rmk: The Hardware license USB key has to be inserted into the host computer. Motive 2.3.0 is installed in the motion capture room.

   The different panels of Motive layout are briefly explained in[32].

2. Camera settings and calibration[33] are not mandatory each time but only when needed (hardware setup change, light or temperature change, a long period without calibrating, or when a rigid body object is blinking...).

   Rmk: For best tracking results, prepare and clean up the capture environment, remove unnecessary obstacles, cover open windows and minimize incoming sunlight, and remove or cover items with reflective surfaces or illuminating features.

   For calibration, use the `Calibration Layout` mode on the top-right corner of Motive:

   - Masking: Apply mask if needed in the `Camera Calibration Pane` or the `Camera Preview Pane` (extraneous reflections or unnecessary markers have to be removed from the capture volume before calibration), see the red masks for cameras 3 and 4 in Fig. 5.1c for instance.
   - Wanding: Click `Start Wanding` and move the calibration wand doing waves in front of the cameras repeatedly, allowing all cameras to see the markers until each camera captures enough sample frames to compute their respective position and orientation in the 3D space. The wand and camera results are depicted in Fig. 5.1a and 5.1d.
   - Calibration results: Press the `Calculate` button and `Apply` if the calibration results reach the desired performance.
   - Ground plane and origin: Place the calibration square (see Fig. 5.1b) at the desired origin in the motion capture room and click the `Set Ground Plane` button.

3. Place the retro-reflective markers onto your TurtleBots (avoiding having similar configurations for different robots), then create a rigid body object[34]. Use the `Create Layout` mode on the top-right corner of Motive:

   - In the `Assets Pane`, add a `New Rigid Body`, select the markers in the `Perspective View` (five markers here), then in the `Builder Pane` choose a name (i.e. `TURTLEBOTi` hereafter, with `i` the TurtleBot number) and `Create` the asset, see Fig. 5.1e and Fig. 5.1b. Eventually, change the pivot point if the center of gravity of the robot is different from the pivot point of the markers.

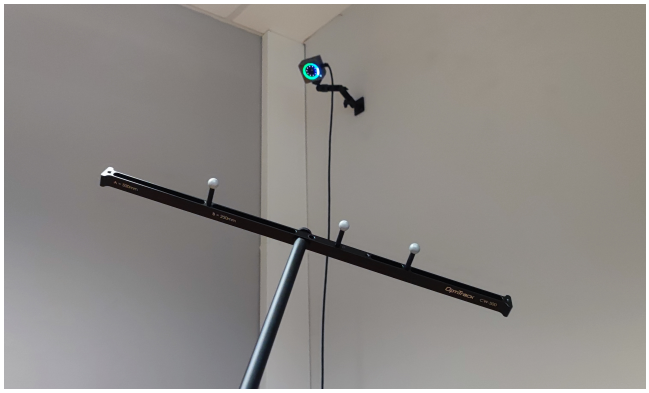
4. Use the `Capture Layout` mode on the top-right corner of Motive to visualize your robot, see Fig. 5.1f.

5. You can either export data (for post-processing for instance) or stream data onto external applications in real-time, with different protocols[35]. Here we will use VRPN to communicate with ROS and Matlab/Simulink, see next section.

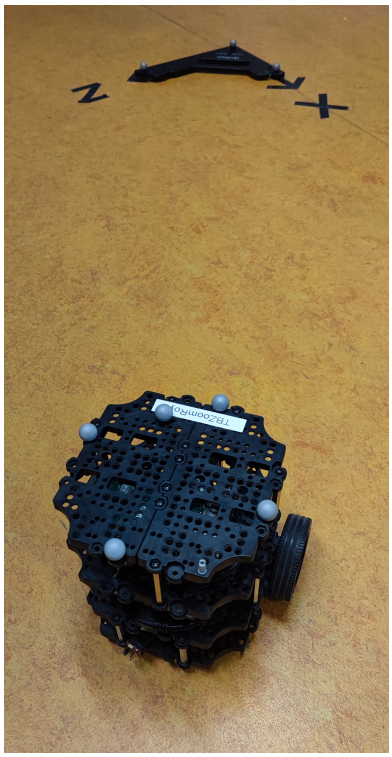
5.2.2 VRPN

VRPN[36] is the *virtual reality peripheral network*. The streaming engine is implemented in the Motive software to stream VRPN data over the network.

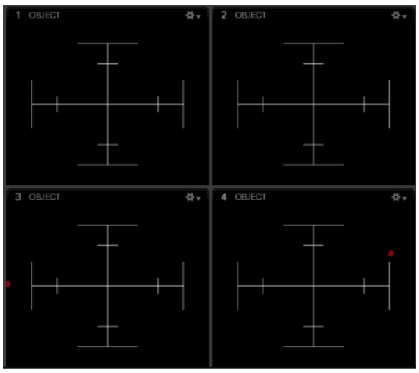
[32] OptiTrack support: Quick start guide: getting started or the more detailed Hardware setup and Motive documentation
[33] OptiTrack support: Calibration
[34] OptiTrack support: Markers and Assets
[35] OptiTrack and ROS
[36] *vrpn-client-ros* ROS package



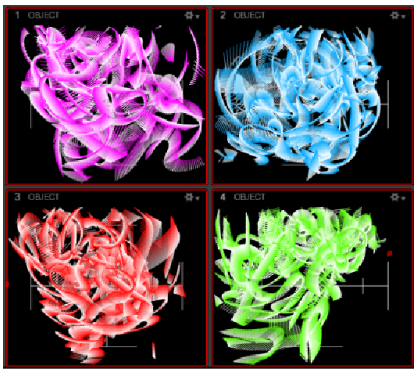
(a) Calibration wand with a camera during wanding.



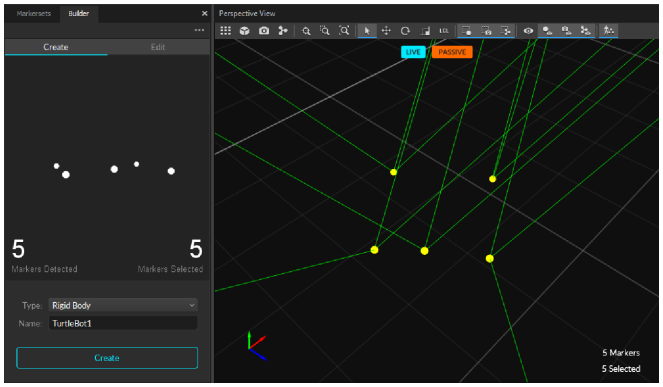
(b) TurtleBot with its markers + origin of the OptiTrack's space with the calibration square.



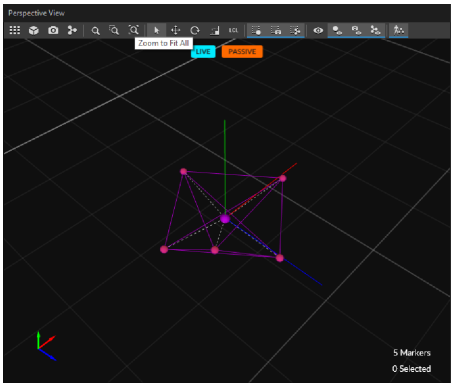
(c) Masking in the camera preview (red masks).



(d) Wanding results in the camera preview.



(e) Create a new rigid body.



(f) Capture the rigid body asset corresponding to the TurtleBot.

Figure 5.1: Motive's interface.

Rmk: Firewall or anti-virus software can block network traffic, so it is important to make sure these applications are disabled or configured to allow access to both server (Motive) and client applications (ROS Master).

1. Only 6-DoF rigid body data can be streamed via VRPN. In Motive, select the rigid body objects (TurtleBots) you want to stream, they have to be in the `Assets` in the `Project Pane` (checkbox checked) to be streamed.

   Rmk: The `Name` for the objects to be tracked must not have any white spaces. To rename an object click with the right mouse button on the object and select `Rename Asset`. Each robot name is denoted `TURTLEBOTi` (see section 5.2.1).

2. Activate the VRPN streaming engine in Motive, by opening the `Data Streaming Pane` in Motive and clicking on `View->Data Streaming`. In the `Optitrack Streaming Engine` group, enable the `Broadcast Frame Data` and select the `Local Interface` as `IP_MOCAP`, see Fig. 5.2.

   Also, be aware to select the `Up Axis` as `Z Up` (default is `Y Up` that is not compatible with the TurtleBot model).

   Rmk: By selecting the `Z Up` mode, you can directly use the information `pose.position.x` and `pose.position.y` from the topic `/vrpn_client_node/TURTLEBOTi/pose` to measure the position of the TurtleBot robot (otherwise, by using the `Y Up` mode, you should use `z` and `x` instead), see section 5.3.1. The orientation is not concerned because of quaternion use.

   Check the checkbox `Broadcast VRPN` in the `VRPN Streaming Engine` group to activate the OptiTrack streaming engine and with the following settings the streamed data is configured. If necessary, set the `VRPN Broadcast Port` (default: 3883, make sure you either turn off the Windows Firewall or create outbound rules for the VRPN port).

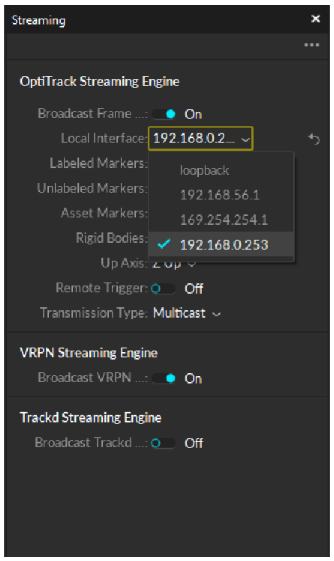


Figure 5.2: Motive's interface to activate and configure the VRPN.

3. Installation on the ROS Master's Raspberry Pi (remember the ROS distribution is **Noetic** here):

   ```
   $ sudo apt-get install ros-noetic-vrpn-client-ros
   ```

4. Configure the `vrpn-client-ros` package to start the VRPN client for ROS, inspire from the basic launch file `/opt/ros/noetic/share/vrpn_client_ros/launch/sample.launch` (see section 2.4.3 to see how works a launch file). Make a copy of the launch file, called `vrpn_turtlebot1.launch` in the example right after, where the pose of one robot `TURTLEBOT1` is streamed, and modify this copied launch file:

```
<launch>
  <arg name="server" default="IP_MOCAP"/>
  <node pkg="vrpn_client_ros" type="vrpn_client_node" name="vrpn_client_node"
      output="screen">
    <rosparam subst_value="true">
      server: $(arg server)
      port: 3883
      frame_id: world
      # Use the VRPN server's time or the client's ROS time:
      use_server_time: false
      # Broadcast transforms of tracker pose on /tf:
      broadcast_tf: true
      # Must either specify refresh frequency > 0.0, or a list of trackers to create:
      refresh_tracker_frequency: 0.0
      trackers:
      - TURTLEBOT1 # First tracker
      #- TURTLEBOT2 # Second tracker (if needed)
      #- ...
      #- TURTLEBOTn # n-th tracker (if needed)
    </rosparam>
  </node>
</launch>
```

where IP_MOCAP specifies the host computer's IP address (see section 3.2).

The `refresh_tracker_frequency` value specifies the frequency for how often the client looks for new objects, this means that new objects that Motive will stream in the future will be recognized automatically (when its value is positive). Comment the `trackers` lines.
If set to `0.0` (default), the `trackers` have to be specified manually, like here for the robot TURTLEBOT1.

The `use_server_time` specifies if publishing the pose using the VRPN server's timestamp, or the local ROS time (when `false` like here).

The `frame_id` specifies the world frame id for the trackers (default "`world`").

Rmk: Further configuration settings can be found on the `vrpn_client_ros` wiki page[36].

→ The launch file (i.e. `vrpn_turtlebot1.launch` or another copy) has to be adapted to the robot(s) to control.

5.3 Practical Work

5.3.1 VRPN Data Streaming

1. Make sure `roscore` is running in the ROS Master's Raspberry Pi. Then, run the `vrpn_client_ros` node[37]:

   ```
   $ roscore &
   $ roslaunch vrpn_client_ros vrpn_turtlebot1.launch server:=IP_MOCAP
   ```

2. OptiTrack's data should be received under topics that look like `/vrpn_client_node/TURTLEBOT1/pose`. Use `rostopic list` to list them and the `echo` command to display the results:

   ```
   $ rostopic list
   $ rostopic echo /vrpn_client_node/TURTLEBOT1/pose
   ```

   which result should be of the form:

   ```
   header:
     seq: 3438
     stamp:
       secs: 1519816203
       nsecs: 344498680
     frame_id: "world"
   pose:
     position:
       x: -1.3609367609
       y: -1.55265676975
       z: 0.809030890465
     orientation:
       x: -0.0133089488372
       y: 0.00888985395432
       z: 0.605775594711
       w: -0.795474588871
   ```

   Rmk: Be careful to select `Z Up` as the `Up Axis` mode in the VRPN data streaming configuration in Motive (see section 5.2.2).

3. The `info` command also gives the message type:

   ```
   $ rostopic info /vrpn_client_node/TURTLEBOT1/pose
   ```

   which is in the form:

   ```
   Type: geometry_msgs/PoseStamped
   Publishers:
    * /vrpn_client_node (http://TURTLEBOT1:39547/)
   Subscribers: None
   ```

   Rmk: If the robot is moving, it is possible to compare OptiTrack's data with the TurtleBot's odometry result (in another terminal):

   ```
   $ rostopic echo /odom
   ```

→ Try to make a real TurtleBot move using a `teleop` and print its position `(x,y)` provided by the OptiTrack motion capture system.

[37] Motion capture setup and wifi communications

5.3.2 Control a TurtleBot from Matlab/Simulink with Motion Capture Feedback

- Instructions can be adapted from section 4.3.2 to collect OptiTrack's data in Matlab/Simulink, by subscribing to the topic `/vrpn_client_node/TURTLEBOT1/pose` instead of `/odom`.

→ Try to make a real TurtleBot draw a square using either Matlab or Simulink (extending the practical work in section 2.4.2 and 4.3.2).
→ Try to print the pose `(x,y,theta)` of the square corners (using the motion capture system).
→ Try to control two TurtleBots, where the first one is controlled by `turtlebot3_teleop_key` while the second one follows it with motion capture feedback.

5.3.3 *Advanced Feedback Control

- As in section 4.3.3, to control a TurtleBot robot with advanced feedback control strategies, the reader can refer to [2]: an *introduction to mobile robotics*, which gives the model of a nonholonomic wheeled mobile robot (such as TurtleBot robots) and different path following and trajectory tracking techniques.
- The OptiTrack feedback can also be useful to validate odometry or vision-based strategies with accurate ground truth.

Bibliography

[1] A. Hably, R. Tang, J. Dumon, and A. Carriquiry, "Robotme: A drone platform for control education," in *45th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2019.

[2] S. Durand, "Introduction on mobile robotics." Lecture notes, INSA Strasbourg.