Foundations of robotics – **Guide to practical sessions**

Fabio Amadio

March 23 and 30, 2026 – *École des Mines de Nancy*

Contents

1 System Setup and Development Environment

Follow the steps described below to properly setup your laptop before the start of the practical sessions and to download all the necessary software and materials required for the exercises.

For any questions or issues, feel free to contact the teaching staff.

1.1 Install Ubuntu

Ubuntu is the most widely used Linux distribution and provides extensive community and documentation support. Ubuntu – and Linux-based operating systems in general – are the standard platforms used in robotics research and development. Most robotics frameworks, tools, and middleware are designed primarily for Linux environments. Students interested in further exploring robotics are strongly encouraged to become familiar with Linux, as proficiency with this operating system is fundamental.

Dual-boot configuration

Install the latest Long Term Support (LTS) release, Ubuntu 24.04.3, to ensure stability and long-term compatibility. Refer to the official installation guide for instructions on setting up Ubuntu alongside your existing operating system (dual-boot configuration).

IMPORTANT

During the installation, selecting the option to erase the disk will permanently delete all existing data. Students are strongly advised to back up their data before proceeding.

WSL: Windows Subsystem for Linux

As an alternative, it is possible to use the Windows Subsystem for Linux (WSL) WSL lets developers install a Linux distribution (such as Ubuntu, OpenSUSE, Kali, Debian, Arch Linux, etc) and use Linux applications, utilities, and Bash command-line tools directly on Windows, unmodified, without the overhead of a traditional virtual machine or dualboot setup. Refer to the official installation guide for instructions on how to setup WSL on your Windows machine (Windows 11 recommended).

1.2 Install Docker

Docker is a containerization platform that allows applications and their dependencies to be packaged into lightweight, portable containers. This ensures reproducible software environments across different systems and simplifies deployment, testing, and development workflows.

Docker is available in two main forms: **Docker Engine**, which provides the core container runtime and is typically used directly on Linux systems, and **Docker Desktop**, which bundles Docker Engine with additional tools and a graphical interface, mainly targeting Windows and macOS.

Docker Engine (Native Ubuntu)

If you are working on native Ubuntu, you can install Docker Engine by following the official installation guide. Specifically, follow the **Install using the apt repository** section, together with the post-installation steps (to allow to to run Docker without root).

Docker Desktop (Windows + WSL)

If you are working on WSL, you need to install **Docker Desktop** on your Windows system. You can follow the official installation guide – simply download and run the provided `.exe` installer.

IMPORTANT

Make sure to enable WSL integration. During the installer, check the following option:
"**Use WSL 2 instead of Hyper-V (recommended)**".

After completing the installation, Docker can be used directly from the WSL terminal as if you were working on a native Linux system – **provided that Docker Desktop is running in the background while you are using WSL**.

Verify Docker installation

Open a terminal and run:

```
docker --version
```

If Docker is correctly installed, this command prints the installed Docker version.

To verify that Docker can run containers, execute:

```
docker run hello-world
```

A successful installation is confirmed if Docker prints a welcome message and exits without errors.

1.3 Code editor

On Native Ubuntu

Feel free to use your preferred code editor; if you do not have one, you can install **Visual Studio Code** (VS Code) via the Snap package manager: `sudo snap install code --classic`.

On Windows + WSL

When working with WSL, it is strongly recommended to use **Visual Studio Code** (VS Code) together with the *WSL extension*. VS Code runs on Windows while providing a seamless development experience inside the Linux environment.

Install VS Code on Windows from the official website, then install the WSL extension from the VS Code extensions marketplace. From within WSL, navigate to your project directory and launch VS Code using: `code .`

This setup allows editing files stored in the WSL filesystem while using Linux tools, compilers, and Docker directly from the integrated terminal.

1.4 Download the Lab Materials

Source code

All the source code used in the following exercises is hosted on **GitHub** in the course repository, which can be downloaded locally by cloning it with `git`.

If `git` is not already installed, you can install it by running:

```
sudo apt update
sudo apt install git
```

Once `git` is available, clone the course repository from GitHub:

```
git clone https://github.com/fabio-amadio/foundations-of-robotics-lab-material.git
```

The repository `foundations-of-robotics-lab-material` will be cloned into the current directory. It contains all the source code needed for carrying out the practical sessions, separated in three directories, `intro_ros2`, `mobile_robotics`, `robotic_manipulation`, one for each set of exercises.

All activities are designed to run inside pre-configured Docker containers to ensure a consistent and reproducible environment.

Docker images

A Docker image is a pre-built package that contains an application together with all the software it needs to run. You can think of it as a blueprint used to create Docker containers, allowing the same program to run in the same way on different computers.

Images are built from a `Dockerfile`. A `Dockerfile` is a simple text file that contains a list of instructions describing, step by step, how to build a Docker image. In practice, a `Dockerfile` acts as a recipe that tells Docker how to create an image in a reproducible and automated way.

The Docker images you will use in these exercises have already been built and hosted on GitHub, so you do not need to build them yourself (which can take some time). You can download them using:

```
docker pull ghcr.io/fabio-amadio/intro-ros2:2026  # Intro to ROS 2 tutorial
```

```
docker pull ghcr.io/fabio-amadio/turtlebot3-sim:2026  # Mobile Robotics tutorial
```

```
docker pull ghcr.io/fabio-amadio/panda-ros2:2026  # Robotic Manipulation tutorial
```

1.5 Docker-Based Development Workflow

Throughout the following exercises, you will work inside Docker containers that **mount**[1] the local directories containing the ROS 2 packages you will be working with.

Therefore, you can edit the files directly on your host machine: since the directories are mounted into the container, any changes you make locally are immediately reflected inside the container.

Terminator

In the proposed exercises, you will use a utility script, `run_docker.sh`, which starts the container and opens an interactive session using a `terminator` terminal.

`terminator` is an advanced terminal emulator that will replace the standard Ubuntu terminal for this course. It allows multiple terminal sessions to be managed within a single window, which is particularly useful when running several processes simultaneously.

A key feature of Terminator is the ability to split the terminal window. This can be done either by right-clicking within a terminal pane and selecting `Split Horizontally` (key binding: `Ctrl+Shift+O`) or `Split Vertically` (key binding: `Ctrl+Shift+E`). This layout makes it easy to run commands in parallel, as well as to monitor and interact with multiple processes simultaneously.

Try to run a container

To verify that everything is working correctly, navigate into each exercise repository:

`. intro_ros2`

`. mobile_robotics`

`. robotic_manipulation`

and try running the container using the provided utility script:

```
bash run_docker.sh
```

[1] **Mounting a volume** in Docker means making a directory from the host system accessible inside a container, allowing data sharing and persistence by synchronizing a host path with a container path.

2 ROS 2 fundamentals

ROS 2 (Robot Operating System 2) is an open-source middleware framework designed to support the development of modular, scalable, and distributed robotic systems. It provides a standardized software infrastructure that simplifies communication between different components of a robot, enabling developers to focus on algorithm design rather than low-level system integration. The primary purpose of ROS 2 is to facilitate the development, testing, and deployment of robotics applications across a wide range of platforms, from simulation environments to real robotic hardware.

ROS 2 is released in different distributions, each with a defined support cycle. In this course, we will use **ROS 2 Humble Hawksbill**, which, although not the most recent release, is a Long Term Support (LTS) distribution and remains widely used and actively supported.

NOTE

Together, we will explore only a subset of the core concepts, specifically those strictly required to carry out the exercises. If you wish to gain a more comprehensive understanding of how ROS 2 works, we recommend studying the **"Concepts"** section of the official documentation and exploring the **"Tutorials"** and **"How-to Guides"**. Moreover, minimal Python and C++ examples can be found in the following GitHub repository: `https://github.com/ros2/examples`

2.1 Nodes

A **node** is the basic unit of computation in ROS 2 and represents a participant in the ROS graph. Each node is designed to perform a single logical task and can communicate with other nodes running in the same process, in different processes, or on different machines.

Nodes exchange data through **topics**, perform synchronous interactions via **services**, and handle long-running tasks through **actions** (services and actions will not be covered here). They can also expose **parameters** to configure their behavior at runtime.

Communication between nodes is established automatically through a **distributed discovery mechanism** provided by the ROS 2 middleware. Nodes announce their presence on the network within the same `ROS_DOMAIN_ID` (an environment variable used to logically isolate ROS 2 communication domains), dynamically discover peers, and notify others when they join or leave the system.

2.2 Topics

Topics are used in ROS 2 for continuous data streams such as sensor measurements and robot state. ROS 2 adopts a **publish/subscribe** communication model, where data producers (**publishers**) and consumers (**subscribers**) exchange messages via named topics. Multiple publishers and subscribers can communicate over the same topic, and published data is delivered to all subscribers.

The system is **anonymous**, meaning subscribers do not need to know which publisher produced the data, enabling modularity and easy component replacement. It is also **strongly typed**, as message field types are strictly defined and enforced, ensuring type safety and consistency across the system.

ROS Messages

ROS 2 nodes communicate using well-defined interfaces described with an Interface Definition Language (IDL), which allows ROS tools to automatically generate message-related code in multiple programming languages. Among the available interface types, **messages** are **used for one-way data exchange**, typically **over topics**, where no response is expected.

Many commonly used message types are already defined in ROS 2 and can be used directly by users. The most widely used message packages include:

- `std_msgs` for basic data types such as integers, floats, and strings;
- `geometry_msgs` for geometric primitives such as points, vectors, poses, and transforms;

- `sensor_msgs` for sensor-related data such as images, point clouds, laser scans, and IMU measurements;
- `navigation_msgs` for navigation-related data, such as paths, maps, and localization information.

Messages are defined in plain-text `.msg` files as a list of fields, each specified by a data type and a name. Users can also define custom message types to meet the specific requirements of their applications (For further details, refer to the tutorial **"Implementing custom interfaces"**).

2.3 Parameters

In ROS 2, **parameters** are associated with individual nodes and are used to configure node behavior at startup and during runtime without modifying the source code. Each parameter consists of a key, a value, and a descriptor. The key is a string representing the parameter name, and the value can be one of the following types: `bool`, `int64`, `float64`, `string`, `byte[]`, `bool[]`, `int64[]`, `float64[]`, or `string[]`. By default, descriptors are empty, but they can include parameter descriptions, valid value ranges, type information, and additional constraints.

2.4 Launch

ROS 2 systems often consist of multiple nodes running across different processes or machines, making manual startup impractical. The **launch** system automates the execution and configuration of multiple nodes using a single command. System behavior is defined in a launch file, which specifies which nodes to run, their parameters, arguments, and execution context, and can be written in XML, YAML, or Python.

2.5 Transformations and tf-frames

In ROS 2, transforms are used to describe the spatial relationships between different coordinate frames in a robotic system. You can think of TFs as coordinate frames attached to key elements of the robot (such as the base, sensors, and joints) as well as to the environment (most commonly the fixed frames `odom` and `map`), as depicted in Fig. 1 Each frame has a pose (position and orientation) defined relative to another frame.

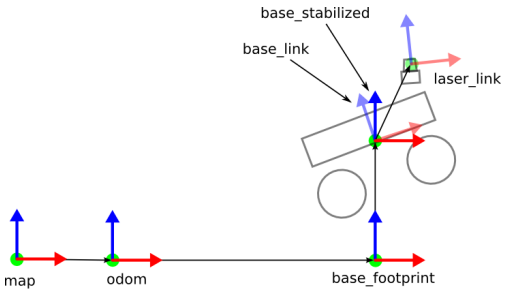


Figure 1: Example *TF tree* (from wiki.ros.org).

All frames are connected in a **tree structure** called the *TF tree*, which must satisfy the following rules:

- Each edge represents a rigid transform between two frames
- The tree contains no cycles
- Every frame has exactly one parent, except for the root

Transforms are published by **nodes** that estimate or know relative poses, and they can be divided into two main categories:

- **Static** transforms are published once and never change over time (e.g., to describe sensor mounting)
- **Dynamic** transforms are published continuously and change over time (e.g., for robot motion)

The main component for handling transforms in ROS 2 is the `tf2` library. It allows positions and orientations to be transformed from one frame to another. Through the `tf2` API – specifically `tf2_ros::Buffer` and `tf2_ros::TransformListener` – any node can answer queries of the form:

> "Where is frame `B` with respect to frame `A` at time t?"

For a more complete introduction to `tf2`, please refer to the official ROS 2 tutorial: **introducing `tf2`**.

2.6 ROS 2 filesystem and build workflow

ROS 2 projects are typically organized in a **workspace**, a directory that contains one or more **packages** (your reusable software units). The standard toolchain relies on `colcon` to build the workspace and on `ament` as the underlying build system.

Workspace layout

A ROS 2 workspace is usually named `ros2_ws` (or similar) and contains at least a `src/` folder:

```
ros2_ws/
|-- src/        # your ROS 2 packages live here
|-- build/      # build artifacts (generated by colcon)
|-- install/    # install space + environment setup scripts (generated by colcon)
|-- log/        # build logs (generated by colcon)
```

Packages

A ROS 2 package is a self-contained unit (nodes, libraries, launch files, interfaces, config, ...). You can create a package using either CMake or Python. Refer to the official tutorial for further details on package structure and creation.

Build the workspace

From the workspace root:

```
cd ~/ros2_ws
colcon build  # build all the packages inside 'src'
```

Useful options (especially for development):

```
colcon build --packages-select <pkg_name>  # build only the selected package
```

Source the environment

ROS 2 uses shell setup scripts to expose built packages (executables, Python modules, message types, launch files, ...) to your terminal session.

Source the ROS 2 distribution (e.g., Humble):

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

Tip: This instruction is usually added to `~/.bashrc` so that it is executed automatically in every new terminal.

Source your workspace overlay after building:

```
source ~/ros2_ws/install/setup.bash
```

Tip: You must re-source `install/setup.bash` in every new terminal (or add it to `/.bashrc` if you are always working in the same workspace). You must also re-source the workspace whenever you recompile it after adding new nodes, launch files, or other resources.

Run what you built

After sourcing:

```
ros2 pkg list  # show the list of all installed packages
ros2 run <pkg_name> <executable_name>  # run a single node
ros2 launch <pkg_name> <launch_file>  # run one or multiple nodes using launch
```

NOTE

In all subsequent exercises, you will work inside Docker containers where the workspace and required packages have already been setup. You will be asked to modify specific files within these packages in order to complete the exercises.

3 Intro to ROS 2: Write publisher and subscriber nodes

In this exercise, we will use `rclpy`, the Python client library for ROS 2, to create nodes, timers, publishers, and subscribers, as well as to declare and load parameters. Finally, you will write your first launch file to run multiple nodes using a single command.

From the `intro_ros2` repository, open a terminal inside the container using the script:

```
bash run_docker.sh
```

This script runs the container in an interactive session within a `terminator` terminal, while mounting the `exercise_pub_sub` folder – containing the ROS 2 package you will work with – into the pre-existing workspace `ros2_ws` inside the container.

Tip: Remember that you can open multiple terminals by right-clicking inside the `terminator` window and selecting `Split Horizontally` or `Split Vertically`.

Package structure

ROS 2 Python packages follow the standard Python package structure (modules, `setup.py`, package directory, etc.), with a few ROS-specific additions, like ROS-specific metadata (package.xml), and optional ROS resources (launch files, message definitions, etc.).

```
exercise_pub_sub/
|-- exercise_pub_sub/
|   |-- __init__.py
|   |-- pub_node.py  # publisher node to complete
|   |-- sub_node.py  # subscriber node to complete
|-- launch/
|   |-- pub_sub.launch.py  # launch file to complete
|-- LICENSE
|-- package.xml
|-- resource/
|   |-- exercise_pub_sub
|-- setup.cfg
|-- setup.py
|-- test/
   |-- test_copyright.py
   |-- test_flake8.py
   |-- test_pep257.py
```

Inside `exercise_pub_sub` you can find the executables `pub_node.py` and `sub_node.py`.

The `setup.py` file tells ROS 2 and Python how the package is structured, what files it contains, and which executables it provides. The `entry_points` section defines the executables (nodes) that can be run from the command line. Each entry maps a command name to a Python function:

```
   entry_points={
      'console_scripts': [
         'pub_node = exercise_pub_sub.pub_node:main',
         'sub_node = exercise_pub_sub.sub_node:main'
      ],
   },
```

Thanks to these entry points, you can run the nodes using `ros2 run`. Without defining entry points, ROS 2 would not know which Python scripts correspond to runnable nodes.

Node structure

Let us now examine the executable file `pub_node.py`. The file defines a class called `MinimalPublisher`, which inherits from `Node`, as well as a `main` function.

The `main` function is the entry point of the node and is executed when the node is run from the command line, as specified in the package's `setup.py` file.

Let us now focus on the core calls inside the `main` function:

```
rclpy.init(args=args)
minimal_publisher = MinimalPublisher()
rclpy.spin(minimal_publisher)
```

- `rclpy.init(args=args)`: Initializes the ROS 2 client library. This step sets up communication with the ROS 2 middleware and must be called before creating any ROS 2 nodes.
- `minimal_publisher = MinimalPublisher()`: Creates an instance of the ROS 2 node defined by the `MinimalPublisher` class. From this point on, the node exists within the ROS 2 system and can create publishers, subscribers, timers, and other ROS 2 entities.
- `rclpy.spin(minimal_publisher)`: Keeps the node running and allows it to process incoming events, such as timer callbacks or message handling. This call blocks execution until the node is shut down.

Spinning is a fundamental concept in ROS 2. While the node is spinning, it continuously checks for incoming messages, timer events, and other callbacks, and executes the corresponding callback functions. Without spinning, callbacks are never processed, even if messages are being published on the network.

If the call to `rclpy.spin()` is omitted, the program will typically terminate immediately after creating the node. As a result, the node will not be able to publish or receive messages, making it effectively inactive.

3.1 `TODO`: Write the publisher node

Complete `pub_node` by publishing a `std_msgs/String` message on the `chatter` topic every second. The message should contain a counter that increases with each published message. Finally, print on terminal the content of every published message.

NOTE

Remember that you can edit the files directly on your host machine: since the package directory is mounted into the container, any changes you make locally are immediately reflected inside the container as well.

How to define a publisher

In the import section:

```
from std_msgs.msg import String
```

In the node definition:

```
self.publisher = self.create_publisher(String, 'topic_name', 10)
```

NOTE

When creating a publisher or subscriber, the last argument (e.g., `10`) specifies the queue size, that is, how many messages can be buffered if they cannot be processed immediately. Using `10` is a reasonable default for most simple applications.

Once the publisher has been defined, you can publish a message in this way:

```
msg = String()  # create a message of type String
msg.data = "your-string"  # fill the data with a string
self.publisher.publish(msg)  # publish the message!
```

How to print log messages

`rclpy` logging provides structured, severity-based output that integrates with the ROS 2 ecosystem, enabling time-stamped, node-aware, and filterable messages. Compared to `print`, it is better suited for debugging, monitoring, and maintaining multi-node ROS 2 applications. Here some example to print log messages from a ROS 2 node.

```
self.get_logger().info('This is a log message at INFO severity')
self.get_logger().warning('This is a log message at WARN severity')
self.get_logger().error('This is a log message at ERROR severity')
```

How to define a timer

In the node definition:

```
timer_period = 0.5  # seconds
self.timer = self.create_timer(timer_period, self.timer_callback)
```

Then it is necessary to define the *callback* method (e.g., the function that is called at each timer period)

```
def timer_callback(self):
   self.get_logger().info('Timer callback')
```

Try it out

Once you are done, compile and source the workspace. From the workspace root folder (`ros2_ws`) run:

```
colcon build
source install/setup.bash
```

Then, run the `pub_node` node:

```
ros2 run exercise_pub_sub pub_node
```

You should see the printed messages running on the terminal. Similarly to

```
[INFO] [1766427814.985814700] [minimal_publisher]: Publishing: "0"
[INFO] [1766427815.970615753] [minimal_publisher]: Publishing: "1"
```

Now, open a second terminal inside the container. You can inspect the list of available topics by running

```
ros2 topic list
```

getting this output

```
/chatter
/parameter_events
/rosout
```

You can also read the messages send over the topic `chatter` directly from the command line by running

```
ros2 topic echo /chatter
```

getting an output similar to this

```
data: 'Counter: 12'
---
data: 'Counter: 13'
---
data: 'Counter: 14'
```

It is also possible to measure the frequency at which messages are published by running

```
ros2 topic hz /chatter
```

getting an output similar to this

```
average rate: 1.000
  min: 1.000s max: 1.000s std dev: 0.00016s window: 2
```

3.2 TODO: Write the subscriber node

Complete `sub_node` by subscribing to the `chatter` topic and printing the content of each message received.

How to create a topic subscription

In the node definition:

```
self.subscription = self.create_subscription(
    String,
    'topic-name',
    self.listener_callback,
    10)
```

Don't forget to define the callback method that is called when a message is received:

```
def listener_callback(self, msg):
    # Process the msg
```

After you finished writing the `sub_node` node (don't forget to re-build the workspace), you can run it by:

```
ros2 run exercise_pub_sub sub_node
```

If the `pub_node` is running on another terminal, you should see `sub_node` printing messages like the ones below at each message received:

```
[INFO] [1767094241.838977321] [minimal_subscriber]: I heard: "Counter: 12"
[INFO] [1767094241.838977321] [minimal_subscriber]: I heard: "Counter: 13"
```

Tip: The `rqt_graph` provides a GUI plugin for visualizing the ROS computation graph (e.g., the network of ROS 2 elements processing data together at the same time). You can open it with the following command (on another terminal):

```
ros2 run rqt_graph rqt_graph
```

In the case of this simple publisher-subscriber example you should see a scheme similar to Fig. 2.

Figure 2: Example `rqt_graph`.

3.3 TODO: Declare parameters

In its current state, the node behavior cannot be customized without modifying the source code, as all values are effectively "hard-coded". By introducing parameters, users could configure aspects such as the publishing period and the topic name at runtime, without changing the code.

Now, try to add parameters to the two previous nodes. In particular:

- in the `pub_node`, declare and load parameters `topic_name` and `timer_period`.
- in the `sub_node`, declare and load parameter `topic_name`.

How to declare and load parameters inside a node

In the node definition:

```
self.declare_parameter("param1_name", "default-value")  # type: str
self.declare_parameter("param2_name", 123.4)  # type: float
param1_value = self.get_parameter("param1_name").value
param2_value = self.get_parameter("param2_name").value
```

Try it out

You can pass parameter values to the `ros2 run` command using the `-ros-args -p` option. For example:

```
ros2 run exercise_pub_sub pub_node --ros-args -p timer_period:=0.1 -p topic_name:=tmp
```

```
ros2 run exercise_pub_sub sub_node --ros-args -p topic_name:=tmp
```

With these calls you can modify the speed of publication and the topic name without modifying the source code of the `pub_node` and `sub_node`.

3.4 TODO: Write the launch file

Starting each node manually using `ros2 run` does not scale well for more complex applications involving a lot of nodes, each requiring specific parameter configurations.

Launch files offer a convenient and scalable way to start, configure, and manage multiple ROS 2 nodes simultaneously from a single command.

A Python launch file in ROS 2 defines how nodes are started and configured.

It contains a `generate_launch_description()` function which returns a `LaunchDescription` object containing a list of `launch.actions`.

Launch arguments are declared with `DeclareLaunchArgument` and accessed using `LaunchConfiguration`, allowing parameters to be set at launch time. More specifically:

- `DeclareLaunchArgument` declares a launch-time variable and optionally assigns a default value.
- `LaunchConfiguration` retrieves the value of that argument so it can be used elsewhere in the launch file (e.g. passed to node parameters, remappings, conditions).

Nodes are started with the `Node` action, which specifies the executable and its configuration.

Complete the `pub_sub.launch.py`

Check the file `intro_ros2/exercise_pub_sub/launch/pub_sub.launch.py`. It is a minimal launch file that runs the `pub_node` getting the `topic_name` and the the `timer_period` parameters as arguments.

Simply add a `Node` action to run the `sub_node` passing the `topic_name` parameter.

Try it out

Launch files can be executed using the `ros2 launch` command.

In our case, to launch the nodes with the default parameter values, run:

```
ros2 launch exercise_pub_sub pub_sub.launch.py
```

To display the list of available launch arguments, add the `-show-args` flag:

```
ros2 launch exercise_pub_sub pub_sub.launch.py --show-args
```

Finally, you can override launch arguments as follows:

```
ros2 launch exercise_pub_sub pub_sub.launch.py timer_period:=0.1 topic_name:=tmp
```

textbfTip: Remember to run `colcon build` after modifying your launch file!

4 Mobile Robotics: Explore the *TurtleBot3 Burger* Simulator

In this tutorial, we will work with a simulated ***TurtleBot3 Burger*** running in the **Gazebo Classic** simulator.

The *TurtleBot3 Burger* is a small, differential-drive mobile robot widely used for education. It is equipped with wheel encoders that provide odometry information and with a 2D LiDAR used for obstacle detection and mapping.

4.1 Launch the Gazebo simulator

Gazebo simulates the robot kinematics and dynamics, as well as realistic sensor data such as laser scans and odometry. The robot is controlled by sending velocity commands on the `/cmd_vel` topic and perceives the environment through laser scan data published on the `/scan` topic.

From the `mobile_robotics` repository, open a terminal inside the container using the script:

```
bash run_docker.sh
```

This script runs the container in an interactive session within a `terminator` terminal, while mounting the `exercise_navigation` folder into the pre-existing workspace `turtlebot3_ws` inside the container.

Tip: Remember that you can open multiple terminals by right-clicking inside the `terminator` window and selecting `Split Horizontally` or `Split Vertically`.

You can start the simulation by running

```
ros2 launch turtlebot3_gazebo turtlebot3_simple_walls.launch.py
```

With this call you will open Gazebo simulating the mobile robot inside a dummy world containing three simple walls as obstacles (Fig. 3). You can explore the simulation environment. To navigate the scene:

- **Left-click and drag** to pan the camera;
- **Scroll the mouse wheel** (or **right-click and drag**) to zoom in and out;
- **Middle-click and drag** to rotate the camera.

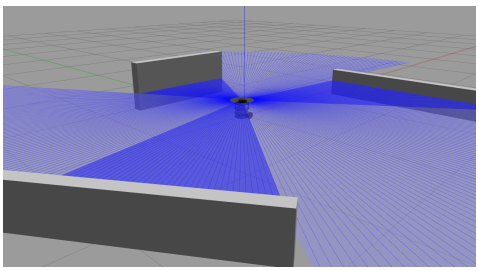


Figure 3: Simulated *TurtleBot3 Burger* (blue rays represent simulated laser scan).

Now, open another terminal and check the available topics by calling `ros2 topic list`.

Try to inspect some of the topics using the commands `ros2 topic echo <topic-name>` and `ros2 topic info <topic-name>`.

The robot can be controlled by sending velocity commands over the `cmd_vel`, while Gazebo provides simulated sensor data over topics: `joint_states` (encoders), `scan` (LiDAR), `imu` (IMU).

Try to move the mobile base by publishing a velocity command

```
ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist "{angular: {z: 1.0}}"
```

The simulated *TurtleBot3 Burger* start following the received command until a new command is received.

4.2 Visualizing the Robot with RViz2

RViz2 is the main visualization tool in ROS 2 and is commonly used to inspect the robot model, coordinate frames, and sensor data.

Open a terminal and start RViz2 with:

```
rviz2
```

Setting the Fixed Frame

Once RViz2 opens up, use the `Fixed Frame` tab (top left) to determine from which frame's perspective you are visualizing the data. Select the `odom` frame. Choosing the correct fixed frame is essential, since RViz2 can only display data that can be transformed into the selected fixed reference frame.

Visualizing the TF Tree

To display the *TF tree*:

- Click `Add`
- Select `TF` from the list of display types
- Press `OK`

This visualization shows the TF tree and the relationships between all coordinate frames.

Tip: Enabling the `Show Names` option also displays the names of the coordinate frames. From the `Frames` tab, you can enable or disable the visualization of individual frames.

Visualizing the Robot Model

To visualize the robot in RViz2

- Click `Add`
- Select `RobotModel`
- Ensure the `Description Source` parameter is set to `Topic`
- Ensure the `Description topic` parameter is set to `robot_description`

Visualizing the Laser Scan

To display laser scan data:

- Click `Add`
- Select `LaserScan`
- Set the `Topic` field to the laser scan topic (e.g., `/scan`)

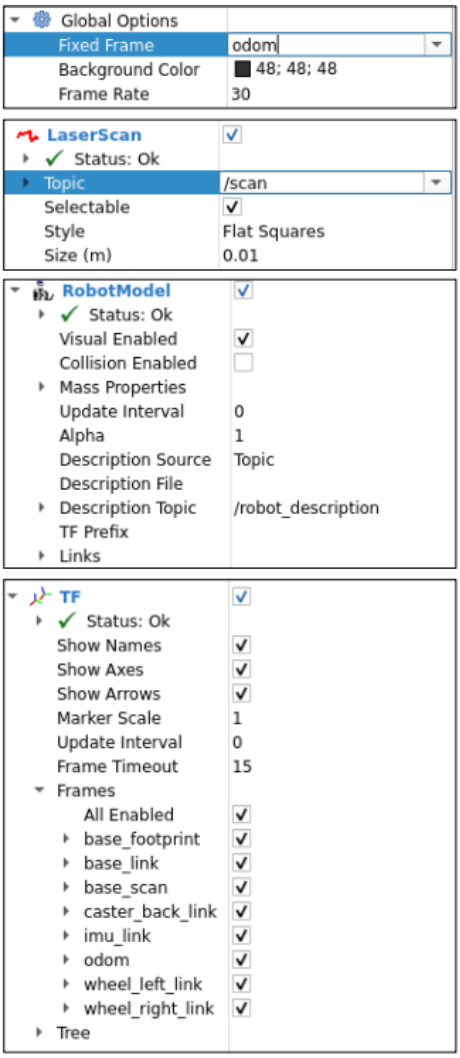


Figure 4: Example RViz2 visualizations settings for the `Fixed Frame`, the `LaserScan`, the `RobotModel`, and the `TF` (from top to bottom).

After configuring these displays (check Fig. 4), RViz2 provides a real-time visualization of the robot, its coordinate frames, and sensor data.

To avoid reconfiguring RViz2 at every startup, visualization settings can be saved and reused. The launch file below starts RViz2 with all previously defined elements already configured.

```
ros2 launch exercise_navigation tb3_rviz2.launch.py
```

5 Mobile Robotics: Command the robot with the keyboard

In this exercise, you will implement a ROS 2 node that listens for keyboard input and publishes velocity commands to the `cmd_vel` topic, enabling manual control of a robot in simulation.

Keyboard input is monitored using the `pynput` Python library.

The file `keyboard_teleop.py`, located in the folder `exercise_navigation`, already contains:

- the structure of a ROS 2 node,
- a publisher for `geometry_msgs/msg/Twist` messages,
- keyboard `listener` with `on_press` and `on_release` callbacks,
- some incomplete methods with `TODO` comments.

Your task is to complete the missing parts of the code so that the node behaves as described below.

5.1 `TODO`: Monitor keyboard events and track pressed keys

You will use four keys, each one associated to actions: **move forward**, **move backward**, **rotate left**, **rotate right**.

Use the keys you prefer, depending on your keyboard layout. A common choice is to use the WASD keys (with QWERTY layout) or the ZQSD keys (with AZERTY layout), as depicted in Fig. 5.

- QWERTY: `W` forward, `S` backward, `A` left, `D` right
- AZERTY: `Z` forward, `S` backward, `Q` left, `D` right

Multiple keys can be pressed at the same time, allowing to combine forward/backward motions with rotations. Opposite commands should cancel each other.

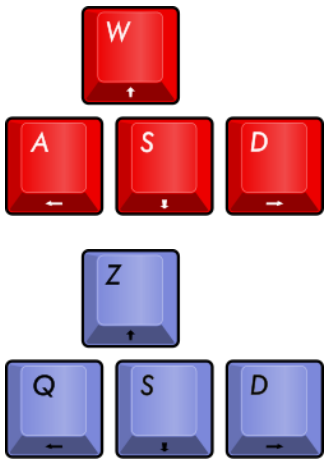


Figure 5: Motion directions with WASD or ZQSD keys.

Define the set of allowed keys

Inside the `__init__` function, define in variable `self.allowed` the set of keys used to control the robot, WASD or ZQSD.

```
self.allowed = ("w", "a", "s", "d")
```

Define keyboard callbacks

Take a look at the `main` function. In addition to creating the node and spinning it, the function contains the following code:

```
listener = keyboard.Listener(
    on_press=node.on_press,
    on_release=node.on_release,
)
listener.start()
```

These instructions create a `pynput.keyboard.Listener`, which monitors keyboard activity and executes the `node.on_press` and `node.on_release` methods whenever a key is pressed or released, respectively.

NOTE

A keyboard listener is implemented as a `threading.Thread` and therefore runs in parallel with the main ROS 2 spinning thread. For this reason, the listener must be explicitly started by calling `listener.start()`.

Now, complete the `on_press` and `on_release` methods in the main class by carrying out the following steps:

- Extract the typed character from the argument `key` using the utility function `self.key_to_char`;
- Check whether the character belongs to the set of allowed characters; if it does not, simply `return`;
- Add the character to `self.pressed` when a key is pressed, or remove it when the key is released.

Tip: `self.pressed` and `self.allowed` are **Python sets**. You can add or remove elements from a set using the methods `add` and `discard`, and check whether an element belongs to a set using the `in` operator.

5.2 `TODO`: Compute and publish the proper velocity command

Velocity commands are published on the `cmd_vel` topic using a `Twist` message. Note that the publisher, `self.pub`, has already been defined for you.

You can inspect the definition of the `Twist` message by running:

```
ros2 interface show geometry_msgs/msg/Twist
```

A `Twist` message can be used to describe the linear and angular velocity of a robot in space. In this exercise, we are only interested in:

- `linear.x`, which controls forward and backward motion;
- `angular.z`, which controls rotational motion.

You will use fixed linear and angular speeds, defined using ROS 2 parameters (see the `__init__` method):

- `linear_speed` (default: 0.25 m/s)
- `angular_speed` (default: 1.2 rad/s)

Now, complete the `self.publish_cmd_vel` method – which is called at the end of both the `on_press` and `on_release` methods – by computing the appropriate linear and angular velocity commands based on the currently pressed keys (stored in `self.pressed`). Fill in the corresponding fields of the `Twist` message and publish it using `self.pub`.

Once you have completed the implementation, re-build the `exercise_navigation` with

```
colcon build --packages-select exercise_navigation
```

and test the keyboard teleoperation node by running:

```
ros2 run exercise_navigation keyboard_teleop
```

You can also experiment with different linear and angular velocities by overriding the default parameter values at runtime using the `-ros-args -p` option.

6 Mobile Robotics: Point-to-point navigation

In this exercise, you will setup a ROS 2 node that drives the robot toward a desired goal position by following a point-to-point straight line.

This behavior will be defined through a simple *Finite State Machine* (FSM) – scheme in Fig. 6 – consisting of two states:

- `IDLE`: the robot does not move and waits for a goal;
- `GO_TO_GOAL`: the robot has received a goal and attempts to reach it by publishing on `/cmd_vel`.

The transitions between states are:

- `IDLE` → `GO_TO_GOAL`: when a new goal is received;
- `GO_TO_GOAL` → `IDLE`: when the goal has been reached (distance to goal is lower than `goal_tolerance`).

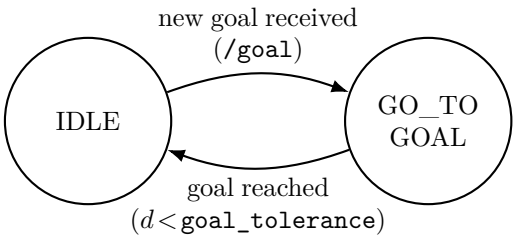


Figure 6: FSM for point-to-point navigation.

The file `point2point_nav.py`, located in the `exercise_navigation` folder, already contains the skeleton of the ROS 2 node `point2point_nav`, which you are expected to complete.

The `point2point_nav` node includes:

- a `Twist` message publisher on the `/cmd_vel` topic (`self.cmd_pub`);
- a `PoseStamped` message subscriber on the `/goal` topic (`self.goal_sub`), with callback function `self.on_goal`;
- a timer that calls the `step` method every 0.05 s;
- a `tf2_ros.Buffer` and a `tf2_ros.TransformListener`, which allow querying the *TF tree*.

The following internal variables are set via ROS 2 parameters to handle basic node configuration:

- `self.world_frame`: fixed reference frame used for navigation (default: `odom`);
- `self.base_frame`: robot base reference frame (default: `base_link`);
- `self.linear_speed`: maximum forward linear velocity of the robot (default: 0.20 m/s);
- `self.angular_speed`: maximum angular velocity of the robot (default: 0.6 rad/s);
- `self.goal_tolerance`: below this distance threshold the goal is considered reached (default: 0.05 m).

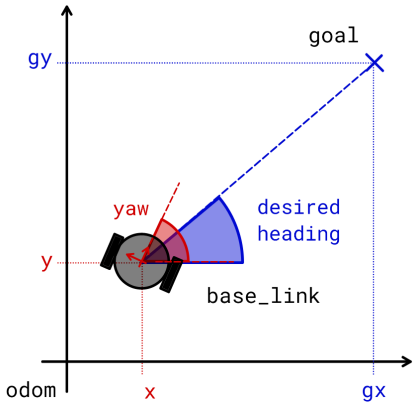


Figure 7: Scheme of the point-to-point navigation with a differential drive mobile robot.

And two internal variables are used to manage the FSM:

- `self.goal`: the most recently received goal (a `PoseStamped` message);
- `self.state`: track of the current FSM state (`"IDLE"` or `"GO_TO_GOAL"` `str`).

Finally, a set of utility functions – `angle_wrap`, `yaw_from_quat`, and `clamp` – is also provided.

6.1 `TODO`: Get robot pose from the *TF tree*

Complete the `get_pose` method by first retrieving the transformation from the `odom` frame to the `base_link` frame. You can obtain it by calling the `lookup_transform` method of the `tf_buffer`:

```
tf = self.tf_buffer.lookup_transform(
    target_frame="odom",
    source_frame="base_link",
    time=rclpy.time.Time(),
    timeout=Duration(seconds=0.2),
)
```

NOTE

You can find more details on how to access frame information in the official `tf2` documentation.

This function returns a `TransformStamped` message, here stored in the variable `tf`.

You can access:

- the translation through `tf.transform.translation`, which is a `Vector3` message;
- the rotation through `tf.transform.rotation`, which is a `Quaternion` message.

Finally, extract and return the tuple `(x, y, yaw)` from `tf`. You can use the provided utility function `yaw_from_quat` to compute `yaw` from the `Quaternion` message.

6.2 `TODO`: Store the desired goal and compute distance

Every time a new goal is published on the topic `/goal`, the callback method `on_goal` is triggered. Complete this method by storing the received message in `self.goal` (it will be used later) and by setting `self.state` to `GO_TO_GOAL`.

Fill in also the utility method `goal_distance(self, x, y)`, which returns the Euclidean distance between the point `(x, y)` and the current goal. You can access the goal coordinates from the fields `self.goal.pose.position.x`, and `self.goal.pose.position.y`.

6.3 `TODO`: Write a *go-to-goal* Controller

Define the `go_to_goal(self, x, y, yaw)` method, whose purpose is to compute and publish a velocity command that drives a mobile robot toward a goal position.

This method implements a ***go-to-goal*** control law for a differential-drive mobile robot. The controller computes the desired heading toward the goal using simple geometric relationships and applies a proportional control action on the heading error. The forward velocity is modulated based on the magnitude of the heading error, causing the robot to slow down or stop while re-orienting when misaligned.

Follow the steps below to implement the controller:

1. **Read the goal position**
   The goal coordinates (g_x,g_y) are extracted from the stored goal message.

2. **Compute the desired heading**
   Compute the angle of the straight line connecting the current robot position (x,y) to the goal:

   $$\theta_d = \text{atan2}(g_y - y, g_x - x)$$

   This angle represents the direction the robot should face in order to move directly toward the goal.

3. **Compute the orientation error**
   Compute the error as the difference between the desired heading and the current `yaw` angle θ:

   $$\theta_{\text{err}} = \theta_d - \theta$$

   Remember to wrap the error angle to the interval $[-\pi,\pi]$ in order to avoid discontinuities at $\pm\pi$.

   Tip: You can use the `angle_wrap` utility function provided.

4. **Compute the angular velocity (steering control)**
   Use a proportional controller to compute the angular velocity:

   $$\omega_z = k_\omega \cdot \theta_{\text{err}}$$

   where k_ω is the proportional gain (a value of 1.5 is recommended). Remember to clamp the resulting velocity to the maximum allowed angular speed to prevent excessive rotation.

   Tip: You can use the `clamp` utility function provided.

5. **Compute the linear velocity (forward motion)**
   Reduce the linear velocity when the robot is not well aligned with the goal:

   $$v_x = v_{\max} \cdot \max\left(0, 1 - \frac{|\theta_{\text{err}}|}{\theta_{\max}}\right)$$

   The maximum forward velocity is scaled by a coefficient that depends on the current angular error. This causes the robot to slow down when the angular error is large, and to stop and re-orient only when the error exceeds $\theta_{\max}$. You can use $\theta_{\max} = 70\text{deg}$ (convert it to radians in the code).

6. **Publish the velocity command**
   Publish the computed linear and angular velocities (v_x,ω_z) on the `/cmd_vel` topic. You can use the `publish_cmd(self, vx, wz)` helper method.

6.4 `TODO`: Handle the control loop

Finally, ensure that the correct commands are sent from the control loop. Check the `step` method (which is periodically called by the timer) and complete it.

If the goal has been reached (i.e., the distance to the goal is below `self.goal_tolerance`), stop the robot (call `stop`) and set the state to `IDLE`. Otherwise, call the `go_to_goal` method implemented before.

6.5 Test the point-to-point navigation

To try out the navigation node you just finished. First, re-build the `exercise_navigation` package with

```
colcon build --packages-select exercise_navigation
```

and start the `point2point_nav` node by running:

```
ros2 run exercise_navigation point2point_nav
```

RViz2 offers a useful tool to publish goal pose messages. Open it with the pre-configured visualization settings using:

```
ros2 launch exercise_navigation tb3_rviz2.launch.py
```

Click on the `2D Goal Pose` button (Fig. 8) and then click somewhere around the robot. This tool is configured to publish a `PoseStamped` message on the topic `/goal`, containing the 2D position of the clicked point expressed in the fixed frame `odom`.

If everything is working correctly, you should see the robot moving toward the selected goal (try to avoid sending the robot into the obstacles!).

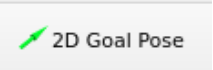


Figure 8: `2D Goal Pose` button.

7 Mobile Robotics: Avoiding obstacles – *Bug 2* navigation

In the previous exercise, you implemented a simple navigation algorithm capable of driving the robot toward a desired goal by following a straight line. However, this approach does not allow the robot to avoid obstacles. In this exercise, you will address this limitation by implementing the so-called *Bug 2* navigation algorithm.

Bug 2 is a simple reactive navigation algorithm that drives a mobile robot to a goal using only local sensing (e.g., a 2D lidar) and a global reference line. It guarantees progress toward the goal in many practical situations while remaining easy to implement.

Key idea: the M-line

Let the robot start at position S and the goal be G. The **M-line** (motion line) is the straight line connecting S to G. When there are no obstacles blocking the way, the robot simply follows the M-line toward the goal.

Two main behaviors

Bug 2 alternates between two behaviors:

- **Go-to-goal (follow the M-line):** move toward the goal while staying close to the M-line.
- **Wall-following:** when an obstacle blocks the M-line, follow the obstacle boundary until it is possible to return to the M-line.

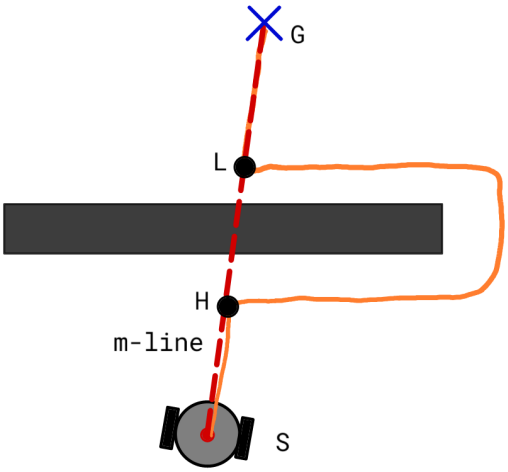


Figure 9: Example of *Bug 2* navigation avoiding an obstacle.

Hit point and leave point

When the robot first detects that an obstacle blocks its direct motion along the M-line, it stores:

- the **hit point** H: the position where the robot first gets sufficiently close to the obstacle.

During wall-following, the robot checks whether it has reached a valid **leave point** L, defined as:

- a point sufficiently close to the M-line,
- that is closer to the goal than the hit point H (i.e., $d(L,G) < d(H,G)$),

Stopping condition

The navigation terminates when the distance to the goal is below a tolerance threshold.

Algorithm 1 Bug 2 Navigation Algorithm

```
Compute the M-line from the start position S to the goal G.
Go-to-goal: move along the M-line toward G.
if an obstacle blocks the path along the M-line then
    Store the current position as the hit point H.
    Switch to wall-following behavior (keep the obstacle on a fixed side, e.g., the left).
end if
while wall-following do
    Track the obstacle boundary while maintaining a safe distance.
    if the robot is on the M-line and closer to the goal than H then
        Leave the obstacle boundary.
        Switch back to go-to-goal behavior.
    end if
end while
Stop when the robot is within a goal tolerance of G.
```

The file `bug_nav.py`, located in the `exercise_navigation` folder, already contains the skeleton of the ROS 2 node `point2point_nav`, which you are expected to complete.

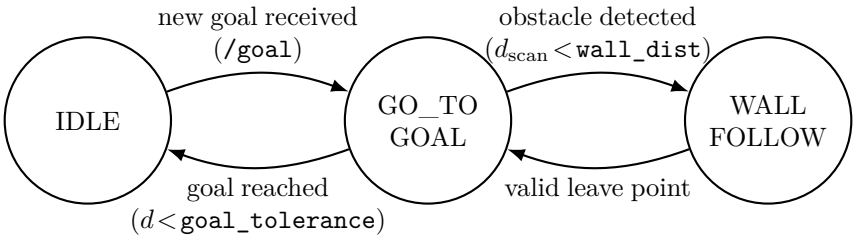


Figure 10: FSM for the *Bug 2* navigation.

The node is an extension of the previous `point2point_nav`, adding a new state, `WALL_FOLLOW`, to the FSM, as shown in the scheme of Fig. 10.

Besides the elements already present in the `point2point_nav` node (included the utlity functions), the `bug_nav` node includes :

- a `LaserScan` message subscriber on the `/scan` topic (`self.scan_sub`), with callback function `self.on_scan`.

Additional internal variables are set via ROS 2 parameters to configure the wall-following behavior:

- `self.wall_dist`: target distance from obstacles (default: 0.4 m);
- `self.mline_tol`: tolerance to leave the M-line (default: 0.1 m).

And other internal variables are used to manage the *Bug 2* logic:

- `self.scan`: the most recently received `LaserScan` msg;
- `self.start`: the start point saved when a new goal is received (a `tuple` of `float`, `(x, y)`);
- `self.hit_point`: the hit point saved when an obstacle blocks the M-line (a `tuple` of `float`, `(x, y)`);
- `self.hit_goal_dist`: the distance of the hit point from the goal (a `float`).

7.1 `TODO`: Define goal and scan callbacks

Complete the `on_goal` callback, by

- saving the received goal message inside `self.goal`;
- saving the start point inside `self.state` (use `self.get_pose()`);
- re-init `self.hit_point` and `self.hit_goal_dist` by setting them to `None`;
- set `self.state` to `"GO_TO_GOAL"`.

Complete the `on_scan` callback, by

- saving the received `LaserScan` message inside `self.scan`.

7.2 `TODO`: Process distance measurements from `LaserScan` message

In ROS 2, 2D LiDARs publish data using the `sensor_msgs/LaserScan` message. This message represents a single 2D scan of distance measurements taken at fixed angular increments around the robot.

A `LaserScan` message mainly contains:

- `angle_min`: starting angle of the scan (in radians) – in our case 0 rad;
- `angle_max`: ending angle of the scan (in radians) – in our case 2π rad;
- `angle_increment`: angular distance between consecutive measurements (in radians);
- `ranges`: an array of distance measurements (in meters), where each element corresponds to a specific angle.

Each range measurement `ranges[i]` corresponds to the angle:

$$\theta_i = \texttt{angle\_min} + i \cdot \texttt{angle\_increment}$$

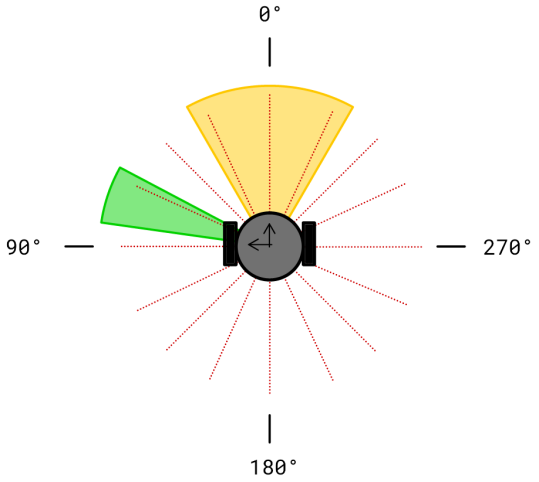


Figure 11: Laser scan from the *Turtlebot3 Burger* robot with highlighted the front and left circular sectors.

Commonly, it is assumed that 0 rad corresponds to the direction directly in front of the robot, and that angles increase in the counterclockwise direction (as shown in Fig. 11).

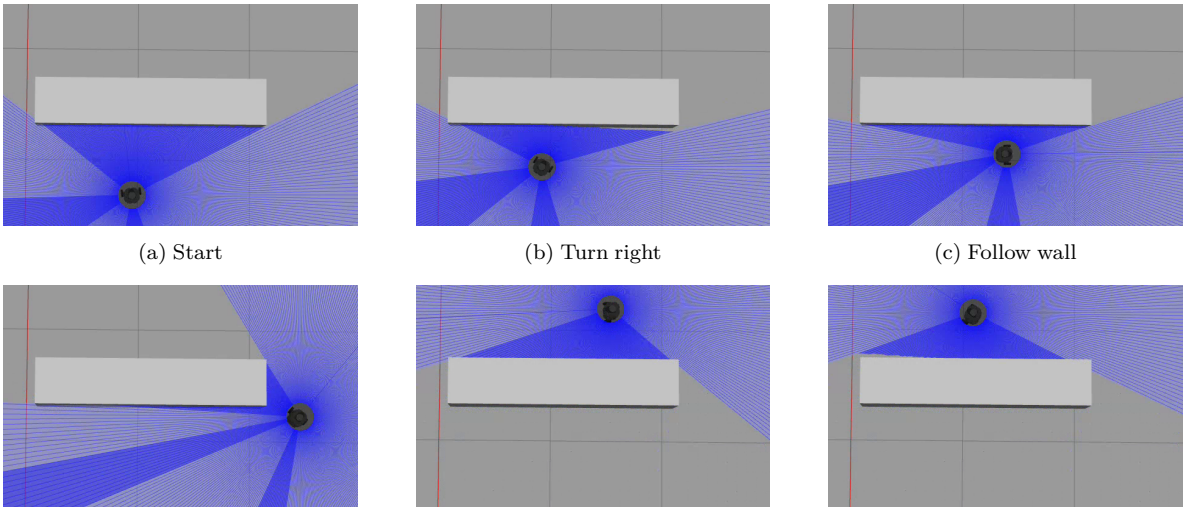
Some range values may be infinite or invalid (e.g., when no obstacle is detected within the sensor range), and should be handled accordingly.

You can inspect the full description of the `LaserScan` message by running

```
ros2 interface show sensor_msgs/msg/LaserScan
```

Now, let's complete the `min_range_in_sector` method. Its purpose is to compute the minimum obstacle distance measured by the laser scanner within the desired circular sector centered.

The circular sector is defined by the arguments `center_deg`, which specifies the center angle (in degrees), and `half_width_deg`, which defines half of the angular width of the sector (in degrees).



(a) Start (b) Turn right (c) Follow wall

(d) Turn around wall (e) Continue wall-following (f) Rejoin M-line

Figure 12: Example execution of the Bug 2 navigation algorithm.

The method should return the minimum value in the `self.scan.ranges` array between the two indices corresponding to the requested circular sector.

Tip: Some important points to keep in mind

- The limiting angles `center_deg` ± `half_width_deg` must be normalized to the interval $[0,2\pi]$. You can use the *modulo* operator `%`.
- The index corresponding to a given angle can be computed as `index = round(angle / angle_increment)`.
- If the sector is centered around 0 deg, it may happen that `i_start` > `i_end`. In this case, it is necessary to handle the two intervals separately: `[0, i_end + 1]` and `[i_start, -1]`.

7.3 `TODO`: Compute the distance from the M-line

Now, complete the method `dist_to_mline`, which is repeatedly called during the wall-following phase to check whether the robot has reached a valid leave point.

Tip: Recall that the M-line is the line connecting the start point and the goal point, saved in `self.start` and `self.goal`, respectively. First compute the line coefficients from these two points, then compute the distance from the current robot position `(x, y)` to the M-line using the appropriate formula.

7.4 `TODO`: Make the robot follow the wall

Finally, let us define the command velocities to send to the robot in order to follow the wall. Implement this basic strategy inside the `follow_left_wall` method:

- if an obstacle is detected in front (`self.front_dist() < self.wall_dist`), turn the robot right (`vx = 0.0` and `wz = -self.angular_speed`);
- if the wall is **too far** on the left (`self.left_dist() > 1.25 * self.wall_dist`), turn the robot left to get closer to the wall (`vx = 0.0` and `wz = self.angular_speed`);
- otherwise, go forward (`vx = self.linear_speed` and `wz = 0.0`).

Tip: You can use the utility methods `self.front_dist()`, `self.left_dist()`, and `self.publish_cmd(vx, wz)` to easily obtain obstacle distances and publish command velocities.

7.5 Test the *Bug 2* navigation

To try out the *Bug 2* navigation node you just finished. First, re-build the `exercise_navigation` package with

```
colcon build --packages-select exercise_navigation
```

start the `bug_nav` node by running:

```
ros2 run exercise_navigation bug_nav
```

and open RViz2 using:

```
ros2 launch exercise_navigation tb3_rviz2.launch.py
```

NOTE

Take a look at the `step` method to see how transitions between the two phases are handled.

You can send goals to the robot using the `2D Goal Pose` button. Try sending a goal behind one of the walls: you should see the robot stop in front of it and circumnavigate it before reaching the goal (as in Fig. 12).

FINAL REMARKS

What was implemented in this session is simply a first example of a basic autonomous navigation strategy. As such, it is not best suited to handle real-world scenarios, since several simplifying assumptions have been made regarding sensing, environment structure, and robot behavior.
If you are interested in exploring this topic in more depth, you are encouraged to look into **Nav2**, the standard ROS 2 navigation stack, which is widely adopted in both research and industrial applications. Visit the **Nav2 official documentation** for introductory materials and tutorials.

8 Robotic Manipulation: Preliminaries

8.1 Cartesian impedance controller

A Cartesian impedance controller regulates the dynamic relationship between the robot end-effector and a desired task-space reference by imposing a virtual mass–spring–damper behavior in Cartesian space.

You can imagine it as a virtual mass–spring–damper system connecting the robot end-effector to its desired pose reference. In the presence of a deviations or external disturbances, the robot does not behave rigidly but instead reacts compliantly according to the specified stiffness and damping.

To achieve this behavior, it is necessary to explicitly command joint torques. This feature is not available on every robotic manipulator – industrial robots commonly provide only a position control interface, which prevents direct control of interaction forces. Collaborative robots (*cobots*), instead, are designed to allow torque-level control, making them well suited for compliant and interaction-rich tasks.

Task-space impedance control

Let $\mathbf{x}$ and $\mathbf{x}_d$ be the current and desired end-effector *position*, and let $\mathbf{Q}$ and $\mathbf{Q}_d$ be the current and desired end-effector *orientation* expressed as unit quaternions.

The **translational error** is computed as: $\mathbf{e}_p = \mathbf{x}_d - \mathbf{x}$

The **rotational error** is computed from the quaternion difference: $\mathbf{Q}_e = \mathbf{Q}^{-1}\mathbf{Q}_d$

From $\mathbf{Q}_e$, the 3D orientation error vector is obtained by extracting its vector part: $\mathbf{e}_o = \begin{bmatrix} Q_{e,x} & Q_{e,y} & Q_{e,z} \end{bmatrix}^\top$

Quaternion differences produce errors expressed in the current end-effector frame. To ensure frame consistency, $\mathbf{e}_o$ must be expressed in the robot base frame by applying the current end-effector rotation matrix $\mathbf{R}$: $\mathbf{e}_o \leftarrow \mathbf{R}\mathbf{e}_o$

Translational and rotational error can then be stacked forming the **full 6D task-space error**:

$$\mathbf{e} = \begin{bmatrix} \mathbf{e}_p \\ \mathbf{e}_o \end{bmatrix} \in \mathbb{R}^6$$

For our control law, it is also necessary to have the **end-effector twist** vector. It can be obtained by mapping joint velocities $\dot{\mathbf{q}}$ using the end-effector Jacobian $\mathbf{J}(\mathbf{q})$:

$$\dot{\mathbf{x}} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}$$

A virtual spring–damper system is imposed in task space, generating the **Cartesian control wrench**:

$$\mathbf{F}_{\text{task}} = \mathbf{K}\mathbf{e} - \mathbf{D}\dot{\mathbf{x}}$$

where $\mathbf{K}$ and $\mathbf{D}$ are the Cartesian stiffness and damping matrices (positive definite 6×6), respectively.

This Cartesian wrench is mapped to joint torques using the Jacobian transpose:

$$\boldsymbol{\tau}_{\text{task}} = \mathbf{J}^\top \mathbf{F}_{\text{task}}$$

Nullspace control

For redundant manipulators, multiple joint configurations can realize the same end-effector pose. This redundancy is exploited by adding a secondary control action in the nullspace of the Jacobian.

The **nullspace projection matrix** is defined as follows (the dependence of the Jacobian on $\mathbf{q}$ is omitted for clarity):

$$\mathbf{N} = \mathbf{I} - \mathbf{J}^\dagger \mathbf{J}$$

where $\mathbf{J}^\dagger$ denotes the pseudoinverse of the Jacobian matrix.

A joint-space impedance behavior is used to regulate the robot toward a desired configuration $\mathbf{q}_d^{\text{ns}}$ in the task null space, while the task-space controller enforces tracking of the desired end-effector pose.

$$\boldsymbol{\tau}_{\text{ns}} = \mathbf{N}(\mathbf{K}_{\text{ns}}(\mathbf{q}_d^{\text{ns}} - \mathbf{q}) - \mathbf{D}_{\text{ns}}\dot{\mathbf{q}})$$

where **K**ns and **D**ns are diagonal matrices (i.e., matrices with all off-diagonal elements equal to zero) containing the null-space stiffness and damping gains along their diagonals.

Final control law

The total commanded joint torque is obtained by summing the task-space and nullspace contributions:

$$\boldsymbol{\tau}_d = \boldsymbol{\tau}_{\text{task}} + \boldsymbol{\tau}_{\text{ns}}$$

This control structure allows the robot to achieve compliant behavior at the end effector while simultaneously satisfying secondary objectives in joint space, resulting in stable and flexible interaction control.

8.2 Linear interpolation of translations and rotations

In many robotics applications, it is useful to smoothly transition between two poses over time. While translations and rotations are both part of a pose, they must be interpolated differently due to their distinct mathematical properties.

Linear interpolation of translations

Translations live in a Euclidean space, and can therefore be interpolated using standard linear interpolation. Given an initial position $\mathbf{p}_0$ and a final position $\mathbf{p}_1$, a linear interpolation is defined as:

$$\mathbf{p}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1, \qquad t \in [0,1]$$

This produces a straight-line motion between the two points at constant speed.

Spherical Linear Interpolation (SLERP)

Rotations do not belong to a Euclidean space, but to the special orthogonal group $SO(3)$. As a consequence, applying linear interpolation directly to rotation representations (e.g., quaternions, Euler angles or rotation matrices) may lead to incorrect or unintuitive behavior.

To interpolate rotations properly, it is common to use **Spherical Linear Interpolation** (SLERP). SLERP interpolates between two orientations along the shortest path on the unit quaternion hypersphere, producing smooth rotational motion with constant angular velocity.

Given two unit quaternions $\mathbf{Q}_0$ and $\mathbf{Q}_1$, SLERP can be expressed as:

$$\mathbf{Q}(t) = \left(\mathbf{Q}_1 \mathbf{Q}_0^{-1}\right)^t \mathbf{Q}_0, \qquad t \in [0,1]$$

Here, $(\cdot)^t$ denotes a quaternion power, defined through the exponential and logarithmic maps on the unit quaternion manifold (since rotations belong to $SO(3)$).

NOTE

Interpolation is useful both for **planning smooth Cartesian trajectories** between two poses and for filtering reference pose signals. In practice, when implementing a robot control loop, it is common to **filter the desired pose values before computing the control actions**. This helps avoid sudden "*jumps*" in the reference signals, which could otherwise lead to abrupt velocity or torque changes and potentially cause instability. In other terms, the pose specified by the user is treated as a target, while the actual reference pose – used to compute the control torques – is filtered so as to gradually converge to the desired set-point.

8.3 Eigen: a C++ library for linear algebra

Eigen is a commonly-adopted C++ library for linear algebra.

The most common Eigen operations are summarized below.

Vector and matrix types

Eigen provides fixed-size types (recommended when sizes are known) and dynamic-size matrix types.

```
Eigen::MatrixXd x;                // dynamic-size matrix
Eigen::Matrix<double, 6, 6> M;    // fixed-size matrix (6x6)
Eigen::Matrix<double, 3, 1> v;    // fixed-size vector (3x1)
```

Some aliases are defined for commonly used sizes:

```
Eigen::Vector2d x;       // 2x1 vector of doubles
Eigen::Vector3d y;       // 3x1 vector of doubles
Eigen::VectorXd z;       // dynamic-size vector of double
Eigen::Matrix3d M;       // 3x3 matrix of doubles
```

Resizing dynamic objects

Dynamic-size vectors and matrices must be resized before use:

```
Eigen::VectorXd v;
v.resize(7);       // now v has size 7
v.setZero();       // set all entries to 0

Eigen::MatrixXd M;
M.resize(6, 6);    // now M is 6x6
M.setIdentity();   // only valid for square matrices
```

Common initializations

Eigen provides convenient constructors and helper functions:

```
Eigen::VectorXd v = Eigen::VectorXd::Zero(7);
Eigen::VectorXd w = Eigen::VectorXd::Ones(7);

Eigen::MatrixXd I = Eigen::MatrixXd::Identity(7, 7);
Eigen::MatrixXd Z = Eigen::MatrixXd::Zero(6, 7);

Eigen::Vector3d a(1.0, 2.0, 3.0);   // fixed-size vector constructor
```

Assigning and Accessing Elements

The operator « is used to assign multiple values at once:

```
Eigen::Matrix<double, 6, 1> x;
x << 1.0, 2.0, 3.0, 4.0, 5.0, 6.0;  // 6 values
```

You can access single entries using parentheses `(i,j)` for matrices and `(i)` for vectors:

```
double s = v(0);        // first element
v(2) = 10.0;           // modify third element

double aij = M(1, 3);   // row 1, col 3
M(0, 0) = 1.0;
```

Extracting Subvectors and Submatrices

Eigen allows selecting the first and last components of a vector:

```
Eigen::Matrix<double, 6, 1> v(1.0, 2.0, 3.0, 4.0, 5.0, 6.0);

Eigen::Vector3d first = v.head(3);  // first 3 elements: [1 2 3]
Eigen::Vector3d last  = v.tail(3);  // last 3 elements:  [4 5 6]
```

Use `block()` to extract or assign a submatrix. For fixed-size blocks, use template arguments:

```
// dynamic-size block: block(top-left row i, top-left col i, row size, col size)
Eigen::MatrixXd B = M.block(0, 0, 3, 3);

// fixed-size block: block<row size, col size>(top-left row i, top-left col i)
Eigen::Matrix3d R = T.block<3, 3>(0, 0);
Eigen::Vector3d p = T.block<3, 1>(0, 3);
```

Transpose and basic linear algebra

Eigen uses natural syntax for matrix operations:

```
Eigen::MatrixXd A, B;
Eigen::VectorXd x, y;

y = A * x;          // matrix-vector product
B = A.transpose();  // transpose
B = A.inverse();    // inverse
```

Quaternions

Eigen provides quaternion support through `Eigen::Quaterniond`:

```
// Initialization
Eigen::Quaterniond q1, q2;
q1.coeffs() << 0.0, 0.0, 0.0, 1.0;     // IMPORTANT: (x, y, z, w) ordering here!
q2 = Eigen::Quaterniond::Identity();

// Access components
double qx = q1.x(); double qy = q1.y(); double qz = q1.z(); double qw = q1.w();

// Normalize
q1.normalize();

// Inverse and multiplication
Eigen::Quaterniond q_inv = q1.inverse();
Eigen::Quaterniond q_prod = q1 * q2;

// SLERP interpolation
double t = 0.3;  // t in [0, 1]
Eigen::Quaterniond q_interp = q1.slerp(t, q2);
```

9 Robotic Manipulation: Write the robot controller

In this tutorial, we will work with a simulated ***Franka Emika Panda Robot*** running in the **MuJoCo** simulator.

The *Franka Emika Panda* is a 7-degree-of-freedom collaborative robotic manipulator widely used in research and education. It features torque-controlled joints with integrated force sensing and supports

real-time control, making it suitable for teaching basic concepts in robotic kinematics, dynamics, and control.

The controller is implemented through a `ros2_control` plugin written in C++.

From the `robotic_manipulation` repository, open a terminal inside the container using the script:

```
bash run_docker.sh
```

This script runs the container in an interactive session within a `terminator` terminal, while mounting the `franka_example_controllers` folder into the pre-existing workspace `humble_ws` inside the container.

You will need to complete the implementation of the controller plugin `franka_example_controllers/ExerciseCartesianImpedanceController` – which implements a Cartesian impedance controller – by filling in the file `exercise_cartesian_impedance_controller.cpp` located in the `src` folder.

In C++, the `.cpp` file contains the actual implementation of the controller logic, while the corresponding header file (`exercise_cartesian_impedance_controller.hpp` inside the `include` folder) declares the class, its functions, and its member variables. The header file specifies *what* the controller provides, whereas the source file specifies *how* it is implemented.

Take a look at both files to become familiar with their structure and to understand how the class declaration in the header file relates to its implementation in the source file.

After each code modification, recompile the package by running

```
colcon build --packages-select franka_example_controllers
```

and check the build output for compilation or syntax errors.

9.1 TODO: Setup the desired pose topic subscriber

Our controller receives the desired end-effector pose from `PoseStamped` messages published on the topic `"/cartesian_impedance/desired_pose"`.

The subscriber is created inside the `ExerciseCartesianImpedanceController::on_init()` function, but the associated callback function still needs to be completed.

Within the function `ExerciseCartesianImpedanceController::equilibriumPoseCallback` use the contents of the incoming `msg` to update the controller's internal state.

In particular, extract the position and orientation from the message and assign them to the member variables `position_d_target_` (of type `Vector3d`) and `orientation_d_target_` (of type `Quaterniond`), respectively.

Tip: If you do not remember the definition of the `PoseStamped` message, you can inspect it by running the following ROS 2 command in a terminal inside the container:

```
ros2 interface show geometry_msgs/msg/PoseStamped
```

9.2 TODO: Compute the task torques

The function `ExerciseCartesianImpedanceController::update` implements the control loop.

The code already provides robot state acquisition and the relevant variable declarations. In particular, you will work with:

- `error`: 6D task-space error (to be computed),
- `tau_task`: task-space torque contribution (to be computed),
- `position_d_`: reference end-effector position (given),
- `current_position_`: current end-effector position (given),

- `orientation_d_`: reference end-effector orientation as a quaternion (given),
- `current_orientation_`: current end-effector orientation as a quaternion (given),
- `current_transform`: base-to-end-effector homogeneous transformation (given),
- `jacobian`: task-space (end-effector) Jacobian (given),
- `task_stiff_`: task-space stiffness matrix (given),
- `task_damp_`: task-space damping matrix (given),
- `dq`: current joint velocities (given).

Complete the code inside the `TODO` block to compute the control torques.

Tip: Implementation guidelines.

1. The error vector first three entries correspond to the translational error and the last three to the rotational error (use `head()` and `tail()` to access the subvectors).
2. After computing the orientation error from quaternions, rotate it to express it in the desired reference frame (use the rotation matrix from the base-to-end-effector homogeneous transformation).
3. The end-effector twist can be computed from joint velocities via the Jacobian.

9.3 `TODO`: Compute the null-space torques

Now continue by computing the null-space torque contribution.

You will work with the following variables:

- `tau_nullspace`: null-space torque contribution (to be computed),
- `q`: current joint positions (given),
- `dq`: current joint velocities (given),
- `ns_stiff_`: null-space stiffness matrix (given),
- `ns_damp_`: null-space damping matrix (given).

To project a joint-space control action into the null space of the task, you will need the (damped) Jacobian pseudo-inverse. You can compute it using the provided `pseudoInverse` function and then build the null-space projector.

Tip: The pseudo-inverse matrix `M_pinv_` is passed to the function by reference and is filled inside the function. This means that you must first declare an `Eigen::MatrixXd` variable and then pass it as an argument to the function, rather than expecting the function to return it.

9.4 `TODO`: Filter the reference pose

As a final step inside the function `ExerciseCartesianImpedanceController::update`, you need to filter the reference end-effector position and orientation used at the next control iteration.

This is done by applying one step of linear interpolation for the position and SLERP for the orientation. You will work with the following variables:

- `position_d_`: filtered reference end-effector position (to be updated here),
- `position_d_target_`: target position set-point (updated by `equilibriumPoseCallback`),
- `orientation_d_`: filtered reference end-effector orientation as a quaternion (to be updated here),
- `orientation_d_target_`: target orientation set-point (updated by `equilibriumPoseCallback`),
- `filter_param_`: smoothing parameter controlling how fast the filtered reference converges to the set-point (larger values yield faster convergence; initialized to `0.05` in the `.hpp` file).

Tip: Implementation guidelines.

1. Referring to the notation introduced in 8.2, the filtered reference position/quaternion corresponds to the initial value (marked with 0), while the target set-point corresponds to the final value (marked with 1). The parameter `filter_param_` plays the role of t.
2. Remember that you can use the `slerp` function provided by `Eigen::Quaterniond` to easily interpolate between two quaternions.

9.5 Test the Cartesian impedance controller

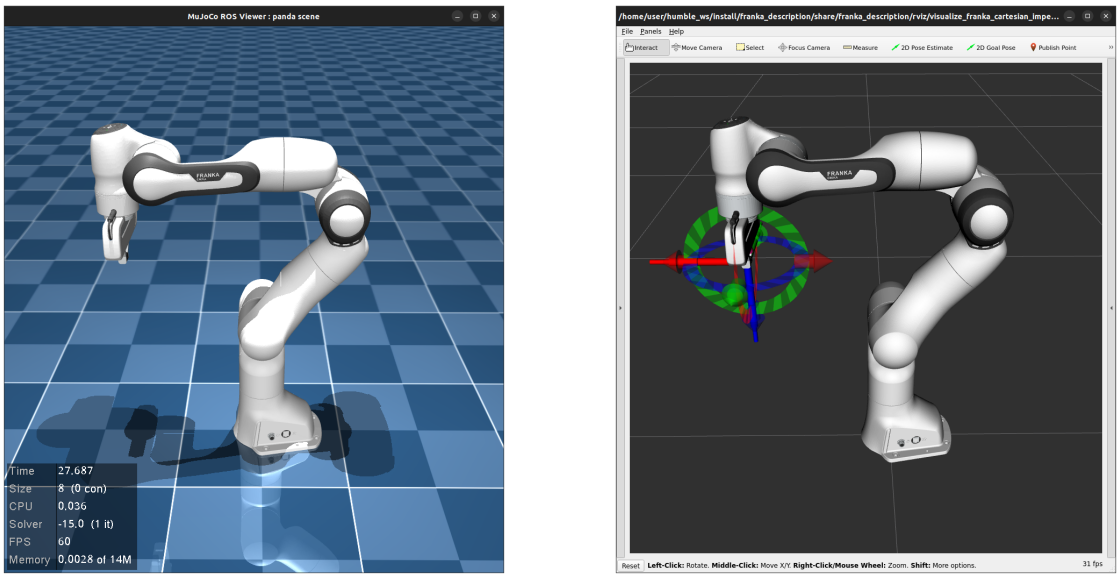
To test the Cartesian impedance controller you have just implemented, first re-build the `franka_example_controllers` package by running

```
colcon build --packages-select franka_example_controllers
```

Then, launch the simulator – with the controller already loaded and running – together with RViz2 by executing

```
ros2 launch franka_bringup exercise_cartesian_impedance.launch.py
```

This will open up two windows, one for the MuJoCo simulator, and one for RViz2, as shown in Fig. 13.



(a) Franka Panda simulated in MuJoCo

(b) Franka Panda visualization in RViz2

Figure 13: Cartesian impedance control experiment: MuJoCo simulation (left) and RViz2 visualization (right).

To interact with the robot, first select the MuJoCo window and **press the spacebar** to **unpause** the simulation.

Then, switch to the RViz2 window. You will see a so-called **interactive marker** located at the tip of the robot end-effector. An interactive marker provides a simple graphical interface for specifying the desired end-effector pose. By dragging the arrows and rotation rings, you can translate and rotate the marker in 3D space.

By right-clicking on the marker, a pop-up menu appears with two entries:

- A `Reset` button, which instantly places the marker at the current end-effector pose.
- A `Send goals` checkbox, which enables the publishing of the current interactive marker pose as the target set-point for the Cartesian impedance controller.

After enabling the `Send goals` option, you can modify the desired end-effector pose by moving the interactive marker, and the controller will smoothly drive the robot toward the commanded pose.

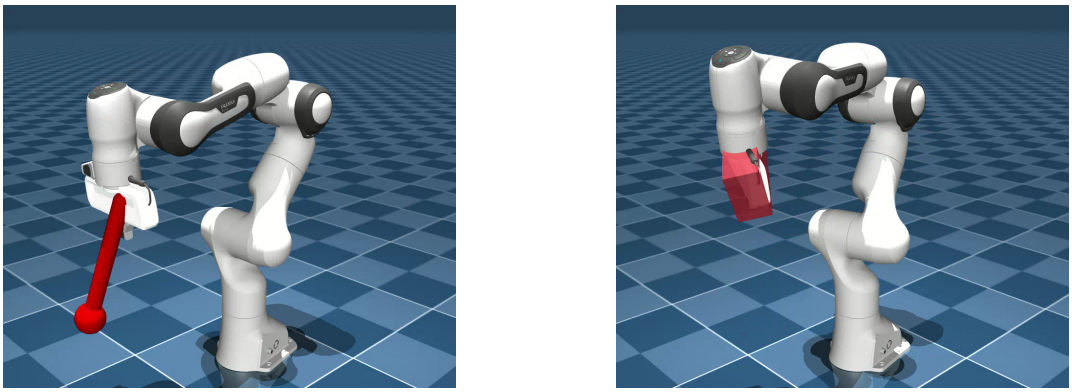
Simulate external interactions

The MuJoCo viewer allows you to freely move the camera to inspect the robot and its motion. You can rotate the camera by dragging the mouse while holding the left mouse button, translate the view by dragging with the right mouse button, and zoom in or out using the mouse wheel.

To simulate external interactions with the robot, you can apply forces or torques directly to individual robot links. First, select a link by double-left-clicking on the desired body in the MuJoCo window (try this with the gripper link).

Once a link is selected, you can apply an external force by holding `Ctrl` and dragging the mouse while holding the right mouse button; alternatively, hold the left mouse button to apply a torque.

This allows you to intuitively perturb the robot and observe the compliant behavior of the Cartesian impedance controller in response to external disturbances, as shown in Fig. 14.



(a) Applying an external force

(b) Applying an external torque

Figure 14: Simulating external interactions in MuJoCo by applying forces (left) and torques (right) to the robot.

NOTE

Try to modify the controller gains defined in the `exercise_cartesian_impedance_controller.hpp` file (don't forget to re-build the package) and observe how these changes affect the robot behaviour when you apply external forces and torques.

Visualize topics with *PlotJuggler*

PlotJuggler is a powerful visualization tool that allows you to plot and inspect ROS 2 topics in real time.

While the robot simulation is running, you can start *PlotJuggler* in a separate terminal with:

```
ros2 run plotjuggler plotjuggler
```

To configure the visualizer, follow these steps:

- Press the `Start` button in the left panel under `Streaming`.
- In the pop-up menu, select the topics `/cartesian_impedance/current_pose`, `/cartesian_impedance/desired_pose`, and `/cartesian_impedance/filt_ref_pose`, then press `OK`.
- Under `Timeseries list`, you will see the namespace `cartesian`. Click on the arrow icon to expand it and explore all the included topics and their fields.
- Drag and drop numerical fields (for example, `/cartesian_impedance/current_pose.pose.position.x`) into the plotting area to visualize the data online as it is published.

Try comparing `current_pose`, `desired_pose`, and `filt_ref_pose` to observe how the filtering you implemented smooths the reference signal and how the actual end-effector pose tracks the set-point.

10 Robotic Manipulation: Write a linear trajectory planner

In this tutorial you will implement a linear Cartesian trajectory planner node. The node can be used to drive the robot end-effector to a desired goal pose along a straight-line trajectory in task space.

A summary of the steps the node must perform is given below:

1. Read the desired goal pose and motion duration from the ROS 2 parameter `"goal"`.
2. Subscribe to `"/cartesian_impedance/current_pose"` to obtain the current end-effector pose, which will be used as the start pose.
3. Compute the interpolated poses between start and goal over the specified duration, using linear interpolation for position and SLERP for orientation.
4. Publish the interpolated poses at a fixed rate to the controller reference topic `"/cartesian_impedance/desired_pose"`.

The file `exercise_pose_publisher.py`, located in the `franka_simple_publishers` folder, already contains the skeleton of the ROS 2 node `exercise_pose_publisher` for you to complete.

Take a moment to inspect the code structure and understand its overall logic. The publisher, subscriber, and timer have already been handled for you; your main task is to correctly load the user input and implement the interpolation computation.

10.1 `TODO`: Declare, load and process the `"goal"` parameter

We will receive the desired goal position, orientation, and time-to-reach from the user via the ROS 2 parameter `goal`, of type `float64[]` (an array of `float`).

We adopt the following convention: the first entry is the duration, the next three entries define the goal position, and the last four entries define the goal orientation as a quaternion: `[T, x, y, z, qx, qy, qz, qw]`.

How to declare and read an array parameter of type `float`

In ROS 2, parameters of type `float64[]` can be used to pass arrays of floating-point values to a node. They are declared by providing a list of `float` as a default value and are read back as a standard Python list. In practice, it is convenient to convert the parameter value into a `numpy` array, since this allows for easier numerical manipulation and slicing.

```
# Declaration
self.declare_parameter("array_param", [1.0, 2.0, 3.0, 4.0])
# Reading (convert to NumPy array)
goal = np.array(self.get_parameter("array_param").value)
```

Tip: When defining the default value for the `"goal"` parameter, choose values that are physically meaningful and safe, such as a positive time duration and a reachable end-effector position and orientation (for instance, something close to the starting pose).

Validate and store the `"goal"` parameter

Extract the data from the read parameter and save them in `self.T` (duration), `self.goal_pos` (position), and `self.goal_quat` (quaternion).

When reading input parameters, it is important to check their feasibility and to handle possible user mistakes explicitly. In this case, you should verify in particular that:

- the `"goal"` parameter has size 8; otherwise, raise an error,
- the specified time duration is strictly positive; otherwise, raise an error,
- the orientation quaternion is valid; the received quaternion should always be normalized, and if its norm is near zero, an error should be raised.

In Python, errors are signaled by raising exceptions, and different exception types are used depending on the nature of the problem. In this context, `ValueError` is the most appropriate choice, since the parameter exists and has the correct type, but its value is not admissible.

```
if <condition>:
    raise ValueError("error message")
```

10.2 `TODO`: Compute interpolated pose at each time step

Once the goal parameter has been correctly loaded and validated, you need to compute the reference pose to be published at each timer callback. This is done inside the `timer_cb()` method, which is executed periodically at the chosen publishing rate.

The current trajectory time is stored in `self.t` and is incremented at each invocation. To keep the interpolation well-defined, the time variable is saturated so that it never exceeds the total duration `self.T`. Your task is to compute, at each time step, the interpolated end-effector pose between the start pose and the goal pose.

Position interpolation

The position is interpolated linearly between the start and goal positions. First compute a normalized interpolation parameter

$$\alpha = \frac{t}{T} \in [0,1],$$

which represents the progress along the trajectory. Then compute the interpolated position as a convex combination of the start and goal positions–as previously described in 8.2–and store the result in the `pos` variable.

Orientation interpolation

The orientation must be interpolated using spherical linear interpolation (SLERP), implemented by the `Slerp` class in `scipy.spatial.transform`. A minimal usage example is shown below:

```
from scipy.spatial.transform import Rotation as R, Slerp

slerp = Slerp([0.0, 1.0], R.from_quat([q0, q1]))
rot = slerp(alpha)
quat = rot.as_quat()   # quaternion in (x, y, z, w) format
```

The resulting quaternion must be stored in the variable `quat`. The interpolated position `pos` and orientation `quat` are then used to populate and publish the reference `PoseStamped` message.

10.3 Test the linear trajectory planner

To test the linear trajectory planner you have just implemented, first re-build the `franka_simple_publishers` package by running

```
colcon build --packages-select franka_simple_publishers
```

Then, as before, launch the robot simulator together with RViz2:

```
ros2 launch franka_bringup exercise_cartesian_impedance.launch.py
```

Finally, start the trajectory planner node, which will immediately begin publishing reference poses:

```
ros2 launch franka_simple_publishers exercise_pose_publisher.launch.py
```

You should observe the robot moving toward a default target pose, following a linear Cartesian trajectory similar to the one shown in Fig. 15.

The launch file is configured to use a `LaunchArgument`, allowing the desired goal pose to be specified from the command line.

To send a custom goal, simply provide the `goal` argument when launching the node:

```
ros2 launch franka_simple_publishers exercise_pose_publisher.launch.py \
  goal:="[3.0, 0.6, 0.2, 0.3, 1.0, 0.0, 0.0, 0.0]"
```

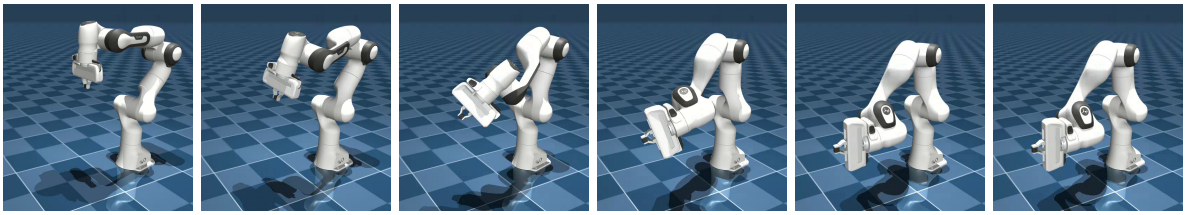


Figure 15: Sequence of end-effector poses along a linear Cartesian trajectory generated by the planner.

FINAL REMARKS

What was implemented in this session is a first example of a basic Cartesian control and planning approach. For the sake of time and simplicity, several important aspects of real-world robot control were deliberately neglected, such as hardware abstraction, controller composition, safety mechanisms, and advanced motion planning.
If you are interested in exploring robotic control and manipulation in more depth, you are encouraged to look into **ros2_control** and **MoveIt**.

- **ros2_control** is the standard ROS 2 framework for hardware abstraction and controller management. An introduction and reference documentation can be found in the **ros2_control official documentation**.
- **MoveIt** provides high-level tools for motion planning, kinematics, collision checking, and manipulation. Introductory materials and hands-on tutorials are available in the **MoveIt official documentation** and in the **MoveIt tutorials**.