Robotics Lab - Lecture Notes

Jonathan Cacace

May 2, 2021

Contents

Preface

This document contains the lecture notes for the *Robotics Lab* class taken at the University of Naples Federico II for automation engineering master's degree during the 2020/21 academic year. This course aims to give an overview of the fundamental tools and techniques used to program advanced robotics systems (both industrial and mobile). After a brief introduction of the technologies commonly used to program robots (e.g. Linux, c++, git), the Robot Operating System (ROS) framework is introduced and deeply explored. Simulation software will help the course attendees to implement and test state-of-art robotic algorithms and their control software.

1

Chapter 1 - Robot Operating System

Robot Operating System (ROS) represents a flexible framework, providing various tools and libraries to write robotic software. It offers several powerful features such as message passing, distributing computing, code reusing, and implementation of state-of-the-art algorithms for robotic applications.

1.0.1 History of ROS

The first version of ROS was release in 2007 by Willow Garage, a robotics research laboratory located in California. In that year, Willow Garage was developing PR2 robot (see Fig.1.1), one of the first robots running ROS as a programming framework.

ROS was born as an open source software and several developers outside Willow Garage participated in its development. One of the main reasons motivating the development of ROS was to simplify the role of robotic programmers. In particular, robotic applications are typically composed of similar parts of software. The most common problem of robotics at the time was that they spent too much time to re-implement the software infrastructure required to build complex robotics algorithms (basically, drivers to the sensors and actuators, and communications between different programs inside the same robot) and too little time dedicated to building intelligent robotics programs that were based on that infrastructure. For this reason, to overcome this problem ROS offers different features, like easy process communication, code reuse, and software modularity. These features made ROS particularly suitable for pr2 programming since it was composed of several heterogeneous sensors (e.g. sonar, lidars, mobile base, etc…) and hardware components. Nowadays ROS is managed by *OSRF* (Open Source Robotic Foundation) and it is released under BSD License.

Today, ROS represents the standard for robot programming and it is already integrated into many robots and used by many universities and

Figure 1.1: PR2 Willow Garage robot. One of the first ROS-enabled robot.

companies.

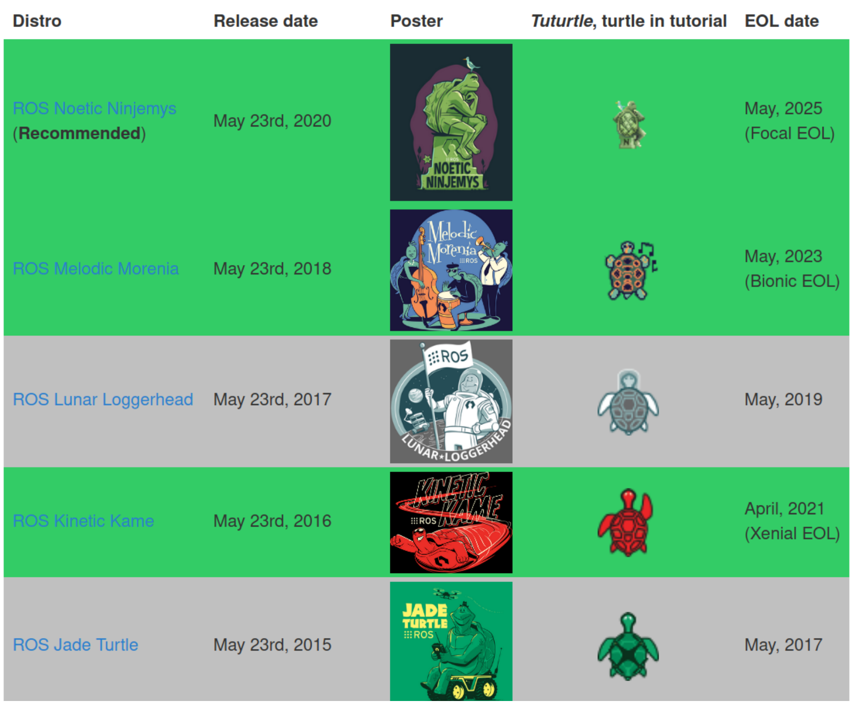
1.0.2 ROS Distributions

ROS updates are released with new ROS distributions. A new distribution of ROS is composed of an updated version of its core software and a set of new/updated ROS packages. ROS follows the same release cycle of Ubuntu Operating System: a new version of ROS is released every six months. Typically, for each Ubuntu LTS (Long Time Support) version, an LTS version of ROS is released. LTS stands for *Long Term Support* and means that the released software will be maintained for a long time (5 years in case of ROS and Ubuntu). The current LTS version of ROS at writing time is *Noetic Ninjemys*, and it's supported by Ubuntu 20.04. The list of recent ROS distribution is shown in Fig. 1.2.

1.0.3 Robot Operating System

ROS 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 building, writing, and running code across multiple computers. The correct definition for ROS is a robotic *middleware*, a software that connects different software components or applications, as shown in Fig. 1.3.

ROS officially runs on a Unix-based platform. Some experimental versions have been released for Windows and MacOS. The suggested version



Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (**Recommended**)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015			May, 2017

Figure 1.2: Recent ROS release.

for Linux is Ubuntu. Before discussing the features and working principles of ROS, let's take an overview of the main elements of ROS software. The computation in ROS is done using a network of processes called ROS **nodes**. This computation network along with additional functionalities can be called *computation graph.* The main concepts in ROS computation graph are *Nodes, Master, Parameter server, Messages, Topics, Services, and Bags.* Each concept in the graph contributes in different ways. Let's focus on the two main elements that are the nodes and the master:

- **Nodes**: Nodes are the processes that perform computation (the executable). Each ROS node is written using ROS client libraries implementing different ROS functionalities, such as the communication methods between nodes, which is particularly useful when different nodes of our robot must exchange information between them. Using the ROS communication methods, they can communicate with each other and exchange data. One of the aims of ROS nodes is to build simple processes rather than a large process with all the functionality (modularity).

- **Master**: The ROS Master is a special ROS node that provides the name registration and lookup to the rest of the nodes. Nodes will not be able to find each other, exchange messages, or invoke services

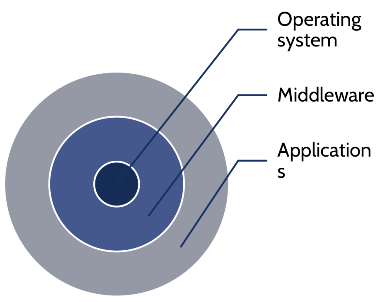


Figure 1.3: Middleware concept.

without a ROS Master. In a distributed system, we should run the master on one computer, and other remote nodes can find each other by communicating with this master.

The typical structure of ROS applications is shown in Fig. 1.4. In particular, in each ROS system, only one Master node can be active. All the other nodes exchange information between them thanks to the ROS master. In practice, a ROS node is nothing more than a program written by developers in one of the supported programming languages. Currently, the supported languages are C++, Python, Matlab, and Java. The ROS functionalities in all the programming languages are the same. The ROS Master is much like a DNS

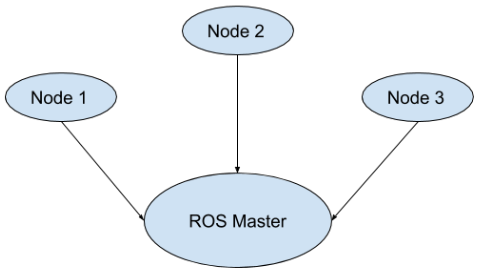


Figure 1.4: ROS application structure.

server, associating unique names and IDs to ROS elements active in our system. When a new node is launched in the ROS system, it will start looking for the ROS Master and register the name of the node in it. So, the ROS Master handles the details of all the nodes currently running on the ROS system.

Now that the basic idea of ROS has been discussed, let's start to better detail its components.

The elements of ROS are summarized in Fig. 1.5 and are described in the following:

- **Plumbing**: ROS allows communication between processes (nodes). In particular, it provides publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed (local and remote) computing systems. For example, consider that your application uses data from a camera, you can use the ROS node deployed by the vendor of your camera to use the data in your application.

- **Tools**: ROS provides an extensive set of tools to configure, manage, debug, visualize data, log and test your robotic application.
- **Capabilities**: ROS provides a broad collection of libraries that implement useful robot functionalities, like manipulation, control, and perception. In addition, ROS can be connected to other external software like OpenCv, PCL, and so on, thanks to proper wrappers (i.e. developers can avoid to re-invent the wheel).
- **Ecosystem**: ROS is supported and improved by a large community, with a strong focus on integration and documentation. On ROS webpage: `ros.org` you can find basic and advanced tutorial to learn how to program in ROS, while di Q&A website (`answers.ros.org`) allow you to directly ask you solution for your own problems (and contains thousand of question already answered).

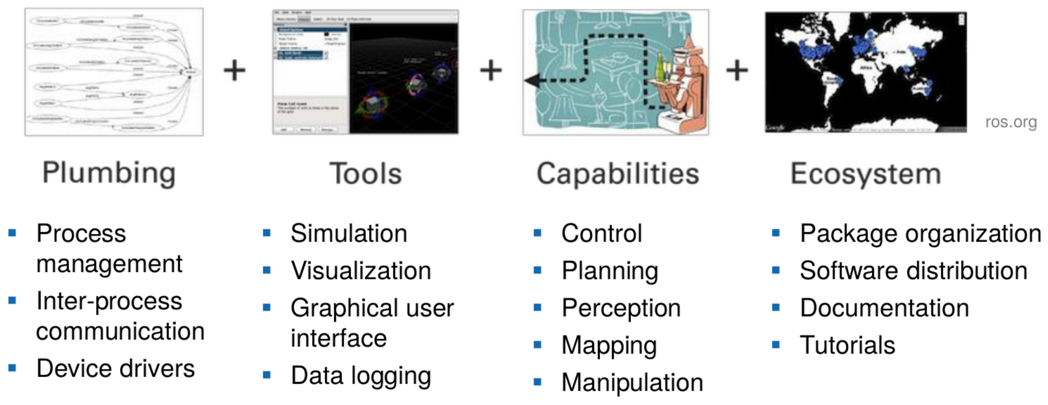


Figure 1.5: ROS components.

At this point we are ready to discuss the philosophy of ROS.

The philosophy of ROS is that you have several individual programs (modules) implemented in your robotic system (these programs can be located on the same machine or **distributed** over the network) and can communicate with each other using defined API like ROS messages, services, and others. Each module can be written in any preferred programming language supported by ROS (at current stage: C++, Python, Matlab, and Java). ROS is free software and its core is open source.

Following this philosophy, you can be easily able to design modular software with several independent interchangeable modules solely responsible for a small task in the overall software. The advantages the modularity in your code are many. First of all, debug a small part of code and functionalities is easier. Besides, will be more easily update your software substituting only the legacy part of your software.

Let's discuss an example of a simple robotic application programmed with ROS. We can consider the navigation of a mobile robot that has to track and follow a given visual target. Such application can be quite complex to program from scratch and can be composed of several modules, as depicted in Fig. 1.6. In this context, we can classify three application layers. One responsible to handle robot sensors, like the vision sensor (*Camera*), the laser scanner (*Lidar*) and the *Encoders* of the wheels. The second layer instead, is responsible for the robot navigation. To perform such a task, the robot must be able to *Localize* itself into the environment, to *Map* the obstacles, and to generate a control strategy to accomplish a given task considering sensor data. Finally, the lower level of your application consists of the integration with the hardware of the robot. Thanks to the large number of versatile ROS modules the developers can concentrate on the programming of the control algorithm. Several sensors are already supported by ROS, like standard USB cameras, depth sensors (from kinect, asus or intel), laser scanners, and so on. Similarly, state-of-art mapping and localization algorithms are already deployed to receive the data from the sensors. One important element of this software architecture is that to exchange information between multiple modules (plumbing) in ROS a set of *standard* messages are used. In this context, another cause for reflection regards the maintainability of the code. Update or change sensors or pieces of code in this architecture is very easy since the communication interface between the modules will remain the same.

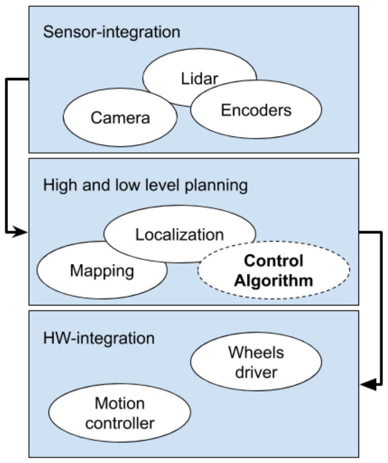


Figure 1.6: Example of ROS node in a robotic mobile navigation task.

Now, let's talk more concretely about ROS software infrastructure starting with the *message-passing*. In the following, two kind of communication protocols are described: the publish-subscribe and services.

Publish-subscribe

Two processes (ROS Nodes) can communicate in different ways. The first communication protocol discussed here is an asynchronous communication protocol based on the publish/subscribe paradigm in which a process streams a series of data that can be read by one or more processes. This communication relies on an entity called **topic**. In particular, each message in ROS is transported using named buses called topics. When a node sends a message through a topic, then we can say the node is publishing a topic, while when a node receives a message through a topic, then we can say that the node is subscribing to a topic. The publishing node and subscribing node are not aware of each other's existence we can even subscribe to a topic that might not have any publisher. In short, the production of information and consumption of it is decoupled. The publish/subscribing communication is described in Fig. 1.7. In this context, the publisher and subscriber nodes register to the ROS Master. The publisher node creates a topic specifying its name that must be unique in the ROS system and the type of message to publish. Differently, the subscriber node requests the data from the topic as long as it specifies the correct message type. This protocol is useful especially when a node must share a continuous stream of information. For example, a node that must grab data from a camera sensor should broadcast the sequence of the images taken from the sensor using a ROS publisher. In this case, we are not interested in the of the communication bridge.

ROS Messages

ROS communication relies on a set of standard and custom data structures called ROS messages. Datatypes are described using a simplified message description language called ROS messages. These datatype descriptions can be used to generate source code for the appropriate message type in different target languages. The message definition consists of a typical data structure composed of two main types: fields and constants. The field is split into field types and field names. The field type is the data type of the transmitting message and the field name is the name of it. The constants define a constant value in the message file. In the following, an example of a message definition to share the pose (position and orientation) of the robot is shown. It is a `geometry_msgs::PoseStamped` message.

```
std_msgs/Header header
        uint32 seq
        time stamp
        string frame_id
```

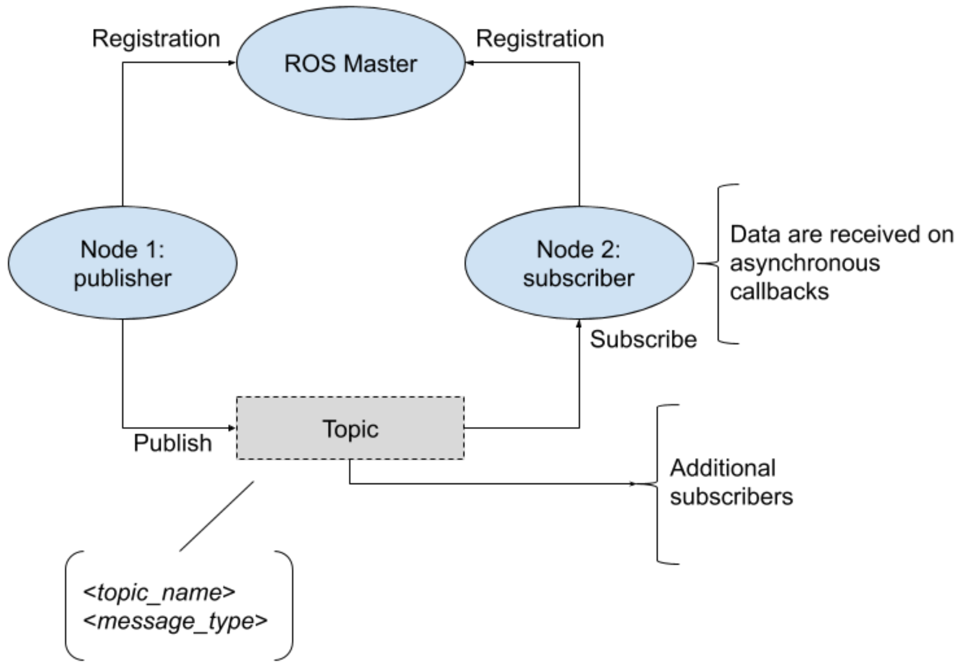


Figure 1.7: ROS publish/subscribe protocol.

```
geometry_msgs/Pose pose
        geometry_msgs/Point position
                float64 x
                float64 y
                float64 z
        geometry_msgs/Quaternion orientation
                float64 x
                float64 y
                float64 z
                float64 w
```

This example represents the definition of structured data to boradcast. The first part of the message is present in several ROS messages and represents an *header* containing information about the publishing time of the message and its reference frame (i.e. the fixed or dynamic reference frame concerning the pose of the object is specified). The rest of the message contains information about the 6D position of the robot. As you can easily see, the structure of the message is composed using basic data types (string, float and so on). Table 1.1 shows some of the built-in field types that we can use in our message.

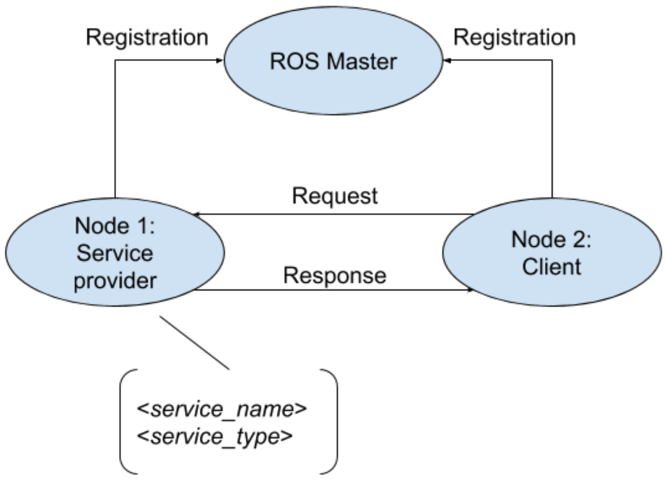


Figure 1.8: ROS service communication.

Primitive type	Serialization	C++	Python
bool(1)	Unsigned 8-bit int	uint8_t(2)	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int (3)
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string(4)	std::string	string
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	rospy.Time ros::Duration	rospy.Duration

Table 1.1: Built-in type for ROS messages

ROS Services

ROS also supports asynchronous communication protocol: ROS services. ROS Services establish a request/response communication between nodes. One node will send a request and wait until it gets a response from the other. Similar to the message definitions we have to define the service definition. A service description language is used to define the ROS service types.

An example of service description format is as follows:

```
#Request message type
string str
---
#Response message type
string str
```

The first section is the message type of the `request` that is separated by − − − and in the next section is the message type of the `response`. In this example, both Request and Response are strings. Of course, the definition of service could contain a structured ROS message (like the one used for the pose of the robot) as well. This type of communication protocol should be used when a node is specialized in doing specific tasks, like some complex calculations. For example, this node could be responsible to calculate the inverse of a matrix. In this way, all the other modules of your system that need to invert a given matrix could directly use this service, without replicating the inversion operation in their source code.

Visualization

ROS provides different tools to support developers in their work. Among these tools, Ros Visualization (RViz) is very useful. It is a 3D visualizer to graphically display the contents of a topic using `viualization_markers` messages. RViz can visualize robot models, the environments they work in, and sensor data directly from ROS topics. Built-in and custom plugins can be loaded on RViz to add additional functionalities like motion planning or motion control.

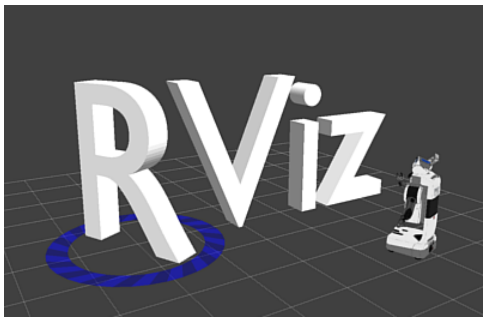


Figure 1.9: ROS Visualization (RViz).

Simulation

Another important feature of ROS is the simulation. In particular, ROS is strictly integrated with Gazebo simulator `http://gazebosim.org/`, a multi-robot simulator for complex indoor and outdoor robotic simulation. In the Gazebo environment, we can simulate complex robots, robot sensors, and a variety of 3D objects. Gazebo already has simulation models of popular robots, sensors, and a variety of 3D objects in their repository. Also, several plugins already exist to interact with the simulated using ROS. Using Gazebo we can simulate the real sensor mounted on our robots receiving the same stream of data (thanks to the same ROS message definition between the real and simulated sensors). An example of the Gazebo interface is shown in Fig. 1.10, where a wheeled mobile robot is simulated. A huge amount of robotic simulators today exist. Differently, almost all of them

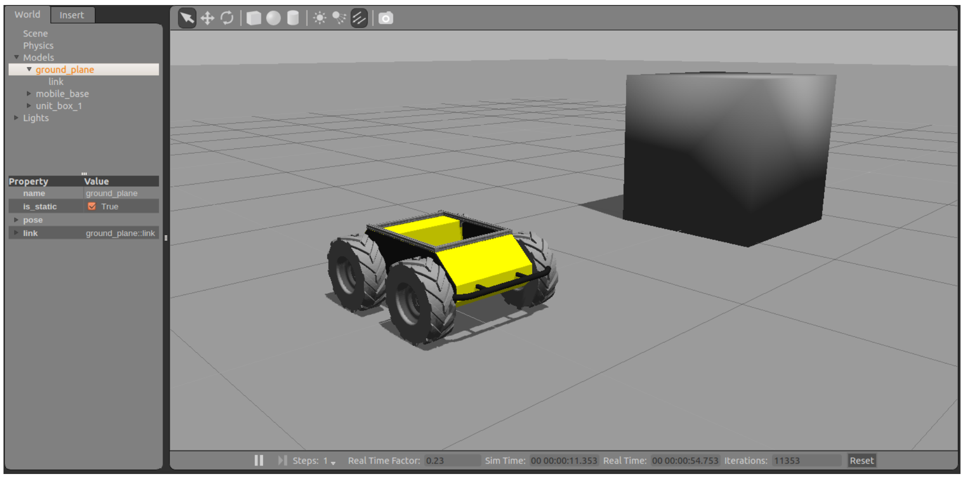


Figure 1.10: Gazebo ROS simulator.

can be integrated with ROS. Among them, we can refer to *CoppeliaSim* and *Webots*.

Summary

In this chapter, we introduced Robot Operating System detailing its history and the motivation that brought to its development. The main element of the ROS computation graph have been discussed: the ROS master node and the executable nodes. Finally, two communication protocols allowing ROS nodes to exchange messages each other have been introduced. Beyond all its benefits, there are some disadvantages in using ROS. In particular, main issues of ROS concern the its reliability and safety. Firstly, the communication between multiple nodes can be unstable, especially with a big amount of data or with a distributed robotic system. In addition, the ROS master will respond to requests from any device on the network (or host) that can connect to it. Any host on the network can publish or subscribe topics, list or modify parameters, and so on. If an unauthorized user can connect to the ROS master, they could leak sensitive information (such as data from sensors or cameras), or even send commands to move a robot, which creates both a privacy and a safety risk. These and other issues bring to the development or ROS 2, the second version of ROS that focus on the use of ROS in real industrial scenarios.

In the next chapter, an overview of the technologies used to program robots will be provided. In particular, we will learn the basic usage of a unix-based operating system, the basic concept of C++ programming and the compilation, and an overview of version control tools like git.

2

Robotics programming technologies

Nowadays, there exist different setup to program robotic systems. Some of these rely on proprietary languages developed by robot providers, like `KRL` and sunrise for kuka industrial robots, or `RoboDK` for Universal Robots. However, these languages don't allow to develop intelligent robotic application to perform complex tasks, but they are considered to be implement cyclical motion in a completely known environment. Differently, sometimes developers need to program robot considering different high level or low level prospective. For this reason, to develop advanced robotic applications in which the robot is able to plan new actions based on the state of its operative environment, standard programming languages must be used. In this lesson we mainly considered `C++` because is fast and versatile, can be used both for high level reasoning (geometrical reasoning, image elaboration, etc ...) and for low level control. Before to recall some basic concepts on c++, we briefly discuss some basic concepts and commands used in Linux.

2.1 Linux Operating System

Linux is a family of open source Unix-like operating systems based on the Linux kernel, an operating system kernel free and open-source, monolithic and Unix-like kernel, released on 1991. Typically different versions of Linux are packaged into distributions. Nowadays, exist hundreds of linux distributions usable on embedded, server and desktop devices. One of the most popular linux distributions for desktop computers is called *Ubuntu.* Ubuntu is an African sub-Saharan term meaning *humanity* or more specifically it is philosophy: 'the belief in a universal bond of sharing that connects all humanity' or 'I am because we are'. Ubuntu is released every six months, with long-term support (LTS) releases every two years. The most recent long-term support release at writing time is 20.04 LTS, which is supported until 2023 under public support.

Independently from the distributions, all versions o linux share a list of commands invokable via the linux console (terminal). Be able to use linux console for a robotics software developer is fundamental for several reasons. First of all, typically robot based on Linux OS are not endowed of a graphical interface or developers joint the robot remotely using the Secure Shell (ssh) connection. In addition, a list of commands from the command line can be extremely useful to properly configure the robotic system.

2.1.1 Install Linux

The convenient way to install linux is with multi boot options. This setup allows to install multiple operating system on the same hard disk. Another way to use linux is relying on Virtual Machine. This solution in our case presents some disadvantages, mainly because of the requirements needed by the simulation software.

2.1.2 Basic Linux commands

In this section, a list of recurrent important linux commands are reported.

- **pwd**: when you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the “pwd” command. It gives us the absolute path, which means the path that starts from the root.
- **ls**: use the ls command to know what files are in the directory you are in. You can see all the hidden files by using the command ls -a.
- **cd**: Use the cd command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in cd Downloads. Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is. To go back from a folder to the folder before that, you can type “cd ..” . The two dots represent back.
- **mkdir & rmdir**: Use the mkdir command when you need to create a folder or a directory. To delete a directory containing files, use rmdir.
- **rm**: Use the rm command to delete files and directories. Use "rm -r" to delete just the directory. It deletes both the folder and the files it contains when using only the rm command.
- **touch**: the touch command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, “touch new.txt”.

- **cp**: Use the cp command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.
- **mv**: Use the mv command to move files through the command line. We can also use the mv command to rename a file. For example, if we want to rename the file "text" to "new", we can use "mv text new". It takes the two arguments, just like the cp command.
- **locate**: the locate command is used to locate a file in a Linux system. This command is useful when you don't know where a file is saved or the actual name of the file. Using the -i argument with the command helps to ignore the case (it doesn't matter if it is uppercase or lowercase). So, if you want a file that has the word "hello", it gives the list of all the files in your Linux system containing the word hello when you type in locate -i hello. If you remember two words, you can separate them using an asterisk (*). For example, to locate a file containing the words hello and this, you can use the command locate -i *hello*this. In order to have an update representation of the filesystem and be able to find even the newest files contained in your machine you should insert the following command:

  ```
  $ sudo updatedb
  ```

  The first time this command could take a bit of time.
- **echo**: the echo command helps us move some data, usually text into a file. For example, if you want to create a new text file or add to an already made text file, you just need to type in, echo hello, my name is alok » new.txt. You do not need to separate the spaces by using the backward slash here, because we put in two triangular brackets when we finish what we need to write.
- **cat**: use the cat command to display the contents of a file. It is usually used to easily view programs.
- **nano**: nano is a text editors already installed in your Linux command line. The nano command is a good text editor that denotes keywords with color and can recognize most languages. It is one of the simplest text editor usable via command line.
- **sudo**: a widely used command in the Linux command line, sudo stands for SuperUser Do. So, if you want any command to be done with administrative or root privileges, you can use the sudo command. For example, if you want to edit a file like viz. alsa-base.conf, which needs root permissions, you can use the command: sudo nano alsa-base.conf.

- **chmod**: use chmod to make a file executable and to change the permissions granted to it in Linux. Imagine you have a python code named numbers.py in your computer. You'll need to run `python numbers.py` every time you need to run it. Instead of that, when you make it executable, you'll just need to run numbers.py in the terminal to run the file. To make a file executable, you can use the command chmod +x numbers.py in this case. Another situation in which this command is particularly useful is when your application needs to access to `USB` devices. In such case the device should have executable privileges.
- **ping**: use ping to check your connection to a server.
- You can power off or reboot the computer by using the command sudo halt and sudo reboot.
- You can use the clear command to clear the terminal if it gets filled up with too many commands.

2.1.3 Basic Linux concepts

In this subsection we report some basic concepts common to all linux operating systems.

- Command auto completion: auto completion represents a fundamental feature of linux console. In particular, TAB key can be used to fill up in terminal. For example, You just need to type `cd Doc` and then TAB and the terminal fills the rest up and makes it `cd Documents`.
- filesystem: most Linux systems use a standard layout for files so that system resources and programs can be easily located. This layout forms a directory tree, which starts at the / directory, also known as the *root directory*. Directly underneath / are important subdirectories: /bin, `/etc`, `/dev`, and `/usr`, among others. These directories in turn contain other directories which contain system configuration files, programs, and so on. In particular, each user has a home directory, which is the directory set aside for that user to store his or her files. A user has completely control of its user space (/home/user). Differently, for the higher level of the filesystem the operations must be performed with the use of superuser privileges (sudo in ubuntu).
- .bashrc: the linux shell is called bash (Bourne Again SHell). It is a command processor that typically runs in a text window where the user types commands. Bash can also read and execute commands from a file, called a shell script. Like all Unix shells, it supports piping, here documents, command substitution, variables, and control structures for condition-testing and iteration. When a new interactive linux shell

is open a series of configuration files are elaborated. In particular, bash reads and executes `/etc/bash.bashrc` and then `~/.bashrc`. For this reason, all the system configuration that you want to automatically load can be placed in the `bashrc` file. This file is placed in the home directory of the user. in addition, it is a hidden file (in fact its name starts with a dot): `/home/user/.bashrc`.

- Environment variables: in linux based systems environment variables are a set of dynamic named values, stored within the system that are used by applications launched in shells. In simple words, an environment variable is a variable with a name and an associated or a list of values. Environment variables allow you to customize how the system works and the behavior of the applications on the system. For example, the environment variable can store information about the default text editor or browser, the path to executable files, or the system locale and keyboard layout settings. In bash a variable can be set in the following way:

  ```
  $ export V=environment
  ```

  While, to print the content of a variable you should refer to the variable name using the $ character:

  ```
  $ echo $V
  ```

  In the following a list of commands used to handle environment variables are reported:

  ```
  $ echo $VARIABLE  #To display value of a variable
  $ env  #Displays all environment variables
  $ VARIABLE_NAME=variable_value  #Create a new variable
  $ unset variable_value  #Remove a variable
  $ export Variable=value  #To set value of an environment variable
  ```

- APT: to install new software a convenient way is to use the package manager. In ubuntu (derived from debian) the package manager is called APT (Advanced Packaging Tool). APT simplifies the process of managing software on Unix-like computer systems by automating the retrieval, configuration and installation of software packages, either from precompiled files or by compiling source code. In order to install a new software you should use the following syntax:

  ```
  $ sudo apt-get install [PACKAGE NAME]
  ```

If the package exists in the apt repository, the list of dependencies of such package will be also automatically installed. If you want to check if a software is contained in the apt repository you could use the following command:

```
$ apt-cache search [PACKAGE NAME]
```

Otherwise, if you know only initial part of the package you can used the auto completion.

Other usage modes of apt and apt-get that facilitate updating installed packages include:

- **update** is used to resynchronize the package index files from their sources. The lists of available packages are fetched from the location(s) specified in /etc/apt/sources.list. For example, when using a Debian archive, this command retrieves and scans the Packages.gz files, so that information about new and updated packages is available.
- **upgrade** is used to install the newest versions of all packages currently installed on the system from the sources enumerated in /etc/apt/sources.list. Packages currently installed with new versions available are retrieved and upgraded; under no circumstances are currently installed packages removed, or packages not already installed retrieved and installed. New versions of currently installed packages that cannot be upgraded without changing the install status of another package will be left at their current version.

In order to add additional software list to the apt repository you should edit the files included in `/etc/apt` directory. In particular, on Ubuntu and all other Debian based distributions, the apt software repositories are defined in the `/etc/apt/sources.list` file or in separate files under the `/etc/apt/sources.list.d/` directory. If you already installed ROS in your system, you should know that the first instruction of ROS tutorial is the following:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list'
```

This line uses the `echo` command to create a file called `ros-latest.list` filled with the address of the repository of ROS packages. At this point, the apt command should be able to see the packages contained in the ROS repository. More information about how to add other repository to APT will be provided when the installation of ROS is discussed.

2.2 Introduction to C++ programming

C++ is a general-purpose programming language created as an extension of the C programming language. One of C++'s strengths is that it can be used to write programs for nearly any processor. It is a high-level language: when you write a program in it, the shorthand are sufficiently expressive that you don't need to worry about the details of processor instructions. In addition, C++ does give access to some lower-level functionality than other languages (e.g. memory addresses). For this reason, C++ can be used both for high level robot programming (reasoning, planning and so on) and for robot low level control. The source code of a C++ program is written in a text file. The process in which a program goes from text files to processor instructions is depicted in Fig. 2.1. In particular, object files are intermediate files that represent an incomplete copy of the program: each source file only expresses a piece of the program, so when it is compiled into an object file, the object file has some markers indicating which missing pieces it depends on. The linker takes those object files and the compiled libraries of predefined code that they rely on, fills in all the gaps, and spits out the final program, which can then be run by the operating system. The compiler and linker are just regular programs. The step in the compilation process in which the compiler reads the file is called *parsing*. In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time. This is one of the reasons C++ code runs far faster than code in many more recent languages.

```
/*
* First C++ program that says hello (hello.cpp)
*/

#include <iostream>      //IO operations
using namespace std;

// Program entry point
int main() {
// Say Hello
cout << "hello, world" << endl;
// Terminate main()
```

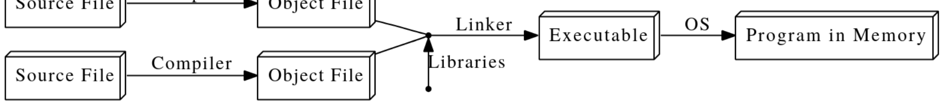


Figure 2.1: C++ compilation process

```
return 0;
} // End of main function
```

The classical form of a single process C++ program is reported in the above algorithm. This is nothing more than a simple program that prints the string "hello, world". In such code, `cout` function print out some piece of text to the screen, while to specify e certain context the namespace keyword is used: In C++, identifiers can be defined within a context called a namespace. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the scope resolution operator (::). Here, we're telling the compiler to look for cout in the std namespace, in which many standard C++ identifiers are defined. If we do this, we can omit the std::prefix when writing `cout`.

C++ syntax is very similar to C and other compiled languages. In this section, we highlight some useful functionalities of C++ using external libraries.

Pointers

After declared a variable in C++, the computer associates its name with a particular location in memory where the value of the variable is stored. When in the code such variable is referred, the computer firstly look up the address the correspond to the variable name, then go to the location in memory to retrieve the value it contains.

Mainly, C++ allows us to perform these steps independently:

- &x evaluates to the address of x in memory
- *(&x)takes the address of x and dereferences it – it retrieves the value at that location in memory. *(&x)thus evaluates to the same thing as x.

Pointers allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself. In particular in C++ pointers are particularly useful to:

- Pass-by-reference variables is more efficient.
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations.
- Return multiple values from a single function.

To declare a pointer variable named `ptr` that points to an integer variable named `x`:

```
int *ptr = &x;
```

`int *ptr` declares the pointer to an integer value, which we are initializing to the address of x. We can have pointers to values of any type.

The following block of code shows a simple example in which pointers are used to pass variables by reference.

```
void squareByPtr(int *numPtr){
*numPtr= *numPtr**numPtr;
}

int main() {
int x=5;
squareByPtr(&x);
std::cout << x << std::endl; //Prints 25
}
```

The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways: when declaring a pointer, * is placed before the variable name to indicate that the variable being declared is a pointer - say, a pointer to an int or char, not an int or char value. Then, when using a pointer that has been set to point to some value, *is placed before the pointer name to dereference it, to access or set the value it points to. A similar distinction exists for &, which can be used either to indicate a reference datatype `(int &x;)`, or to take the address of a variable `(int *ptr=&x;)`.

Smart pointers (Shared pointers)

A smart pointer represents a class of objects aiming at simplify the usage of pointers. In particular, smart pointers prevent most situations of memory leaks by making the memory deallocation automatic and providing feature like automatic memory management or bounds checking. Such features are intended to reduce bugs caused by the misuse of pointers, while retaining efficiency.

In C++, a smart pointer is implemented as a template class that mimics, by means of operator overloading, the behaviors of a traditional (raw) pointer, (e.g. dereferencing, assignment) while providing additional memory management features.

Among different features of smart pointers, in this document we are interested in the `std::shared_ptr`. C++11 introduces `std::shared_ptr`, defined in the header <memory>. A `shared_ptr` is a container for a raw pointer. It maintains reference counting ownership of its contained pointer in cooperation with all copies of the `shared_ptr`. An object referenced by the contained raw pointer will be destroyed when and only when all copies of the `shared_ptr` have been destroyed.

```
//Allocates 1 integer and initialize it with value 5.
```

```
std::shared_ptr<int> p0(new int(5));
//Valid, allocates 5 integers.
std::shared_ptr<int[]> p1(new int[5]);
//Both now own the memory.
std::shared_ptr<int[]> p2 = p1;
//Memory still exists, due to p2.
p1.reset();
//Deletes the memory, since no one else owns the memory.
p2.reset();
```

One important feature of `shared_ptr` is that multiple threads can safely simultaneously access different `shared_ptr` that point to the same object. In particular, `shared_ptr` is considered when multiple owners should access to the same object in memory. A `shared_ptr` object effectively holds a pointer to the resource that it owns or holds a null pointer. A resource can be owned by more than one `shared_ptr` object; when the last `shared_ptr` object that owns a particular resource is destroyed, the resource is freed.

Classes

A class represents a user-defined data type which groups together related pieces of information. If you consider a geometric vector, a vector consists of 2 points: a start and a finish, each point itself has an x and y coordinate. We can create the following class to represent different type of vectors:

```
class Vector {
private:
double xStart;
double xEnd;
double yStart;
double yEnd;
};
```

Of course, similar results can be obtained using a simple data structure. To improve the class functionalities we should implement some methods in the vector class. Some functions are closely associated with a particular class, like the calculation of the norm of a vector:

```
class Vector {
public:
float get_norm();
private:
double xStart;
double xEnd;
double yStart;
double yEnd;
};
```

In addition, a class need a constructor: a method that is called when an instance is created. In our case, we can consider to initialize the member of the class (the points of the vector) when an instance of the class is created:

```
class Vector {
public:
Vector( float xstart_, float xend_,
float ystart_, float yend_);
float get_norm();
private:
double xStart;
double xEnd;
double yStart;
double yEnd;
};
```

In this way, to initialize a new vector and get its norm, we must implement the functions of its class:

```
Vector::Vector( float xstart_, float xend_,
float ystart_, float yend_) {
xStart = xstart_;
xEnd = xend_;
yStart = ystart_;
yEnd = yend_;
}


float Vector::get_norm() {
return sqrt(pow((xEnd - xStart),2) + pow((yEnd - yStart),2))
}


int main() {
Vector v(0, 0, 2, 2);
std::cout << v.get_norm() << std::endl;
return 0;
}
```

As for the class modifiers, we can choose three different type for the class members:

- *public*: members are accessible from outside the class
- *private*: members cannot be accessed (or viewed) from outside the class

- *protected*: members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Data hiding is a software development technique specifically used in object-oriented programming to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes. To implement data hiding you can follow two simple rules: make all the data members private and create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

```
void Vector::set_xend( float xend ) {
xEnd = xend;
}

float Vector::get_xend() {
return xEnd;
}
```

2.2.1 Compilation using make

To generate executable files we need to compile one or more source files. One of the most common way is to use the `GNU make` program and (under linux) the `GCC` compiler. GCC, formerly for "GNU C Compiler", has grown over times to support many languages such as C (gcc), C++ (g++), Objective-C, Objective-C++, Java (gcj), etc.... It is now referred to as "GNU Compiler Collection". In this section we will discuss different tools to compile C++ programs. In particular, we introduce the command line tool of gcc to compile programs, then we discuss the `make` utility used to automatize the compilation process.

GCC

The GNU C and C++ compiler are called gcc and g++, respectively. Considering the following block of code:

```
// hello.c
#include <stdio.h>

int main() {
printf("Hello, world!\n");
return 0;
}
```

To compile the hello.c:

```
$ gcc hello.c
```

This command generates an executable that by default under linux OS is called `a.out`. This file can be executed from unix console using the following commands:

```
$ chmod a+x a.out
$ ./a.out
```

To specify the output filename, use -o option:

```
$ gcc -o hello.exe hello.c
```

Consider that `gcc` and `g++` share the same syntax, so to compile a C++ program, just used `g++` instead of `gcc`.

In case your program has multiple source files, like f1.cpp and f2.ccp, you could compile them in a single command:

```
$ g++ -o program f1.cpp f2.cpp
```

with the option `-c` you can compile multiple different object source files separately into object file, and link them together in the later stage. In this case, changes in one file does not require re-compilation of the other files:

```
$ g++ -c f1.cpp
$ g++ -c f2.cpp
$ g++ -o program f1.o f2.o
```

An important element of program compilation is represented by the shared and static libraries that can be used in your source code. In particular, a library is a collection of pre-compiled object files that can be linked into your programs via the linker. Examples are the system functions such as printf() and sqrt(). There are two types of external libraries: static library and shared library.

- A static library has file extension of ".a" (archive file). When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable.

- A shared library has file extension of ".so" (shared objects). When your program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as dynamic linking. Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. The shared library codes can be upgraded without the need to recompile your program.

When compiling the program, the compiler needs the header files to compile the source codes; the linker needs the libraries to resolve external references from other object files or libraries. This could be a tedious step because the compiler and linker will not find the headers/libraries unless you set the appropriate options. For each of the headers used in your source (via #include directives), the compiler searches the so-called include-paths for these headers. The include-paths are specified via `-Idir` option (or environment variable `CPATH`). Since the header's filename is known (e.g., iostream.h, stdio.h), the compiler only needs the directories.

As for the linker, it searches the so-called library-paths for libraries needed to link the program into an executable. The library-path is specified via `-Ldir` option (or environment variable `LIBRARY_PATH`). In addition, you also have to specify the library name. In Unix, the library libxxx.a is specified via -lxxx option. In this context, the linker needs to know both the directories as well as the library names. Hence, two options need to be specified. To summarize, GCC uses the following environment variables:

- `PATH`: For searching the executables and run-time shared libraries (.so).
- `CPATH`: For searching the include-paths for headers. It is searched after paths specified in -I<dir> options. `C_INCLUDE_PATH` and `CPLUS_INCLUDE_PATH` can be used to specify C and C++ headers if the particular language was indicated in pre-processing.
- `LIBRARY_PATH`: For searching library-paths for link libraries. It is searched after paths specified in -L<dir> options.

Make

The `make` utility automates building process of executable from source code. `make` uses a so-called `makefile`, which contains rules on how to generate executable. Let's see a first example to build the `hello.c` program into executable using make utility.

Create the following file named "makefile" (without any file extension), which contains rules to build the executable, and save in the same directory as the source file.

```
all: hello

hello: hello.o
gcc -o hello hello.o

hello.o: hello.c
gcc -c hello.c

clean:
```

```
rm hello.o hello
```

To compile the program, run the `make` command in the same directory of the `makefile`.

```
$ make
```

makefile is typically used when you have a complex compilation structure for your program (multiple sources, libraries and so on). For this reason, several variables can be used to simply the content of the makefile. Automatic variables are set by make after a rule is matched. There include:

- `$@`: the target filename.
- `$*`: the target filename without the file extension.
- `$<`: the first prerequisite filename.
- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- `$+`: similar to `$^`, but includes duplicates.
- `$?`: the names of all prerequisites that are newer than the target, separated by spaces.

The previous makefile can be re-written as:

```
all: hello

# $@ matches the target;
# $< matches the first dependent
hello: hello.o
gcc -o $@ $<

hello.o: hello.c
gcc -c $<

clean:
rm hello.o hello
```

CMake

Using `make` to compile complex and multi-platform projects could be not an easy task. For this reason, a generator of build system is often used to simplify the work of a software developer. The most famous generator of build-systems and also the one adopted by ROS is called `CMake`. `CMake` is a cross-platform free and open-source software tool for managing the build process of software using a compiler-independent method. It is used in conjunction with native build environments such as Make, Qt Creator, Ninja, Apple's Xcode, and Microsoft Visual Studio.

The build process with `CMake` takes place in two stages. Simple configuration files placed in each source directory (called `CMakeLists.txt` files) are used to generate standard build files (e.g., `makefiles`) which are used in the usual way. Another nice feature of `CMake` is that it generates a cache file that is designed to be used with a graphical editor. For example, when `CMake` runs, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.

Considering `hello.c` source file of the previous example, to compile it in `CMake` you should create a *txt* file named `CMakeLists.txt`:

```
# Specify the minimum version for CMake

cmake_minimum_required(VERSION 2.8)

# Project's name

project(hello)
# Set the output folder where your program will be created
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})

# The following folder will be included
include_directories("${PROJECT_SOURCE_DIR}")
```

In this file we used the following global variables:

- `CMAKE_BINARY_DIR`: if you are building in-source, this is the same as `CMAKE_SOURCE_DIR`, otherwise this is the top level directory of your build tree

- `CMAKE_SOURCE_DIR`: this is the directory, from which cmake was started, i.e. the top level source directory

- `EXECUTABLE_OUTPUT_PATH`: set this variable to specify a common place where CMake should put all executable files (instead of `CMAKE_CURRENT_BINARY_DIR`)

- `EXECUTABLE_OUTPUT_PATH`: set this variable to specify a common place where `CMake` should put all executable files (instead of `CMAKE_CURRENT_BINARY_DIR`), for example `SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)`. `LIBRARY_OUTPUT_PATH`: set this variable to specify a common place where CMake should put all libraries

- `PROJECT_SOURCE_DIR`: contains the full path to the root of your project source directory, i.e. to the nearest directory where `CMakeLists.txt` contains the `PROJECT()` command.

Finally, to compile the source code you should add this final line to your `CMakeLists.txt`:

```
add_executable(hello ${PROJECT_SOURCE_DIR}/hello.c)
```

Now you are ready to compile the `hello.c` source file. At this point, you will have the folder with the following files:

```
$ ls
$ CMakeLists.txt hello.c
```

The common way to compile with `CMake` tools, is to create a temporary folder in which all the compilation file are put. In this way, you can delete the temporary compilation file to share the entire folder of your project:

```
$ mkdir build && cd build
$ cmake ..
$ make
```

Sometimes, you should need to install custom libraries in your system. This is particularly useful when you want to use functions and headers in different source files. In `CMake` the `install` command is used. This command generates installation rules for a project. Rules specified by calls to this command within a source directory are executed in order during installation.

```
#libs
add_library(mylib SHARED ${LIB_SRC})
#install
install(TARGETS mylib DESTINATION /usr/lib)
install(FILES ${LIB_HEADER} DESTINATION /usr/include/mylib)
```

In this case, after the `make` command, to perform the install step you should type the following command:

```
$ sudo make install
```

2.3 git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Today, Git is the most widely used modern version control system in the world.

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. So ideally, we can place any file in the computer on version control. A Version Control System (VCS) allows you to revert files back to a previous state, revert the entire project back to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also means that if you screw things up or lose files, you can generally recover easily. The most famous

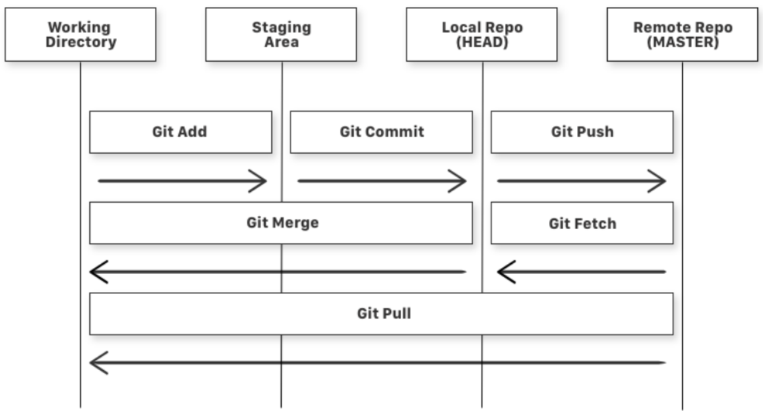


Figure 2.2: Git workflow

implementation of Git is github (`www.github.com`). In this section we introduce the basic concept of Git. First of all, the *Remote Repository* is where you send your changes when you want to share them with other people, and where you get their changes from, while the *Development Environment* is what you have on your local machine: the three parts of it are your *Working Directory*, the *Staging Area* and the *Local Repository.* The workflow of git among these directories is shown in Fig. 2.2. First of all, what is a repository? A repository is nothing but a collection of source code. If you consider a file in your Working Directory, it can be in three possible states. The following commands are used to manage the workflow:

- `git add` is a command used to add a file that is in the working directory to the staging area.

- `git commit` is a command used to add all files that are staged to the local repository.

- `git push` is a command used to add all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.

- `git fetch` is a command used to get files from the remote repository to the local repository but not into the working directory.
- `git merge` is a command used to get the files from the local repository into the working directory.
- `git pull` is command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge .

As for the main commands to start with git, you can follow these commands:

- Create a new repository:

  ```
  $ git init
  ```

- Checkout a repository: create a working copy of a local repository by running the command

  ```
  $ git clone /path/to/repository
  ```

- You can propose changes (add it to the Index) using:

  ```
  $ git add <filename> or git add *
  ```

- To actually commit these changes use:

  ```
  $ git commit -m "Commit message"
  ```

- Your changes are now in the `HEAD` of your local working copy. To send those changes to your remote repository, execute:

  ```
  $ git push origin master
  ```

  Change `master` to whatever branch you want to push your changes to.

- If you have not cloned an existing repository and want to connect your repository to a remote server, you need to add it with:

  ```
  $ git remote add origin <server>
  ```

- branching: branches are used to develop features isolated from each other. The master branch is the "default" branch when you create a repository. Use other branches for development and merge them back to the master branch upon completion. To create a new branch named *branch* and switch to it using:

```
$ git checkout -b branch
```

To switch back to master

```
$ git checkout master
```

And delete the branch again

```
$ git branch -d branch
```

A branch is not available to others unless you push the branch to your remote repository

```
$git push origin <branch>
```

- To update your local repository to the newest commit and fetch and merge remote changes in your working directory, execute:

  ```
  $ git pull
  ```

  while, to merge another branch into your active branch (e.g. master), use:

  ```
  $ git merge <branch>
  ```

Typically, a repository can be private or public. In the first case the owner and the developers of the project are the solely ones that can see the project and download its source code. In this context, you need to authenticate during the `clone` operations. This can be done in two different ways: with `https` authentication: just inserting user name and password of your account or using `SSH` authentication. Using the `SSH` protocol, you can connect and authenticate to remote servers and services. After you've checked for existing SSH keys, you can generate a new SSH key to use for authentication, then add it to the ssh-agent. To configure your GitHub account to use your new (or existing) SSH key, you'll also need to add it to your GitHub account.

3

Starting with ROS programming (Part 1)

In this lesson, we discuss two ROS packages implementing the communication protocols available in ROS: the publish-subscribe and the service. For each package, we also show some useful command line tool used in ROS to handle the execution of ROS nodes.

3.1 Environment configuration

You need to install ROS in your system before to start programming with it. In our lessons we use ROS `Noetic`, however similar steps can be follow to install other ROS distributions.

The first step is the setup your computer to accept software from `packages.ros.org`.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80'
--recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Then you should update your APT repository:

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

Finally, you are ready to install ROS. There are many different libraries and tools in ROS. The full version of ROS includes all the packages commonly used in robotic programming and can be installed with the following command:

```
$ sudo apt install ros-noetic-desktop-full
```

This step will take some time. Moreover, the latter command doesn't install all the packages available in ROS. To find additional packages, use:

```
$ apt-cache search ros-noetic
```

Now ROS is installed in your system. However, you are not able to used ROS command yet. In fact, at this point if you try to run a ROS command like `roscd` in your linux shell, you will get the following error:

```
$ Command 'roscd' not found, did you mean:

$ command 'rosco' from deb python-rosinstall

$ Try: sudo apt install <deb name>
```

This happen because the ROS environment is not correctly configured. By default, ROS in installed in the following directory:

```
$ /opt/ros/${ROS_VERSION}
```

To properly load the environment you have to `source` the setup file included in the installation directory of ROS.

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

Now you should be able to use the ROS commands. However, with this setup the user workspace is set to a directory owned by the super user (i.e. `/opt/ros/noetic/share`). So, before to continue, you should create your own workspace in the user space:

```
$ cd ~
$ mkdir -p ros_ws/src
$ cd ros_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
```

The `catkin_make` is the compilation command. Now you have created a ROS workspace called `ros_ws`. If you check the content of this directory, it contains three folders: `build`, `devel` and `src`. In the `src` directory you must create or download new ROS packages. If a package is not placed there, it will not be compiled. The `build` directory instead contains the compilation file, while the `devel` folder contains the compiled libraries.

To set `ros_ws` workspace as the default workspace you need to source the `setup.bash` file contained in the `devel` folder. To source this file automatically when a new linux shell is opened, you could be put the source command `bashrc` file:

```
$ echo "source ~/ros_ws/devel/setup.bash" >> ~/.bashrc
```

To test if everything is properly configure, you could try to enter the `roscd` command in a linux shell. If everything is right, this command will move you in the `devel` folder of your ROS workspace.

3.2 Create a ROS package

As already stated, all ROS packages, either created from scratch or downloaded from other code repositories, must be placed in the src folder of the ROS workspace, otherwise they can not be recognized by the ROS system and compiled.

To create a ROS package, switch to the catkin workspace `src` folder and create the package, using the following command:

```
$ catkin_create_pkg package_name [dependency1] [dependency2]
```

Try to create a simple node implementing the publish/subscribe communication protocol. Call this package: `ros_topic`:

```
$ catkin_create_pkg ros_topic roscpp std_msgs
```

As dependencies, we specified the following:

- `roscpp`: This is the C++ implementation of ROS. It is a ROS client library which provides APIs to C++ developers to make ROS nodes with ROS topics, services, parameters, and so on. We are including this dependency because we are going to write a ROS C++ node. Any ROS package which uses the C++ node must add this dependency.
- `std_msgs`: This package contains basic ROS primitive data types, such as integer, float, string, array, and so on. We can directly use these data types in our nodes without defining a new ROS message.

After ran this command, a new directory appears in your ROS workspace. A typical structure of an ROS package is shown in Fig. 3.1.

- `config`: All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and it is a common practice to name the folder config to keep the configuration files in it.
- `include/package_name`: This folder consists of headers and libraries that we need to use inside the package.
- `script`: This folder keeps executable Python scripts. In the block diagram, we can see two example scripts.
- `src`: This folder stores the C++ source codes.

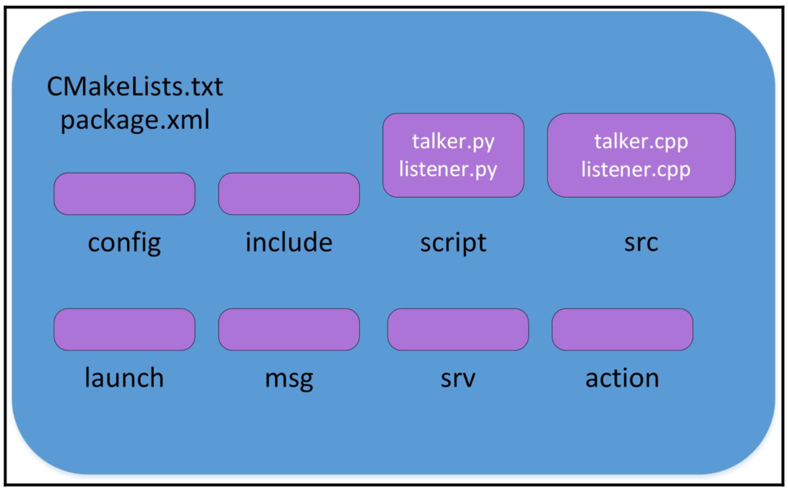


Figure 3.1: Structure of a typical ROS package

- `launch`: This folder keeps the launch files that are used to launch one or more ROS nodes.
- `msg`: This folder contains custom message definitions.
- `srv`: This folder contains the services definitions.
- `action`: This folder contains the action files.
- `package.xml`: This is the package manifest file of this package. In particular, this file defines properties about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages.
- `CMakeLists.txt`: This files contains the directives to compile the package.

After that the `ros_topic` has been created, you should be able to use the first ROS command introduced here, the `roscd`. In particular, `roscd` command is used to change the current directory using a package name or a special location. If we give the argument a package name, it will switch to that package folder (i.e. `~/ros_ws/devel`). Consider that after created or downloaded a new package in your workspace, you should inform the ROS system about it, updating the ROS filesystem using the command:

```
$ rospack profile
```

Now you can try to reach the ROS package folder using the following command:

```
$ roscd ros_topic
```

If everything is working properly, you will be move in the directory of the package:

```
$ ~/ros_ws/src/ros_topic
```

After that the ROS package has been successfully created, we can start adding nodes to it. A ROS package can contain multiple ROS nodes. We will create two nodes, one to publish a topic and one to subscribe to a topic. Let's start with the publisher one.

To create a new node in the `ros_topic` package, move in the `src` directory of the package and create an empty source file:

```
$ roscd ros_topic/src
$ touch ros_publisher.cpp
```

The aim of this node is to publish an integer value on a topic called `/numbers`.

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

int main(int argc, char **argv) {
ros::init(argc, argv,"ros_topic_publisher");
ros::NodeHandle nh;
ros::Publisher topic_pub =
nh.advertise<std_msgs::Int32 >("/numbers",10);
ros::Rate rate(10);
int count = 0;
while (ros::ok()) {
std_msgs::Int32 msg;
msg.data = count++;
ROS_INFO("%d",msg.data);
topic_pub.publish(msg);
rate.sleep();
}
return 0;
}
```

In the above code, we firstly include the header files needed to use ROS api (`roscpp`): the `ros/ros.h` is the main header of ROS, while the `std_msgs/Int32.h` is the standard message definition of the integer datatype. In order to initialize a ROS node with a given name, we used the following code line:

```
ros::init(argc, argv,"ros_topic_publisher");
```

This line is mandatory for all ROS nodes, otherwise will be impossible to use all ROS api functions. The name provided in the `init` function should be unique and will be used by the ROS master to handle it. Each ROS node typically has a `NodeHandle`, an object used to communicate with the whole ROS system. To declare it we use the following line of code:

```
ros::NodeHandle nh;
```

Then, we can create the object representing the topic publisher:

```
ros::Publisher topic_pub = nh.advertise
<std_msgs::Int32>("/numbers", 10);
```

This will create a topic publisher and name the topic `/numbers` with a message type `std_msgs::Int32`. The second argument is the buffer size. It indicates how many messages need to be put in a buffer before sending. It should be set to high if the data sending rate is high is used to set the frequency of sending data.

Another important feature of ROS is represented the `ros::Rate` object. This object is used to run loops at a desired frequency. Note that the `Rate` takes into account the elapsed time between the end of the previous loop and the new loop. When you create the `Rate` object you specify also the desired loop rate (in `Hetz`). To create an infinite while loop, we exploit the `ros::ok()` function, that returns zero when Ctrl+C is pressed. The message that we want to publish is a `std_msgs::Int32`, so, after declared it we fill its data field. How is composed a `std_msgs::Int32`? If you want to know how a message is composed, you can used the `rosmsg` command. The inbuilt tools called `rosmsg` is used to get information about ROS messages. Here are some parameters used along with `rosmsg`:

```
$ rosmsg show [message]: This shows the message description
$ rosmsg list: This lists all messages installed in your system
$ rosmsg md5 [message] : This displays md5sum of a message
$ rosmsg package [package_name] : This lists messages
in a package
```

So, considering our initial aim, to show how is composed the `std_msgs::Int32` message, we can use the `rosmsg show` command. So open a new terminal and insert the following command:

```
$ rosmsg show std_msgs/Int32

Output:
$ int32 data
```

So to fill the contents of `std_msgs::Int32` message, you need to refer to the data field.

To publish the message to the `/numbers` topic we use the method `publish`, who takes as input the message to broadcast on your topic.

Compile and run a ROS node

To compile a ROS package you need to edit its `CMakeLists.txt` file. In particular, we have to inform the building tool about what source file must be compiled and its dependencies. To compile the ROS publisher node, add the following lines at the end of the `CMakeLists.txt`:

```
#This will create executables of the nodes
add_executable(topic_publisher src/ros_publisher.cpp)

#This will link executables to the appropriate libraries
target_link_libraries(topic_publisher ${catkin_LIBRARIES})
```

At this point, you can use the `catkin_make` command to build the package. We can first switch to a workspace:

```
$ cd ~/ros_ws
```

Build `ros_topic` package as follows:

```
$ catkin_make
```

Consider that the `catkin_make` command compiles all the package in your workspace. Sometimes, you could have draft versions of other package that bring to compilation errors or unsatisfied dependencies. In this way the compilation could fail. To compile only one package and not the entire workspace you can use the `DCATKIN_WHITELIST_PACKAGES` argument. With this option, it is possible to set one or more packages enabled to be compiled.

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="pkg1,pkg2,..."
```

Note that is necessary to revert this configuration to compile other packages not specified in the `WHITELIST`. In fact, after set the `WHITELIST` it will remain saved in your system configuration, and you can directly execute the `catkin_make` command to compile only the packages specified in the `WHITELIST`. Differently, to bring again the list to the initial configuration you can use the following command:

```
$ catkin_make -DCATKIN_WHITELIST_PACKAGES=""
```

Now you are ready to run the publisher node. First of all, to execute ROS nodes you must activate a `roscore` in your system. To do this, on a linux terminal, run the following command:

```
$ roscore
```

This command runs the ROS master node on your local machine. Consider that this command locks your terminal, so to run other commands you need to open other linux shells.

To run the publisher node, you can use the `rosrun` command:

```
$ rosrun ros_topic topic_publisher
```

The output on the linux shell shows the `INFO` about the integer that are going to be published on `/numbers` topic. We can use two additionally commands to debug and understand the working of the nodes: `rosnode` and `rostopic`.

```
$ rosnode info [node_name]: This will print the
information about the node
$ rosnode kill [node_name]: This will kill a
running node
$ rosnode list: This will list the running nodes
$ rosnode machine [machine_name] : This will list
the nodes running on a particular
machine or a list of machines
$ rosnode ping: This will check the connectivity of a node
$ rosnode cleanup: This will purge
the registration of unreachable nodes
```

In particular, the output of `rosnode info /ros_topic_publisher` will provide information about the published and subscribed topics:

```
Node [/ros_topic_publisher]
Publications:
* /numbers [std_msgs/Int32]
* /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
* /ros_topic_publisher/get_loggers
* /ros_topic_publisher/set_logger_level

contacting node http://jcacace-Inspiron-7570:43001/ ...
Pid: 19478
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound
* transport: TCPROS
```

This is useful to understand the input and output of a node. As for the `rostopic` command, it can be used to get information about ROS topics. Here is the syntax of this command:

```
$ rostopic bw /topic: This command will display
the bandwidth used by the given topic.
```

```
$ rostopic echo /topic: This command will
print the content of the given topic in a human
readable format. Users can use the "-p"
option to print data in a csv format.
$ rostopic find /message_type: This command will
find topics using the given message type.
$ rostopic hz /topic: This command will display
the publishing rate of the given topic.
$ rostopic info /topic: This command will print
information about an active topic.
$ rostopic list: This command will list all active
topics in the ROS system.
$ rostopic pub /topic message_type args: This
command can be used to publish a value to a
topic with a message type.
$ rostopic type /topic: This will display the
message type of the given topic.
```

We can use these commands on the output of our publisher node. In particular, check which kind of topics are published by the node:

```
$ rostopic list

Output:
/numbers
/rosout
/rosout_agg
```

And check its content:

```
$ rostopic echo /numbers

Output:
data: 609
---
data: 610
---
data: 611
```

Before to discuss the subscriber node, let's introduce additional ROS graphical tools. In the first versions of ROS only few graphical tools were considered to plot data or display the connections between ROS nodes. In 2012, the first version of ROS including the `rqt` graphical interface has been released.

`rqt` is a software framework that implements the various GUI tools in the form of plugins. One can run all the existing GUI tools as dockable windows within rqt. The tools can still run in a traditional standalone method, but rqt makes it easier to manage all the various windows on the screen at one moment. To start `rqt` just type this command in your linux shell:

```
$ rqt
```

This command will open a new windows, as depicted in Fig. 3.2. In the

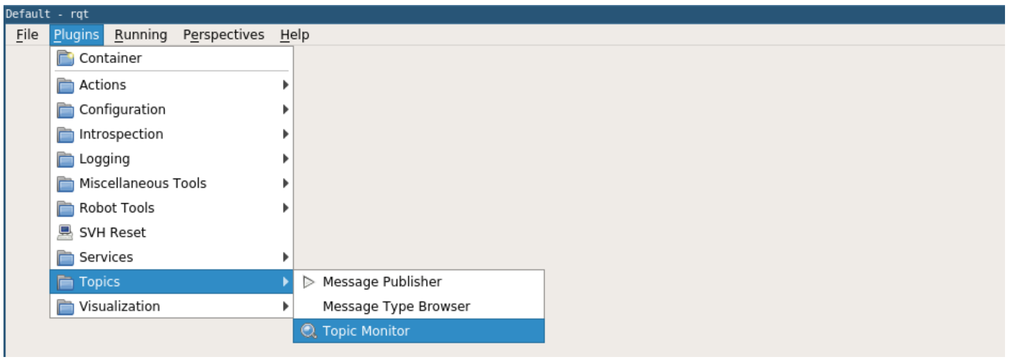
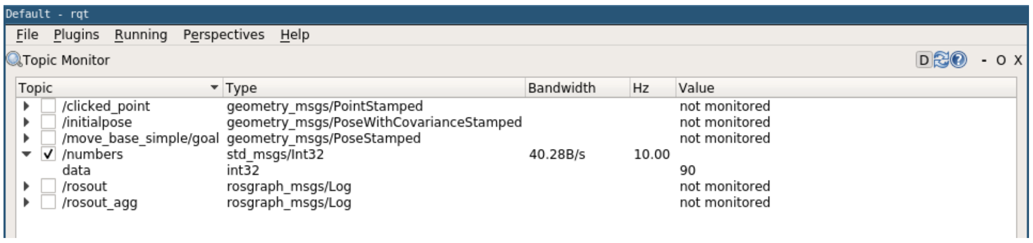


Figure 3.2: rqt window.

`rqt` start window you can load any desired plugin present in your system. You can also add custom plugins. Try to use the `Topic Monitor` plugin to inspect the data published on the `/numbers` topic. Open the plugin, and check the checkbox on this topic, as shown in Fig 3.3. As you can



Topic	Type	Bandwidth	Hz	Value
/clicked_point	geometry_msgs/PointStamped			not monitored
/initialpose	geometry_msgs/PoseWithCovarianceStamped			not monitored
/move_base_simple/goal	geometry_msgs/PoseStamped			not monitored
✓ /numbers	std_msgs/Int32	40.28B/s	10.00	
data	int32			90
/rosout	rosgraph_msgs/Log			not monitored
/rosout_agg	rosgraph_msgs/Log			not monitored

Figure 3.3: Topic monitor plugin.

see, we obtained the same result of `rostopic echo` command, but in a simpler/another way.

ROS Subscriber

After ran the publisher node, you can create a subscriber node that use the data published on `/numbers` topic.

Let's now create a new source code file called `ros_subscriber.cpp`.

```
$ roscd ros_topic/src
$ touch ros_subscriber.cpp
```

A sample code to read the `std_msgs::Int32` data is here reported.

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

class ROS_SUB {

public:
ROS_SUB();
void topic_cb( std_msgs::Int32ConstPtr data);
private:
ros::NodeHandle _nh;
ros::Subscriber _topic_sub;
};

ROS_SUB::ROS_SUB() {
_topic_sub = _nh.subscribe
("/numbers", 0, &ROS_SUB::topic_cb, this);

}

void ROS_SUB::topic_cb( std_msgs::Int32ConstPtr data) {
ROS_INFO("Listener: %d", data->data);

}

int main( int argc, char** argv ) {
ros::init(argc, argv, "ros_subscriber");
ROS_SUB rs;
ros::spin();
return 0;
}
```

Differently from the publisher node, in this example we use a class called `ROS_SUB`. In the constructor of the class, we declare a subscriber for the `std_msgs::Int32` data:

```
_topic_sub = _nh.subscribe ("/numbers", 0, &ROS_SUB::topic_cb, this);
```

In this line of code, we have to specify the name of the topic to read, the buffer and the callback function that receives the data. When you use class methods as subscribers, you must specify the class in were the method belong (`&ROS_SUB::topic_cb`) but also its context. In this case, we used `this`, which means that the subscriber will refer to the class it is part of. To update ROS topics the `ros::spin()` function is used. In particular, we have two functions that let all the callbacks get called for your subscriber: `ros::spin()` and `ros::spinOnce()`. The main differences between these two functions is that the first one is blocking function. The code after the `spin()` will never be executed. In addition, it will implement an infinite loop that makes your program alive over time.

As shown in the previous example, you can now modify the `CMakeLists.txt` file to add this new node (executable) and compile it with the `catkin_make` command. Now, you can launch both the nodes, the publisher and subscriber. To do this, type the following commands on three different linux shells:

```
$ roscore
$ rosrun ros_topic topic_publisher
$ rosrun ros_topic topic_subsciber
```

Now, you can use also the `rostopic` command to have more information about the connection between the publisher and subscriber.

```
rostopic info /numbers

Output:
Type: std_msgs/Int32

Publishers:
* /ros_topic_publisher (http://jcacace-Inspiron-7570:41703/)

Subscribers:
* /ros_subscriber (http://jcacace-Inspiron-7570:34901/)
```

This command provide information about the type of the message (`std_msgs::Int32`), but also the publisher (`ros_topic_publisher`) and the subscribers (`ros_subscriber`) active in the your ROS system.

Sometimes you just need to publish some data used to test a subscriber node (or send desired data like commanded velocity of similar). Of course, in such context implement a ROS node from scratch could be a waste of time. For this reason, we can directly publish a data using the command line tool:

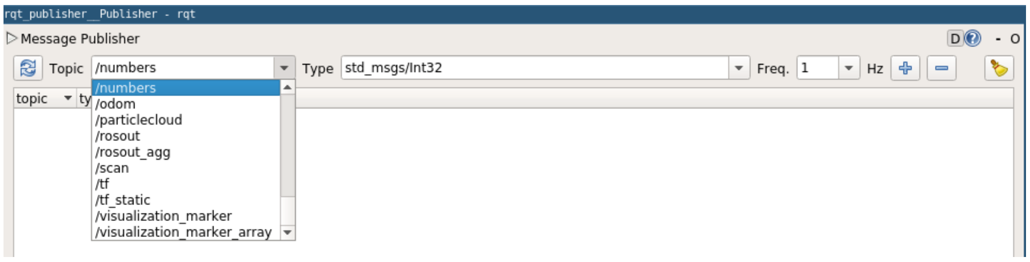
```
$ rostopic pub /numbers std_msgs::Int32 "data: 13" -r 10
```

This command publishes the number 13 on the topic `numbers` with a publishing rate of 10 Hz.

Again, try to use the graphical tools of ROS to obtain the same results of `rostopic pub` command. We can use the `rqt_publisher` plugin of the `rqt` interface. In this case, try to directly load this plugin from the command line.

```
$ rosrun rqt_publisher rqt_publisher
```

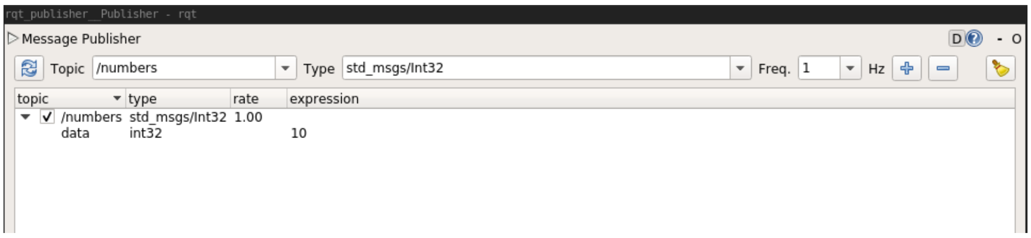
This command will open the `rqt` interface with the `rqt_publisher` plugin. As shown in Figs. 3.4 and 3.5, from this window you can select and add a topic to publish, specify its value and publishing rate and finally publish the desired value.



Topic: /numbers   Type: std_msgs/Int32   Freq.: 1 Hz

```
/numbers
/odom
/particlecloud
/rosout
/rosout_agg
/scan
/tf
/tf_static
/visualization_marker
/visualization_marker_array
```

Figure 3.4: rqt publisher plugin.



Topic: /numbers   Type: std_msgs/Int32   Freq.: 1 Hz

```
topic         type            rate   expression
▾ ✓ /numbers  std_msgs/Int32  1.00
    data      int32                  10
```

Figure 3.5: Publish on /numbers topic using rqt_publisher plugin.

3.3 ROS Service

In this section, we are going to create a new ROS package to implement ROS servic e protocol. The service nodes we are going to create can send a string message as a request to the server and the server node will send another message as a response. Differently from the example of previous section, in which the ROS publisher used a standard message already present in the installation of ROS (the `std_msgs::Int32`), in this case we have to define the service message exchanged between the client and the server. Let's start creating a new ROS package called `ros_service`

```
$ roscd && cd ..
$ cd src
$ catkin_create_pkg ros_service roscpp std_msgs
message_generation message_runtime
```

We add two additionally dependencies for this package, the `message_generation` and `message_runtime`, packages used to handle the building and run-time usage of custom messages.

Before to create the source code of the ROS nodes, let's add a custom `service` message. Create a new folder called `srv` in the package directory and add a `srv` file called `service.srv`. The definition of this file is as follows:

```
string in
---
string out
```

In this case, both the Request and Response filed of the service are strings. To use this service, we need to compile it. For this reason, you have to uncomment the following lines of the `CMakeLists.txt` file as shown here:

```
## Generate services in the 'srv' folder
add_service_files(
FILES
service.srv
)
```

and

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

After making these changes, we can build the package using `catkin_make` and using the following command, we can verify the procedure:

```
$ rossrv show ros_service/service
```

If we see the same content as we defined in the file, we can confirm it's working.

Now, let's create the service server and client. Move in the `src` folder of the `ros_service` package and create a new source file:

```
$ roscd ros_service/src
$ touch service_server.cpp
```

The content of the server is listed below:

```
#include "ros/ros.h"
#include "ros_service/service.h"
#include <iostream>
#include <sstream>

using namespace std;

```

```
bool service_callback
(ros_service::service::Request &req, ros_service::service::Response &res) {
std::stringstream ss;
ss << "Received Here";
res.out = ss.str();
ROS_INFO("From Client [%s], Server says
[%s]",req.in.c_str(),res.out.c_str());
return true;
}

int main(int argc, char **argv) {
ros::init(argc, argv, "service_server");
ros::NodeHandle n;
ros::ServiceServer service = n.advertiseService("service", service_callback);
ROS_INFO("Ready to receive from client.");
ros::spin();
return 0;
}
```

The `ros_service/service.h` header is a generated header, which contains our service definition and we can use this in our code. The server callback function is executed when a request is received on the server. The server can receive the request from clients with a message type of `ros_service::service::Request` and sends the response in the `ros_service ::service::Response` type. Finally, in order to offer the service, we need to include this line of code:

```
ros::ServiceServer service =
n.advertiseService("service", service_callback);
```

This code line instantiates a service called `service`, while the callback for it is a function called `service_callback`.

Like in the previous example, start this service and try to handle it using the commands provided by the ROS framework. Just add the compilation directives in the `CMakeLists.txt` file:

```
add_executable(service_server  src/service_server.cpp)
target_link_libraries(service_server ${catkin_LIBRARIES} )
```

Then, compile with `catkin_make` command:

```
$ roscd
$ cd ..
$ catkin_make
```

Finally, after ran a `roscore`, start the service:

```
$ roscore
$ rosrun ros_service service_server
```

At this point, we can check that the service instantiated in this node is active. Use the following command to inspect the service active in your ROS system:

```
$ rosservice list

Output:
/rosout/get_loggers
/rosout/set_logger_level
/service
/service_server/get_loggers
/service_server/set_logger_level
```

You can also call this service using the following command:

```
$ rosservice call /service "in_: 'Call'"

out: "Received Here"
```

The command `rosservice call` takes into account the name of the service to call and the list of arguments. In this case, it takes a string.

Let's now create the service client node.

```
$ roscd ros_service/src
$ touch service_client.cpp
```

The following code calls the service declared in the previous example:

```
#include "ros/ros.h"
#include <iostream>
#include "ros_service/service.h"
#include <iostream>
#include <sstream>

using namespace std;

int main(int argc, char **argv) {

ros::init(argc, argv, "service_client");
ros::NodeHandle n;
ros::Rate loop_rate(10);
ros::ServiceClient client =
n.serviceClient<ros_service::service>("service");
while (ros::ok()) {
```

```
ros_service::service srv;
std::stringstream ss;
ss << "Sending from Here";
srv.request.in = ss.str();
if (client.call(srv)) {
cout << "From Client:
["<<    srv.request.in << "],
Server says [" <<
srv.response.out << "]" << endl;
}
else {
ROS_ERROR("Failed to call service");
return 1;
}
ros::spinOnce();
loop_rate.sleep();
}
return 0;
}
```

To create a service client of the `service` type, we can use the following code line:

```
ros::ServiceClient client =
n.serviceClient<ros_service::service>("service");
```

While, to send the service call to the server we can use the following line:

```
if (client.call(srv))
```

This line returns true if the service is successfully called, false otherwise.

Create custom messages

Similarly to the ROS service message (`srv`), you can create custom messages. ROS already provides a comprehensive set of messages to handle several situation of robotic programming (i.e. `sensor_msgs`, `geometry_msgs`, `nav_msgs`, etc...). However, in some situation could be useful to define your own ROS messages. The message definitions are stored in a .msg file the `msg` folder of your package. Let's create a custom message in the `ros_topic` package:

```
$ roscd ros_topic
$ mkdir msg && cd msg
$ touch demo.msg
```

In this message we want to group together a string and an integer:

```
string name
int32 data
```

When the `ros_topic` package was created, we hadn't planned to add custom message. For this reason, we have to manually add the `message_generation` dependency in the `CMakeLists.txt`. Open this file adding the`message_generation` in the `find_package` command:

```
find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation
)
```

And decomment the following line and add the custom message file:

```
add_message_files(
FILES
demo.msg
)
generate_messages(
DEPENDENCIES
std_msgs
)
```

As usual, to use the added message, you have to compile the `ros_topic` package.

4

Working with ROS actionlib

In ROS services, the user implements a request/reply interaction between two nodes, but if the reply takes too much time or the server is not finished with the given work, we have to wait until it completes, blocking the main application while waiting for the termination of the requested action. In addition, the calling client could be implemented to monitor the execution of the remote process. In these cases, we should implement our application using `actionlib`. This is another protocol in ROS in which we can preempt the running request and start sending another one if the request is not finished on time as we expected. `Actionlib` packages provide a standard way to implement these kinds of preemptive tasks. It is is highly used in robot manipulation and mobile robot navigation. We can see how to implement an action server and action client implementation. There is another method in ROS in which we can preempt the running request and start sending another one if the request is not finished on time as we expected. Like ROS services, in `actionlib`, we have to specify the action data type. The action specification is stored inside the action file having an extension of `.action`. This file must be kept inside the action folder, that as usual must be placed inside the ROS package. The action file has the following parts:

- *Goal*: The action client can send a goal that has to be executed by the action server. This is similar to the request in the ROS service. For example, if a robot arm joint wants to move from 45 degrees to 90 degrees, the goal here is 90 degrees.

- *Feedback*: When an action client sends a goal to the action server, it will start executing a callback function. Feedback is simply giving the progress of the current operation inside the callback function. Using the feedback definition, we can get the current progress. In the preceding case, the robot arm joint has to move to 90 degrees; in this case, the feedback can be the intermediate value between 45 and 90 degrees in which the arm is moving.

- *Result*: After completing the goal, the action server will send a final result of completion, it can be the computational result or an acknowledgment. In the preceding example, if the joint reaches 90 degrees it achieves the goal and the result can be anything indicating it finished the goal.

Action client and server implement a state machine to manage the execution of the process. In particular, goals are initiated by an ActionClient. Once a goal is received by an ActionServer, the ActionServer creates a state machine (reported in Fig. 4.1) to track the status of the goal. The action server can

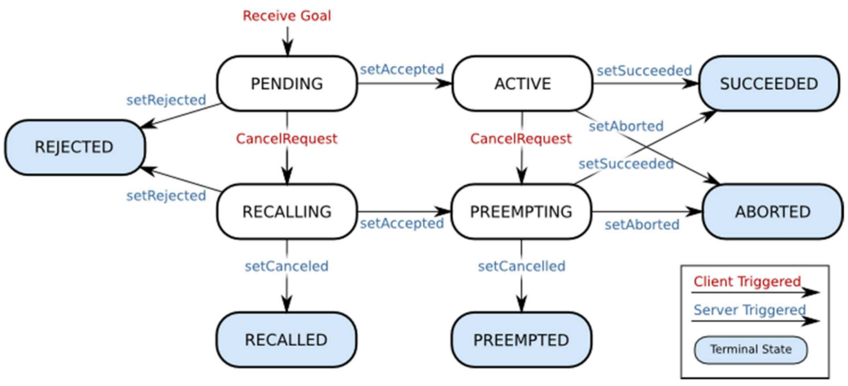


Figure 4.1: Action server state machine

be in the following states:

- *Pending*: The goal has yet to be processed by the action server.
- *Active*: The goal is currently being processed by the action server.
- *Recalling*: The goal has not been processed and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled
- *Preempting*: The goal is being processed, and a cancel request has been received from the action client, but the action server has not confirmed the goal is canceled.
- *Rejected*: The goal was rejected by the action server without being processed and without a request from the action client to cancel.
- *Succeeded*: The goal was achieved successfully by the action server.
- *Aborted*: The goal was terminated by the action server without an external request from the action client to cancel.
- *Recalled*: The goal was canceled by either another goal, or a cancel request, before the action server began processing the goal.

- *Preempted*: Processing of the goal was canceled by either another goal, or a cancel request sent to the action server.

We can discuss a demo action server and action client here. The demo action client will send a number as the goal. When an action server receives the goal, it will count from 0 to the goal number with a step size of 1 and with a 1 second delay. If it completes before the given time, it will send the result; otherwise, the task will be preempted by the client. The feedback here is the progress of counting. The action file of this task is as follows. Let's create another ROS package in order to implement such action. We call this package `ros_action`.

```
$ roscd
$ cd ../src
$ catkin_create_pkg ros_action roscpp std_msgs
actionlib actionlib_msgs
```

Now create the action folder to store the action files:

```
$ roscd ros_action
$ mkdir action && cd action
$ touch demo.action.
```

Edit the content of `demo.action` file as following:

```
#goal definition
int32 count
---
#result definition
int32 final_count
---
#feedback
int32 current_number
```

Here, the `count` value is the goal in which the server has to count from zero to this number. `final_count` is the result, in which the final value after completion of a task and `current_number` is the feedback value. It will specify how much the progress is.

Let's start to create the action server and client sources, creating in the source directory of the package two `cpp` files: `action_server.cpp` and `action_client.cpp`.

```
$ roscd ros_action/src
$ touch action_server.cpp
$ touch action_client.cpp
```

In the following the code of the server is reported:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <actionlib/server/simple_action_server.h>
#include "ros_action/demoAction.h"
#include <iostream>
#include <sstream>

class action_class {

private:
ros::NodeHandle nh_;
//NodeHandle instance must be created before this line.
Otherwise strange error may occur.
actionlib::SimpleActionServer<ros_action::demoAction> as;
//create messages that are used to published
//feedback/result
ros_action::demoFeedback feedback;
ros_action::demoResult result;

std::string action_name;
int goal;
int progress;

public:
action_class(std::string name) :
as(nh_, name, boost::bind
(&action_class::executeCB, this, _1), false),
action_name(name) {
as.registerPreemptCallback(
boost::bind(&action_class::preemptCB, this));
as.start();
}

void preemptCB(){
ROS_WARN("%s got preempted!", action_name.c_str());
result.final_count = progress;
as.setPreempted(result,"I got Preempted");
}

void executeCB(const ros_action::demoGoalConstPtr &goal) {
if(!as.isActive() || as.isPreemptRequested()) return;
ros::Rate rate(5);
ROS_INFO("%s is processing the goal %d",
action_name.c_str(), goal->count);
for(progress = 1 ; progress <= goal->count; progress++){
//Check for ros
if (!ros::ok()) {
result.final_count = progress;
as.setAborted(result,"I failed !");
ROS_INFO("%s Shutting down",action_name.c_str());
break;
}

if(!as.isActive() || as.isPreemptRequested()){
return;
}

if(goal->count <= progress) {
ROS_INFO("%s Succeeded at getting to goal %d",
action_name.c_str(), goal->count);
result.final_count = progress;
as.setSucceeded(result);
}
else {
ROS_INFO("Setting to goal %d /
%d",feedback.current_number,goal->count);
feedback.current_number = progress;
as.publishFeedback(feedback);
}
rate.sleep();
}
}
};

int main(int argc, char** argv) {
ros::init(argc, argv, "demo_action");
ROS_INFO("Starting Demo Action Server");
action_class demo_action_obj(ros::this_node::getName());
ros::spin();
return 0;
}
```

The salient parts of the code are explained. Start with the header inclusion. The first header is the standard action library to implement an action server node, while the second header is generated from the stored action files:

```
#include <actionlib/server/simple_action_server.h>
#include "ros_action/demoAction.h"
```

Than a class implementing the server definition, the action feedback and result is declared:

```
class action_class {
private:
ros::NodeHandle nh_;
actionlib::SimpleActionServer<ros_action::demoAction> as;
ros_action::demoFeedback feedback;
ros_action::demoResult result;
```

In order to make the action server available over the ROS system, our class must extend the action constructor, by taking an argument such as Nodehandle, the action name, and the server callback. Then, the callback in case o preemption must be specified. The preemtCB is the callback name executed when there is a preempt request from the action client.

```
action_class(std::string name) :
as(nh_, name, boost::bind
(&action_class::executeCB, this, _1), false),
action_name(name) {
as.registerPreemptCallback(boost::bind(&action_class::preemptCB, this
as.start();
}
```

The callback is executed when the action server receives a new goal and can terminate in different ways. The action could fail its execution, so we abort the action callback:

```
if (!ros::ok()) {
result.final_count = progress;
as.setAborted(result,"I failed !");
ROS_INFO("%s Shutting down",action_name.c_str());
break;
}
```

Differently, when the action is concluded we can set a `succeeded` its end state:

```
if(goal->count <= progress) {
ROS_INFO("%s Succeeded at getting to goal %d",
action_name.c_str(), goal->count);
result.final_count = progress;
as.setSucceeded(result);
}
```

Finally, in the main function of the action server we need to instantiate a class object:

```
action_class demo_action_obj(ros::this_node::getName());
```

As specified in the class definition, the input argument of the class constructor is the name of the action. In this case, we used the name of the node itself as action name.

Let's now start to create the action client. Edit the content of the `action_client.cpp` as following:

```
#include "ros/ros.h"
#include <iostream>
#include <actionlib/client/simple_action_client.h>
#include <actionlib/client/terminal_state.h>
#include "ros_action/demoAction.h"

int main (int argc, char **argv) {

ros::init(argc, argv, "demo_action_client");
if(argc != 3){
ROS_INFO("%d",argc);
ROS_WARN("Usage: demo_action_client <goal> <time_to_preempt_in_sec>");
return 1;
}

// create the action client
// true causes the client to spin its own thread
actionlib::SimpleActionClient<ros_action::demoAction> ac("demo_action", true)

ROS_INFO("Waiting for action server to start.");

//wait for the action server to start
ac.waitForServer(); //will wait for infinite time

ROS_INFO("Action server started, sending goal.");

//send a goal to the action
ros_action::demoGoal goal;
goal.count = atoi(argv[1]);

ROS_INFO("Sending Goal [%d] and Preempt time of [%d]",goal.count, atoi(argv[2
ac.sendGoal(goal);

//wait for the action to return
bool finished_before_timeout = ac.waitForResult(ros::Duration(atoi(argv[2])))
//Preempting task
ac.cancelGoal();

if (finished_before_timeout) {
actionlib::SimpleClientGoalState state = ac.getState();
ROS_INFO("Action finished: %s",state.toString().c_str());
//Preempting the process
ac.cancelGoal();
}
else
ROS_INFO("Action did not finish before the time out.");
//exit
return 0;
}
```

To declare the action client object we use the following line of code:

```
actionlib::SimpleActionClient<ros_action::demoAction>
ac("demo_action", true);
```

Then we ask the client to wait an infinite time if there is no action server running on the system. It will exit only when there is an action server running on the system:

```
ac.waitForServer();
```

Finally, we call the server of the action specifying the goal to reach:

```
ros_action::demoGoal goal;
goal.count = atoi(argv[1]);
ac.sendGoal(goal);
```

Finally, we required the client to preempt the action if the server doesn't terminate after a certain amount of time:

```
bool finished_before_timeout =
ac.waitForResult(ros::Duration(atoi(argv[2])));
```

As you can see from the source code, the goal and the time to wait the server termination are required as a command line input. We will see how to call this program in the following.

As usual to compile the client and server you should edit the `CMakeLists.txt` file. Other than the executable, we should require to the build tool to compile the action file as well. To do this, uncomment the related section of the make file:

```
## Generate actions in the 'action' folder
add_action_files(
FILES
demo.action
)
## Generate added messages and services with any dependencies listed
generate_messages(
DEPENDENCIES
std_msgs
actionlib_msgs
)
```

And add the executable file:

```
add_executable ( action_server src/action_server.cpp)
target_link_libraries(action_server ${catkin_LIBRARIES} )
add_executable ( action_client src/action_client.cpp)
target_link_libraries(action_client ${catkin_LIBRARIES} )
```

After `catkin_make`, we can run these nodes using the following commands:
Run `roscore`:

```
$ roscore
```

Launch the action server node:

```
$ rosrun ros_action action_server
```

Launch the action client node with two arguments (i.e. reach the number 13 in maximum 10 seconds):

```
$ rosrun ros_action action_client 13 10
```

4.0.1 ROS Action messages

Internally the `actionlib` communication works very similar to topic communication. In fact, several new topics appear after ran the server:

```
$ rostopic list

Output:
/demo_action/cancel
/demo_action/feedback
/demo_action/goal
/demo_action/result
/demo_action/status
```

These topics are directly published by the `actionlib` client. Each topic accepts a different part of the action definition, for example, the `/demo_action/goal` topic, accepts a `ros_action/demoActionGoal` message:

```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
actionlib_msgs/GoalID goal_id
```

```
time stamp
string id
ros_action/demoGoal goal
int32 count
```

4.0.2 Additional ROS tools

In this section a set of useful additional tool of ROS are discussed.

Bagfile

A `bag` is a file format used in ROS for storing ROS message data. Bags have an important role in ROS and a variety of tools have been written to allow you to store, process, analyze, and visualize them. It represents the main logging system for ROS data and can be used to save and later work on a stream of topic data. For example, if you are working with a camera sensor, you can just record the sensor output placed on its scene and work with the captured data without the hardware.

To create a new bagfile you can use the following command:

```
$ rosbag record [TOPICS] [OPTIONS]
```

In particular, you can choose the `-a` option to record all topics active in your system, while the option `-O` allows you to specify the bagfile name. If the name of the file is not specified, the current date is used to naming it. The complete list of `rosbag` options is available here: `http://wiki.ros.org/rosbag/Commandline`.

ROS parameter server

When programming a robot, we might have to define robot parameters, such as robot controller gains P, I, and D. ROS provides a parameter server, which is a shared server in which all ROS nodes can access parameters from this server. A node can read, write, modify, and delete parameter values from the parameter server. We can store these parameters in a file and load them into the server. The server can store a wide variety of data types and can even store dictionaries. The programmer can also set the scope of the parameter, that is, whether it can be accessed by only this node or all the nodes.

The `rosparam` tool is used to get and set the ROS parameter from the command line. To set a value in the given parameter:

```
$ rosparam set [parameter_name] [value]
```

To retrieve a value from the given parameter:

```
$ rosparam get [parameter_name]
```

In order to retrieve the value of a parameter from source code, in C++ you can use the `param` function:

```
int my_num;
nh.param("my_num", my_num, 13);
```

This function accepts as arguments the name of the parameter, the variable to fill with its value and a default value. The default value is used when the requested parameter is not present in the parameter server.

We can include these lines in a new version of our publisher node, who start to publish the integer starting from `my_num` value.

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

int main(int argc, char **argv) {
ros::init(argc, argv,"ros_topic_publisher");
ros::NodeHandle nh("~");
ros::Publisher topic_pub =
nh.advertise<std_msgs::Int32>("/numbers", 10);

int my_num;
nh.param("my_num", my_num, 42);

ros::Rate rate(10);
int count = my_num;
while (ros::ok()) {
std_msgs::Int32 msg;
msg.data = count++;
ROS_INFO("%d",msg.data);
topic_pub.publish(msg);
ros::spinOnce();
rate.sleep();
}
return 0;
}
```

After properly compiled, try to set a desired value for `my_int`.

Using ROS Launch files

Until now, we just used `rosrun` command to start a node. Another way to do this is using the `roslaunch` command using the ROS `launch` files.

Launch files are a very useful feature for launching more than one node. We can write all nodes inside an XML-based file called `launch` files parsing it with `roslaunch`. This command will also automatically start the ROS Master and the parameter server. So, in essence, there is no need to start the `roscore` command and individual node; if we launch the file, all operations are made in a single command.

Launch files can be also used to set ROS parameters. We can try to set the initial value for the ROS publisher using the `launch` file. Just create a new file in the `ros_topic` package (the common way is to put it into a sub-directory called `launch`) and paste the following contents.

```
<?xml version="1.0" ?>

<launch>
<node pkg="ros_topic" name="ros_topic_publisher"
type="topic_publisher_param" output="screen">

<param name="my_num" value="13" type="int" />

</node>
</launch>
```

In this file we require to start a new node represented by the executable: `topic_publisher_param`, from package `ros_topic`. As param for this node we specified the value of `my_num`, similarly to the previous example, in which this param has been set using the command line tool.

Catkin build

A different way to compile ROS workspace is using `catkin build`. To install it you can use `APT`:

```
$ sudo apt-get install python-catkin-tools
```

While, to build a specific package:

```
$ catkin build <target_package>
```

The main advantages of `catkin build` are that you can call it from everywhere and always build packages seperately. In addition, `catkin clean` cleans everything. Differently, with `catkin_make` the same operation is done deleting the build and devel directories. However, you can't mix `catkin build` and `catkin_make`.

5

Robot Modeling

5.0.1 starting with robot modeling

In previous lessons of Robotics Lab course we just discuss about the capabilities of ROS showing initial examples to use its plumbing features. In this lesson instead, we start to see how ROS is so much useful in robot programming when you need functionalities already implemented in other ROS packages. In particular, several robotic software need for the knowledge about the kinematic and dynamic structure of the robot to work. For example, to solve problems like collision checking, forward or inverse kinematics and so on, automatic system need to know how the robot is composed and configured.

Nevertheless, robot modeling is important also for visualization and simulation. In this context, graphically display the configuration of a robot in its environment helps developers to debug their applications. Finally, simulation engines (like the one used in this course: Gazebo) use robot models to spawn a simulated robot into the environment.

The aim of this lesson is twofold:

- Understand how robot models are used in robot programming
- Learn how to create or improve a robot model and interfacing its with other ROS packages

Consider that designing the model of a robot from scratch is not easy, since its kinematic chain could be quite complex. In addition, you need a way to properly characterize its dynamics. Automatic model generator tools exist that start from the 3d CAD model of the robot to generate a model file. However, understand how a robot model file is typically composed appears to be extremely useful when you want to adjust it or add additional elements to its model, like a custom gripper or sensors.

The most popular file format to describe the model of a robot is called **URDF** (Unified Robot Description Format) and its also the one officially supported by ROS.

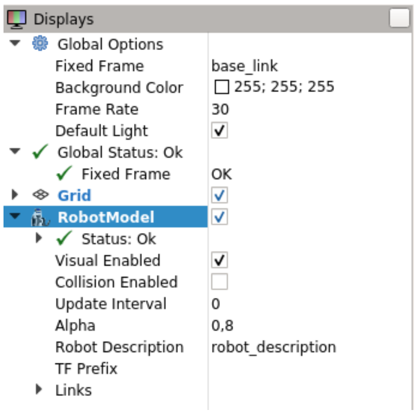
5.0.2 RViz with robot model

Before starting to see how robot model are composed, let's discuss some tools used to interact with such models.

The first tool discussed here is called RViz (ROS Visualization). As already said, this tool is very useful in ROS because is able to show information available in ROS system to a graphical way. You can start `RViz` as a ros node:

```
$ rosrun rviz rviz
```

After ran this command, if the `roscore` is active a new window is opened.



```
Displays
▾ Global Options
    Fixed Frame            base_link
    Background Color       ☐ 255; 255; 255
    Frame Rate             30
    Default Light          ✓
▾ ✓ Global Status: Ok
    ✓ Fixed Frame          OK
▸ Grid                     ✓
▾ RobotModel               ✓
  ▸ ✓ Status: Ok
    Visual Enabled         ✓
    Collision Enabled      ☐
    Update Interval        0
    Alpha                  0,8
    Robot Description      robot_description
    TF Prefix
  ▸ Links
```

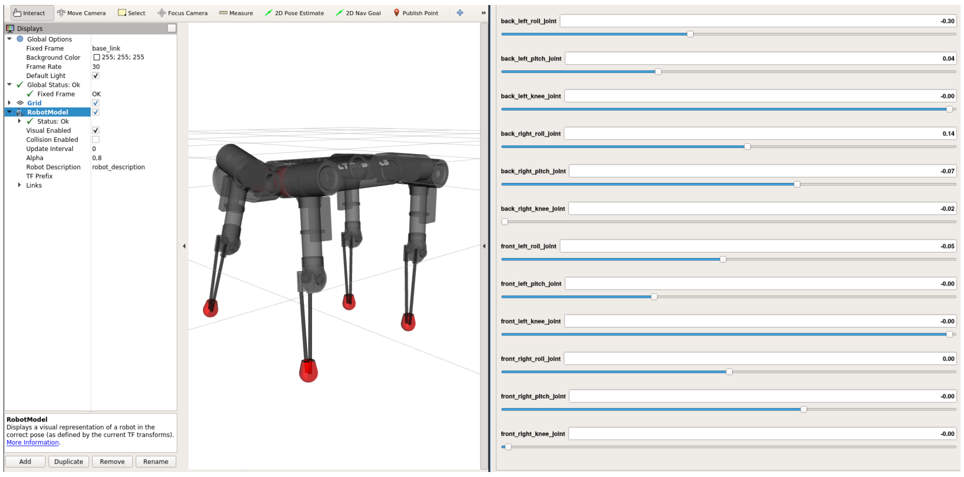
Figure 5.1: RViz display panel

The most useful panel of RViz is the display panel (show in Fig. 5.1). In this panel you can configure the plugin loaded in your system that are responsible to display in RViz window. In particular, in the Global Options section, you must select the fixed frame in which the information are displayed. Be careful, if the fixed frame or the link between this frame and the information to display doesn't exist an error message is reported.

Regarding the model of the robot, RViz is able to display the model of the robot and move its joints as well as specified in he model file, using the plugin `RobotModel`. In this lesson we will see how to obtain a result similar to the one shown in Fig. 5.2. In this figure you can also see that is possible to modify the position of each joint of the robot considering the joint limit specified in its model. In this case we are not applying any controller, but just change the robot configuration in the visualization.

5.1 Robot modeling using URDF

The most important package to model robot using ROS is the `URDF` package. This package contains a C++ parser for the Unified Robot Description



```
Global Options
  Fixed Frame           base_link
  Background Color      255; 255; 255
  Frame Rate            30
  Default Light         ✓
Global Status: Ok
  Fixed Frame           OK
Grid                    ✓
RobotModel              ✓
  Status: Ok
  Visual Enabled        ✓
  Collision Enabled
  Update Interval       0
  Alpha                 0,8
  Robot Description     robot_description
  TF Prefix
  Links
```

RobotModel
Displays a visual representation of a robot in the correct pose (as defined by the current TF transforms). More Information.

```
back_left_roll_joint      -0.30
back_left_pitch_joint      0.04
back_left_knee_joint      -0.00
back_right_roll_joint      0.14
back_right_pitch_joint    -0.07
back_right_knee_joint     -0.02
front_left_roll_joint     -0.05
front_left_pitch_joint    -0.00
front_left_knee_joint     -0.00
front_right_roll_joint     0.00
front_right_pitch_joint   -0.00
front_right_knee_joint    -0.00
```

Figure 5.2: RobotModel plugin in RViz.

Format (`URDF`), which is an XML file representing the robot model thanks to a set of rules that specify the elements of the robot. In particular, `URDF` represents the kinematic and dynamic description of the robot, the visual representation of the robot, and the collision model of the robot.

The following tags are the commonly used URDF tags to compose a URDF robot model:

- *robot*: This tag encapsulates the entire robot model that can be represented using URDF. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot. As shown in Fig. 5.3, a robot model consists of connected links and joints.

```
<robot name="<name of the robot>"
<link> ..... </link>
<link> ...... </link>
<joint> ....... </joint>
<joint> ........</joint>
</robot>
```

- *link*: The link tag represents a single link of a robot. Using this tag, we can model a robot link and its properties. The modeling includes the size, the shape, and the color, and it can even import a 3D mesh to represent the robot link. We can also provide the dynamic properties of the link, such as the inertial matrix and the collision properties

```
<link name="<name of the link>">
<inertial >...........</inertial>
<visual> ............</visual>
<collision >.........</collision>
</link>
```

The Visual section represents the real link of the robot, and the area surrounding the real link is the Collision section. The Collision section encapsulates the real link to detect collision before hitting the real link.

- *joint*: The joint tag represents a robot joint. We can specify the kinematics and the dynamics of the joint, and set the limits of the joint movement and its velocity. The joint tag supports the different types of joints, such as revolute, continuous, prismatic, fixed, floating, and planar.

```
        <joint name="<name of the joint>">
        <parent link="link1"/>
        <child link="link2"/>
        <calibration .... />
        <dynamics damping ..../ >
        <limit effort .... />
        </joint>
```

  A joint is formed between two links; the first is called the Parent link, and the second is called the Child link.

In the next lessons we will see how to include additional tags to specify the sensors of the robot.

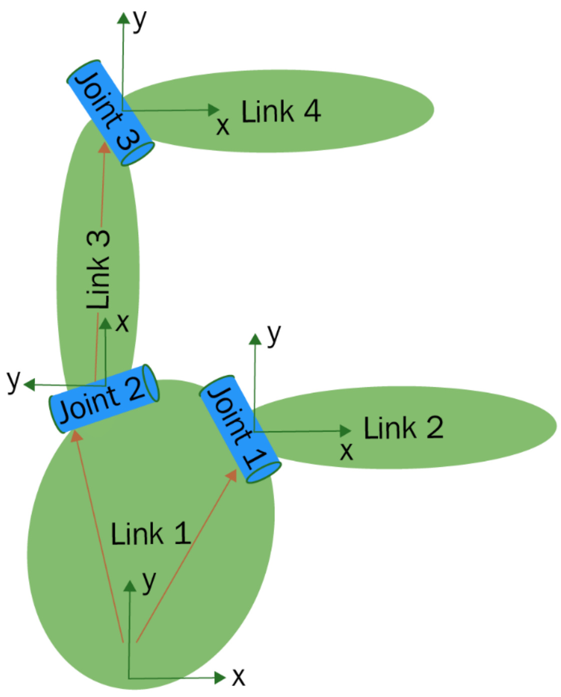


Figure 5.3: Robot model represented by the URDF.

5.1.1 Pan-tilt robot model

Before creating the URDF file for the robot, let's create an ROS package in the ROS workspace using the following command:

```
$ catkin_create_pkg rl_robot_description_pkg roscpp tf geometry_msgs urdf rviz xacro
```

The package mainly depends on the urdf and xacro packages. If these packages have not been installed on to your system, you can install them using the package manager:

```
$ sudo apt-get install ros-$ROS_DISTRO-urdf
$ sudo apt-get install ros-$ROS_DISTRO-xacro
```

We can create the urdf file of the robot inside this package and create launch files to display the created urdf in RViz. Before creating the urdf file for this robot, let's create `urdf` and `launch` folders inside our package. The urdf folder can be used to keep the model files that we are going to create.

The first robot mechanism that we are going to design is a pan-and-tilt mechanism, as shown in Fig. 5.4.

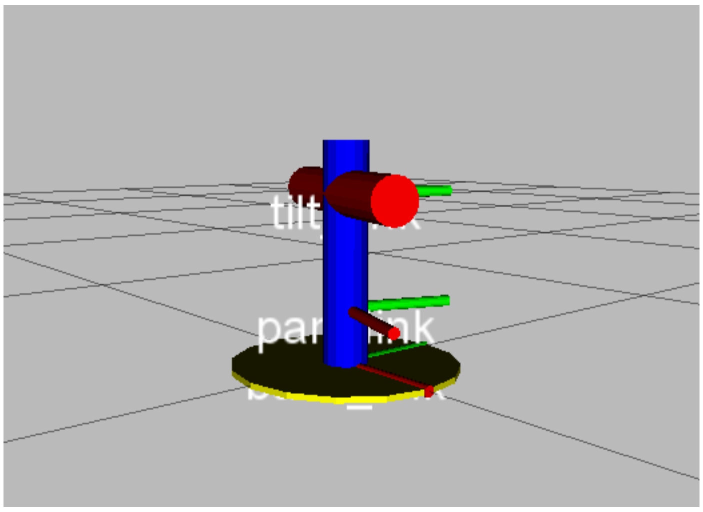


Figure 5.4: Pan tilt mechanism.

The following `tag` declare a new RobotModel called *pan_tilt*.

```
<?xml version="1.0"?>
<robot name="pan_tilt">
```

The first link of the robot is the `base_link`. Typically, the `base_link` is the fixed frame of the robot and it represents the first link of the kinematic chain. In this example, the shape of the robot is drawn using basic 3d shape (a cylinder). In other cases, the geometry and its collision is directly get by the CAD model of the robot part.

```
<link name="base_link">
<visual>
<geometry>
<cylinder length="0.01" radius="0.2"/>
</geometry>
<origin rpy="0 0 0" xyz="0 0 0"/>
<material name="yellow">
<color rgba="1 1 0 1"/>
</material>
</visual>
<collision>
<geometry>
<cylinder length="0.03" radius="0.2"/>
</geometry>
<origin rpy="0 0 0" xyz="0 0 0"/>
</collision>
<inertial>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0"
iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
```

We need to connect two links of the robot. The links are connected using a joint. The first joint to specify is the pan joint (rotation around the z axis). The main elements of the `joint` tag are the parent and the child links that specify which links must be connected. The type of the joint specifies its behavior in the kinematic chain. The rotation angle of this joint is specified with the `axis` tag.

```
<joint name="pan_joint" type="revolute">
<parent link="base_link"/>
<child link="pan_link"/>
<origin xyz="0 0 0.1"/>
<axis xyz="0 0 1" />
<limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
<dynamics damping="50" friction="1"/>
</joint>
```

Link the `base_link` the pan link is represented by a cylinder.

```
<link name="pan_link">
<visual>
<geometry>
<cylinder length="0.4" radius="0.04"/>
</geometry>
```

```
<origin rpy="0 0 0" xyz="0 0 0.09"/>
<material name="red">
<color rgba="0 0 1 1"/>
</material>
</visual>
<collision>
<geometry>
<cylinder length="0.4" radius="0.06"/>
</geometry>
<origin rpy="0 0 0" xyz="0 0 0.09"/>
</collision>
<inertial>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
```

The rest of the model file is similar to its first part. Another rotational join (rotating around the y axis) and a final link.

```
<joint name="tilt_joint" type="revolute">
<parent link="pan_link"/>
<child link="tilt_link"/>
<origin xyz="0 0 0.2"/>
<axis xyz="0 1 0" />
<limit effort="300" velocity="0.1" lower="-4.64" upper="-1.5"/>
<dynamics damping="50" friction="1"/>
</joint>
```

```
<link name="tilt_link">
<visual>
<geometry>
<cylinder length="0.4" radius="0.04"/>
</geometry>
<origin rpy="0 1.5 0" xyz="0 0 0"/>
<material name="green">
<color rgba="1 0 0 1"/>
</material>
</visual>
<collision>
<geometry>
<cylinder length="0.4" radius="0.06"/>
</geometry>
<origin rpy="0 1.5 0" xyz="0 0 0"/>
</collision>
```

```
<inertial>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
</robot>
```

In an `URDF` file the name of the joints and link must be unique. Since this file is only interpreted (and not compiled) any error in its format will be reported only when the model of the robot is loaded. Actually, there are some tools to check and debug the `URDF` file, link the `check_urdf` command. This command accepts as input the file to check. In our case:

```
$ roscd rl_robot_description_pkg/urdf/
$ check_urdf pan_tilt.urdf
```

This is the output of the command:

```
robot name is: pan_tilt
---------- Successfully Parsed XML ---------------
root Link: base_link has 1 child(ren)
child(1):  pan_link
child(1):  tilt_link
```

Using this command you can check if the kinematic chain defined in the robot model file is correct. If we want to view the structure of the robot links and joints graphically, we can use the following command:

```
$ urdf_to_graphiz pan_tilt.urdf
```

This command will generate a pdf file of the kinematic chain, as depicted in Fig. 5.5.

5.1.2 Display Robot Models in RViz

We are now ready to display our model in RViz. This is particularly useful to understand if the robot model appears to be correct. In fact, using RViz we are able to see how the shape of the robot is and test the connection between the robotic links. To visualize the robot model in RViz, we must load the robot model in a ROS param and start some nodes that will be discussed at the end this lesson.

Create a `.launch` file in the package directory:

```
$ roscd rl_robot_description_pkg
$ mkdir launch && cd launch
$ touch display.launch
```

The content of this file is reported in the following:

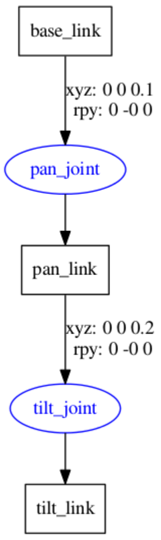


Figure 5.5: Graph of joint and links in the pan-and-tilt mechanism.

```
<?xml version="1.0" ?>

<launch>
<param name="robot_description"
textfile="$(find rl_robot_description_pkg)/urdf/pan_tilt.urdf" />
```

These lines load the robot model into a ROS parameter called `robot_description`. In order to be much general as possible, to reach the location of the `URDF` file the `find` command is used. In particular, using the `$find` command we can directly reach the location of the package.

```
<node name="joint_state_publisher"
pkg="joint_state_publisher" type="joint_state_publisher" />
<node name="robot_state_publisher"
pkg="robot_state_publisher" type="robot_state_publisher" />
```

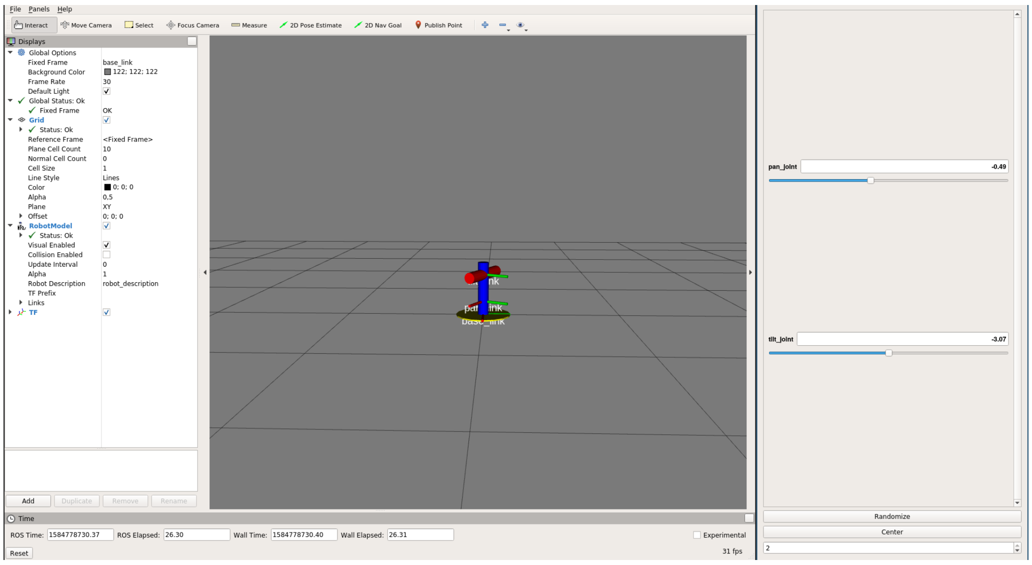
We additionally start two more nodes: the `robot_state_publisher` and the `joint_state_publisher` that will be later discussed.

```
<node name="rviz" pkg="rviz" type="rviz"
args="-d $(find rl_robot_description_pkg)/urdf.rviz" required="true" />
</launch>
```

Finally, we launch `RViz` program. We can also specify a configuration file located in the same directory of the package of the robot model. In this way, the visualization settings can be automatically loaded when `RViz` starts. The output of the `display.lauch` file is show in Fig. 5.6

```
$ roslaunch rl_robot_description_pkg display.launch
```

To use the GUI shown in the picture you should launch the `display_gui.launch` file in which the `joint_state_publisher` node is substituted by the `joint_state_publisher_gui` one. Let's see what is behind the visualiza-



```
Global Options
  Fixed Frame          base_link
  Background Color     122; 122; 122
  Frame Rate           30
  Default Light        ✓
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
  Status: Ok
  Reference Frame      <Fixed Frame>
  Plane Cell Count     10
  Normal Cell Count    0
  Cell Size            1
  Line Style           Lines
  Color                0; 0; 0
  Alpha                0,5
  Plane                XY
  Offset               0; 0; 0
RobotModel             ✓
  Status: Ok
  Visual Enabled       ✓
  Collision Enabled
  Update Interval      0
  Alpha                1
  Robot Description    robot_description
  TF Prefix
  Links
TF                     ✓

ROS Time: 1584778730.37  ROS Elapsed: 26.30  Wall Time: 1584778730.40  Wall Elapsed: 26.31
31 fps

pan_joint   -0.49
tilt_joint  -3.07
2
```

Figure 5.6: Display pan tilt robot using RViz

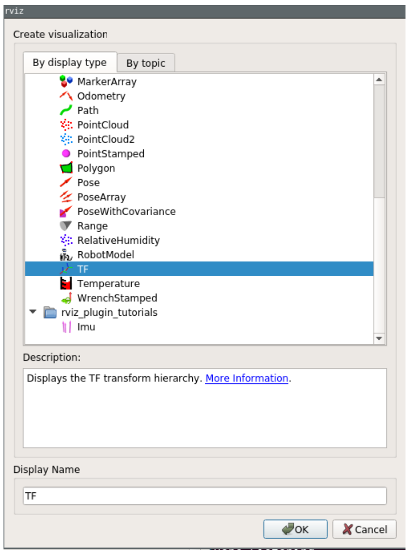
tion process of RViz. This is important to understand because the same information needed by RViz visualization are used by all the other external packages to calculate robot kinematic or allow mobile robot navigation.

The first element discussed here is the `TF` package. In particular, to exploit the model of a robot its configuration must be calculated considering the position of its joints and similarly, the full pose of a mobile robot of a mobile robot inside its environment should be available to allow its navigation.

ROS Transformation (tf) package

A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. . . . `tf` package keeps track of all these frames over time, and allows developers to develop queries to retrieve the relative pose of a frame with respect to the other frames or the position of an object not belonging to the robot with respect to the robotic end effector:

- How is oriented the head of the head of the robot with respect to its base frame?
- What is the pose of the object in my gripper relative to my base?



Description:

Displays the TF transform hierarchy. More Information.

Display Name

TF

Figure 5.7: Add tf plugin.

- What is the current pose of the base frame in the map frame?

Information about the coordinate frames of a robot is available to all ROS components. Like other ROS basic packages, `tf` can be used both with command line tools (implemented as ROS nodes) and using `Python` and `C++` API. In particular, command line tools include:

- `view_frames`: visualizes the full tree of coordinate transforms. This tool is a graphical debugging tool creating a PDF graph of the full transform tree. To test this command just launch the `display.launch` file above discussed and run the following commands:

```
$ rosrun tf view_frames
```

  After few seconds a pdf file is generated in your current directory. In linux you can open it using `evince` command:

```
$ evince frames.pdf
```

- `tf_monitor`: monitors transforms between frames: Similarly to `view_frames` command, `tf_monitor` print information about the current coordinate transform tree to console. Without arguments, this argument prints the result for all frames. To get a specific transformation between two know frames you can use the following syntax:

```
$ tf_monitor <source_frame> <target_target>
```

Note that this command only print information about the tree and not its contents.

- `tf_echo`: prints specified transform to screen. To use this command the syntax is:

  ```
  $ tf_echo <source_frame> <target_frame>
  ```

  In our example to retrieve the pose of the `tilt_link` relative to the `base_link` you can run the following command:

  ```
  $ rosrun tf tf_echo /base_link /tilt_link
  ```

  The output of this command is here reported:

  ```
  - Translation: [0.000, 0.000, 0.300]
  - Rotation: in Quaternion [0.983, 0.034, 0.180, -0.006]
  in RPY (radian) [-3.142, -0.361, 0.069]
  in RPY (degree) [-180.000, -20.708, 3.977]
  ```

`tf` can also be visualized using RViz adding the proper plugin. Like the robot model plugin, you can add `tf` plugin using the `Add` button on RViz interface, as show in Fig. 5.7.

In ROS a frame is a coordinate system and the coordinate systems in are always in 3D right-handed, with X forward, Y left, and Z up. You can see how important is the fixed frame setup of RViz. In fact, you can decide the reference system in which `tf` are displayed.

As for the API, `tf` includes several data type to represent object poses like:

- Quaternion: `tf::Quaternion`
- Transform: `tf::Transform`
- Vector: `tf::Vector3`
- Point: `tf::Point`
- Pose: `tf::Pose`

`tf` offers different features to handle the coordinate frame of your robot. In this lesson, we focus on the possibility to retrieve information by the `tf` tree. Let's create a new package to print the current pose of the `tilt_link` relative to the `base_link`

```
$ roscd rl_robot_description_pkg/src
$ touch tf_listener.cpp
```

We will mainly use two functions from `tf` API:

- `waitForTransform`:

```
        bool tf::TransformListener::waitForTransform
         (const std::string
        &target_frame, const std::string &source_frame,
        const ros::Time &time, const ros::Duration &timeout)
```

  This function test if `source_frame` can be transformed to `target_frame` at time `time`. This method returns a bool whether the transform can be evaluated. This is a blocking method that returns when the elapsed time reaches the `Duration`.

- `lookupTransform`:

```
        void tf::TransformListener::lookupTransform
        (const std::string &target_frame,
        const std::string &source_frame,
         const ros::Time &time, StampedTransform &transform)
```

  This function fills `transform` data with the transform from `source_frame` to `target_frame` at time. This method is the core functionality of the `tf` library, however most often the `transform*` methods will be used by the end user. This methods is designed to be used within `transform*()` methods. The direction of the transform returned will be from the `target_frame` to the `source_frame`. Which if applied to data, will transform data in the `source_frame` into the `target_frame`.

The content of `tf_listener.cpp` is here reported:

```
int main( int argc, char** argv ) {
ros::init(argc, argv, "tf_example");
ros::NodeHandle nh;
//Wait ROS node starts
sleep(1);
```

As usual, the first part of the code initializes the ROS node.

```
//Declare the listener to use c++ tf API
tf::TransformListener listener;
//Declare the tranfsorm object to store tf data
tf::StampedTransform transform;
```

Then, we declare the objects needed to use `tf` API: a listener for the ROS transforms and the object to store rotation and translation data.

```
for(int i=0; i<10; i++ ) {
try {
//We want the current transform
ros::Time now = ros::Time::now();
```

We specify the time in which we require the transform. To get the current rototranslation use: `ros::Time::now()`.

```
if( listener.waitForTransform("/base_link", "/pan_link", now, ros::Du
listener.lookupTransform("/base_link", "/pan_link",  now, transform);
std::cout << "Translation: " << transform.getOrigin().x() << " " << tr
std::cout << "Rotation: " << transform.getRotation().w() << " " << tra
```

Here, we check if the desired transform is available in the `tf` tree. If the transform is present we require the value of the transform that is stored into the transform object.

```
//Convert quaternion to euler angles
tf::Matrix3x3 m( transform.getRotation() );
double roll, pitch, yaw;
m.getRPY(roll, pitch, yaw);

std::cout << "RPY: " <<
roll << ", " << pitch << ", " << yaw << std::endl;
}
} else { ROS_WARN("Transform not ready"); }
```

Finally, we change the representation system for link orientation, from quaternion to euler angles.

```
catch (tf::TransformException ex){
ROS_ERROR("%s",ex.what());
ros::Duration(1.0).sleep();
}
ros::Duration(1.0).sleep();
}
}
```

After compiled the node, you can run this code after launched the `display.launch` file that load the model of the robot and fill the `tf` tree.

```
$ roslaunch rl_robot_description_pkg display.launch
$ rosrun rl_robot_description_pkg tf_listener
```

In the next lessons of this course we will learn more about ROS `tf` system, but now continue to analyze the `pan-tilt` example. In particular, we saw how the kinematic chain of our robot model is updated when the values of its joints changes. The obvious question that we should made now is: who is publishing `tf` in our ROS `tf` tree? Remember that we launched two additional nodes in the `display.launch` file: `joint_state_publisher` and `robot_state_publisher`.

robot state publisher

robot_state_publisher is a fundamental package or ROS. It allows you to publish the state of a robot to tf. tf are published using the information contained into the URDF file describing the robot. In particular, robot_state_publisher uses the URDF specified by the parameter robot_description and the joint positions from the topic joint_states to calculate the forward kinematics of the robot and publish the results via tf.

So, to work properly this package needs for two input:

- Robot model: loaded into the ROS parameter server
- Joint position: published as sensor_msgs/JointState

We already seen how can load the model of the robot into the parameter server. As for the joint position, robot_state_publisher expects that a sensor_msgs/JointState is published on a topic called joint_states. sensor_msgs/JointState is structured as follow:

```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

In principle, for each joint of the robot, a properly joint_states message stores its name, position, velocity and effort. Notice that only fields required to make things work are joint name and position.

So, at this point we understood how the tf using the information about the position of the joint of the robot and its kinematic structure. But, the last question to reply is: who is publishing the position of the joints?

joint state publisher

This package publishes sensor_msgs/JointState messages for a robot. The package reads the robot_description parameter, finds all of the non-fixed joints and publishes a JointState message with all those joints defined. In our example, we used this package in conjunction with the robot_state_publisher node to also publish transforms for all joint states. In particular, in our pan-tilt example we used the GUI plugin to inform the joint_state_publisher package about the value of each joint.

In conclusion, the chain to retrieve information from our robot model is closed:

- joint_state_publisher uses URDF to get information about the name and the type of each joint, then it publish a joint_states message containing such values.
- robot_state_publisher uses the URDF model to get information about the kinematic structure of the robot and the joint_states message to fill the tf tree.
- RViz uses the URDF to display the robot model, and the tf tree to display the relative position of each joint of the robot.

5.1.3 Robot modeling using XACRO

URDF structure is not flexible when you need to create a complex robot model. Some of the main features that URDF is missing are simplicity, reusability, modularity, and programmability. If someone wants to reuse a URDF block 10 times in his robot description, he can copy and paste the block 10 times. If there is an option to use this code block and make multiple copies with different settings, it will be very useful while creating the robot description. The URDF is a single file and we can't include other URDF files inside it. This reduces the modular nature of the code. All code should be in a single file, which reduces the code's simplicity. Also, if there is some programmability, such as adding variables, constants, mathematical expressions, and conditional statements, in the description language, it will be more user friendly. xacro is a file description format to meet these conditions. It mainly:

- Simplify URDF: The xacro is the cleaned-up version of URDF. It creates macros inside the robot description and reuses the macros. This can reduce the code length. Also, it can include macros from other files and make the code simpler, more readable, and more modular.
- Programmability: The xacro language supports a simple programming statement in its description. There are variables, constants, mathematical expressions, conditional statements, and so on that make the description more intelligent and efficient.

xacro file format represents an updated version of URDF. However these two formats have the same power of expression. You can always convert a xacro file in an URDF one and vice versa.

In the following some features of xacro file format are discussed:

- **Property**: We can declare variables used anywhere in the code and are helpful to change the constants of your robot model. For example, if you want to define the length of a link or a size of a wheel you can use a parameter:

```
<xacro:property name="base_link_length" value="0.1" />
```

- You can also fit properties into blocks:

```
<xacro:property name="front_left_origin">
<origin xyz="0.3 0 0" rpy="0 0 0" />
</xacro:property>

<pr2_wheel name="front_left_wheel">
<xacro:insert_block name="front_left_origin" />
</pr2_wheel>
```

- Using `xacro` file you can also use math expressions:

```
<xacro:property name="radius" value="4.3" />
<circle diameter="${2 * radius}" />
```

- and conditional blocks:

```
<xacro:if value="<expression>">
<... some xml code here ...>
</xacro:if>
<xacro:unless value="<expression>">
<... some xml code here ...>
</xacro:unless>
```

- However, the most important element of `xacro` format is the possibility to define macro blocks. For example, you can create a general inertial block to use in our model:

```
<xacro:macro name="inertial_matrix" params="mass">
<inertial>
<mass value="${mass}" />
<inertia ixx="0.5" ixy="0.0" ixz="0.0"
iyy="0.5" iyz="0.0" izz="0.5" />
</inertial>
</xacro:macro>
```

  Here, the macro is named `inertial_matrix`, and its parameter is the mass mass. The mass parameter can be used inside the inertial definition using `${mass}`. We can replace each inertial code with a single line, as given here:

```
<xacro:inertial_matrix mass="1"/>
```

6

Robot Modeling using xacro

In previous section with learn how to model robot using `URDF` file format. `URDF` structure is not flexible when you need to create a complex robot model. Some of the main features that URDF is missing are simplicity, reusability, modularity, and programmability. If someone wants to reuse a URDF block 10 times in his robot description, he must copy and paste it 10 times. If there is an option to use this code block and make multiple copies with different settings, it will be very useful while creating the robot description. The URDF is a single file and we can't include other URDF files inside it. This reduces the modular nature of the code. All code should be in a single file, which reduces the code's simplicity. Also, if there is some programmability, such as adding variables, constants, mathematical expressions, and conditional statements, in the description language, it will be more user friendly. `xacro` is a file description format to meet these conditions. In particular:

- Simplify URDF: The `xacro` is the cleaned-up version of URDF. It creates macros inside the robot description and reuses the macros. This can reduce the code length. Also, it can include macros from other files and make the code simpler, more readable, and more modular.

- Increase robot model programmability: The `xacro` language supports a simple programming statement in its description. There are variables, constants, mathematical expressions, conditional statements, and so on that make the description more intelligent and efficient.

6.0.1 xacro format

`xacro` file format represents an updated version of `URDF`. However these two formats have the same power of expression. You can always convert a `xacro` file in an `URDF` one and vice versa.

In the following some features of `xacro` file format are discussed:

- **Property**: We can declare variables used anywhere in the code and are helpful to change the constants of your robot model. For example, if you want to define the length of a link or a size of a wheel you can use a parameter:

  ```
  <xacro:property name="base_link_length" value="0.1" />
  ```

- You can also fit properties into blocks:

  ```
  <xacro:property name="front_left_origin">
  <origin xyz="0.3 0 0" rpy="0 0 0" />
  </xacro:property>

  <pr2_wheel name="front_left_wheel">
  <xacro:insert_block name="front_left_origin" />
  </pr2_wheel>
  ```

- Using `xacro` file you can also use math expressions:

  ```
  <xacro:property name="radius" value="4.3" />
  <circle diameter="${2 * radius}" />
  ```

- and conditional blocks:

  ```
  <xacro:if value="<expression>">
  <... some xml code here ...>
  </xacro:if>
  <xacro:unless value="<expression>">
  <... some xml code here ...>
  </xacro:unless>
  ```

- However, the most important element of `xacro` format is the possibility to define macro blocks. For example, you can create a general inertial block to use in our model:

  ```
  <xacro:macro name="inertial_matrix" params="mass">
  <inertial>
  <mass value="${mass}" />
  <inertia ixx="0.5" ixy="0.0" ixz="0.0"
  iyy="0.5" iyz="0.0" izz="0.5" />
  </inertial>
  </xacro:macro>
  ```

  Here, the macro is named `inertial_matrix`, and its parameter is the mass. The mass parameter can be used inside the inertial definition using `${mass}`. We can replace each inertial code with a single line, as given here:

```
<xacro:inertial_matrix mass="1"/>
```

In order to show the capabilities of `xacro` modeling, try to re-write the `pan-tilt` robot model discussed in *Lesson 5* using `xacro` file format. Move into the `rl_robot_description_pkg/urdf` directory and create the following two files:

```
$ roscd rl_robot_description_pkg/urdf/
$ touch pan_tilt_macro.xacro
$ touch pan_tilt.xacro
```

The general idea is that `pan_tilt.xacro` defines the structure of our robot (like a source file) while the `pan_tilt_macro.xacro` contains a set of definition of possible robot parts (like header file).

Let's start to see the content of the `pan_tilt_macro.xacro` file:

As usual we include the prolog of the file, and also open a new `robot` tag. In this case we have to specify the namespace of our xml file (xmlns means XML namespace).

```
<?xml version="1.0"?>
<robot name="pan_tilt_macro" xmlns:xacro="http://wiki.ros.org/xacro">
```

Then we define the `macro` to instantiate the `base_link` of the robot. In this case we are not using the power of `xacro` definition because our `base_link` is unique in our robot. For this reason except its name, the other element of this link are static (its visual, geometry, collision, inertia, and so on . . .). As you may note, the body of this `macro` is exactly the same of the one contained into the `URDF` file.

```
<xacro:macro name="base_link_macro" params="base_link_name">
<link name="${base_link_name}">
<visual>
<geometry>
<cylinder length="0.01" radius="0.2"/>
</geometry>
<origin rpy="0 0 0" xyz="0 0 0"/>
<material name="yellow">
<color rgba="1 1 0 1"/>
</material>
</visual>
<collision>
<geometry>
<cylinder length="0.03" radius="0.2"/>
</geometry>
<origin rpy="0 0 0" xyz="0 0 0"/>
</collision>
<inertial>
```

```
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
</xacro:macro>
```

We are ready to define something that can be reused several time to build a robot model. In `pan-tilt` robot we had two joints and two links. Try to define general elements that can be instantiated multiple times to build a robot with multiple DOF.

Let's define a `macro` to create a joint for our robot:

```
<xacro:macro name="pan_tilt_joint" params="name type parent child *or
<joint name="${name}" type="${type}">
```

Our `macro` is called `pan_tilt_joint`. Be careful, don't confuse the `macro` name with the element name. This `macro` block has multiple parameters that can be distinguish between two classes:

- Simple parameters: just a string or a numerical value, like the name or the type of the joint. We also want to specify the links connected by this joint.
- Block parameters: in order to insert custom blocks, like the center of the joint or its rotation axes, we must pass a block adding the * symbol before the parameter name.

```
<parent link="${parent}" />
<child link="${child}" />
<xacro:insert_block name="origin" />
<xacro:insert_block name="axis" />
<limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
<dynamics damping="50" friction="1"/>
</joint>
</xacro:macro>
```

At this point, is clear that we can instantiate this block customizing its main characteristics. Let's see the definition of the `macro` link:

```
<xacro:macro name="pan_tilt_link" params="name *geometry *origin">
<link name="${name}">
<visual>
<xacro:insert_block name="geometry" />
<xacro:insert_block name="origin" />
<material name="red">
<color rgba="0 0 1 1"/>
</material>
```

```
</visual>
<collision>
<xacro:insert_block name="geometry" />
<xacro:insert_block name="origin" />
</collision>
<inertial>
<mass value="1"/>
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
</link>
</xacro:macro>
```

Finally, we can close the content of this file terminating the `robot` tag:

```
</robot>
```

As for the file defining the whole robot structure using such macro, here is reported its content:

```
<robot name="pan_tilt" xmlns:xacro="http://wiki.ros.org/xacro">
<xacro:include filename="$(find rl_robot_description_pkg)/urdf/pan_tilt_macro
```

First of all, we must include the `pan_tilt_macro.xacro` file to use its macro. Then, we can start to define the structure of the `pan-tilt` robot:

```
<xacro:base_link_macro
base_link_name="base_link">
</xacro:base_link_macro>
```

As already stated, the `base_link` of the robot accepts as parameter only the name of the link.

We can start including our first joint: the `pan_joint`. Here is also clear the difference between simple and block parameters.

```
<xacro:pan_tilt_joint
name="pan_joint"
type="revolute"
parent="base_link"
child="pan_link">
<origin xyz="0 0 0.1" />
<axis xyz="0 0 1"/>
</xacro:pan_tilt_joint>
```

Then we can instantiate the `pan_link` and the tilt joint and link:

```
<xacro:pan_tilt_link name="pan_link">
<geometry>
<cylinder length="0.4" radius="0.04"/>
</geometry>
<origin xyz="0 0 0" rpy="0 0 0.09"/>
```

```
</xacro:pan_tilt_link>

<xacro:pan_tilt_joint
name="tilt_joint"
type="revolute"
parent="pan_link"
child="tilt_link">
<origin xyz="0 0 0.2" />
<axis xyz="0 1 0"/>
</xacro:pan_tilt_joint>

<xacro:pan_tilt_link name="tilt_link">
<geometry>
<cylinder length="0.4" radius="0.06"/>
</geometry>
<origin xyz="0 0 0" rpy="0 1.5 0.0"/>
</xacro:pan_tilt_link>
</robot>
```

So we have seen how he `xacro` definition improves the code readability and reduces the number of lines compared to `URDF`. However, all ROS software, like RViz robot model plugin, `robot_state_publisher`, and so on require an `URDF` file.

After designing the `xacro` file, we can use the following command to convert it to a `URDF` file:

```
$ rosrun xacro xacro pan_tilt.xacro > pan_tilt_generated.urdf
```

Be careful to refer to your **main** `xacro` file, and not to that one containing the definition of the macro blocks.

Finally, to display the `xacro` model in RViz, you can create another `launch` file in the `rl_robot_description_pkg` package referring to the new `URDF` file generated by the `xacro`:

```
<?xml version="1.0" ?>
<launch>
<arg name="model" />
<param name="robot_description" textfile="$(find rl_robot_description_
<param name="use_gui" value="true"/>
<node name="joint_state_publisher" pkg="joint_state_publisher" type="
<node name="robot_state_publisher" pkg="robot_state_publisher" type="
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find rl_robot_des
</launch>
```

Let's call this file `display_xacro.launch` and execute it with the following command:

```
$ roslaunch rl_robot_description_pkg display_xacro.launch
```

In the rest of the course we mainly consider `xacro` file to model our robots.

7

Simulation in robotics

Simulate robots represents a fundamental step in the development of robotic applications for several reasons. First of all, simulation helps developers to test different solutions to perform a task without risks. Also, simulation is useful when we have to test our algorithms and we haven't access to the robotic hardware.

Several features are nice of have in a simulation environment. In particular, good robotic simulators enable a natural and realistic behavior of our robots, both into the dynamics reaction of the world objects and the motion of the robot. Also it's very important that very few changes must be made in your programs to move it from simulation to the real robot. ROS supports different robotic simulation tools, we will discuss about `Coppelia Sim` and `Gazebo`.

7.0.1 Coppelia Sim

`Coppelia Sim` is a multi-platform robotic simulator developed by Coppelia Robotics. It offers many simulation models of popular industrial and mobile robots ready to be used, and different functionalities that can be easily integrated and combined through a dedicated API. In addition, it can operate with ROS using a communication interface that allows us to control the simulation scene and the robots via topics and services. `Coppelia Sim` can be used as a standalone software, while an external plugin must be installed to work with ROS.

This simulator born from the `V-REP` (Virtual Robot Experimentation Platform) project. `Coppelia Sim`, with integrated development environment, is based on a distributed control architecture: each object/model can be individually controlled via an embedded script, a plugin or a ROS node, a remote API client, or a custom solution. This makes `Coppelia Sim` very versatile and ideal for multi-robot applications. Controllers can be written in `C/C++`, `Python`, `Java`, `Lua` or `Matlab`

7.0.2 Starting with Coppelia Sim

The first thing to do to start interfacing with `Coppelia Sim` is install it. Actually, you need only to download and extract it in a desired folder. If you are using `Ubuntu 20.04` download the last version of `Coppelia Sim` at this link: `https://coppeliarobotics.com/files/CoppeliaSim_Edu_V4_2_0_Ubuntu20_04.tar.xz`. Then extract it in your development folder.

Consider that, `CoppeliaSim` is not a free software. It is under a commercial license. We can use in this course thanks to our educational affiliation. So, just download the educational version of `CoppeliaSim`.

Could be convenient to rename the `Coppelia Sim` directory with an intuitive name:

```
$ mv CoppeliaSim_Edu_V4_2_0_Ubuntu20_04.tar.xz  CoppeliaSim
```

To start `Coppelia Sim` you need to start the `coppeliaSim` script into the simulator folder:

```
$ cd CoppeliaSim
$ ./coppeliaSim
```

It's highly probable that the first time you start the simulator you will get the following error:

```
./coppeliaSim: error while loading shared libraries:
liblua5.1.so: cannot open shared object file: No such
file or directory
```

You should be able to recognize this error: `coppeliaSim` is not able to fine a `shared library` called `liblua`. This could happen for two reasons. Or this library is not installed in your system or `coppeliaSim` doesn't know where to search for it. In this case, you need to install such library:

```
$ sudo apt-get install liblua5.1.0
```

After installed such library, you can retry to launch the `CoppeliaSim`. However you should get the following error:

```
$ ./coppeliaSim: /usr/lib/x86_64-linux-gnu/libQt5Core.so.5:
version ‘Qt_5.12’ not found (required by ./coppeliaSim)
```

Another error related to the libraries. These libs are included into the simulator folder, so in this case, we need to inform Linux where to find them. We need to properly set the well known environment variable called: `LD_LIBRARY_PATH`. The must include the `CoppeliaSim` folder to the other directory already contained in it.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/CoppeliaSim
```

Of course, to make this modification permanent, you must to add this line to your `.bashrc` file.

We are now ready to start using `CoppeliaSim` running the starting script:

```
./coppeliaSim
```

We are now ready to start interfacing ROS with `CoppeliaSim`.

7.0.3 CoppeliaSim - ROS interface

In previous versions of `CoppeliaSim`, when it was called `V-REP`, the interface with ROS was developed by means a plugin called `RosPlugin`.

Nowadays, the ROS Interface is part of the `CoppeliaSim` API framework. Make sure not to mix up the ROS Interface with the `RosPlugin`, which is an older, deprecated interface in `CoppeliaSim`. The ROS Interface duplicates the C/C++ ROS API with a good fidelity. This makes it the ideal choice for very flexible communication via ROS, but might require a little bit more insight on the various messages and the way ROS operates.

`CoppeliaSim` can act as a ROS node that other nodes can communicate with via ROS services, ROS publishers and ROS subscribers. The ROS Interface functionality in `CoppeliaSim` is enabled via a plugin: libsimExtROSInterface.*. The plugins can easily be adapted to your own needs. The plugins are loaded when `CoppeliaSim` is launched, but the load operation will only be successful if `roscore` was running at that time. Make sure to inspect CoppeliaSim's console window or terminal for details on plugin load operations.

To check if ROS interface is correctly loaded, run a roscore and, when the simulator starts, check into your linux shell for the following lines:

```
Plugin 'ROSInterface': loading...
Plugin 'ROSInterface': load succeeded.
```

If you are not able to see such lines, you must configure your system copying the ROS interface into the main directory of `CoppeliaSim`. After joined the simulator directory, use the following command

```
$ cp compiledRosPlugins/libsimExtROSInterface.so .
```

Now you can restart `CoppeliaSim`.

7.0.4 CoppeliaSim GUI

A comprehensive documentation about the `CoppeliaSim GUI` can be found at this link `https://www.coppeliarobotics.com/helpFiles/en/userInterface.htm`. Please, refer to this page if you are not able to find elements considered in this lesson.

7.0.5 Programming ROS scene with CoppeliaSim

This section discusses how to start programming using ROS API in `CoppeliaSim`. Start opening a new scene of CoppeliaSim: by default the simulator opens a new scene at its start, otherwise this can be also made using the menu bar on the top of the windows `[File -> New scene]`. To start programming a robot or a device into the simulation scene, you need to add it into the scene hierarchy.

The goal of our first example is just to understand how ROS plumbing capabilities can be used into CoppeliaSim. So, just include a new object with a primitive shape: right click on the simulation scene then: `[Add -> Primitive Shape -> Plane]`.

In our case, CoppeliaSim works thanks to a set of scripts based on `Lua` programming language that must be associated to scene objects. So, to enable ROS functionalities into CoppeliaSim, we need to call ROS API into such scripts.

Let's crate our first Lua script: right click on the just create object, then: `Add -> Associated child script -> Not threaded`. The main difference between this two kind of scripts is that threaded scripts are run into separate threads, so have their framerate and so on. Differently, not thread scripts have a series of function blocks that are called on particular events. A new child script has this form:

```lua
function sysCall_init()
-- do some initialization here
end

function sysCall_actuation()
-- put your actuation code here
end

function sysCall_sensing()
-- put your sensing code here
end

function sysCall_cleanup()
-- do some clean-up here
end
```

Following the comments into the script, we have 4 main function blocks:

- `sysCall_init()`: called only when the simulation starts. Here we can initialize our stuff
- `sysCall_actuation()`: called following the framerate of the simulation. Used to actuate the robotic joints

- `sysCall_sensing()`: called when the sensors of your simulation scene are updated.
- `sysCall_cleanup()`: called when your scene is stopped.

So, our goal is to create a script publishing an integer value. Later, we will create another object with another script that reads such value printing it into the status bar.

Starting with the initialization function, we firstly check if the ROS plugin is properly working. This is made checking the value of the `simROS` variable:

```
if simROS then
...
else
...
end
```

`simROS` is `true` if the ROS plugin is working. In this case, we can initialize the publisher:

```
if simROS then
print("ROS interface correctly loaded")
pub=simROS.advertise('/number', 'std_msgs/Int32')
else
print("<font color='#F00'>
ROS interface was not found. Cannot run.</font>@html")
end
```

As for the publishing part, we can fill the actuation function with the following code:

```
function sysCall_actuation()
int_data = {}
int_data['data'] = 13
simROS.publish(pub, int_data)
end
```

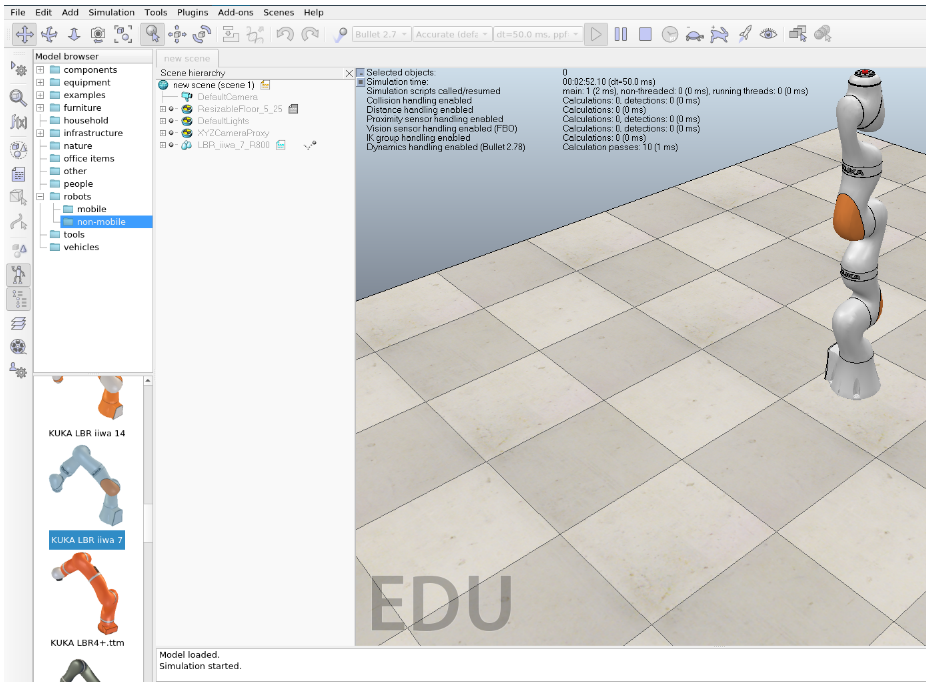
In this function we declare a new variable, called `int_data`. We fill such variable thanks to the knowledge about the structure of the message that we want to publish: we know that the `std_msgs/Int32` message has a field called data, so we assign this field in the following way:

```
int_data['data'] = value
```

Differently, if we commit some mistakes in the variable definition (referring to an unknown data field for example), we could have an error similar to this:

```
Lua runtime error: [string "CHILD SCRIPT Plane"]:13:
read__std_msgs__Int32: unexpected key: qdata
(simROS.publish @ 'RosInterface' plugin)
stack traceback:
[C]: in function 'publish'
[string "CHILD SCRIPT Plane"]:13: in function
```

Now our script is complete. We can use the `play` button on the simulator bar and check the result using the `rostopic echo` command.



```
Selected objects:                          0
Simulation time:                           00:02:52.10 (dt=50.0 ms)
Simulation scripts called/resumed          main: 1 (2 ms), non-threaded: 0 (0 ms), running threads: 0 (0 ms)
Collision handling enabled                 Calculations: 0, detections: 0 (0 ms)
Distance handling enabled                  Calculations: 0 (0 ms)
Proximity sensor handling enabled          Calculations: 0, detections: 0 (0 ms)
Vision sensor handling enabled (FBO)       Calculations: 0, detections: 0 (0 ms)
IK group handling enabled                  Calculations: 0 (0 ms)
Dynamics handling enabled (Bullet 2.78)    Calculation passes: 10 (1 ms)
```

```
Model loaded.
Simulation started.
```

Figure 7.1: Kuka LBR iiwa 7 in CoppeliaSim

We are also ready to write our subscriber. Just create another basic object and add a not threaded script. Like in the publisher example, we check if ROS plugin is properly working. Then we declare a subscriber:

```
function sysCall_init()
if simROS then
print("ROS interface correctly loaded")
sub=simROS.subscribe('/number', 'std_msgs/Int32', 'intMessage_callback
else
print("<font color='#F00'>ROS interface was not found. Cannot run.</fo
end
end
```

The subscriber function needs for three arguments: the topic name, the topic type and the callback function. In the following the callback function si reported.

```
function intMessage_callback(msg)
print ( "data", msg["data"] )
end
```

In this function (that we added to our script), we just get the data field of the ROS topic in order to print its value into the status bar. We can save this scene using the menu bar of the simulator window: `File -> Save scene`. Extensions for `CoppeliaSim` scene is `.ttt`, let's all this scene: `pub_sub.ttt` and try something of more complex.

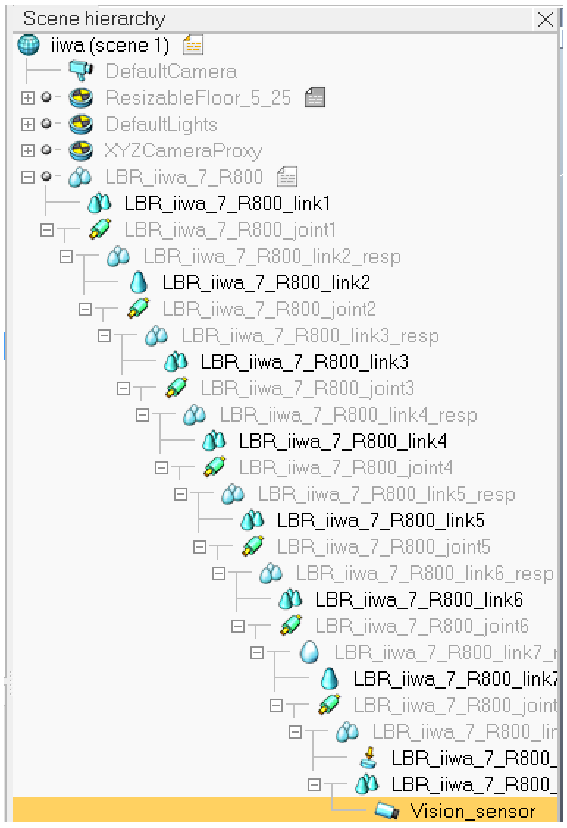


Figure 7.2: Kuka with vision sensor

A very useful thing allowed by simulators is the possibility to simulate sensors over robots. We are going to implement two examples:

- : we will add a vision sensor on a seven degree of freedom manipulator and publish its content over ROS network.

- Example 2: we will implement a ROS node to control the robotic joints.

Example 1

Vision sensors are commonly used in robotics to implement advanced applications. They are cheap and allow a robot to be aware about the its working environment. Let's start preparing our scene. Open a new scene and select a robot to import into the environment. For example, you can choose the `Kuka LBR iiwa 7` as show in Fig. 7.1. The `Kuka LBR iiwa 7` model already contains a script demonstrating its motion capabilities. Just delete that script and attach a new vision sensor to its end effector:

- To spawn a new robot in the simulation scene, select it from the Model browser (in our case, `robots->non mobile`) and drop it into the scene
- To remove the associated script, right click on robot model [Edit -> remove -> associated child script].
- To add a vision sensor, unfold the robot components and select the last link of the robot (`LBR_iiwa_7_R800_link8`). Click with the right button of the mouse on its name and select [Add -> Vision sensor -> Prespective View].
- Could also happen that the model is not placed on the correct link. To solve this just drug and drop it in the correct location into the scene hierarchy. You scene hierarchy must be similar to the one show in Fig. 7.2.
- Use the object/item shift button to correctly attach the sensor on the end effector of the robot.
- You can also modify the camera parameters to have more resolution of field of view.
- Finally, add the non-threaded script to manage the vision sensor and the robot motion as well.

Here the contents of the script. Start with the initialization function:

```
function sysCall_init()
jointHandles={-1,-1,-1,-1,-1,-1,-1}
for i=1,7,1 do
jointHandles[i]=sim.getObjectHandle('LBR_iiwa_7_R800_joint'..i)
end
```

We need object handlers to control the behavior of a joint. In our simulation we have seven joints called `LBR_iiwa_7_R800_joint0`, `LBR_iiwa_7_R800_joint1`, and so on. We can initialize seven different handlers. To get elements from the simulation scene, you must used the `sim` object, that is declared by default into the `Lua` script.

```
sim.setJointTargetPosition( jointHandles[1], 0.0   )
sim.setJointTargetPosition( jointHandles[2], 0.0   )
sim.setJointTargetPosition( jointHandles[3], 0.0   )
sim.setJointTargetPosition( jointHandles[4], -1.57 )
sim.setJointTargetPosition( jointHandles[5], 0.0   )
sim.setJointTargetPosition( jointHandles[6], 0.27  )
sim.setJointTargetPosition( jointHandles[7], -1.57 )
```

The default position of this robot is called candle position. All its joints are set to 0. Just use the `setJointTargetPosition` function to find a convenient position for our task. This function accepts as input the handler of the joint and the target position in radians. We need a handle also for the vision sensor:

```
vision_sensor = sim.getObjectHandle('Vision_sensor')
```

Then, we are ready to declare our publisher of the vision sensor:

```
if simROS then
print("<font color='#0F0'>ROS interface was found.</font>@html")
img_pub = simROS.advertise('/image', 'sensor_msgs/Image')
```

To advertise a data om `Lua`, the `advertise` function accepts two input, the name of the topic and its type.

```
-- treat uint8 arrays as strings (much faster, tables/arrays are kind of slow
simROS.publisherTreatUInt8ArrayAsString(img_pub)
else
print("<font color='#F00'>ROS interface was not found. Cannot run.</font>@htm
end
end
```

In the particular case of the image type we can ask for the encoding type of the pixel of the image. We require to consider the full list of the `uint8` elements as a string.

Finally, in the sensing part we can manage the vision sensor data, in order to fit them in a ROS message and publish the camera data over ROS network. Before to continue let's analyze the `sensor_msgs::Image` data type. As usual, see the structure of the message using the `rosmsg` command:

```
$ rosmsg show sensor_msgs/Image
```

Here is reported its content:

```
std_msgs/Header header
uint32 seq
```

```
time stamp
string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

We have some information about the image and the data type, like its encoding and the dimension of the image (`height` and `width`). Regarding image pixel, they are stored into a one dimensional vector. The dimension of this vector is depends from the resolution of the image.

```
function sysCall_sensing()
local data,w,h=sim.getVisionSensorCharImage(vision_sensor)
```

We firstly get data by the vision sensor. In particular, some of return value of the `getVisionSensorCharImage` function are the pixel data and the dimension of the image. As already made for the `std_msgs/Int32` example, fill the image data structure and publish it.

```
d={}
d['header']={stamp=simROS.getTime(), frame_id="camera_frame"}
d['height']=h
d['width']=w
d['encoding']='rgb8'
d['is_bigendian']=1
d['step']=w*3
d['data']=data
simROS.publish(img_pub ,d)
end
```

After start the simulation, you can check that the image topic is present in your ROS system. We are also ready to use another tool from ROS: `image_view`. To display the content of a ROS image data, you can use the `image_view` package:

```
$ rosrun image_view image_view image:=/image
```

You need also to specify the topic name where the image is store `image:=TOPIC_NAME`. This command will open a new window displaying the topic image. You also can obtain the same result using the `rqt_image_view` plugin. Save this scene with a desired name and let's continue implementing our second example to move of the joint of the robot.

Example 2

In this example our goal is to move the first link of the kuka robot to implement the pan behavior already seen in this course. We will also need to modify the `CoppeliaSim` script to link the information used by the ROS package. In particular we need to read the position of the joint to control, and to set the desired command. Let's modify the `Lua` script adding a publisher and a subscriber to such robot elements. In the initialization part of the script add these lines:

```
jp_pub = simROS.advertise('/iiwa/j0/pos', 'std_msgs/Float32')
-- treat uint8 arrays as strings (much faster, tables/arrays are kind of slow
sub = simROS.subscribe('/iiwa/j0/cmd', 'std_msgs/Float32', 'jCmd_callback')
```

As for the callback, it can be implemented in the following way:

```
function jCmd_callback( msg )
sim.setJointTargetPosition( jointHandles[1], msg['data'] )
end
```

So, we want only to control the first joint of the robot. After checked that everything is properly working, using the ROS tools that we already learned, create a ROS package to store our code:

```
$ roscd && cd ../src
$ catkin_create_pkg pan_control roscpp std_msgs tf
```

We also include the `URDF` of the robot taken by the kuka ROS development package, you will find it into the `urdf` directory of the course material. We will use this model for future exercises. Create the source code to implement the pan behavior. We also take advantage of this example to see a convenient structure for our robot control programs. The proposed structure uses `boost::thread` to manage the parallel execution of different section the code. In particular using `boost::thread` implementation, we can easily define class member functions as thread that share with all the other thread of the program the variable space.

Let's start discussing the `pan_control.cpp` file:

```
#include "ros/ros.h"
#include "std_msgs/Float32.h"
#include "boost/thread.hpp"
using namespace std;
```

As you can see, we include two new headers: the one to publish the joint command as a `Float32` ROS message and the one to use the `boost thread`.

```
class PAN_MOTION {
public:
PAN_MOTION();
void ctrl_loop();
void joint_data_sub(std_msgs::Float32ConstPtr );
void run();
private:
ros::NodeHandle _nh;
ros::Publisher  _j_cmd_pub;
ros::Subscriber _j_pos_sub;
double _left_lim;
double _right_lim;
float  _cj_pos;
double  _p;
bool   _first_jpos_data;
};
```

In the definition of the class we have some variables that can be passed to our node using the ROS parameters server. In particular:

- `_left_lim`: is the angle limit in the left rotation of the pan motion
- `_right_lim`: is the angle limit in the right rotation of the pan motion
- `_p`: is a proportional gain

We initialize the variables in the constructor of the class, along with the publisher and subscriber:

```
PAN_MOTION::PAN_MOTION () {
_j_cmd_pub = _nh.advertise< std_msgs::Float32 > ("/iiwa/j0/cmd", 0);
_j_pos_sub = _nh.subscribe("/iiwa/j0/pos", 0, &PAN_MOTION::joint_data_

_nh.param(ros::this_node::getName() + "/left_lim", _left_lim, 1.2);
_nh.param(ros::this_node::getName() + "/right_lim", _right_lim, -1.2
_nh.param(ros::this_node::getName() + "/p", _p, 0.005);

_first_jpos_data = false;
}
```

Since the parameters are loaded into the ROS parameter server in relation with the node name, we use the function ros::this_node::getName() to retrieve the name of the node.

Regarding the topic subscriber, we just save the value of the joint angle into a class variable and inform our control program that the information about the joint position are now available:

```
void PAN_MOTION::joint_data_sub(std_msgs::Float32ConstPtr j_pos ) {
_cj_pos = j_pos->data;
_       first_jpos_data = true;
}
```

Finally, we are ready to run the main control loop:

```
void PAN_MOTION::ctrl_loop() {

cout << "Pan motion controller, move your joint from " << _left_lim << " to:

while (!_first_jpos_data) sleep(1);
ROS_INFO("Joint position info arrived!");
```

Before commanding the robot, we need to wait to know the current position of its joint. So, we wait until the first message in the relative callback is not arrived.

```
double set_point = _left_lim;
std_msgs::Float32 cmd;
double ref;
float threshold = 0.01;
ros::Rate r(100);
ref = _cj_pos;
```

Than we can initialize some stuff like the control rate (100Hz) and the current position of the joint before starting to control it. The `set_point` variable defines the end point of the pan motion. In this context, we assume that the first rotation must be made in the left direction.

```
while( ros::ok() ) {
float e = set_point - _cj_pos;
while ( fabs( e ) > threshold ) {
e = set_point - _cj_pos;
ref = ref + e*_p;
cmd.data = ref;
_j_cmd_pub.publish( cmd );
r.sleep();
}
```

In the control routine, we calculate the error between the desired setpoint and the current position of the joint. Then, we just implement a simple `P` controller. The motion along a certain direction is considered completed when this error is lesser than a given threshold. Finally, we have to updated the setpoint:

```
set_point = (set_point == _left_lim ) ? _right_lim : _left_lim;
}
}
```

Our code is quite complete. However, how could we call the control loop? We added in the class definition a `run()` function:

```
void PAN_MOTION::run() {
boost::thread ctrl_loop_t(&PAN_MOTION::ctrl_loop, this);
ros::spin();
}
```

We call this function in the main of our application. The aim of this function is to start all the thread implemented in our system and finally, run the `spin` function in order to link the callbacks of our ROS node with the ROS network. We also use the `spin` to implement an infinite loop.

You can test this program starting the simulation on `CoppeliaSim` and then running this script with a proper launch file that also loads the needed parameters into the ROS parameter server.

7.0.6 Exercise 7

Create a new ROS package to publish the `tf` tree of the `kuka iiwa` simulated on `CoppeliaSim`.

8

Gazebo ROS

Gazebo is a multi-robot simulator for complex indoor and outdoor robotic simulations. We can simulate complex robots, robotic sensors, and a variety of 3D objects. Gazebo already has simulation models of popular robots, sensors, and 3D objects in its repository (`https://bitbucket.org/osrf/gazebo_models/`). We can directly use these models without having to create new ones. Gazebo has a good interface to ROS, which exposes the whole control of the simulation scene. We can install Gazebo without ROS, and we should install the ROS-Gazebo interface to communicate from ROS to Gazebo.

The default version installed from Noetic ROS is Gazebo 11 and it is interfaced with ROS thanks to the following packages:

- `gazebo_ros_pkgs`: This contains wrappers and tools for interfacing ROS with Gazebo
- `gazebo-msgs`: This contains messages and service data structures for interfacing with Gazebo from ROS
- `gazebo-plugins`: This contains Gazebo plugins for sensors, actuators, and so on.
- `gazebo-ros-control`: This contains standard controllers to enable the communication between ROS and Gazebo

To check if Gazebo is properly installed use the following commands:

```
$ roscore
$ rosrun gazebo_ros gazebo
```

These commands will open the Gazebo GUI.

In future lessons, we will see what is behind the Gazebo ROS environment and how robot models are programmed to communicate with the ROS network. However, let's start discussing gazebo interface.

After launched gazebo using the `rosrun` command, some topics are published containing information about the simulation scene and the simulated models. In particular, you can check the topic `/gazebo/model_states` that contains geometrical information about the model spawned into the scene. The types of topics published by gazebo are part of the `gazebo_ros` messages. In the `model_states` case, the type of the message is here reported:

```
string[] name
geometry_msgs/Pose[] pose
        geometry_msgs/Point position
                float64 x
                float64 y
                float64 z
        geometry_msgs/Quaternion orientation
                float64 x
                float64 y
                float64 z
                float64 w
geometry_msgs/Twist[] twist
        geometry_msgs/Vector3 linear
                float64 x
                float64 y
                float64 z
        geometry_msgs/Vector3 angular
                float64 x
                float64 y
                float64 z
```

The size of this message depends on the number of the model active in the simulated environment. Remember that both robots and objects are models. Each model is characterized by its position and velocity. This topic contains the exact pose of your models like a sort of oracle. For this reason, such information typically is used only for ground truth or debug, since in the real world no one can inform your programs about the exact pose of an object (also the GPS is characterized by measure error).

A fundamental step of Gazebo programming regards the configuration of your robot to work with the simulation. In `CoppeliaSim` this requires much more effort, for this reason, we have not discussed how to import your robot in the simulation. Differently, with Gazebo this step is more simple and will help us to deeply understand how to interact with the robot or how to import new robots in the Gazebo world.

8.0.1 Configure a robotic arm for Gazebo simulation

In this section, we will discuss how to modify the `xacro` model file of a 7 DOF manipulator to properly work in Gazebo. To this aim, consider the following `xacro` files modeling a kuka iiwa industrial robot:

- The main file of the kuka robot is the `kuka_iiwa.xacro`

```
<?xml version="1.0"?>
<robot name="kuka_iiwa"
xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename=
"$(find iiwa_description)/urdf/utilities.xacro" />
<xacro:include filename="$(find kuka_iiwa_support)/urdf/
kuka_iiwa_macro.xacro"/>
<xacro:arg name=
"robot_name" default="iiwa"/>
<link name="world"/>

<xacro:iiwa7 robot_name="$(arg robot_name)" parent="world">
<origin xyz="0 0 0" rpy="0 0 0" />
</xacro:iiwa7>
</robot>
```

  In this file we include two additional `xacro` files, one where the `macro` are defined and another one in which are stored some mathematical utilities.

  The robot model accepts the name of the robot as input parameter when it is converted in a URDF.

- The macro block that can be used to build the iiwa robot are contained into the the `kuka_iiwa_macro.xacro` file. We briefly see only a small part of this file, since it is very long. The authors of this robot preferred to define a separate macro block for each joint and link of the robot. However, the full content of this file can be found in the `urdf` folder of the `kuka_iiwa_support` package provided with the course material.

```
<xacro:macro name="iiwa7" params="parent robot_name *origin">
<!--joint between {parent} and link_0-->
<joint name="${parent}_${robot_name}_joint" type="fixed">
<xacro:insert_block name="origin"/>
<parent link="${parent}"/>
<child link="${robot_name}_link_0"/>
</joint>
<link name="${robot_name}_link_0">
<inertial>
```

```
<origin xyz="-0.1 0 0.07" rpy="0 0 0"/>
<mass value="5"/>
<inertia ixx="0.05" ixy="0" ixz="0" iyy="0.06" iyz="0"
</inertial>
<visual>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<mesh filename="package://iiwa_description/meshes/iiwa7/
 visual/link_0.stl"/>
</geometry>
<material name="Grey"/>
</visual>

<collision>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<mesh filename="package://iiwa_description/meshes/iiwa7/
 collision/link_0.stl"/>
</geometry>
<material name="Grey"/>
</collision>
</link>

<joint name="${robot_name}_joint_1" type="revolute">
<parent link="${robot_name}_link_0"/>
<child link="${robot_name}_link_1"/>
<origin xyz="0 0 0.15" rpy="0 0 0"/>
<axis xyz="0 0 1"/>
<limit lower="${-170 *
PI / 180}" upper="${170 * PI / 180}"
 effort="${max_effort}" velocity="${max_velocity}" />
</joint>
<link name="${robot_name}_link_1">
 ...
```

So, such files are quite similar to the ones we saw in Lesson 6. One difference regards how the geometry and the collision of the links are specified. In this case, the shape of a link is directly taken by its CAD file (the `stl` file). Let's see what happens if we try to directly spawn in a Gazebo scene this robot without any modification.

Let's start creating a proper launch file to do this work. First of all, we should specify some arguments that are used by Gazebo simulation:

```
<arg name="paused" default="false"/>
```

```
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>
<arg name="robot_name" default="kuka_iiwa"/>
```

Then, to start gazebo we use a launch file included into another package: the `gazebo_ros` package. To include a launch file in another launch file, we should use the tag `include`:

```
<!-- We resume the logic in empty_world.launch -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
<arg name="debug" value="$(arg debug)" />
<arg name="gui" value="$(arg gui)" />
<arg name="paused" value="$(arg paused)"/>
<arg name="use_sim_time" value="$(arg use_sim_time)"/>
<arg name="headless" value="$(arg headless)"/>
</include>
```

Finally, we can load the robot model in the ROS parameter server:

```
<!-- Load the URDF into the ROS Parameter Server -->
<param name="robot_description" command="
$(find xacro)/xacro '$(find kuka_iiwa_support)/urdf/kuka_iiwa.xacro'" />
```

And use the `urdf_spawner` node to spawn (make appear) the robot in the simulation scene.

```
<!-- Run a python script to the send a service call to gazebo_ros to
spawn a URDF robot -->
<node name="urdf_spawner" pkg="gazebo_ros"
type="spawn_model" respawn="false" output="screen"
args="-urdf -model kuka_iiwa -param robot_description"/>
```

Fig. 8.1 shows what happen if we try to use this launch file:

```
$ roslaunch kuka_iiwa_support gazebo.launch
```

What is happening is that the robot is not able to contrast the gravity, since it has no motor controller defined in its model. In few words, the robot can not be actuated. So, what we will do is to modify the `xacro` file of the robot to add a set of controllers.

Just duplicate the `xacro` file renaming the new file as `kuka_iiwa_ctrl.xacro` and `kuka_iiwa_macro_ctrl.xacro` respectively. One way to actuate the robot is using ROS controllers. To do this, we should define one thing called `<transmission>`. We will define a `<transmission>` element for each joint of the robot.

Here is the macro to define new transmissions:

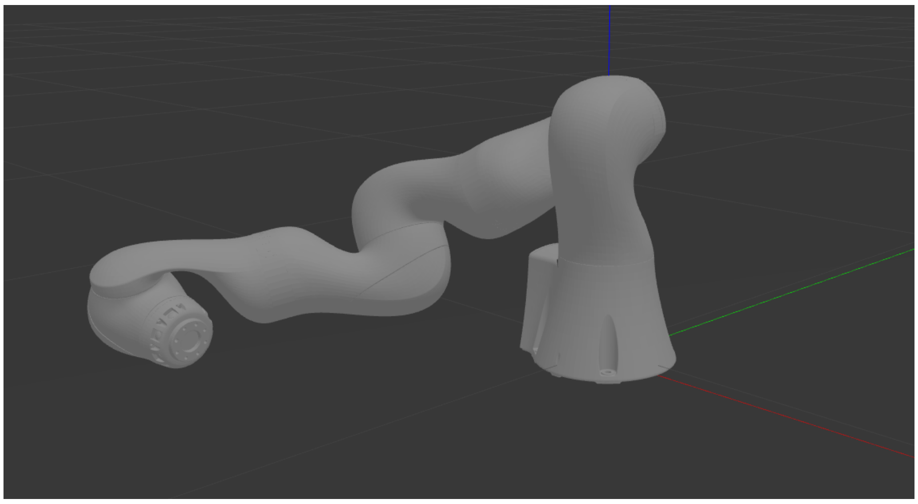


Figure 8.1: Kuka iiwa in ROS gazebo

```
<xacro:macro name="transmission_block" params="joint_name">
<transmission name="${joint_name}_tran1">
<type>transmission_interface/SimpleTransmission</type>
<joint name="${joint_name}">
<hardwareInterface>hardware_interface/
PositionJointInterface</hardwareInterface>
</joint>
<actuator name="${joint_name}_motor">
<hardwareInterface>hardware_interface/
PositionJointInterface</hardwareInterface>
<mechanicalReduction>1</mechanicalReduction>
</actuator>
</transmission>
</xacro:macro>
```

Where:

- `<joint name = "">` is the joint in which we link the actuators.
- `<type>` element is the type of transmission. Currently, `transmission_interface /SimpleTransmission` is only supported.
- `<hardwareInterface>` element is the type of hardware interface to load (position, velocity, or effort interfaces).

This macro should be placed in the `kuka_iiwa_macro.xacro` file, before the block in which the robot is defined.

Of course, we have only defined the macro block. To instantiate it modify the robot model with the following lines:

```
<xacro:transmission_block joint_name="${robot_name}_joint_1"/>
<xacro:transmission_block joint_name="${robot_name}_joint_2"/>
<xacro:transmission_block joint_name="${robot_name}_joint_3"/>
<xacro:transmission_block joint_name="${robot_name}_joint_4"/>
<xacro:transmission_block joint_name="${robot_name}_joint_5"/>
<xacro:transmission_block joint_name="${robot_name}_joint_6"/>
<xacro:transmission_block joint_name="${robot_name}_joint_7"/>
```

Sadly, just adding the transmission tags is not enough for our goal. In fact, we need to enable the `gazebo_ros_control` plugin to load the hardware interface and allow us to control robot joints. The question is... what is a plugin?

8.0.2 Gazebo plugins

Plugins are a commonly used term in the computer world. They are modular pieces of software that can add a new feature to the existing software application. The advantage of plugins is that we don't need to write all the features in the main software; instead, we can make an infrastructure on the main software to accept new plugins to it. Using this method, we can extend the capabilities of the software to any level. Plugin files are runtime libraries, such as shared objects (.so) which are built without linking to the main application code. Plugins are separate entities that do not have any dependencies with the main software. The main advantage of plugins is that we can expand the application capabilities without making many changes in the main application code.

In our context, Gazebo plugins help us to control the robot models, sensors, world properties, and even the way Gazebo runs. Gazebo plugins are a set of C++ code, which can be dynamically loaded/unloaded from the Gazebo simulator. Using plugins, we can access all the components of Gazebo, and also it is independent of ROS.

We will discuss on how to implement a Gazebo plugin in future lessons. For now, just consider the possibility to add plugins to our robot simulated in Gazebo. In our case, we want to include the `gazebo_ros_control` plugin to our kuka iiwa.

8.0.3 Use the gazebo_ros_control plugin

After adding the transmission tags, we should add the `gazebo_ros_control` plugin in the simulation model to parse the transmission tags and assign appropriate hardware interfaces and the control manager. The following code adds the `gazebo_ros_control` plugin to the `kuka_iiwa_macro_ctrl.xacro` file:

```
<gazebo>
```

```
<plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
<robotNamespace>kuka_iiwa</robotNamespace>
<robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
<legacyModeNS>true</legacyModeNS>
</plugin>
</gazebo>
```

This is the common way in which Gazebo plugins can be enabled into robot models spawned in Gazebo scene. Here, the <plugin> element specifies the plugin name to be loaded, which is `libgazebo_ros_control.so`. The <robotNamespace> element can be given as the name of the robot; if we are not specifying the name, it will automatically load the name of the robot from the URDF. We can also specify the controller update rate (<controlPeriod>) and the type of robot hardware interface (<robotSimType>). The default hardware interfaces are JointStateInterface, PositionJointInterface, EffortJointInterface and VelocityJointInterface.

So, <gazebo> element is an extension to the URDF used for specifying additional properties needed for simulation purposes in Gazebo. There are three different types of <gazebo> elements: one for the <robot> tag, one for <link> tags, and one for <joint> tags.

For example, if we want to add colors to the shape of our robot we can include a <gazebo> element referring to a given link:

```
<gazebo reference="${robot_name}_link_0">
<material>Gazebo/Black</material>
</gazebo>
```

At this point, the robot is able to react to the world gravity, so it will remain stable on the initial position (the candle position). However, we have not the possibility to control it. Let's see how to enable the possibility to control the joints of the robot using the `ros_control` package.

8.0.4 Control a simulated robot using ros_control

To move each joint of the kuka iiwa, we need to assign a ROS controller. In particular, for each joint we need to attach a controller that is compatible with the hardware interface mentioned inside the transmission tags of the `xacro` file. A ROS controller mainly consists of a feedback mechanism that can receive a set point and control the output using the feedback from the actuators.

Consider that a ROS controller interacts with the hardware using the hardware interface. The aim of the hardware interface is to act as a mediator between ROS controller output and the real or simulated hardware, allocating the resources to control it considering the data generated by the ROS controller. So, when we are ready to move our application on a real robot, we can maintain the same software structure without carrying out any modification to our source code previously tested in the simulation scene.

The `ros_control` framework is composed by multiple packages:

- `control_toolbox`: This package contains common modules (PID and Sine) that can be used by all controllers.
- `controller_interface`: This package contains the interface base class for controllers-
- `controller_manager`: This package provides the infrastructure to load, unload, start, and stop controllers.
- `controller_manager_msgs`: This package provides the message and service definition for the controller manager.
- `hardware_interface`: This contains the base class for the hardware interfaces.
- `transmission_interface`: This package contains the interface classes for the transmission interface (differential, joint state, position, and velocity).

We also can used different type of ROS controllers:

- `joint_position_controller`: This is a simple implementation of the joint position controller.
- `joint_state_controller`: This is a controller to publish joint states.
- `joint_effort_controller`: This is an implementation of the joint effort (force) controller.

Fig. 8.2 shows how `ros_control` interacts with ROS controller, robot hardware interface, and simulator/real hardware. We can see the third-party tools, such as the navigation and MoveIt packages. These packages can give the goal (set point) to the mobile robot controllers and robotic arm controllers. These controllers can send the position, velocity, or effort to the robot hardware interface. The hardware interface allocates each resource to the controllers and sends values to each resource

The hardware interface is decoupled from actual hardware and simulation. The values from the hardware interface can be fed to Gazebo for simulation or to the actual hardware itself. The hardware interface is a software representation of the robot and its abstract hardware. The resource of the hardware interfaces are actuators, joints, and sensors. Some resources are read-only, such as joint states, IMU, force-torque sensors, and so on, and some are read and write compatible, such as position, velocity, and effort joints.

Try now to interface the kuka iiwa arm with the ros controllers.

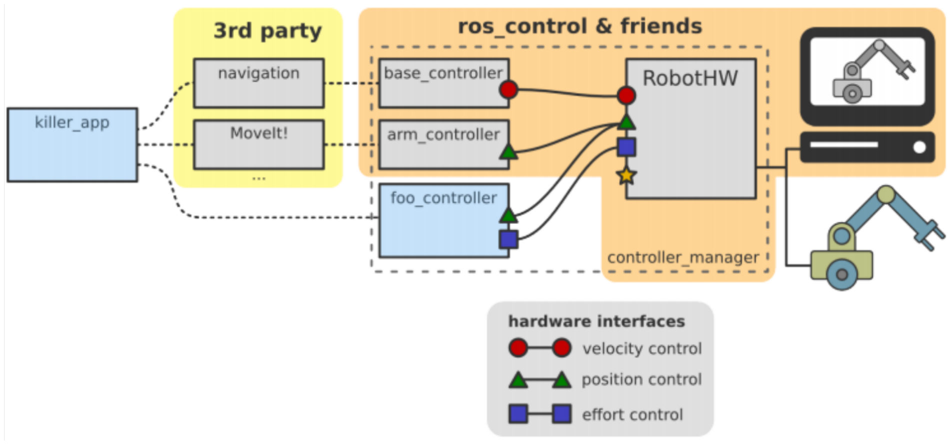


Figure 8.2: Interaction of ROS controllers with Gazebo and real robots

8.0.5 Interfacing joint state controllers and joint position controllers to the arm

Interfacing robot controllers to each joint is a simple task, only two steps are needed:

- Configure the ros controllers using a proper configuration file (`yaml` format)
- Load the configuration file and launch the ros controller node using a `launch` file.

The first task is to write a configuration file for desired controllers. We want to start two controllers: the joint_state_controller/JointStateController controller that provides the information about the joint state and a set of position_controllers/JointPositionController to control the position of each robot joint. Let's create the `yaml` configuration file. We can store it into a `conf` directory of our model package:

```
$ roscd kuka_iiwa_support
$ mkdir conf
$ cd conf && touch kuka_iiwa_controller.yaml
```

Here is reported the content of the configuration file:

```
kuka_iiwa:
```

The first line represents the namespace of the ros controllers.

```
# Publish all joint states -----------------------------------
joint_state_controller:
type: joint_state_controller/JointStateController
publish_rate: 50
```

Then, we are ready to configure the first controller: the joint_state_controller. The configuration requires the name of the controller (in this case joint_state_controller) that is chosen by you and the type of the controller, that can be selected between the available controller type. In this case we chose a JointState-Controller

```
# Position Controllers ---------------------------------------
joint1_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_1
pid: {p: 100.0, i: 0.01, d: 10.0}
joint2_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_2
pid: {p: 100.0, i: 0.01, d: 10.0}
joint3_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_3
pid: {p: 100.0, i: 0.01, d: 10.0}
joint4_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_4
pid: {p: 100.0, i: 0.01, d: 10.0}
joint5_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_5
pid: {p: 100.0, i: 0.01, d: 10.0}
joint6_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_6
pid: {p: 100.0, i: 0.01, d: 10.0}
joint7_position_controller:
type: position_controllers/JointPositionController
joint: iiwa_joint_7
pid: {p: 100.0, i: 0.01, d: 10.0}
```

Finally we can declare a position controller for each joint of the robot.

Let's now fill the launch file to start the controllers. The only thing to add to our launch file is the reported code:

```
<rosparam file="$(find kuka_iiwa_support)/conf/kuka_iiwa_controller.yaml" com
```

We firstly load the configuration file of our controller. Then we can run the node `controller_spawner` of the package `controller_manager`:

```
<node name="controller_spawner" pkg="controller_manager" type="spawner" respa
```

```
joint_state_controller
joint1_position_controller
joint2_position_controller
joint3_position_controller
joint4_position_controller
joint5_position_controller
joint6_position_controller
"/>
```

The arguments of this node are the name of the controllers specified into the `yaml` file. Of course, we have also change the `xacro` file to save into the proper ROS parameter.

After launched this file, you can understand what's happening from the unix shell. In particular, you are informed about all the controller that are loaded:

```
[INFO] [1585730377.037825, 0.224000]: Loading controller:
joint_state_controller
[INFO] [1585730377.047918, 0.234000]: Loading controller:
joint1_position_controller
[INFO] [1585730377.057779, 0.244000]: Loading controller:
joint2_position_controller
[INFO] [1585730377.062676, 0.249000]: Loading controller:
joint3_position_controller
[INFO] [1585730377.068737, 0.255000]: Loading controller:
joint4_position_controller
[INFO] [1585730377.073687, 0.260000]: Loading controller:
joint5_position_controller
[INFO] [1585730377.079606, 0.266000]: Loading controller:
joint6_position_controller
```

... spwaned:

```
[INFO] [1585730377.085591, 0.272000]: Controller Spawner:
Loaded controllers: joint_state_controller,
joint1_position_controller,
joint2_position_controller,
joint3_position_controller,
joint4_position_controller,
joint5_position_controller,
joint6_position_controller
```

...and started:

```
[INFO] [1585730377.088452, 0.275000]: Started controllers:
joint_state_controller,
joint1_position_controller,
joint2_position_controller,
joint3_position_controller,
joint4_position_controller,
joint5_position_controller,
joint6_position_controller
```

Now that everything is ready, you can check the new control topic appeared in your ROS system. In particular, the `ros_control` package publishes the state of the joints:

```
/kuka_iiwa/joint_states
```

while, accepts the desired position requiring a `std_msgs::Float32` data.

```
/kuka_iiwa/joint1_position_controller/command
/kuka_iiwa/joint2_position_controller/command
/kuka_iiwa/joint3_position_controller/command
/kuka_iiwa/joint4_position_controller/command
/kuka_iiwa/joint5_position_controller/command
/kuka_iiwa/joint6_position_controller/command
```

Try to move a desired joint using one of these topics.

8.0.6 CoppeliaSim vs Gazebo ROS

Before to continue studying ROS Gazebo, try to compare it with CoppeliaSim, in order to choose the most suitable solution tool for our aim.

- World/robot modeling: CoppeliaSim offers a lot of models that can be easily inserted in the scene. These models range from infrastructure objects like walls and doors, to furniture, and even terrain models. In this point, Gazebo is far behind. It does not offer many world modeling features out-of-the-box. It does provide a building editor which is very practical to design mazes and basic infrastructure. However, new models can be imported or modified in Gazebo thanks to the URDF and xacro specification. In one sense we could say that Gazebo model import is more complex, but at the same time it is also more powerful.

- ROS integration, CoppeliaSim and Gazebo are natively integrated in ROS. However, in CoppeliaSim you should program Lua (a new language program) scripts to exchange information with the ROS network. In Gazebo you can program directly in C++.

- Realism: Gazebo is more ready for the deployment of your application on real robots. You can use ROS controllers with the same hardware interface for the simulated and real robot to directly deploy your application. Also the sensors are simulated in a more realistic way.

- Simplicity: CoppeliaSim is more simple to use respect to Gazebo. Many things can be done with its User interface. You can also disable the dynamics of the robots, in order to avoid strange behavior if the urdf of your robot contains errors.

9

Mobile robotics

One of the first macro classification among the different robotics field can be made between industrial and mobile robotic. In this context, the object of the classification is the robot itself. We can say that and industrial robot is a autonomous system used for manufacturing and it is capable of movements along and around three axes. Typically industrial robots are huge and dangerous manipulators or flexible, human-robot interaction enabled arms (like the Kuka IIWA seen in previous lessons). Differently, mobile robots are autonomous machines capable of locomotion. Of course, several other classifications can be made considering the type of locomotion and the task that is supposed that a robot should accomplish. In particular, regarding locomotion we can mainly considered wheeled, legged and flying robots. In this lesson, lesson we focus on mobile wheeled robots.

Wheeled robot represent the most popular mobile robots. The most diffuse actuation model for such robots are:

- Steering (or car like): there robots are actuated like cars. A single motor drives two rear wheels moving them in the same direction, while another motor (typically a servo motor) which rotates two frontal wheels.
- Differential drive: two side mounted independently driven wheels which are used for both propulsion and steering.
- Omni-directional: they can move towards all the directions without any constraint

The locomotion of a robot is strictly related to its scope. However, differential drive robots represent a good trade-off between motion capabilities and control complexity. Let's start to see how to build a model for mobile differential driven robot.

9.1 Creating a robot model for the differential drive mobile robot

A differential wheeled robot will have two wheels connected on opposite sides of the robot chassis, which is supported by one or two caster wheels. The wheels will control the speed of the robot by adjusting individual velocity. If the two motors are running at the same speed, the wheels will move forward or backward. If one wheel is running slower than the other, the robot will turn to the side of the lower speed. If we want to turn the robot to the left side, we reduce the velocity of the left wheel, and vice versa.

There are two supporting wheels, called caster wheels, that will support the robot and rotate freely based on the movement of the main wheels.

The URDF model of this robot is present in the cloned ROS package. The final robot model is reported in Fig. 9.1.

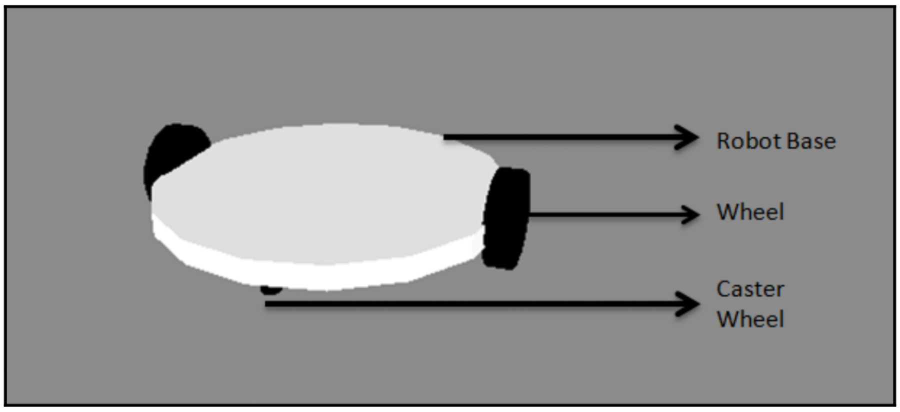


Figure 9.1: Wheeled differential mobile robot

This robot robot has five joints and links. The two main joints connect the wheels to the robot, while the others are fixed joints connecting the caster wheels and the base footprint to the body of the robot. We can create the `xacro` file to design this robot into the `urdf` folder of the `rl_robot_description_pkg` ROS package.

As already made for the `pan_tilt` robot, create two `xacro` files.

```
$ roscd rl_robot_description_pkg/urdf/
$ touch touch diff_robot_macro.xacro
$ touch touch diff_robot.xacro
```

Our goal is to define the following elements:

- `base_link`: the base frame of the mobile robot
- `caster_link`: two links representing the passive wheels of the robot

- `caster joint`: two fixed jonins connecting the `base_link` with the casters
- `wheels`: two actuated wheel with continuous joints allowing the motion of the robot

Start analyzing the macro file.

Before starting to include the macros, we define some properties. In particular, we define also the color of the material of the robotic links. This is useful only to display the robot in RViz. As already seen the same will be made to import the robot in Gazebo.

```
<robot name="wheel" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:property name="M_PI" value="3.1415926535897931" />
<xacro:property name="M_PI_2" value="1.570796327" />
<xacro:property name="DEG_TO_RAD" value="0.017453293" />

<!--Material Definition-->
<material name="Black">
<color rgba="0.0 0.0 0.0 1.0"/>
</material>

<material name="Red">
<color rgba="0.8 0.0 0.0 1.0"/>
</material>

<material name="White">
<color rgba="1.0 1.0 1.0 1.0"/>
</material>

<material name="Blue">
<color rgba="0.0 0.0 0.8 1.0"/>
</material>
```

Then, we define some properties to characterize the structure of the robot. In particular, the dimension and the messes of the wheels and casters.

```
<!-- Base properties -->
<xacro:property name="base_height" value="0.02" />
<xacro:property name="base_radius" value="0.15" />
<xacro:property name="base_mass" value="5" />

<!-- caster wheel proprieties -->
<xacro:property name="caster_height" value="0.04" />
<xacro:property name="caster_radius" value="0.025" />
<xacro:property name="caster_mass" value="0.5" />
```

```

<!-- Wheels -->
<xacro:property name="wheel_radius" value="0.04" />
<xacro:property name="wheel_height" value="0.02" />
<xacro:property name="wheel_mass" value="2.5" />

<xacro:property name="base_x_origin_to_wheel_origin" value="0.25" />
<xacro:property name="base_y_origin_to_wheel_origin" value="0.3" />
<xacro:property name="base_z_origin_to_wheel_origin" value="0.0" />
```

We are now ready to include the caster joint macro. We define this joint as a fixed joint.

```
<xacro:macro name="caster_joint" params="name parent child *origin">
<joint name="${name}" type="fixed" >
<parent link="${parent}" />
<child link="${child}" />
<xacro:insert_block name="origin" />
</joint>
</xacro:macro>
```

Then, we can define the link representing the caster:

```
<xacro:macro name="caster_link" params="name *origin">
<link name="${name}">
<visual>
<xacro:insert_block name="origin" />
<geometry>
<sphere radius="${caster_radius}" />
</geometry>
<material name="Black" />
</visual>
<collision>
<geometry>
<sphere radius="${caster_radius}" />
</geometry>
<origin xyz="0 0.02 0" rpy="${M_PI/2} 0 0" />
</collision>
<inertial>
<mass value="${caster_mass}" />
<origin xyz="0 0 0" />
<inertia  ixx="0.001" ixy="0.0" ixz="0.0"
iyy="0.001" iyz="0.0"
izz="0.001" />
</inertial>
</link>
```

```
</xacro:macro>
```

Now, we define the wheels of the robot. In this case, wa define a cylindrical link (the wheel) and a continuous joint allowing the infinite rotation of the link. Notice how we considered one macro for both wheels using mathematical expressions.

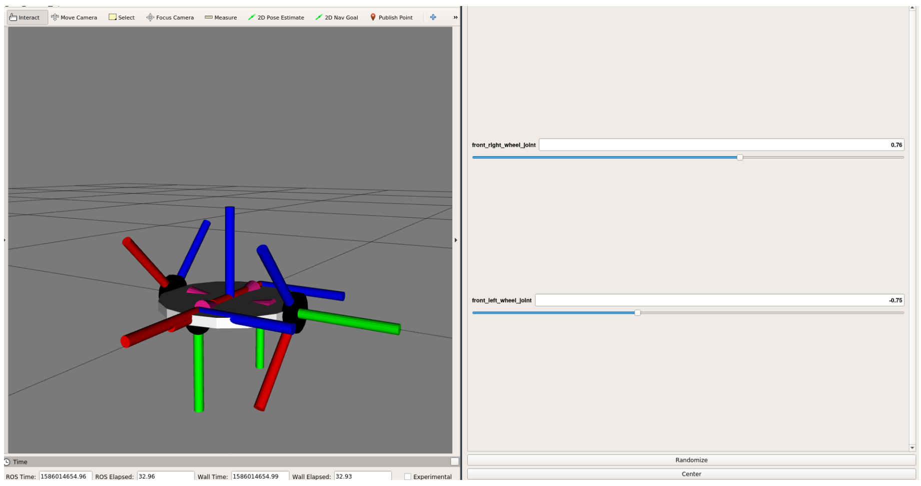
```
<xacro:macro name="wheel" params="fb lr parent translateX translateY flipY">
<link name="${fb}_${lr}_wheel">
<visual>
<origin xyz="0 0 0" rpy="${flipY*M_PI/2} 0  0 " />
<geometry>
<cylinder length="${wheel_height}" radius="${wheel_radius}" />
</geometry>
<material name="DarkGray" />
</visual>
<collision>
<origin xyz="0 0 0" rpy="${flipY*M_PI/2} 0 0 " />
<geometry>
<cylinder length="${wheel_height}" radius="${wheel_radius}" />
</geometry>
</collision>
<inertial>
<mass value="${wheel_mass}" />
<origin xyz="0 0 0" />
<cylinder_inertia  m="${wheel_mass}" r="${wheel_radius}" h="${wheel_height}"
</inertial>
</link>

<joint name="${fb}_${lr}_wheel_joint" type="continuous">
<parent link="${parent}"/>
<child link="${fb}_${lr}_wheel"/>
<origin xyz="${translateX * base_x_origin_to_wheel_origin} ${translateY * bas
<axis xyz="0 1 0" rpy="0  0" />
<limit effort="100" velocity="100"/>
<joint_properties damping="0.0" friction="0.0"/>
</joint>

</xacro:macro>
```

As for the contents of the `diff_robot.xacro` file, we firstly define the `base_link` of the robot, then include the element implemented as macro in the macro file:

```
<!--Actual body/chassis of the robot-->
<link name="base_link">
```



front_right_wheel_joint 0.76
front_left_wheel_joint -0.75
ROS Time: 1586014654.96 ROS Elapsed: 32.96 Wall Time: 1586014654.99 Wall Elapsed: 32.93

Figure 9.2: Display differential mobile robot using RViz.

```
<inertial>
<mass value="${base_mass}" />
<origin xyz="0 0 0" />
<cylinder_inertia  m="${base_mass}" r="${base_radius}" h="${base_heig
</inertial>
<visual>
<origin xyz="0 0 0" rpy="0 0 0" />
<geometry>
<cylinder length="${base_height}" radius="${base_radius}" />
</geometry>
<material name="White" />
</visual>
<collision>
<origin xyz="0 0 0" rpy="0 0 0 " />
<geometry>
<cylinder length="${base_height}" radius="${base_radius}" />
</geometry>
</collision>
</link>

<xacro:caster_joint
name="caster_front_link"
 parent="base_link"
 child="caster_front_link">
<origin xyz="0.115 0.0 0.007" rpy="${-M_PI/2} 0 0"/>
</xacro:caster_joint>

```

```
<xacro:caster_link
name="caster_front_link">
<origin xyz="0 0.02 0" rpy="${M_PI/2} 0 0" />
</xacro:caster_link>

<xacro:caster_joint
name="caster_back_joint"
 parent="base_link"
 child="caster_back_link">
<origin xyz="-0.135 0.0 0.009" rpy="${-M_PI/2} 0 0"/>
</xacro:caster_joint>

<xacro:caster_link
name="caster_back_link">
<origin xyz="0.02 0.02 0 " rpy="${M_PI/2} 0 0" />
</xacro:caster_link>


<wheel fb="front" lr="right" parent="base_link" translateX="0" translateY="0
<wheel fb="front" lr="left" parent="base_link" translateX="0" translateY="-0
```

In order to check that the robot has been designed correctly, create a proper launch file named `display_mobile_robot.launch` configuring the RobotModel plugin to display the `xacro` model using RViz. The result should be similar to the one shown in Fig. 9.2.

9.1.1 Simulate differential mobile robot in Gazebo

The next step to simulate the mobile robot in Gazebo is to add the transmission and the `ros_control` plugin to properly actuate the wheels of the robot, like we made for the iiwa arm. We need to add two different transmission, one for each joint controlling the wheels. Let's to edit he macro file: `diff_robot_macro.xacro` file. You can add the transmission element into the wheel macro block:

```
<transmission name="${fb}_${lr}_wheel_joint_trans">
<type>transmission_interface/SimpleTransmission</type>
<joint name="${fb}_${lr}_wheel_joint" />
<actuator name="${fb}_${lr}_wheel_joint_motor">
<hardwareInterface>EffortJointInterface</hardwareInterface>
<mechanicalReduction>1</mechanicalReduction>
</actuator>
</transmission>
```

We need to add the `ros_control` plugin. In the iiwa arm case, we just want to control each joint of the robot specifying the desired position. When we control a mobile robot, typically we want to specify a desired velocity for its base. In particular, the linear and rotational velocity are reasonable input for a mobile base. In ROS this control input travels using a `geometry_msgs::Twist`:

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

The published command must be translate into a desired velocity for the two wheels of the robot. This step is typically made with a proper allocation matrix considering the kinematic of the robotic platform. In Gazebo, this behavior can be modeled using the `libgazebo_ros_diff_drive` plugin. We can add this plugin in the `diff_robot.xacro` file:

```
<gazebo>
<plugin name="differential_drive_controller" filename="libgazebo_ros_
<legacyMode>true</legacyMode>
<rosDebugLevel>Debug</rosDebugLevel>
<publishWheelTF>false</publishWheelTF>
<robotNamespace>/</robotNamespace>
<publishTf>1</publishTf>
<publishWheelJointState>false</publishWheelJointState>
<alwaysOn>true</alwaysOn>
<updateRate>100.0</updateRate>
<leftJoint>front_left_wheel_joint</leftJoint>
<rightJoint>front_right_wheel_joint</rightJoint>
<wheelSeparation>${2*base_radius}</wheelSeparation>
<wheelDiameter>${2*wheel_radius}</wheelDiameter>
<broadcastTF>1</broadcastTF>
<wheelTorque>30</wheelTorque>
<wheelAcceleration>1.8</wheelAcceleration>
<commandTopic>cmd_vel</commandTopic>
<odometryFrame>odom</odometryFrame>
<odometryTopic>odom</odometryTopic>
<robotBaseFrame>base_link</robotBaseFrame>
</plugin>
</gazebo>
```

We can provide the parameters such as the wheel joints of the robot (joints should be of a continuous type), wheel separation, wheel diameters, odometry topic, and so on, in this plugin. Among the different parameters an the `<commandTopic>cmd_vel</commandTopic>` one specifies the command velocity topic to the plugin, which is basically a Twist message in ROS (`sensor_msgs/Twist`). We can publish the Twist message into the `/cmd_vel` topic, and we can see the robot start to move from its position.

As usual, the last step to start our simulation is to write a convenient launch file to start the simulation spawning the mobile robot. Create a launch file into our package:

```
$ roscd rl_robot_description_pkg
$ mkdir launch && cd launch
$ touch spawn_diff_robot.launch
```

Here is the content of the launch file:

```
<?xml version="1.0" ?>
<launch>
<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>

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

<param name="robot_description" command="$(find xacro)/xacro '$(find rl_robot

<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
 args="-urdf -model diff_wheeled_robot -param robot_description"/>
</launch>
```

You can launch this file and discover a list of new topic helping us to interact with the mobile robot.

- `cmd_vel`: the `geometry_msgs::Twist` message used to control the robot. This is the input of the differential drive controller plugin. In particular, the `x` element of the linear part of the message controls the forward velocity of the robot, while the `z` element of the angular part of the message structure controls the angular motion.

- `odom`: This topic publishes the odometry data calculated by the gazebo differential drive plugin. Odometry is the use of data from motion sensors to estimate change in position over time. In wheeled robot the measures get by encoders. We will return on this point, but consider that odometry is not used as is for smart navigation.

What is the difference between the configuration of this robot and the kuka iiwa? In the kuka iiwa case, we use a plugin in gazebo who simulates an hardware interface for `ros_control` package. In this way, if you implement a proper hardware interface for your *real* robot, the software structure will be exactly the same. In this case, we are just using a plugin implemented in gazebo, who accepts the desired body velocity and convert these into the wheels velocity. This means that when you want to move your software on a real robot, you must do additional work. Of course, `ros_control` also include a library to implement a differential drive controller: the `diff_drive_contoller` (`http://wiki.ros.org/diff_drive_controller?distro=melodic`.

9.1.2 Control the differential drive robot

The overall goal of robotics application is to allow robot to perform actions in a complete autonomy or, at least with a minimal cooperation with a human operator. However, directly control your robot could help developers to test the capabilities of the robot and the developed algorithm as well. One of the most simple way to interact with robots is using standard joypad. This is particularly evident in case of mobile robots, since you can use the joypad to move it into the scene. We will see how sometimes is important to explore manually the environment before to run the autonomous application. However, joypad can be useful also in case of robot manipulators. Consider that you want to test a control algorithm for the pose of the end effector, or just test the motion along one joint.

ROS supports several joypad devices. In particular, if you have a bluetooth or USB joypad you could test the joy node (`http://wiki.ros.org/joy`). This node can be installed with the following command:

```
$ sudo apt-get install ros-melodic-joy
```

This package reads the hardware connected to your robot/computer publishing a `sensor_msgs/Joy` data, containing the value of the joypad axes and buttons.

Let's try to simulate the behavior of a joypad using our keyboard, in order to control the differential drive robot modeled in this section. The goal here is to read data from the keyboard and then publish a geometry_msgs/Twist message on the `cmd_vel` topic. Of course, here we have not axes so we must use the keyboard to generate the velocity message. Create a `key_teleop` package:

```
$ roscd && cd .. && cd src
$ catkin_create_pkg key_teleop roscpp geometry_msgs
```

Create a new source file called `key_teleop.cpp`. One way to specify the desired velocity is to use the following keys. We can use the *w* and *x* keys to specify a linear velocity in the forward and backward direction respectively. An additional idea could be that, if the robot is moving along a positive direction and the user requires for a negative direction, the robot stops its linear motion. The same thing happen for the angular rotation motion invoked with the *a* and *d* keys.

As usual, we include the desired header file. We also define two static variable for specifying the linear and angular velocity.

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "boost/thread.hpp"
#include <iostream>
using namespace std;

#define LIN_VEL 0.2
#define ANG_VEL 0.8
```

Then we can define the `KEY_CTRL` class. Among its member the `_fv` and `_rv` store the desired forward and rotation velocities. In its constructor, we instantiate the publisher of the velocity command.

```
class KEY_CTRL {
public:
KEY_CTRL();
void key_input();
void run();
void vel_ctrl();
private:
ros::NodeHandle _nh;
ros::Publisher  _vel_pub;
float _fv; //Forward velocity
float _rv; //Rotational velocity
};
KEY_CTRL::KEY_CTRL() {
_vel_pub = _nh.advertise< geometry_msgs::Twist >("/cmd_vel", 0);
}
```

One of the thread of our node is in charge of get keyboard input. As discussed, we use the w, x, a, d buttons to generate the velocity and the s button to stop the robot.

```
void KEY_CTRL::key_input() {
```

```
string input;
cout << "Keyboard Input: " << endl;
cout << "[w]: Forward direction velocity" << endl;
cout << "[x]: Backward direction velocity" << endl;
cout << "[a]: Left angular velocity" << endl;
cout << "[d]: Right angular velocity" << endl;
cout << "[s]: stop the robot!" << endl;
while (ros::ok()) {
getline( cin, input);
if( input == "w" )
_fv = (_fv < 0.0 ) ? 0.0 : LIN_VEL;
else if( input == "x" )
_fv = (_fv > 0.0 ) ? 0.0 : -LIN_VEL;
else if( input == "a" )
_rv = (_rv > 0.0 ) ? 0.0 : -ANG_VEL;
else if( input == "d" )
_rv = (_rv < 0.0 ) ? 0.0 : ANG_VEL;
else if( input == "s" )
_fv = _rv = 0.0;
}
}
```

In the control function, we just set the output of the node (the `cmd_vel` topic).

```
void KEY_CTRL::vel_ctrl() {
ros::Rate r(10);
geometry_msgs::Twist cmd_vel;
while(ros::ok()) {
cmd_vel.linear.x = _fv;
cmd_vel.angular.z = _rv;
_vel_pub.publish( cmd_vel );
r.sleep();
}
}
```

Finally, as already seen in previous examples, in he `run` function we start the desired threads: the one to get the input and the one to publish the velocities.

```
void KEY_CTRL::run() {
boost::thread key_input_t( &KEY_CTRL::key_input, this );
boost::thread vel_ctrl_t( &KEY_CTRL::vel_ctrl, this );
ros::spin();
}
```

```
int main(int argc, char** argv ) {
ros::init(argc, argv, "key_ctrl");
KEY_CTRL kc;
kc.run();
return 0;
}
```

Compile this package and test the node after launched the simulation scene:

```
$ roslaunch rl_robot_description_pkg spawn_diff_robot.launch
$ rosrun key_teleop key_teleop
```

9.2 Gazebo sensor simulation

An important feature of Gazebo is the possibility to simulate a variety of sensors other than robots. The interface between a real and a simulated sensor is exactly the same. Some of these sensors are already implemented and available in the Gazebo installation, however, a variety of sensors have been released by other developer and can be imported in your simulation scene. Just sensors most commonly used in the development of a robotic application and already included in the default installation of Gazebo. As you can imagine, in Gazebo a sensor is simulated by means of a dedicated plugin.

To test such sensors we could create a proper package to store all the configuration file needed to start them. Create the `gazebo_sensors` package:

```
$ catkin_create_pkg gazebo_sensors roscpp
gazebo_plugins gazebo_ros sensor_msgs tf
```

The idea is to define a base object and attach to it different sensors to test. Create an `urdf` directory and fill the `base.xacro` file with the following content:

```
<?xml version="1.0"?>

<robot name="sensor" xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:property name="box_height" value="0.2" />
<xacro:property name="box_width" value="0.2" />

<link name="world"/>
<joint name="fixed" type="fixed">
<parent link="world"/>
<child link="sensor"/>
</joint>
```

```
<link name="sensor">
<collision>
<origin xyz="0 0 ${box_height/2}" rpy="0 0 0"/>
<geometry>
<box size="${box_width} ${box_width} ${box_height}"/>
</geometry>
</collision>
<visual>
<origin xyz="0 0 ${box_height/2}" rpy="0 0 0"/>
<geometry>
<box size="${box_width} ${box_width} ${box_height}"/>
</geometry>
<material name="orange"/>
</visual>
<inertial>
<origin xyz="0 0 1" rpy="0 0 0"/>
<mass value="1"/>
<inertia
ixx="1.0" ixy="0.0" ixz="0.0"
iyy="1.0" iyz="0.0"
izz="1.0"/>
</inertial>
</link>
<material name="orange">
<color rgba="${255/255} ${108/255} ${10/255} 1.0"/>
</material>
<gazebo reference="sensor">
<material>Gazebo/Blue</material>
</gazebo>
</robot>
```

This model just represents a box of 20 centimeters called `sensor`. We can include this from the model file of the sensors.

Camera sensor

Camera sensors are one of the most common sensors used in robotics. Camera are cheap and reliable sensors that can be used in different contexts. Typically, cameras are used to implement image elaboration algorithms able to detect and recognize shapes or known objects. Also color based approach can be used with camera. One of the biggest problems with such sensors arise behind the fact that it's hard to retrieve depth information by a gathered image. In fact, with 2D images you can only estimate the distance between a target and an image using triangulation techniques.

The simulated camera is implemented by the `libgazebo_ros_camera.so` plugin. The parameters to configure this plugin are:

- `update_rate`: the frequency of the updated of the camera sensor. Commercial devices usually run at 30 or 60 Hz.
- `horizontal_fov`: the field of view. It represents the part of the world that is visible through the camera, expressed in radians.
- `width`: the width of the image
- `height`: the height of the image
- `format`: the format of the image. A colored image format is: R8G8B8
- `near`: how close the images are visible by the camera
- `far`: how far the image are visible by the camera

You can set these parameters in the `xacro` file instantiating the camera. Create the `camer.xacro` file and fill it with the following content:

```
<?xml version="1.0"?>

<robot name="sensor" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find gazebo_sensors)/urdf/base.xacro"/>
<gazebo reference="sensor">
<sensor type="camera" name="camera1">
<update_rate>30.0</update_rate>
<camera name="head">
<horizontal_fov>1.3962634</horizontal_fov>
<image>
<width>800</width>
<height>600</height>
<format>R8G8B8</format>
</image>
<clip>
<near>0.02</near>
<far>300</far>
</clip>
<noise>
<type>gaussian</type>
<mean>0.0</mean>
<stddev>0.007</stddev>
</noise>
</camera>
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
```

```
<alwaysOn>true</alwaysOn>
<updateRate>0.0</updateRate>
<cameraName>/camera</cameraName>
<imageTopicName>image_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
<frameName>camera_link</frameName>
<hackBaseline>0.07</hackBaseline>
<distortionK1>0.0</distortionK1>
<distortionK2>0.0</distortionK2>
<distortionK3>0.0</distortionK3>
<distortionT1>0.0</distortionT1>
<distortionT2>0.0</distortionT2>
</plugin>
</sensor>
</gazebo>
</robot>
```

In this file we can also specify the name of the topic where the images and the camera calibration are published:

- `imageTopicName`: this topic contains the image stream. The type of this message is: `sensor_msgs::Image`.
- `cameraInfoTopicName`: streams information about the calibration of the camera. Typically, image elaboration algorithms require the calibration of the camera to transform pixel information into 2D Cartesian information. Usually, before to use a camera you should calibrate it using standard techniques, of course in the simulated world the calibration is already available using a `sensor_msgs::CameraInfo` message.

To start this sensor use the following `camera.launch` file:

```
<?xml version="1.0" ?>
<launch>

<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>
<arg name="verbose" default="false"/>
<arg name="world_name" default="worlds/empty.world"/> <!-- Note: the

<!-- Start gazebo and load the world -->
<include file="$(find gazebo_ros)/launch/empty_world.launch" >
<arg name="paused" value="$(arg paused)"/>
```

```
<arg name="use_sim_time" value="$(arg use_sim_time)"/>
<arg name="gui" value="$(arg gui)"/>
<arg name="headless" value="$(arg headless)"/>
<arg name="debug" value="$(arg debug)"/>
<arg name="verbose" value="$(arg verbose)"/>
<arg name="world_name" value="$(arg world_name)"/>
</include>

<!-- Spawn the example robot -->
<param name="robot_description" command="$(find xacro)/xacro --inorder '$(fin
<node pkg="gazebo_ros" type="spawn_model" name="spawn_model" args="-urdf -par
<node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_st
</launch>
```

RGB-D sensor

A very popular sensor used in robotics in the last decade is the RGB-D sensor. In this context, the D stands for Depth. One of the first sensor of this type spawned into the marker was the kinect sensor. Recently, Intel is spending a lot of effort to produce new generation sensors for robotics (i.e. Realsense sensors), see Fig. 9.3. The aim of these sensor is to provide both information about the 2d image in the form of colored image (RGB) and also the spatial information relate to each pixel of the image. In particular, users are able to reconstruct the 3d point representing the position of each pixel in the image plane. The format to represent such information is called `PointCloud`, we will discuss about Point Cloud elaboration in next lessons, for now just consider the possibility to spawn this sensor in gazebo and visualize its data. The plugin implementing gazebo a RGB-D sensor is the

Figure 9.3: RGB-D sensor produced by Intel

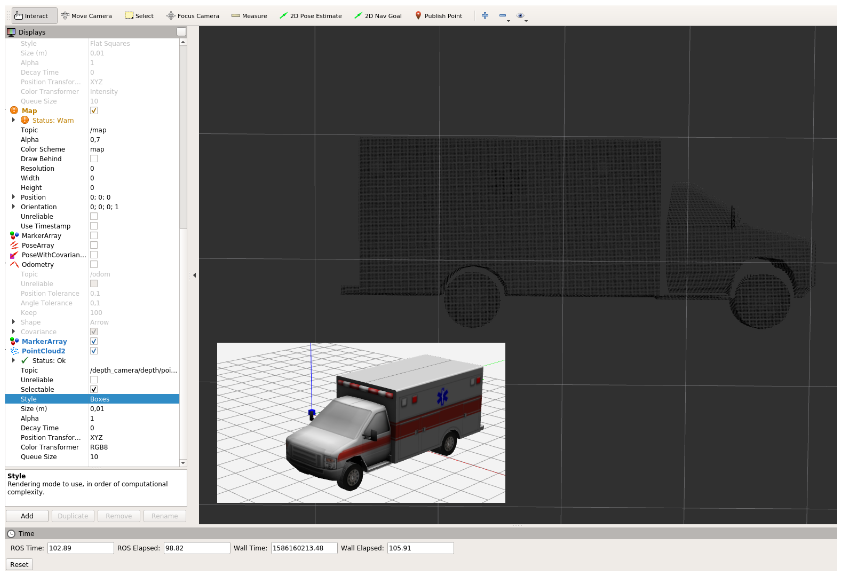
`libgazebo_ros_openni_kinect.so` plugin.

To include it we have only to create a new `xacro` file including the `base.xacro` file and the launch file to import it into gazebo scene. Create the `rgbd.xacro` file:

```
<?xml version="1.0"?>
<robot name="sensor" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find gazebo_sensors)/urdf/base.xacro"/>
<gazebo reference="sensor">
<sensor type="depth" name="$depth_camera">
<always_on>true</always_on>
<update_rate>30.0</update_rate>
<camera>
<horizontal_fov>2</horizontal_fov>
<image>
<format>L8</format>
<width>640</width>
<height>480</height>
</image>
<clip>
<near>0.01</near>
<far>100</far>
</clip>
</camera>
<plugin name="depth_camera_plugin" filename="libgazebo_ros_openni_kin
<alwaysOn>true</alwaysOn>
<baseline>0.11</baseline>
<updateRate>30.0</updateRate>
<cameraName>depth_camera</cameraName>
<imageTopicName>depth_camera/image_raw</imageTopicName>
<cameraInfoTopicName>depth_camera/camera_info</cameraInfoTopicName>
<depthImageTopicName>depth/disparity</depthImageTopicName>
<depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfo
<pointCloudTopicName>depth/points</pointCloudTopicName>
<frameName>rgbd_link</frameName>
<pointCloudCutoff>0.5</pointCloudCutoff>
<distortionK1>0.0</distortionK1>
<distortionK2>0.0</distortionK2>
<distortionK3>0.0</distortionK3>
<distortionT1>0.0</distortionT1>
<distortionT2>0.0</distortionT2>
</plugin>
</sensor>
</gazebo>
</robot>
```

Here the plugin parameters are very similar to the one considered for the camera model. Edit the camera launch file to spawn the `rgbd` sensor instead. After loaded this new sensor in the simulation scene, you could get its data using the following topic:

- `/depth_camera/depth_camera/image_raw`: Like a standard camera, a RGB-D sensor publishes images on a `sensor_msgs::Image`.
- `/depth_camera/depth_camera/camera_info`: Again, like a standard camera a RGB-D sensor publishes images on a `sensor_msgs::CameraInfo` message.
- `/depth_camera/depth/disparity`: This is a particular type of image that is published by all the depth sensors. This image encodes the information about the distance of all the element grabbed by the sensor respect the sensor itself.
- `/depth_camera/depth/camera_info`: A depth camera also has a separate calibration for the published disparity image.
- `/depth_camera/depth/points`: Combining the disparity image and its calibration is possible to generate 3d spatial information for each pixel of the colored image. This information is encoded into a sensor_msgs/PointCloud2 data type. Fig. 9.4 shows the output of pointcloud data generated by the depth sensor after included an ambulance in the gazebo scene.



```
Style                 Flat Squares
Size (m)              0,01
Alpha                 1
Decay Time            0
Position Transfor...  XYZ
Color Transformer     Intensity
Queue Size            10
Map                   ✓
  Status: Warn
  Topic               /map
  Alpha               0,7
  Color Scheme        map
  Draw Behind         ☐
  Resolution          0
  Width               0
  Height              0
  Position            0; 0; 0
  Orientation         0; 0; 0; 1
  Unreliable          ☐
  Use Timestamp       ☐
MarkerArray           ☐
PoseArray             ☐
PoseWithCovarian...   ☐
Odometry              ☐
  Topic               /odom
  Unreliable          ☐
  Position Tolerance  0,1
  Angle Tolerance     0,1
  Keep                100
  Shape               Arrow
  Covariance          ✓
MarkerArray           ✓
PointCloud2           ✓
  Status: Ok
  Topic               /depth_camera/depth/poi...
  Unreliable          ☐
  Selectable          ✓
  Style               Boxes
  Size (m)            0,01
  Alpha               1
  Decay Time          0
  Position Transfor...  XYZ
  Color Transformer   RGB8
  Queue Size          10

Style
Rendering mode to use, in order of computational complexity.

Time
ROS Time: 102.89   ROS Elapsed: 98.82   Wall Time: 1586160213.48   Wall Elapsed: 105.91
```

Figure 9.4: PCL displayed in RViz

LIDAR sensor

A very popular sensor used mainly in robot navigation is the LIDAR sensor. Using such sensors is possible to retrieve the distance of an object respect to the sensor. Laser scanners are very precise with a long operative range (up to 40 meters). In addition, they emits a circular laser beam able to detect objects around the whole circumference of the robot. The main classification in terms of laser scanner for robotic applications arise behind the working space. In particular, we can distinguish between 2D and 3D laser scanners. Their shapes is very compact and similar to the one shown in Fig. 9.5. In recently years, 3d laser scanners became very popular for autonomous driving applications. Let's see how to add a 2D lidar in Gazebo. The plugin

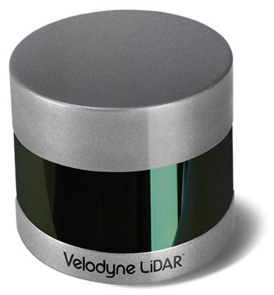


Figure 9.5: Velodyine 3d LIDAR sensor

implementign the sensor is the `libgazebo_ros_laser.so` plugin. As usual, create the `laser.xacro` file to enable the plugin into the scene:

```
<?xml version="1.0"?>

<robot name="sensor" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find sensor_sim_gazebo)/urdf/sensor.xacro"/
<gazebo reference="sensor">
<sensor type="ray" name="head_hokuyo_sensor">
<pose>0 0 0 0 0 0</pose>
<visualize>false</visualize>
<update_rate>40</update_rate>
<ray>
<scan>
<horizontal>
<samples>720</samples>
<resolution>1</resolution>
<min_angle>−1.570796</min_angle>
<max_angle>1.570796</max_angle>
</horizontal>
</scan>
```

```
<range>
<min>0.8</min>
<max>30.0</max>
<resolution>0.01</resolution>
</range>
<noise>
<type>gaussian</type>
<mean>0.0</mean>
<stddev>0.01</stddev>
</noise>
</ray>
<plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_lase
<topicName>/laser/scan</topicName>
<frameName>world</frameName>
</plugin>
</sensor>
</gazebo>
</robot>
```

The most important parameters to configure this plugin are:

- `min_angle`: The minimum angle on the horizontal plane
- `max_angle`: The maximum angle on the horizontal plane
- `min`: the minimum range range to detect objects
- `max`: the maximum range range to detect objects

For example if you want to detect have a full beam on the horizontal plan you should specify your maximum and minimum angles in order to have 6.28 radians of field of view. In our case, the sensor is only able to detect objects placed ahead to it. Write a launch a proper launch file to spawn the laser in gazebo and check the topic list.

Laser scanner publishes its output using a `sensor_msgs::LaserScan` message. This type of message stores a the distance detected by the sensor in a static huge vector (`ranges`), where, in each location is reported the distance between the sensor and scene object relative to a specific angle of the sensor.

```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
```

```
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

You can also visualize using RViz the output of this sensor, just add some objects to the gazebo scene and open RViz adding the visualization of a `LaserScan` message.

IMU sensor

Finally, last sensor proposed in this document is an inertial sensor. The Inertial Measurement Unit sensor is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. Is commonly used on quadrotors to estimate their attitude, but you can also find it on several mobile robots working in outdoor scenarios to estimate its Yaw (orientation around Z axis). This is the `xacro` file for the IMU sensor:

```
<?xml version="1.0"?>

<robot name="sensor" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find gazebo_sensors)/urdf/base.xacro"/>
<gazebo>
<plugin name="imu_plugin" filename="libgazebo_ros_imu.so">
<alwaysOn>true</alwaysOn>
<bodyName>sensor</bodyName>
<topicName>imu</topicName>
<serviceName>imu_service</serviceName>
<gaussianNoise>0.0</gaussianNoise>
<updateRate>20.0</updateRate>
</plugin>
</gazebo>
</robot>
```

The `libgazebo_ros_imu` publishes data on `sensor_msgs/Imu` message, containing the sensor orientation, linear acceleration and angular velocity.

10

Gazebo plugins

Gazebo plugins help us to control the robot models, sensors, world properties, and even the way Gazebo runs. Gazebo plugins are a set of C++ code, which can be dynamically loaded/unloaded from the Gazebo simulator. Using plugins, we can access all the components of Gazebo, and also it is independent of ROS, so that it can share with people who are not using ROS. We can mainly classify the plugins as follows:

- The world plugin: Using the world plugin, we can control the properties of a specific world in Gazebo. We can change the physics engine, the lighting, and other world properties using this plugin.

- *The model plugin*: The model plugin is attached to a specific model in Gazebo and controls its properties. The parameters, such as the joint state of the model,control of the joints, and so on, can be controlled using this plugin.

- The sensor plugin: The sensor plugins are for modeling sensors, such as camera, IMU, and so on, in Gazebo.

- The system plugin: The system plugin is started along with the Gazebo startup. A user can control a system-related function in Gazebo using this plugin.

- The visual plugin: The visual property of any Gazebo component can be accessed and controlled using the visual plugin.

Before starting development with Gazebo plugins, we might need to install some packages. The Gazebo version installed along with ROS Melodic is 9.0, so you might need to install its development package in Ubuntu using the following command:

```
$ sudo apt-get install libgazebo9-dev
```

The Gazebo plugins are independent of ROS and we don't need ROS libraries to build a plugin. However could be useful implement a communication between ROS and Gazebo plugins to simplify the overall control of our robot.

In this chapter we will see how to implement gazebo plugins from scratch and how the inter process communication between different gazebo elements works. Let's start with a basic gazebo plugin.

10.1 Creating a basic world plugin

We will look at a basic Gazebo world plugin and try to build and load it in Gazebo. In this first example, we will not link the plugin with ROS network. Start creating a folder called `hello_world_plugin` in a desired location of your system (you can also use the ROS workspace). We will store the plugin source in the `src` sub-directory of this directory:

```
$ mkdir -p hello_world_plugin/src && cd hello_world_plugin/src
$ touch hello_world.cc
```

`hello_world.cc` source code will be useful to analyze the structure of a basic Gazebo plugin. In the following the plugin code is reported:

```
#include <gazebo/gazebo.hh>
```

We start with the inclusion of the header file. gazebo.hh contains core functionalities of Gazebo. Other commonly used header file for gazebo are:

- gazebo/physics/physics.hh : This is the Gazebo header for accessing the physics engine parameters
- gazebo/rendering/rendering.hh : This is the Gazebo header for handling rendering parameters
- gazebo/sensors/sensors.hh : This is the header for handling sensors

```
namespace gazebo {
//The custom WorldpluginTutorials is inheriting from standard worldPl
class WorldPluginTutorial : public WorldPlugin {
```

Then. we can declare the plugin namespace. All plugins must be part of `gazebo` namespace. In this example, we are implementing a `WorldPlugin`.

```
public: WorldPluginTutorial() : WorldPlugin() {
printf("Hello World!\n");
}
```

By default Gazebo plugins have a function called `Load` that is automatically invoked when the simulator loads the plugin.

```
//The Load function can receive the SDF elements
public: void Load(physics::WorldPtr _world, sdf::ElementPtr _sdf) {
printf("The plugin has been correctly loaded!\n");
}
};
```

```
//Registering World Plugin with Simulator
GZ_REGISTER_WORLD_PLUGIN(WorldPluginTutorial)
}
```

At the end of the code, we must export the plugin using the following statements. The GZ_REGISTER_WORLD_PLUGIN (WorldPluginTutorial) macro will register and export the plugin as a world plugin. The following macros are used to register for sensors, models, and so on:

- `GZ_REGISTER_MODEL_PLUGIN`: This is the export macro for the Gazebo robot model
- `GZ_REGISTER_SENSOR_PLUGIN`: This is the export macro for the Gazebo sensor model
- `GZ_REGISTER_SYSTEM_PLUGIN`: This is the export macro for the Gazebo system
- `GZ_REGISTER_VISUAL_PLUGIN`: This is the export macro for Gazebo visuals

After setting the code, we can edit the CMakeLists.txt for compiling the source of the plugin. Create the CMakeLists.txt in the root directory of your source:

```
$ cd hello_world_plugin
$ touch CMakeLists.txt
```

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)

include (FindPkgConfig)
if (PKG_CONFIG_FOUND)
pkg_check_modules(GAZEBO gazebo)
endif()

include_directories(${GAZEBO_INCLUDE_DIRS})
link_directories(${GAZEBO_LIBRARY_DIRS})
```

We need to inform the compiler about the location of the gazebo headers (i.e. gazebo.hh). We can do this as usual with the `include_directories` command. We use this command including the `GAZEBO_INCLUDE_DIRS` variable, that is filled thanks to the `find_package` instruction.

```
add_library(hello_world src/plugin/hello_world.cc)
target_link_libraries(hello_world ${catkin_LIBRARIES} ${GAZEBO_LIBRAR
```

We are now ready to compile this plugin. In particular, we have to use the `cmake` and `make` commands:

```
$ cd hello_world_plugin
$ mkdir build
$ cd build
$ cmake .. && make
```

In previous lessons, we included plugin in robot models. In this case, we have developed a `World` plugin and could be included into a Gazebo world. Again, in past example, we just loaded the `empty_world` of gazebo ROS package. Differently, let's define a custom world including this plugin.

```
$ roscd hello_world_plugin
$ mkdir world && cd world
$ touch hello.world
```

The world file is a XML file describing the element present in the simulation scene. In this case, we include the `libhello_world.so` plugin.

```
<?xml version="1.0"?>
<sdf version="1.4">
<world name="default">
<plugin name="hello_world" filename="libhello_world.so"/>
</world>
</sdf>
```

We are very close to run our plugin. However, we have to inform the Gazebo program about the location of this new plugin. This is made updating the value of the `GAZEBO_PLUGIN_PATH` variable.

```
export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:
/path/to/plugin/build
```

Finally, run the gazebo plugin:

```
$ cd world
$ gzserver hello.world --verbose
```

Now you should see the output of the plugin when it is created (*Hello World!*) and when it is loaded (*The plugin has been correctly loaded!*).

10.1.1 Model plugin with ROS integration

World plugins are useful when you want to directly control the simulation scene. A more useful plugin type is the `model` plugin. We can implement this kind of plugin to handle the proprietress of a specific model of Gazebo simulation. Let's try to create a model plugin and attach it to the differential driver robot.

As already stated, the differential drive controller of the robot mobile robot defines two topics: the `cmd_vel` and `odom` topics. Let's imagine that we are interest in the velocity of the wheel, how could get such information even tough the differential drive controller doesn't publish this it? We can write a model plugin and add it to the differential drive mobile robot. Move into the `rl_robot_description_pkg`, create a plugin directory in the source folder:

```
$ roscd rl_robot_description_pkg
$ mkdir -p src/plugin && cd src/plugin
$ touch wheel_vel_plugin.cpp
```

Let's describe the contents of this new plugin. The goal of this plugin is to gather information from the wheel joints from Gazebo world and publish its content into a `std_msgs::Float32MultiArray`.

```
#include <ros/ros.h>
#include "std_msgs/Float32MultiArray.h"
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>

using namespace std;
```

We start including the headers. We need ROS stuff but also other functionalities from Gazebo: the `Joint` class, included into the `gazebo/physics/physics.hh` header. Then, we create the class plugin, part of the gazebo namespace:

```
namespace gazebo
{
class WheelsVelPlugin : public ModelPlugin
{
```

We can then start with class member declaration:

```
private: ros::NodeHandle* _node_handle;
private: physics::ModelPtr model;
private: ros::Publisher _w_v_pub;
private: event::ConnectionPtr updateConnection;
private: physics::JointPtr _front_left_wheel_joint;
private: physics::JointPtr _front_right_wheel_joint;
private: std_msgs::Float32MultiArray _w_vel;
```

In the `Load` function we have to instantiate the node handle and initialize the publisher of our topic. To initialize the joint handlers we will use the `GetJoint` function:

```
public: void Load(physics::ModelPtr _parent, sdf::ElementPtr _sdf) {
_node_handle = new ros::NodeHandle();
model = _parent;

_front_left_wheel_joint = this->model->GetJoint("front_left_wheel_joi
_front_right_wheel_joint = this->model->GetJoint("front_right_wheel_j
_w_v_pub = _node_handle->advertise< std_msgs::Float32MultiArray >("/d
_w_vel.data.resize(2);
}
```

In particular, differently from the `hello_world` plugin, we need a cyclic function that will periodically called directly by the simulator engine. We call this function `OnUpdate` and the we use event based calls of this function with the event::Events::ConnectWorldUpdateBegin class function.

```
this->updateConnection = event::Events::ConnectWorldUpdateBegin(std::
```

Let's see how is composed this function:

```
public: void OnUpdate() {
_w_vel.data[0] = _front_left_wheel_joint->GetVelocity(0);
_w_vel.data[1] = _front_right_wheel_joint->GetVelocity(0);
_w_v_pub.publish( _w_vel );
}
```

Finally, just register the plugin:

```
};

// Register this plugin with the simulator
GZ_REGISTER_MODEL_PLUGIN(WheelsVelPlugin)
}
```

Modify the CMakeLists.txt of the package as show in the previous section and compile the package. We are now ready to add this plugin to the robot model. Edit the `diff_robot.xacro` file adding the following lines:

```
<plugin name="libwheels_vel" filename="libwheels_vel.so">
</plugin>
```

Note that, in this case we could avoid to inform the Gazebo engine about the location of the compiled plugin modifying the `GZ_REGISTER_MODEL_PLUGIN` environment variable. This happened because this plugin is compiled using catkin, and the generated shared library is placed in the devel folder of your workspace. This directory is already present in the gazebo configuration.

You can try to run again the differential drive robot and check the contents of the topic published by the plugin.

```
$ roslaunch rl_robot_description_pkg spawn_diff_robot.launch
$ rostopic echo /diff_wheels/vel
```

11

KDL

KDL stands for Kinematics and Dynamics Library. This library is born with the OROCOS (Open Robots COntrol Software) project and can be used to solve different kinematic and dynamic problems. The Kinematics and Dynamics Library (KDL) develops an application independent framework for modelling and computation of kinematic chains, such as robots, biomechanical human models, computer-animated figures, machine tools, etc. It provides class libraries for geometrical objects (point, frame, line,...), kinematic chains of various families (serial, humanoid, parallel, mobile,...), and their motion specification and interpolation.

In particular, KDL can be used for:

- *Kinematics and Dynamics of kinematic chains*: You can represent a kinematic chain by a KDL Chain object, and use KDL solvers to compute anything from forward position kinematics, to inverse dynamics. The kdl_parser includes support to construct a KDL chain from a XML Robot Description Format (URDF) file.
- *Kinematics of kinematic trees*: You can represent a kinematic chain by a KDL Chain object, and use KDL solvers to compute forward position kinematics.

The goal of this section is to provide an overview of the KDL capabilities with the use of kinematic and dynamic solvers. We will discuss two examples:

- Kinematic: in the first example, we will see how KDL can be used to solve forward and inverse kinematics
- Dynamic: in the second example, we use KDL to calculate dynamic parameters of the robotic manipulator and generate the force to control it.

However, both examples need to know the model of the robot and it's passed to KDL framework initializing a KDL Tree. We will see in the next section how to create it using the URDF file.

kdl parser

kdl parser is a ROS package providing an easy way to construct a full KDL Tree object. In particular, this Tree could be build manually specifying Joints and Links or a using the URDF xml description file.

The Tree is initialized directly from the URDF file that is loaded into the ROS parameter server. As usual we can use the launch file to fill the `robot_description` parameter, then the function `treeFromString` made the rest of the work:

```
std::string robot_desc_string;
_nh.param("robot_description", robot_desc_string, std::string());
if (!kdl_parser::treeFromString(robot_desc_string, iiwa_tree)){
ROS_ERROR("Failed to construct kdl tree");
return false;
}
```

We will use this procedure in the following examples.

Inverse kinematics

In this case, the forward and inverse kinematics of the robot are calculate considering the position of its joint. Using the forward kinematic we get the position of the robotic end effector, while with the use of an inverse kinematic solver we calculate the desired joint values to apply to bring the manipulator in a given location. At this point of the lessons only the salient parts of the source code will be described. However the entire source can be found in the `iiwa_kdl` package. For this example, we will use a position controlled robot. You can start the robot with the following command:

```
roslaunch lbr_iiwa_description gazebo_ctrl.launch
```

We need to include the header files to use KDL functions:

```
#include <kdl_parser/kdl_parser.hpp>
#include <kdl/chainfksolverpos_recursive.hpp>
#include <kdl/chainiksolvervel_pinv.hpp>
#include <kdl/chainfksolverpos_recursive.hpp>
#include <kdl/chainiksolverpos_nr.hpp>
```

We must load the robot model, but also initialize its *kinematic chain*:

```
std::string robot_desc_string;
_nh.param("robot_description", robot_desc_string, std::string());
if (!kdl_parser::treeFromString(robot_desc_string, iiwa_tree)){
ROS_ERROR("Failed to construct kdl tree");
return false;
}
```

```

std::string base_link = "lbr_iiwa_link_0";
std::string tip_link  = "lbr_iiwa_link_7";
if ( !iiwa_tree.getChain(base_link, tip_link, _k_chain) ) return false;
```

The chain is specified between two desired links. In this case, we create a chain starting from the first link and finishing in the end effector link.

In order to retrieve the position of the end effector we use the ChainFkSolverPos_recursive object, initialized on the kinematic chain.

```
_fksolver = new KDL::ChainFkSolverPos_recursive( _k_chain );
```

At the same time, the ChainIkSolverPos_NR object is used to calculate the inverse kinematic. Among the arguments required to initialize such object.

```
_ik_solver_vel = new KDL::ChainIkSolverVel_pinv( _k_chain );
_ik_solver_pos = new KDL::ChainIkSolverPos_NR( _k_chain, *_fksolver, *_ik_so
```

Later than you have the joint values you can use the _fksolver object to translate the joint position into the end effector position:

```
_fksolver->JntToCart(*_q_in, _p_out);
```

The position of the end effector is stored into a KDL::Frame object. A Frame is the 4x4 matrix that represents the pose of an object/frame with respect to a reference frame. It contains:

- a Rotation M for the rotation of the object/frame wrt the reference frame.
- a Vector p for the position of the origin of the object/frame in the reference frame

KDL also provides a convenient template to store joint variables. For example, the joint angles can be store in a `JntArray` variable:

```
KDL::JntArray *_q_in;
_q_in = new KDL::JntArray( _k_chain.getNrOfJoints() );
for(int i=0; i<7; i++ )
_q_in->data[i] = js.position[i];
```

We are now ready to control the position of the end effector using the inverse kinematic solver. To this aim, we need to define its target filling a KDL::Frame object:

```
F_dest.p.data[0] = _p_out.p.x() - 0.2;
F_dest.p.data[1] = _p_out.p.y();
F_dest.p.data[2] = _p_out.p.z() - 0.1;
for(int i=0; i<9; i++ )
F_dest.M.data[i] = _p_out.M.data[i];
if( _ik_solver_pos->CartToJnt(*_q_in, F_dest, q_out) != KDL::SolverI::E_NOERR
cout << "failing in ik!" << endl;
```

After run the simulator launch the inverse kinematic example can be ran in this following way:

```
rosrun iiwa_kdl kuka_invkin_ctrl
```

Inverse Dynamics

In some cases, we need to exploit the dynamic of the system to improve the performance of our controllers. In particular, inverse dynamics is a method for computing forces and/or torques based on the kinematics (motion) of a body and the body's inertial properties (mass and moment of inertia). In robotics,inverse dynamics algorithms are used to calculate the torques that a robot's motors must deliver to make the robot's end-point move in the way prescribed by its current task.

In order to test develop an inverse dynamics control algorithm, we need a robot that can be controlled using effort: the hardware_interface EffortJointInterface must be used for the iiwa simulation. To start this new robot model use the following command:

```
$ roslaunch lbr_iiwa_description gazebo_effort_controller.launch
```

Differently from the position controlled robot, if we don't apply a force command to motor joints, the robot falls down to the floor. Let's discuss the code of the `kuka_invdyn_ctrl.cpp` source. In this case, we will not rely on a solver to implement the inverse dynamics controller, but directly calculate the body's inertial properties using KDL template and than implement a PD controller to generate the torque command.

In particular, we firstly calculate the joint position error (the desired position for robot joints against the current one):

```
Eigen::VectorXd e = _initial_q->data - _q_in->data; //Keep initial po
```

While, the velocity error consists in the inverse of the current velocity of the robot (we want that the robot doesn't move)

```
Eigen::VectorXd de = -_dq_in->data; //Desired velocity: 0
```

Then, we calculate the intertia matrix, the coriolis terms and the gravity temrs:

```
_dyn_param->JntToMass(*_q_in, jsim_);
_dyn_param->JntToCoriolis(*_q_in, *_dq_in, coriol_);
_dyn_param->JntToGravity(*_q_in, grav_);
```

To access to dynamic parameters we use the `_dyn_param` object.

```
KDL::ChainDynParam *_dyn_param;
_dyn_param = new KDL::ChainDynParam(_k_chain,KDL::Vector(0,0,-9.81));
```

Finally, we can calculate the effort command:

```
Eigen::VectorXd q_out = jsim_.data * (Kd*de + Kp*e ) + coriol_.data + grav_.d
```

To run this example, after launched the gazebo simulator use the following command

```
rosrun iiwa_kdl kuka_invdyn_ctrl
```

11.0.1 Exercise

In order to merge the knowledge about the ROS Gazebo plugins and KDL, develop a Gazebo plugin to calculate and apply the CLIK algorithm on the IIWA robot simulated in ROS. In particular, the developed plugin must take as input the desired position of the manipulator, while should apply the q values generated with KDL library.

12

EIGEN libraries

In previous section, we used an external library to simplify mathematics and linear algebra operations: EIGEN.

Eigen is a high-level C++ library of template headers for linear algebra, matrix and vector operations, geometrical transformations, numerical solvers and related algorithms.

To install Eigen you just need to download and extract Eigen's source code. In fact, the header files in the Eigen subdirectory are the only files required to compile programs using Eigen. The header files are the same for all platforms. It is not necessary to use CMake or install anything. You can also install Eigen using apt

```
$ sudo apt-get install libeigen3-dev
```

While, to use in a cpp program, you need to include the following lines into the `CMakeLists.txt` file.

```
find_package(Eigen3 REQUIRED)
include_directories(${Eigen_INCLUDE_DIRS})
```

12.0.1 Data types

Using Eigen a convenient way to define matrix and vectors are provided. You can specify both static and dynamic variables. In particular, if you want to set at run time the dimension of a matrix or a vector use the following code.

```
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
```

```
{
MatrixXd m = MatrixXd::Random(3,3);
m = (m + MatrixXd::Constant(3,3,1.2)) * 50;
cout << "m =" << endl << m << endl;
VectorXd v(3);
v << 1, 2, 3;
cout << "m * v =" << endl << m * v << endl;
}
```

While, if you want to set such sizes statically:

```
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
{
Matrix3d m = Matrix3d::Random();
m = (m + Matrix3d::Constant(1.2)) * 50;
cout << "m =" << endl << m << endl;
Vector3d v(1,2,3);

cout << "m * v =" << endl << m * v << endl;
}
```

Using such structures you can access to different functionalities to solve common problem in robotics. Check the *Quick reference guide* to have an overview about the capabilities: `https://eigen.tuxfamily.org/dox/group__QuickRefPage.html`. To have a briefly idea of Eigen functionalities, imagine that you want to calculate the pseudo inverse of a matrix. This task is not trivial using standard libraries or without matlab. The Eigen version is here reported:

```
Eigen::Matrix<float, 6, 4> M_matrix;
M_matrix << 0, 0, 0, 0,
0, 0, 0, 0,
kf, kf, kf, kf,
(0.707*L*kf), (0.707*L*kf), -(0.707*L*kf), -(0.707*L*kf),
-(0.707*L*kf), (0.707*L*kf), (0.707*L*kf), -(0.707*L*kf),
km, -km, km, -km;


Eigen::Matrix<float, 4, 6> M_matrix_pinv = M_matrix.completeOrthogona
```

13

ROS Navigation stack

Navigation stack contains a set of powerful tools and libraries to work mainly for mobile robot navigation. The Navigation stack contains ready-to-use navigation algorithms which can be used in mobile robots, especially for differential wheeled robots. Using these stacks, we can make the robot autonomous, and that is the final concept that we are going to see in the Navigation stack.

The main aim of the ROS Navigation package is to move a robot from the start position to the goal position, without making any collision with the environment. The ROS Navigation package comes with an implementation of several navigation-related algorithms which can easily help implement autonomous navigation in the mobile robots. The user only needs to feed the goal position of the robot and the robot odometry data from sensors such as wheel encoders, IMU, and GPS, along with other sensor data streams, such as laser scanner data or 3D point cloud from sensors such as depth sensor. The output of the Navigation package will be the velocity commands that will drive the robot to the given goal position.

The Navigation stack contains the implementation of the standard algorithms, such as SLAM, A *(star), Dijkstra, amcl, and so on, which can directly be used in our application.

The ROS Navigation stack is designed as generic. There are some hardware requirements that should be satisfied by the robot. The following are the requirements:

- The Navigation package will work better in differential drive and holonomic (total DOF of robot equals to controllable DOF of robots). Also, the mobile robotshould be controlled by sending velocity commands in the form of linear and angular velocity.
- The robot should be equipped with a vision (rgb-d) or laser sensor to build the map of the environment.
- The Navigation stack will perform better for square and circular shaped

mobile bases. It will work on an arbitrary shape, but performance is not guaranteed.

The modules of the Navigation stack are depicted in Fig. 13.1 and are discussed in the following.

`IMG/Ch11/2nav_module.png`

Figure 13.1: Robot navigation modules using ROS

According to the Navigation stack diagram reported in Fig. 13.1, for configuring the Navigation package for a custom robot, we must provide functional blocks that interface to the Navigation stack. The following are the explanations of all the blocks which are provided as input to the Navigational stack:

- Odometry source: Odometry data of a robot gives the robot position with respect to its starting position. The main odometry sources are wheel encoders, IMU, and 2D/3D cameras (visual odometry). The odom value should publish to the Navigation stack, which has a message type of `nav_msgs/Odometry`. The odom message can hold the position and the velocity of the robot. Odometry data is a mandatory input to the Navigational stack.

- Sensor source: We have to provide laser scan data or point cloud data to the Navigation stack for mapping the robot environment. This data, along with odometry, combines to build the global and local cost map of the robot. The main sensors used here are Laser Range finders or Kinect 3D sensors. The data should be of type `sensor_msgs/LaserScan` or `sensor_msgs/PointCloud`.

- sensor transforms/tf: The robot should publish the relationship between the robot coordinate frame using ROS tf.

- `base_controller`: The main function of the base controller is to convert the output of the Navigation stack, which is a twist (geometry_msgs/Twist) message, and convert it into corresponding motor velocities of the robot.

The optional nodes of the Navigation stack are **amcl** and **map server**, which allow localization of the robot and help to save/load the robot map.

13.1 Move base

The move_base node is from a package called move_base. The main function of this package is to move a robot from its current position to a goal position with the help of other navigation nodes. The move_base node inside this package links the global-planner and the local-planner for the path planning, connecting to the rotate-recovery package if the robot is stuck in some obstacle and connecting global costmap and local costmap for getting the map. The move_base node is basically an implementation of SimpleActionServer, which takes a goal pose with message type (geometry_msgs/PoseStamped). We can send a goal position to this node using a SimpleActionClient node. The move_base node subscribes the goal from a topic called move_base_simple/goal, which is the input of the Navigation stack, as shown in the previous diagram. When this node receives a goal pose, it links to components such as global_planner, local_planner, recovery_behavior, global_costmap, and local_costmap, generates the output, which is the command velocity (geometry_msgs/Twist), and sends it to the base controller for moving the robot for achieving the goal pose.

The following is the list of all the packages which are linked by the move_base node:

- global-planner: This package provides libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map. This package has the implementation of path-finding algorithms, such as A*, Dijkstra, and so on, for finding the shortest path from the current robot position to the goal position.
- local-planner: The main function of this package is to navigate the robot in a section of the global path planned using the global planner. The local planner will take the odometry and sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan. The base local planner package is the implementation of the trajectory rollout and dynamic window algorithms.
- costmap-2D: The main use of this package is to map the robot environment. The robot can only plan a path with respect to a map. In ROS, we create 2D or 3D occupancy grid maps, which is a representation of the environment in a grid of cells. Each cell has a probability value that indicates whether the cell is occupied or not. The costmap-2D package can build the grid map of the environment by subscribing sensor values of the laser scan or point cloud and also the odometry values. There are global cost maps for global navigation and local cost maps for local navigation.

The following are the other packages which are interfaced to the move_base node:

- map-server: The map-server package allows us to save and load the map generated by the costmap-2D package.

- AMCL: AMCL (Adaptive Monte Carlo Localization) is a method to localize the robot in a map. This approach uses a particle filter to track the pose of the robot with respect to the map, with the help of probability theory. In the ROS system, AMCL accepts a sensor_msgs/LaserScan to create the map.
- gmapping: The gmapping package is an implementation of an algorithm called Fast SLAM, which takes the laser scan data and odometry to build a 2D occupancy grid map.

After discussing each functional block of the Navigation stack, let's see how it really works. To do this, the robot should publish a proper odometry value, TF information, and sensor data from the laser, and have a base controller and map of the surroundings.

13.1.1 Localization and mapping

To navigate unknown environments a robot must be able to build a map. This process is called mapping. In addition, during the navigation the robot should also be able to localize during the map it is creating: this process is called localization. Since these two steps are made in the same time it is called **SLAM: simultaneous localization and mapping**. One of the most famous tools to do 2D SLAM is called *gmapping*. The ROS Gmapping package is a wrapper of the open source implementation of SLAM, called OpenSLAM (`https://www.openslam.org/gmapping.html`). The package contains a node called slam_gmapping, which is the implementation of SLAM and helps to create a 2D occupancy grid map from the laser scan data and the mobile robot pose. The basic hardware requirement for doing SLAM is a laser scanner which is horizontally mounted on the top of the robot, and the robot odometry data.

Let's try to use gmapping on the differential drive robot. This robot is quite ready to been used with gmapping, since it is already endowed with a laser scanner, and the odometry data are provided by the differential drive robot plugin. Start installing gmapping ROS wrapper:

```
$ sudo apt-get install ros-melodic-gmapping
```

As usual, we need to create a proper launch file to start 2d SLAM.

```
<?xml version="1.0" ?>
<launch>
<arg name="scan_topic" default="/laser/scan" />
<!-- Defining parameters for slam_gmapping node -->
<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output=
```

The node to start is the `slam_gmapping` node from the `gmapping` package. In order to be informed about the output of the node we include the `output="screen"` option. This launch file also contains an argument: the `scan_topic` who reports the name of the topic in which the scan data are published.

```
<param name="base_frame" value="base_link"/>
<param name="odom_frame" value="odom"/>
```

Two additional parameters are used to specify the reference frames of the robotic based and the odometry, respectively. Other parameters are used to tune the behavior of the SLAM algorithm.

```
<param name="map_update_interval" value="5.0"/>
<param name="maxUrange" value="6.0"/>
<param name="maxRange" value="8.0"/>
<param name="sigma" value="0.05"/>
<param name="kernelSize" value="1"/>
<param name="lstep" value="0.05"/>
<param name="astep" value="0.05"/>
<param name="iterations" value="5"/>
<param name="lsigma" value="0.075"/>
<param name="ogain" value="3.0"/>
<param name="lskip" value="0"/>
<param name="minimumScore" value="100"/>
<param name="srr" value="0.01"/>
<param name="srt" value="0.02"/>
<param name="str" value="0.01"/>
<param name="stt" value="0.02"/>
<param name="linearUpdate" value="0.5"/>
<param name="angularUpdate" value="0.436"/>
<param name="temporalUpdate" value="-1.0"/>
<param name="resampleThreshold" value="0.5"/>
<param name="particles" value="80"/>
<param name="xmin" value="-1.0"/>
<param name="ymin" value="-1.0"/>
<param name="xmax" value="1.0"/>
<param name="ymax" value="1.0"/>
<param name="delta" value="0.05"/>
<param name="llsamplerange" value="0.01"/>
<param name="llsamplestep" value="0.01"/>
<param name="lasamplerange" value="0.005"/>
<param name="lasamplestep" value="0.005"/>
<remap from="scan" to="$(arg scan_topic)"/>
</node>
</launch>
```

Gmapping configured as is can be ran on the differential mobile robot using the following commands:

```
$ roslaunch rl_robot_description_pkg spawn_diff_robot.launch
$ roslaunch rl_robot_description_pkg gmapping.launch
```

If everything is properly configured, the robot starts to create the environment map. In our case, the simulation scene is empty so no map can be created. Let's try to create a custom world in gazebo with some walls. Move into the `rl_robot_description_pkg` package and create a worlds directory with a world source file and a models directory.

```
$ roscd rl_robot_description_pkg
$ mkdir models
$ mkdir worlds && cd worlds
$ touch walls.world
```

In the models folder we will put the elements to include into the simulation scene.

models

Models in Gazebo define a physical entity with dynamic, kinematic, and visual properties. In addition, a model may have one or more plugins, which affect the model's behavior. A model can represent anything from a simple shape to a complex robot. Even the ground is a model. Gazebo relies on a database to store and maintain models available for use within simulation. The model database is a community-supported resource, so please upload and maintain models that you create and use. However, you can define custom models. Here we will describe the Gazebo model structure.

Each model is included into a directory and must have a model.config file in its root directory that contains meta information about the model. The format of this model.config is:

```
<?xml version="1.0"?>

<model>
<name>My Model Name</name>
<version>1.0</version>
<sdf version='1.5'>model.sdf</sdf>

<author>
<name>My name</name>
<email>name@email.address</email>
</author>

```

```
<description>
A description of the model
</description>
</model>
```

- <name> Name of the model.
- <version> Version of this model. This is not the version of sdf that the model uses. That information is kept in the model.sdf file.
- <sdf> The name of a SDF or URDF file that describes this model. The version attribute indicates what SDF version the file uses, and is not required for URDFs. Multiple <sdf> elements may be used in order to support multiple SDF versions.
- <author> Name and contacts of the model author.
- <description> Description of the model should include.

By default, Gazebo searches for models into the `~/.gazebo/models/` directory. However, you can add additional custom folder modifying the `GAZEBO_MODEL_PATH` environmental variable. This time, we can do this directly from the launch file, with the following line:

```
<env name="GAZEBO_MODEL_PATH" value="$(find rl_robot_description_pkg)/models:
```

The model provided with this package is the `grey_wall` model.

world

As for the world specification file, it is a `sdf` file format. The main elements of this file are the included models from the gazebo models stack.

```
<?xml version="1.0" ?>
<sdf version="1.5">
<world name="default">
<scene>
<ambient>0.0 0.0 0.0 1.0</ambient>
<shadows>0</shadows>
</scene>
<include>
<uri>model://ground_plane</uri>
</include>
<include>
<uri>model://sun</uri>
</include>

```

```
<include>
<pose> 3.8 −3.04 0.02 0 0 0.0</pose>
<uri>model://grey_wall</uri>
</include>
</world>
</sdf>
```

In this case, we include the grey walls models. To load the world file we must use the `world_name` argument of gazebo ros:

```
<arg name="world_name" value="$(find rl_robot_description_pkg)/worlds
```

The composed world is shown in Fig. 13.2. We are now ready to test

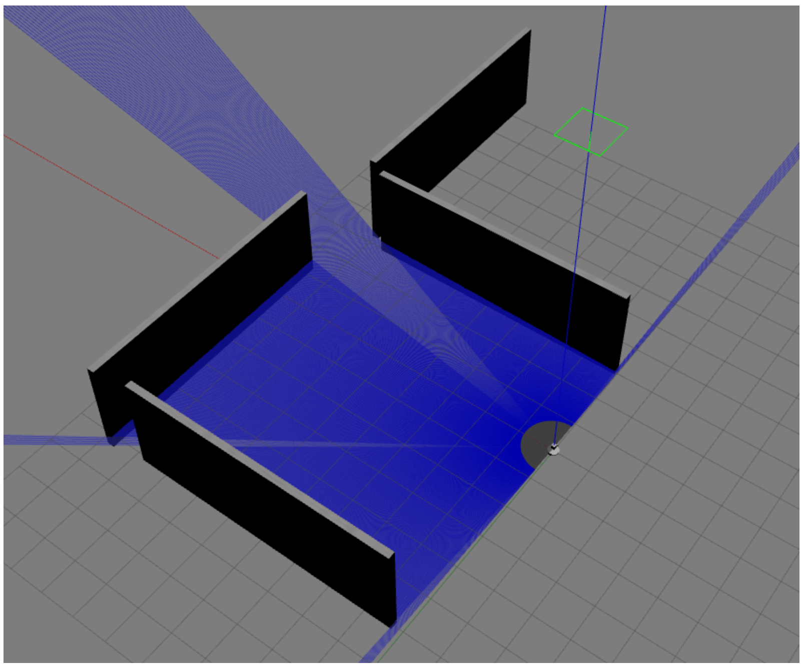


Figure 13.2: Gazebo scene used for SLAM

gmapping on this custom world. Launch the custom world file with the differential drive robot. Then, launch the gmapping.launch launch file. Then use the teleoperation node to move the robot into the environment.

We can launch RViz and add a display type called Map and the topic name as /map.The following image shows the completed map of the environment shown in RViz is reported in Fig. 13.3.

We can save the built map using the following command. This command will listen to the map topic and save into the image. The map server package does this operation.

```
$ rosrun map_server map_saver -f walls
```

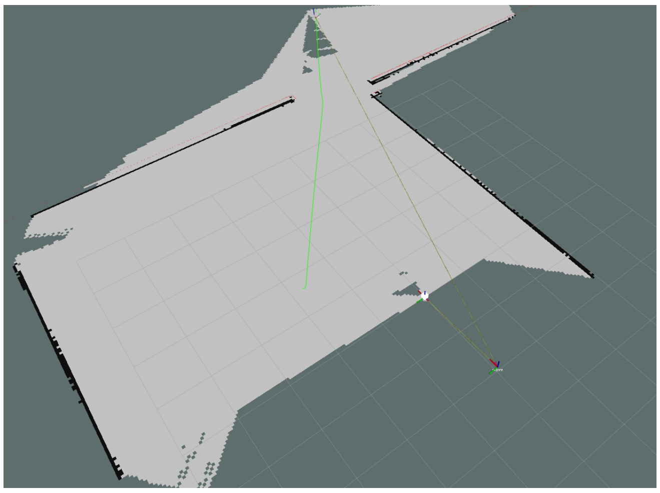


Figure 13.3: Map topic displayed in RViz using the map frame

The map file is stored as two files: one is the YAML file, which contains the map metadata and the image name, and second is the image, which has the encoded data of the occupancy grid map. In order to find and correctly load the map in the ROS system, move these files into the `rl_robot_description_pkg` folder, into a maps directory.

13.1.2 Map representation

There are mainly two ways to represent maps in ROS. The representation type strictly depends on the dimension of the world space. In particular, if you want to represent a 2d world, you can use an Occupancy Grid Map (nav_msgs/OccupancyGrid). If your sensor and your motion planning algorithms work in a 3d world, is better to use Octomap (we will discuss about octomap in further lessons).

grid map An occupancy grid map maps the environment as a grid of cells. The cell size typically ranges from 5 to 50 centimeters and each cell holds a probability value that the cell is occupied in the range between 0 and 100 (0 means that the location is totally free). Other cells can be unexplored and their value is -1.

As for their visualization, considering the Fig. 13.3, the clear gray cells represent free regions of space while the black cells encode obstacles.

13.2 Robot localization

In several cases, robots are programmed to work everyday in the same environment (i.e. a warehouse, a museum and so on). For this reason, could be useful to create a map of the static object of the operative environment of the robot, in order to simplify the role of the robot during its operations. Of course, if the robot is endowed with a map of the environment, the SLAM problem consists only in the Localization part. One popular way to solve localization problem is to use `amcl` algorithm (augmented monte carlo localization) to localize a robot into a given map.

13.2.1 AMCL

The ROS amcl package provides nodes for localizing the robot on a static map. The amcl node subscribes the laser scan data, laser scan based maps, and the TF information from the robot. The amcl node estimates the pose of the robot on the map and publishes its estimated position with respect to the map.

If we create a static map from the laser scan data, the robot can autonomously navigate from any pose of the map using amcl and the move_base nodes. The first step is to create a launch file for starting the amcl node. The amcl node is highly customizable and we can configure it with several parameters.

amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates. On startup, amcl initializes its particle filter according to the parameters provided. Note that, because of the defaults, if no parameters are set, the initial filter state will be a moderately sized particle cloud centered about (0,0,0). The in and out topics are reported in the following:

- Subscribed Topics
  - scan (sensor_msgs/LaserScan): Laser scan data
  - tf (tf/tfMessage): Transforms.
  - initialpose (geometry_msgs/PoseWithCovarianceStamped): Mean and covariance with which to (re-)initialize the particle filter.
  - map (nav_msgs/OccupancyGrid): When the use_map_topic parameter is set, AMCL subscribes to this topic to retrieve the map used for laser-based localization.
- Published Topics
  - amcl_pose (geometry_msgs/PoseWithCovarianceStamped): Robot's estimated pose in the map, with covariance.
  - particlecloud (geometry_msgs/PoseArray): The set of pose estimates being maintained by the filter.
  - tf (tf/tfMessage): Publishes the transform from odom (which can be remapped via the odom_frame_id parameter) to map.

In the previous section we have saved a map of the environment. Let's now try to start the localization algorithm.

Start installing the amcl package in your system.

```
$ sudo apt-get install ros-melodic-amcl
```

Now, we can create the proper launch file for amcl:

```
<launch>
<arg name="map_file" default="$(find rl_robot_description_pkg)/maps/walls.yan
<node name="map_server" pkg="map_server" type="map_server" args="$(arg map_f
```

First of all we need to load the map. To this aim, we use the map_server. We already used this package to save the map, with the map_saver node. In this case, we want to use this node to load the saved map and publish it on a topic called /map.

```
<arg name="use_map_topic"   default="false"/>
<arg name="scan_topic"      default="/laser/scan"/>
<arg name="initial_pose_x"  default="0.0"/>
<arg name="initial_pose_y"  default="0.0"/>
<arg name="initial_pose_a"  default="0.0"/>

<node pkg="amcl" type="amcl" name="amcl">
```

Then we can launch the amcl node.

```
<param name="use_map_topic"              value="true"/>
<param name="odom_model_type"            value="diff"/>
<param name="odom_alpha5"                value="0.1"/>
<param name="gui_publish_rate"           value="10.0"/>
<param name="laser_max_beams"              value="60"/>
<param name="laser_max_range"            value="12.0"/>
<param name="min_particles"              value="500"/>
<param name="max_particles"              value="2000"/>
<param name="kld_err"                    value="0.05"/>
<param name="kld_z"                      value="0.99"/>
<param name="odom_alpha1"                value="0.2"/>
<param name="odom_alpha2"                value="0.2"/>
<param name="odom_alpha3"                value="0.2"/>
<param name="odom_alpha4"                value="0.2"/>
<param name="laser_z_hit"                value="0.5"/>
<param name="laser_z_short"              value="0.05"/>
<param name="laser_z_max"                value="0.05"/>
<param name="laser_z_rand"               value="0.5"/>
<param name="laser_sigma_hit"            value="0.2"/>
<param name="laser_lambda_short"         value="0.1"/>
```

```
<param name="laser_model_type"              value="likelihood_field"/>
<param name="laser_likelihood_max_dist"     value="2.0"/>
<param name="update_min_d"                  value="0.25"/>
<param name="update_min_a"                  value="0.2"/>
<param name="odom_frame_id"                 value="odom"/>
<param name="base_frame_id"                 value="base_link"/>
<param name="resample_interval"             value="1"/>
<param name="transform_tolerance"           value="1.0"/>
<param name="recovery_alpha_slow"           value="0.0"/>
<param name="recovery_alpha_fast"           value="0.0"/>
<param name="initial_pose_x"                value="$(arg initial_pose_x)"/
<param name="initial_pose_y"                value="$(arg initial_pose_y)"/
<param name="initial_pose_a"                value="$(arg initial_pose_a)"/
<remap from="scan"                          to="$(arg scan_topic)"/>
</node>
</launch>
```

You can launch the gazebo simulation and the amcl files and see the result on RViz. Now, the robot is able to create a map and localize inside it. The next step is to test the autonomous navigation using the move_base package.

13.3 navigation using move_base

After getting the current position of the robot, we can send a goal position to the move_base node. The move_base node will send this goal position to a global planner, which will plan a path from the current robot position to the goal position. This plan is with respect to the global costmap, which is feeding from the map server. The global planner will send this path to the local planner, which executes each segment of the global plan. The local planner gets the odometry and the sensor value from the move_base node and finds a collision-free local plan for the robot. The local planner is associated with the local costmap, which can monitor the obstacle(s) around the robot.

Since move_base implements some action server to allow the navigation of the robot, we can write an action client to perform the generation of the path. However, in our first example we can use RViz user interface to perform the robot navigation.

Start writing the launch file for the move_base package. Let's firstly introduce some configuration files involved into the planning process. In particular, we have to configure the 2d costmap, the local and global planners, and the move_base controller. The configuration of these module relies on yaml files and are included into the conf folder of the rl_robot_description_pkg package. In particular, we have the following file:

- `costmap_common_params.yaml`: This file specifies the sensor topics they should listen to for updating the costmap. In addition, we have to include a thresholds on obstacle information like the obstacle_range parameter who determines the maximum range sensor reading that will result in an obstacle being put into the costmap. Finally, we can include information about the sensor information with this line:

  ```
  laser_scan_sensor:
  {sensor_frame: frame_name, data_type: LaserScan,
  topic: topic_name, marking: true, clearing: true}
  ```

- `global_costmap_params.yaml`: In the global costmap parameter file the frame reference for the map (usually we set `/map`) and additionally parameters.

- `local_costmap_params.yaml`: In this file we have the same parameter for the local costmap planner, but considered for the local map.

- `dwa_local_planner_params.yaml`: This file contains the configuration of the DWAPlannerROS base local planner, like the maximum linear and angular velocity, trajectory tolerances and so on.

- `move_base_params.yaml`: This contains configuration of the move_base package.

Let's now write the launch file for move_base: the `move_base.launch`.

```
<launch>
<arg name="odom_topic" default="odom" />

<node pkg="move_base" type="move_base" respawn="false" name="move_base" outpu
<rosparam file="$(find rl_robot_description_pkg)/param/costmap_common_params.
<rosparam file="$(find rl_robot_description_pkg)/param/local_costmap_params.y
<rosparam file="$(find rl_robot_description_pkg)/param/global_costmap_params.
<rosparam file="$(find rl_robot_description_pkg)/param/dwa_local_planner_para
<rosparam file="$(find rl_robot_description_pkg)/param/move_base_params.yaml"

</node>
</launch>
```

In this file we mainly load the configuration files described above. If you don't have move_base installed in your system, you should install it with the following command:

```
$ sudo apt-get install ros-melodic-move-base
ros-melodic-move-base-msgs
```

The move_base package can be used both with the SLAM and AMCL packages. Let's try to use it with the SLAM module. We can start the whole system using the prepared launch files.

```
$ roslaunch rl_robot_description_pkg walls_world.launch
$ roslaunch rl_robot_description_pkg gmapping.launch
$ roslaunch rl_robot_description_pkg move_base.launch
```

At this point, you could be able to send a goal point to the move_base server using RViz. Open Rviz:

```
$ rviz
```

Then, is possible to use the `2D Nav Goal` button of RViz bar to send the goal on the map, as shown in Fig. 13.4. You can also add the visualization

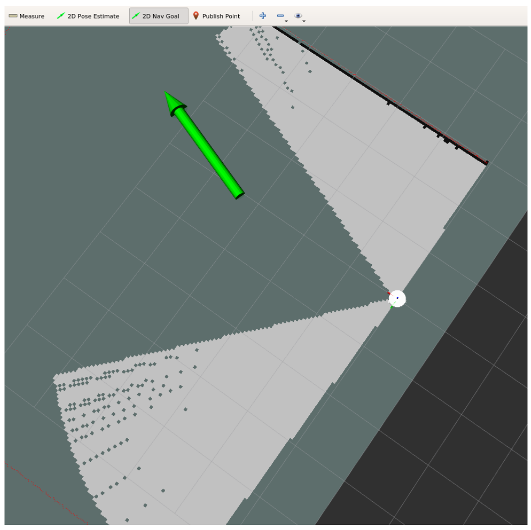


Figure 13.4: RViz navigation plugin.

of the path generated by the local and global planners.

cpp move base client

Of course, is much more useful to use the move_base action from a client code. The following code shows an example of move_base action client.

```
#include "ros/ros.h"
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>


int main( int argc, char** argv) {
```

```

ros::init(argc, argv, "move_base_client");
ros::NodeHandle nh;
actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac("move_base",
ac.waitForServer(); //will wait for infinite time

move_base_msgs::MoveBaseGoal goal;
goal.target_pose.header.frame_id = "map";
goal.target_pose.pose.position.x = 5.0;
goal.target_pose.pose.orientation.w = 1.0;

ac.sendGoal(goal);
bool done = false;
ros::Rate r(10);
while ( !done ) {
if ( ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED ||
ac.getState() == actionlib::SimpleClientGoalState::PREEMPTED ) {
done = true;
}
r.sleep();
}
}
```