Mobile Robotics

Theory and Practice

Stefano Carpin

University of California, Merced

Version 1.0 – August 22, 2025

Contents

Foreword

Do we need another textbook on mobile robotics? Given that you are reading this foreword, the question may seem rhetorical. At least from the author's perspective, the answer is clearly "yes." A more appropriate question might be: *Why do we need another textbook on mobile robotics?* In the following, I will explain why I undertook this project and the gap it aims to fill.

There are many excellent books that present the foundations of mobile robotics, covering mathematical models, algorithms, and technologies. Several of these are cited in this book and provide a solid grounding for those entering this fascinating field. However, one cannot truly claim to master the subject without putting theory into practice, and this is where many of these books fall short.

Over the past several years, ROS has become the de facto standard for developing robot software. Numerous high-quality books and websites explain how to develop robot control systems using ROS. Unfortunately, most of these resources assume that the reader already possesses a solid understanding of robotics theory and technologies. These books are therefore not adequate for the beginner.

This book aims to bridge the gap between these two categories. In addition to providing a concise introduction to key theoretical foundations (kinematics, planning, estimation, etc.), each topic is paired with its practical implementation in ROS. Because neither the student nor the aspiring practitioner may necessarily have easy access to a mobile platform, examples are coupled with suitable Gazebo simulations, thus lowering the entry barrier.

Like any textbook, the selection of topics reflects personal choices shaped by the author's perspective on the field and informed by years of teaching at both the undergraduate and graduate levels. Naturally, due to space constraints and the goal of keeping the content manageable within a single semester, some important topics had to be left out. It should also be reiterated that this is an introductory book, so it purposefully focuses on the foundations and does not aim to cover more advanced topics. However, the expectation is that, starting with the basics provided in this book, readers can progress to more advanced material that did not make it into this volume.

It is also worth noting that this book was developed primarily for undergraduate students majoring in Computer Science and Engineering. As such, it does not assume prior knowledge of subjects typically covered in courses like "Signals and Systems" or "Feedback Control," and the selection of topics covered (or omitted) reflects this assumption. In terms of mathematical background, the standard material offered in a lower-division engineering curriculum should be sufficient to follow the contents of this book. A dedicated appendix provides a brief recap of probability theory, as this is instrumental for the development of

estimation algorithms.

An important question is which programming language should be used for teaching how to program robots, given that ROS supports both C++ and Python. This book adopts C++, primarily because it better serves the students for whom this book was written, but most concepts transcend the specific language. In fact, a future project may entail developing a parallel version using Python.

In an age where generative AI can be used to quickly produce instructional materials (often inaccurate or plainly wrong...) it is worth noting that no content in this book was generated by AI, although AI tools were used for spell-checking.

I am grateful to the graduate students at the University of California, Merced who served as teaching assistants for my robotics class and provided valuable corrections and suggestions (in no particular order): Jose Luis Susa Rincon, Carlos Diaz Alvarenga, Lorenzo Booth, Marcos Zuzuárregui, and Andre Torres Garcia.

Finally, like many other academic writings, a significant portion of this book was written in the evenings, on weekends, during holidays, and at other times that would have been better spent with family. I am indebted to them for the many hours I spent secluded in my home office.

There are certainly numerous typos, mistakes, and other issues commonly found in first editions. Comments, critiques, and suggestions (including any errors you find!) should be sent to `scarpin@ucmerced.edu`. I thank everyone in advance for providing feedback.

All code presented in this book has been developed and tested on a computer running Ubuntu 24.04.1 (Noble) and ROS Jazzy. Code, installation instructions, and additional materials can be downloaded from `https://github.com/stefanocarpin/MRTP` (referred to as the *MRTP GitHub* in the following).

Notation

X	random variable (upper case last letters of the alphabet)
$\bar{X}$	random vector (upper case last letters of the alphabet with bar)
$\boldsymbol{X}$	stochastic process (upper case bold italic letter)
A	set (upper case first letters of the alphabet)
A	event (upper case first letters of the alphabet)
p_X	probability mass function of discrete random variable X
f_X	probability density function of continuous random variable X
$p_{\bar{X}}$	probability mass function of discrete random vector $\bar{X}$
$f_{\bar{X}}$	probability density function of continuous random vector $\bar{X}$
μ_X	expectation of random variable X
σ_X	standard deviation of random variable X
σ_X^2	variance of random variable X
σ_{XY}	covariance of random variables X and Y
r_{XY}	correlation of random variables X and Y
ρ_{XY}	correlation coefficient between random variables X and Y
$\boldsymbol{\mu}_{\bar{X}}$	expectation of random vector $\bar{X}$
$\boldsymbol{\Sigma}_{\bar{X}}$	covariance matrix of random vector $\bar{X}$
$\mathbf{x}$	vector (boldface lowercase letter)
$\mathbf{A}$	matrix (non italic boldface uppercase letter)
$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
${}^{A}\mathbf{p}$	coordinates of point $\mathbf{p}$ expressed in frame A
${}^{A}_{B}\mathbf{R}$	rotation matrix describing the orientation of frame B relative to frame A
${}^{A}_{B}\mathbf{T}$	transformation matrix describing frame B relative to frame A

Chapter 1

Introduction

1.1 Mobile Robots

There exist many definitions for the term *robot* and numerous taxonomies to classify them. Most definitions concur in defining a robot as a system equipped with sensors and actuators and capable of being reprogrammed to perform different tasks. This definition, albeit vague, describes the systems considered in these notes. Robots are also often classified based on characteristics such as the kind of tasks they used for (industrial, service, field, space, medical, military, etc.), their mobility (legged, tracked, wheeled, aerial, etc.), and so on. In these notes we focus on *mobile robots*. There is no universally accepted definition of *mobile robot*, and in a sense if one intends the term *mobile* as the ability to move in space, one could speculate that all robots by definition perform some sort of motion in space and are then mobile. We informally define a mobile robot as a system capable of moving in the environment without being restricted to a preassigned location. This definition excludes industrial manipulators that are anchored to a fixed work station because said systems cannot move from there, i.e., they are restricted to a preassigned location. However, if one considers a mobile manipulator, i.e., a manipulator mounted on a mobile base, then this would be a mobile robot according to our definition. In general the term *mobile robot* is used to indicate robots whose locomotion is based on wheels, but the above definition would be equally appropriate for a legged robot (e.g., a humanoid). Hence, the term has to be interpreted with some flexibility and with the understanding that the boundary defining this category may be fuzzy at times. On the other hand, most of the material presented in the following is rather general and can be applied to many different types of robots, so this ambiguity is not detrimental to the remainder of the discussion. Figure 1.1 shows a mobile robot that is representative of the systems we will consider. It consists of a mobile base with four wheels equipped with a variety of sensors and an onboard computer taking care of all computations necessary to perform its assigned task. The depicted robot is tasked with scouting orchards and take thermal images of leaves to determine water stress in plants. It is capable of autonomously operating for extended periods of time and to navigate through a set of preassigned way points scattered over a large area. The robot is equipped with two motors actuating the left and right wheels, and a linear actuator to raise and lower the thermal camera located in the white case. On the sensing side, the platform features numerous sensors, including an RTK GNSS receiver, two inertial motion units (IMU), a laser range finder, a RGBD camera, and wheel encoders. Once

Figure 1.1: A mobile robot collecting thermal images in a pistachio orchard.

these components are properly interfaced together and a suitable control software system is in place, the system can perform useful tasks that would be too time consuming for a human. For example, many farmers would like to completely scout their orchards frequently to promptly identify critical conditions, but due to labor shortages this is never done, thus generating preventable crop losses. This is a typical task where a mobile robot could be used to complement human expertise. Figure 1.2 shows the user interface displaying the results for given robot scouting mission.

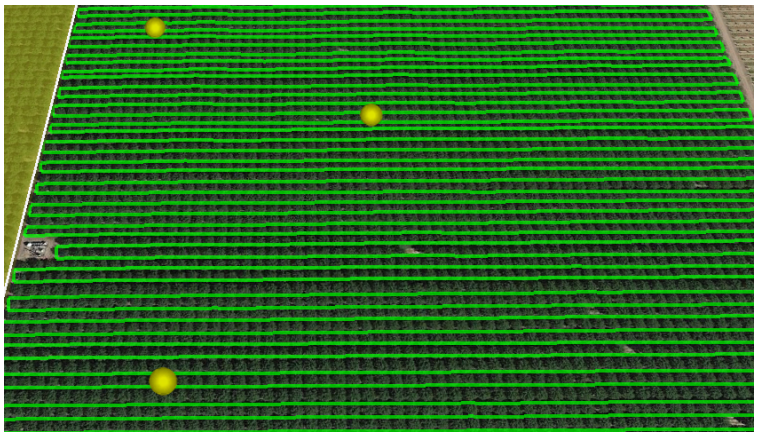


Figure 1.2: User interface for the orchard scouting robot.

Green lines overlayed over an aerial view of an orchard show the path the robot was assigned to follow. Yellow circles indicate the locations where the robot detected some anomalous conditions requiring human intervention. When the user clicks over these dots, a pop-up window appears displaying the data collected at that location and a short description of the problem detected. To autonomously complete a mission like this the robot must solve a variety of problems. The following list specifies some of the different functionalities needed

to complete a mission like the one described above.

Planning : the end user specifies only a few way points, e.g., the turning points at the end of the tree rows, or the boundaries of a rectangular region to scout. The planning module determines a set of intermediate way points to navigate through to either reach each of the the coarsely spaced way points specified by the end user or completely cover the assigned region.

Localization : the robot needs to continuously estimate its position to make sure it is making progress towards the way point it is going to. Knowledge of the pose is also needed to geolocalize the different conditions identified in the orchard so that the end user knows where they occurred. The localization module is in charge of estimating the pose of the robot as it moves through the orchard. Simply querying the GPS receivers is not enough because the returned location is jittery and the loss of line of sight with satellites due to the trees makes them very inaccurate at times. Hence the module integrates various sensors with different accuracy into a single position estimate.

Navigation and Obstacle avoidance : while the planning module determines the intermediate points to navigate through, the navigation and obstacles avoidance module ensures that the robot moves towards the given way point while avoiding obstacles that may be found on the way. This is particularly important given that the operating environment has not been previously preconditioned for robot operation.

The previous list does not provide a complete description of all the software modules running on the laptop mounted on the robot, but it outlines some of the basic capabilities needed to enable autonomous operation. In the following chapters we will study various algorithms to takle these and other associated challenges. Indeed, **the focus of these notes is on mathematical models, algorithms, and implementations**. In other words, **we will exclusively focus on the software components and will not discuss any hardware issue**. The reader will note that the above list of modules misses some important features, like a software module processing the data to determine if relevant conditions are spotted, e.g., an image processing algorithm aiming at detecting the presence of weeds (and possibly the type). While these are important aspects from a practical standpoint to deliver a fully functioning system, they will not be considered in the following because image processing and related disciplines are stand alone areas deserving separate treatment. In our algorithms and models, the output provided by sensor processing algorithms will be considered, but the algorithmic details will be abstracted away.

1.1.1 Terminology

The term *robot* comes from the Czech word *robota* that translates to *forced labor*. This expression traces its root to the play "Russum's Universal Robots" by Karel Čapek (1920). Another term often used to describe robots is *autonomous*. This word comes from the Greek words *auto* (self) and *nomos* (law) and shall be therefore intended as self-governed. The expression *autonomous robot* is pervasively used today. There is no universally accepted

definition of the term robot, and the meaning itself somehow evolves over time, as technology advances. In 1979 the Robotics Institute of America (RIA) defined a robot as "a reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through various programmed motions for the performance of a variety of tasks." The Merriam-Webster dictionary defines a robot[1] as "a machine that resembles a living creature in being capable of moving independently (as by walking or rolling on wheels) and performing complex actions (such as grasping and moving objects)". Wikipedia offers a similar definition[2], i.e., "a machine – especially one programmable by a computer – capable of carrying out a complex series of actions automatically." Key to these definitions of robot is the ability of being (re)programmed, i.e., being reconfigured to perform different tasks. This requirement, albeit important, shall be interpreted with some flexibility. For example, various floor cleaning robots have been developed and became a commercial success. One could however object that the only task they perform is cleaning the floor, and that by reprogramming it is possible to change how they complete their task but not the task they carry out. Therefore, they would not pass a strict definition of robot as *being able to perform a variety of tasks*. The same could be said for autonomous vehicles. Ultimately, however, we are more interested in the computational techniques used to perform complex tasks, and so floor cleaning robots (and autonomous vehicles) will be considered robots in the following.

1.2 Robot Modeling

Mathematical models are pervasively used in science and engineering to analyze and predict the behavior of a variety of systems. Accordingly, a mobile robot can be modeled in many different ways. Figure 1.3 shows one such possible model. It resembles the *agent* perspective presented in [47]. It outlines the mutual interaction between the robot (also called robotic agent) and the environment in which it operates, and it shows their coupling through a loop implemented with sensor and actuators. This is mostly an artificial intelligence (AI) standpoint.

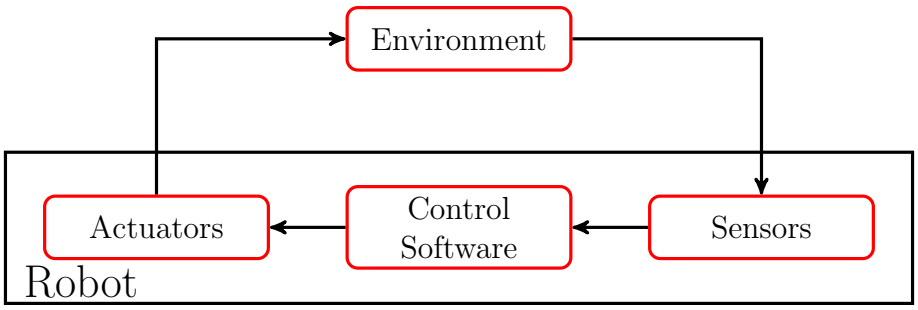


Figure 1.3: An agent view of a mobile robot.

As stated before, in these notes we are mostly interested in the middle block of the robot component, namely the *control software* (also called *controller*, *artificial intelligence*,

[1] https://www.merriam-webster.com/dictionary/robot
[2] https://en.wikipedia.org/wiki/Robot

etc.) This is the module in charge of deciding what to do next (input to provide to the actuators) based on the current values it receives from the sensor and its internal memory. The internal memory typically stores a mission specification defining what the robot is supposed to eventually do (the task it is assigned), as well as some information gathered while executing the task, e.g., a map of the environment incrementally built by the robot while working towards its assigned goals. While the technology defining sensors and actuators may be continuously improving, the algorithms we will present abstract from the low level details characterizing these hardware components and have therefore broader applicability even if the underlying sensing technology evolves over time.

The feedback loop between the robot and the environment suggests that a robot can also be seen as a *dynamical system*, i.e., a system whose state (pose, velocity, etc.) changes over time. Figure 1.4 shows this alternative model, and is pervasively found in control literature.

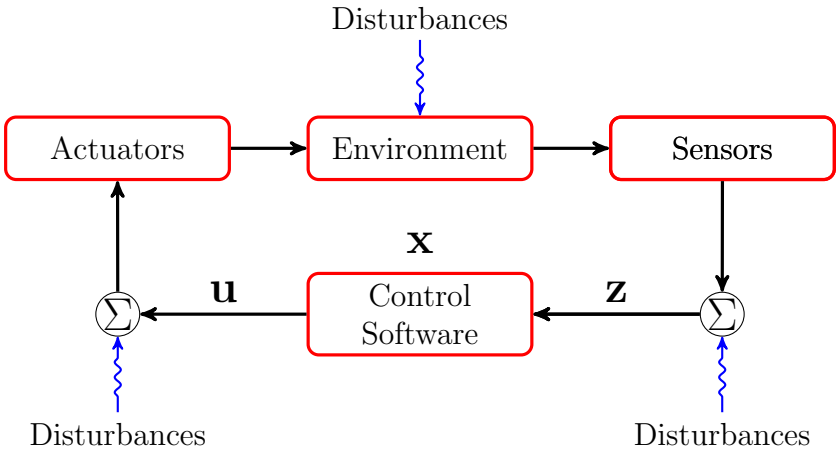


Figure 1.4: A dynamical system view of a mobile robot.

The figure introduces also some interesting concepts and symbols we will use extensively in the future. First, it outlines the presence of *disturbances* interfering with sensors and actuators and influencing the evolution of the environment as well. The *sum* character (Σ) symbolizes that disturbances are added on top of the signals produced and received by the robot. The unavoidable presence of disturbances significantly complicates the task of the controller. However, the reader should not be misled to think that algorithmic challenges would be trivial if noise was to disappear. For example, the motion planning problem is NP-hard even under the unrealistic hypothesis that the system is noise free. Disturbances affect both actuators and sensors. On the actuators side, disturbances manifest themselves through inaccuracies in the execution of the desired command. For example, if the controller commands the actuators to move the robot forward by 1m, the robot will invariably move forward of a distance different from 1m. Some times the difference may be small and even negligible (or even not detectable without highly accurate instruments), while other times the difference may be dramatic. Since these error sources appear every time the controller sends a signal to the actuators and they add up over time, the consequence is that trying to draw conclusions by just computing predictions of the effect of the commands issued by the controller is an approach doomed to failure. This is the reason why this strategy (open loop

control) is almost never used. Sensors are then introduced to make information about the environment available to the control software. Unfortunately, sensors are prone to disturbances too, and provide only noisy measurements. As in the case of actuators, errors affecting sensor readings can at times be small, but may also be very large, depending on the type of sensor used or on the conditions in which it is used (e.g., a GPS receiver used in an urban canyon where it cannot receive signals from a sufficient number of satellites.) Irrespective of the characteristics of the actuators and sensors considered, two general statements can be made and will define the behavior of many algorithms and techniques we will see in the following. Actions increase uncertainty whereas sensor readings decrease it[3]. The interplay and relative magnitude of these changes in an extremely important aspect characterizing the success or failure of many robotic systems (see Figure 1.5).

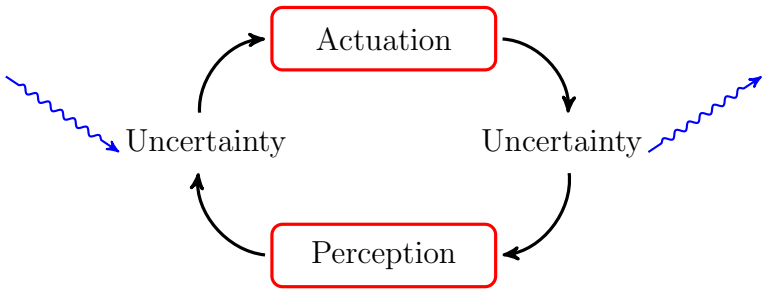


Figure 1.5: The uncertainty cycle between actuation and perception. Actuation generally increases uncertainty, whereas perception usually decreases it.

Figure 1.4 also introduces three symbols we will extensively use in the following, each of them to be interpreted as a vector. The symbol $\mathbf{u}$ is a vector in $\mathbb{R}^p$ and is the set of values that the controller sends to the actuators. In a sense, $\mathbf{u}$ is the output of the control software. The symbol $\mathbf{z}$ is a vector in $\mathbb{R}^q$ and is the set of values that the control software receives from the sensors. Hence, it can be seen as the input to the controller, though the controller typically receives also other inputs through the mission specification and so on. Note that in general $p \neq q$, i.e., the size of the two vectors is different. Finally, $\mathbf{x} \in \mathbb{R}^n$ is the so-called *state* vector. As pointed out in [4], "the state of a system is a collection of variables that summarize the past of a system with the purpose of predicting the future." Hence the state may include variables not only characterizing the robot, but also components external to the robots itself. For example, if a robot is tasked with approaching and grasping an object, the state will most likely going to include some variables defining where the object is (or ia believed to be). In general there is no method to *automatically* determining what should be included in the state, although in many cases this decision is rather simple to make. Note also that the same quantities could be differently represented in the state. In the grasping example, the pose of the object could be included in the state in terms of an external reference system, or it could be relative to the robot. Note moreover that some authors place the sum symbol in other places (e.g., between the actor and the environment) or between the environment and the sensor). These alternative scenarios do not substantially alter the nature of the problems we will deal with in the following and could be accommodated as

[3]In some instance this may not be true, but to simplify the discussion we assume this is the case.

well. The disturbances affecting the environment, actuators, and sensors can be modeled as *random signals* (a concept that will be formalized in Chapter A.) Consequently, the state $\mathbf{x}$ will not be known for sure, but will rather be a random variable itself that evolves over time. In Appendix A we develop appropriate mathematical tools (called stochastic processes) to model this temporal evolution.

1.3 Robots and Dynamical Systems

As formerly stated, "a *dynamic system* is a system whose behavior changes over time" [4]. A robot can therefore be viewed as a dynamic system, since it typically moves in its environment. Consequently, many ideas and concepts developed in *control theory* and in the theory of *feedback systems* are useful when studying robots. *Feedback* refers to a coupling between two systems so that they influence each other. This is illustrated in both Figures 1.3 and 1.4, where it is shown that the output of one subsystem (robot) is an input for the other subsystem (environment) and viceversa (the output of the environment is an input for the robot). In this case we have a *closed loop* system. One of the strengths of closed loop feedback systems is that they are robust to disturbances, and this is a desirable objective for a robot, i.e., we want to make sure it correctly works despite unexpected events. Due to the interactions between different subsystems (e.g., robot and environment), the study of closed loop systems is also more complex. For this reason, some times (e.g., in planning), we consider open loop systems, i.e., abstractions where there is no loop closure (see Figure 1.6). Systems operating in open-loop are less robust and therefore generally not deployed in practice, but their study can be nevertheless useful in some situations. For example, an open loop planner may produce a high-level path that is then followed using a low-level controller using a feedback loop.

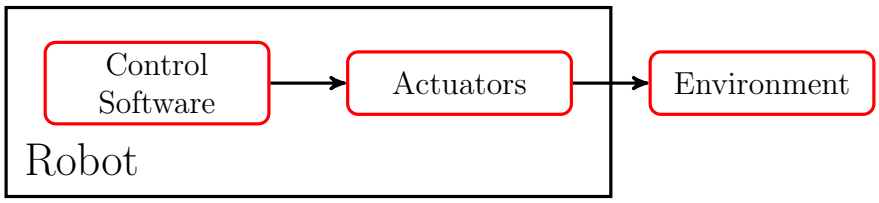


Figure 1.6: A robot operating in open loop.

According to the dynamical system view of a mobile robot (see Figure 1.4), the following relationships can then be written to model the temporal evolution of a mobile robot system

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, t) \tag{1.1}$$

$$\mathbf{z} = h(\mathbf{x}, \mathbf{u}, t) \tag{1.2}$$

where $\dot{\mathbf{x}}$ as usual indicates the derivative with respect to time. Eq. (1.1) is called the *state evolution equation* or *state transition equation*, and Eq. (1.2) is called the *state observation equation*. The use of a differential equation in Eq. (1.1) is natural, since a robot operates in the physical world and is subject to the continuous time laws of physics. However, since

our focus is on writing algorithms executed on digital devices, a discrete time version of the above relations will be often more useful[4], i.e.,

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t, t) \tag{1.3}$$
$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{u}_t, t). \tag{1.4}$$

Eq. (1.3) indicates that the current state $\mathbf{x}_t$ is a function of the previous state $\mathbf{x}_{t-1}$ and the current input $\mathbf{u}_t$. This model will be pervasively used in the discrete planning algorithms presented in chapter 6.

In most instances we will consider so-called *time invariant* systems, in which the above equations do not depend explicitly on time, e.g.,

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) \tag{1.5}$$
$$\mathbf{z} = h(\mathbf{x}, \mathbf{u}) \tag{1.6}$$

and

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t) \tag{1.7}$$
$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{u}_t) \tag{1.8}$$

for the discrete version. Moreover that in many practical situations the reading returned by the sensor is a function of the state only, and therefore the state observation equation can be written as

$$\mathbf{z}_t = h(\mathbf{x}_t) \tag{1.9}$$

where the dependency on $\mathbf{u}$ has been dropped.

For analysis purposes it will be often convenient to consider the special instance where functions f and h are linear. In this case Eq. (1.3),(1.4) can be rewritten as

$$\mathbf{x}_t = \mathbf{A}_t\mathbf{x}_{t-1} + \mathbf{B}_t\mathbf{u}_t \tag{1.10}$$
$$\mathbf{z}_t = \mathbf{C}_t\mathbf{x}_t + \mathbf{D}_t\mathbf{u}_t \tag{1.11}$$

where $\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t, \mathbf{D}_t$ are matrices of appropriate sizes. Similar equations can be rewritten for Eq. (1.1),(1.2). Note that in Eq. (1.10),(1.11) the matrices depend on time, as indicated by the subscript t. In the time invariant case these matrices are instead constant. Linear systems are rarely found in robotic applications, but non linear models can often be satisfactorily approximated through linearization. Therefore the study of linear systems is useful not only to gain insights into nonlinear problems but also from a practical perspective.

The following diagram depicts how the state evolves in the discrete time version:

$$\mathbf{x_0} \xrightarrow{\mathbf{u}_1} \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) \xrightarrow{\mathbf{u}_2} \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) \xrightarrow{\mathbf{u}_3} \mathbf{x}_3 = f(\mathbf{x}_2, \mathbf{u}_3) \rightarrow \ldots \rightarrow \mathbf{x}_n = f(\mathbf{x}_{n-1}, \mathbf{u}_n)$$

[4]Note that although we use the same symbols f and h in these two sets of equations, the functions will in general be different in the continuous and discrete case.

If $\mathbf{x}_0$ is known and no disturbances affect the system, then the sequence is deterministic, i.e., knowledge of $\mathbf{x}_0$ and of the sequence of inputs $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n, \ldots$ allows to determine all states $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n, \ldots$ This simplified version is based on an open loop view will be useful in some planning scenarios, but is unrealistic in robotic applications. In practice, $\mathbf{x}_0$ is most often not known without uncertainty, but rather modeled as a random variable and therefore even if the sequence of inputs $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n, \ldots$ is known, the sequence of states will be a sequence of random variables because of the initial uncertainty. On the other hand, even if $\mathbf{x}_0$ is known, the unavoidable disturbances affecting the system will still make the sequence of states a sequence of random variables. In the most general (and realistic) case, uncertainty stems from both disturbances and imprecise knowledge of the initial state $\mathbf{x}_0$. Despite the uncertainties in the state $\mathbf{x}$, it is important to observe that the sequence of inputs $\mathbf{u}_i$ is instead in general fully known because it is determined by the algorithms we will develop to control the robot. When considering sensor readings as well, the following charts can be considered. In the first one, we display the special case where the sensor reading is a function of the state only, i..e, $\mathbf{z}_t = h(\mathbf{x}_t)$ as per Eq. (1.9)

$$\begin{array}{ccccccccc} \mathbf{x_0} & \xrightarrow{\mathbf{u}_1} & \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) & \xrightarrow{\mathbf{u}_2} & \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) & \xrightarrow{\mathbf{u}_3} & \mathbf{x}_3 = f(\mathbf{x}_2, \mathbf{u}_3) & \longrightarrow & \ldots \\ & & \downarrow & & \downarrow & & \downarrow & & \\ & & h(\mathbf{x}_1) & & h(\mathbf{x}_2) & & h(\mathbf{x}_3) & & \end{array}$$

This configuration will be particularly relevant when studying Bayesian estimation algorithms (Chapter 8). The next configuration instead displays the general case where the output is a function of both the state and the input, as per Eq. (1.4):

$$\begin{array}{ccccccccc} \mathbf{x_0} & \xrightarrow{\mathbf{u}_1} & \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) & \xrightarrow{\mathbf{u}_2} & \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) & \xrightarrow{\mathbf{u}_3} & \mathbf{x}_3 = f(\mathbf{x}_2, \mathbf{u}_3) & \longrightarrow & \ldots \\ & \searrow & \downarrow & \searrow & \downarrow & \searrow & \downarrow & & \\ & & \mathbf{z}_1 = h(\mathbf{x}_1, \mathbf{u}_1) & & \mathbf{z}_2 = h(\mathbf{x}_2, \mathbf{u}_2) & & \mathbf{z}_3 = h(\mathbf{x}_3, \mathbf{u}_3) & & \end{array}$$

From a high level perspective one could say that the objective of the robot control system is to generate a sequence of inputs $\mathbf{u}_1, \mathbf{u}_2, \ldots$ so that eventually a desired state $\mathbf{x}_D$ is achieved, irrespectively of the disturbances encountered. For example, we want to determine a sequence of inputs so that the robot eventually moves to a preassigned goal position expressed as a desired final sate $\mathbf{x}_D$. Because of the unavoidable disturbances, it follows that $\mathbf{u}_1, \mathbf{u}_2, \ldots$ cannot be precomputed upfront, but must rather be computed *on the fly* integrating information collected at run time through the sensors. Hence, a robot control software could be seen as a function π that given the current state $\mathbf{x}_i$ (or an estimate of it) computes the next input, i.e., $\mathbf{u}_{i+1} = \pi(\mathbf{x}_i)$. $\mathbf{u}_{i+1}$ and $\mathbf{x}_i$ then determine the next state $\mathbf{x}_{i+1}$ as per Eq. (1.7). The "problem" with this approach is that π assumes that $\mathbf{x}_i$ is known, but this is generally not the case because of disturbances. Sensors are then used to determine or estimate $\mathbf{x}_i$.

Two concepts from control theory that are very important in robotics are *controllability* and *observability*. Informally speaking, a robot (system) is controllable if for every desired goal state $\mathbf{x}_D$ it is possible to determine a sequence of inputs so that the robot reaches the desired state. The state of the robot is *observable* if through repeated sensor observations

$h(\mathbf{x}_i)$ it is possible to determine the state.

Example 1.1. *Consider a robot operating on the plane, and assume there exists a stationary beacon placed at a known location, say x_B, y_B (see figure 1.7).*

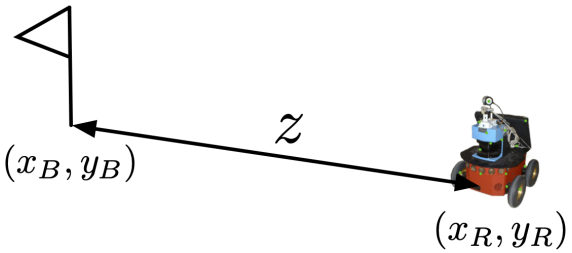


Figure 1.7: Abstraction of a robot equipped with a sensor to measure the distance to a beacon (displayed as a flag) placed at a known location.

The robot is equipped with a sensor returning the distance between the robot and the beacon. To simplify the problem, let us furthermore assume that the robot is stationary and can be abstracted as point robot*, i.e., it can be modeled as a point in the plane whose coordinates are x_R, y_R. Let the state of the robot be $\mathbf{x} = [x_R \; y_R]^T$. In this case the sensor reading is a scalar that can be modeled using Eq.* (1.9)

$$z = \sqrt{(x_B - x_R)^2 + (y_B - y_R)^2}$$

where we dropped the subscript t as we assumed everything is static. In this case with this only sensor the state of the robot is not observable, i.e., if the robot stands still it is not possible to determine x_R, y_R just by repeatedly querying the sensor (no matter what intermediate computation takes place.)

We conclude with one important observation. All equations presented in this section are deterministic, i.e., for the time invariant case the next state is a function of the current state and of the input, and the sensor reading is a deterministic function of the current state (and possibly last input). This approach is useful for modeling purposes, but is almost invariably never happening in reality. The realistic view to keep in mind is the one presented in Figure 1.4, where we outlined that disturbances affect both actuation and sensing. This means that in practice, neither f nor h are deterministic functions of their inputs, but are rather influenced also by stochastic disturbances. Much of the challenges in designing robot control software is in dealing with these uncertainties, and we will extensively discuss this problem in later chapters.

1.4 Robot Software Architectures

Robots are complex systems composed by the interconnection of multiple simpler subsystems. For example, numerous sensors (laser, camera, sonar, etc.) produce data streams that are consumed (processed) by one or more computational units. Each computational unit, in turn, may produce data that can be sent to various actuators (wheels, servos, etc.), or even

back to sensors (e.g., to change a configuration parameter.) Informally speaking, the term *architecture* describes how a complex system is broken into simpler (sub)systems and how these subsystems interact among them. The definition is recursive, i.e., once a complex system is divided into a set of simpler subsystem, each of them can be further decomposed into simpler subsystems, and so on, according to a hierarchical structure. This approach based on hierarchical decomposition is pervasive in engineering and it applies both to the robot hardware and software.

On the software side, particular emphasis is put on how data is exchanged between the various components and numerous ideas can be borrowed from software engineering. It should also be pointed out that robots can (and often should) be seen as distributed systems because they are indeed composed by multiple interacting components exchanging data through some network infrastructure. Consequently, if one focuses on how data is exchanged between components, various architectures developed in software engineering have been used to develop software controlling robots, like client/server and publisher/subscriber. Another aspect inherently related to this view focused on exchanged data is the definition and standardization of the messages exchanged between the various components. Through the years, the robotics community has developed and embraced different methods and frameworks. Recently, however, the Robot Operating System (ROS from now onwards) has emerged as the most commonly used approach. ROS will be introduced in chapter 2 and used throughout this book.

While data exchange is an important aspect when designing or selecting a software architecture, establishing which building blocks should be selected, and how they should interact with each other is equally important, and also part of what goes under the "architecture" definition. From this standpoint, countless choices are possible. One common choice is the *layered* architecture, whereby multiple components implementing different levels of competence are arranged in layers (see Figure 1.8 for an example). From the bottom to the top, each layer provides more complex functionalities building upon those below.

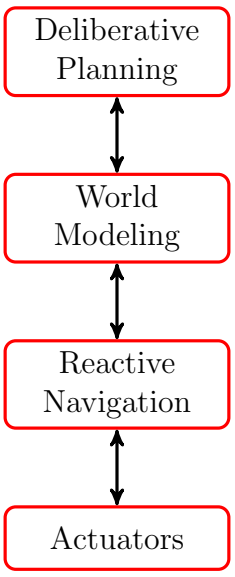


Figure 1.8: A possible example of a (simplified) robot architecture.

In this simplified model, each layer shall be thought as an independent thread of computation exchanging data with one or two other layers, as indicated by the arrows. As we will see, one of the advantages of ROS is that it encourages to decompose the software architecture as a set of software modules independently executing and exchanging messages with other modules.

Further Reading

There exist numerous textbooks devoted to the topic of mobile robotics. Almost all of them start offering a broad perspective to the field and some historic remarks. The reader is referred to the initial chapters in [2, 6, 12, 14, 17, 27, 41, 49, 50] for more details.

With regard to software architectures for robot programming, a chapter in the Springer handbook of robotics provides a good introduction to this topic and includes many useful references [28]. With the emergence of ROS and other frameworks for robot programming, software engineering for robotics has emerged as an independent area. Relevant references in this domain include [10, 24].

While feedback control theory is not strictly needed to follow the contents of this book, the interested reader aiming at a deeper knowledge of the subject is referred to [4, 54] for a comprehensive introduction.

Chapter 2

Introduction to ROS

2.1 ROS

ROS stands for *Robot Operating System* [33]. A thorough introduction to ROS would require a thick book on its own and is beyond the scope of these lecture notes. This chapter covers only some basic concepts useful for getting started. The reader is referred to the references at the end of this chapter for more comprehensive discussions, and to the ROS official website.[1] Despite its name, ROS is *not* an operating system, but rather a platform for developing software applications to control robots. It includes a communication infrastructure and a collection of tools and libraries. This set of resources is often described as a *framework*, or *middleware*, or SDK (software development kit).

Different ROS versions are released as *distributions*, similar to the Ubuntu distributions for Linux. A distribution is a versioned set of tools, packages, and libraries providing a stable codebase for developers of robot software. Accordingly, code developed for ROS typically targets a specific distribution, although it is possible for the same software to work without changes across multiple distributions. At the time of writing, the ROS community has transitioned from ROS 1 to ROS 2, and both coexist. The latest and final ROS 1 distribution is called Noetic Ninjemys and was released in May 2020. In the following, we will exclusively focus on ROS 2. Accordingly, whenever we refer to ROS for brevity, we implicitly mean ROS 2.

While in the past ROS 1 officially supported only Linux, starting with ROS 2, support for other operating systems has also been included, although the level of support varies between versions. All examples presented in the following have been developed and tested on a system running Ubuntu (see the foreword for details about the specific versions). The reader is referred to the ROS official website and to the MRTP GitHub for detailed, up-to-date instructions on how to install and configure ROS to follow and replicate the examples presented below. Table 2.1 shows a subset of the different ROS 2 distributions released so far and the associated supported platforms (older distributions have not been included in the table).

ROS 2 is a vast, powerful, and also complex system, featuring a large and growing number of components. To simplify, we can say that ROS 2 provides the following:

[1] `https://docs.ros.org/`

Distribution	Supported Platforms
Humble	Ubuntu 22.04, Windows 10
Iron	Ubuntu 22.04, Windows 10
Jazzy	Ubuntu 24.04, Windows 10
Kilted	Ubuntu 24.04, Windows 10

Table 2.1: Recent ROS 2 distributions.

- A software middleware enabling secure communications between components using different data exchange patterns (e.g., asynchronous, synchronous, and more);
- A collection of tools to simplify the development and debugging of complex robotic applications;
- Implementations of numerous algorithms solving basic robotic problems that can be composed together to develop more complex functionalities (e.g., localization, navigation, planning, teleoperation, etc.);
- Definitions of various data types (messages) to process and exchange data commonly needed to implement robotic applications (e.g., quaternions, transformation matrices, sensor data, etc.).

In the following, we present some of the most important concepts in ROS. The goal is not to provide a comprehensive introduction to ROS but rather to introduce a subset of concepts to show how algorithms and ideas developed in later chapters can be implemented in practice. Accordingly, some concepts and subsystems will either be simplified or skipped altogether, and the reader is referred to the official documentation for more details.

2.2 Nodes

A *ROS application* consists of various software entities called *nodes*, which work concurrently to complete an assigned task. This approach aligns with the fact that a mobile robot is a composition of various connected subsystems (sensors, actuators, computational units) that often process data in parallel and asynchronously. According to this approach, a robotic application can be decomposed into subsystems, with each subsystem independently executed by one or more nodes.

Figure 2.1 shows a small mobile robot[2] with some of its sensors and actuators highlighted. The robot includes a GPS receiver, a camera, and sonars. Additionally, it has two motors to actuate the left and right wheels (controlled by a single motor controller), as well as a motor to open and close the gripper in front of it. On the back of the robot, we can see the embedded computer (black box) that runs ROS to control the robot. A ROS application for this robot would not consist of a single node directly interacting with all these hardware subsystems but would instead feature multiple nodes, each performing a specific function. For example, we could assign a node to interact with each of the sensors and actuators and

[2] `https://www.nasa.gov/content/meet-the-swarmies-robotics-answer-to-bugs`

run an additional node to integrate all the sensor data, reason about it, and decide which actions should be taken, e.g., where to move or whether to open or close the gripper.

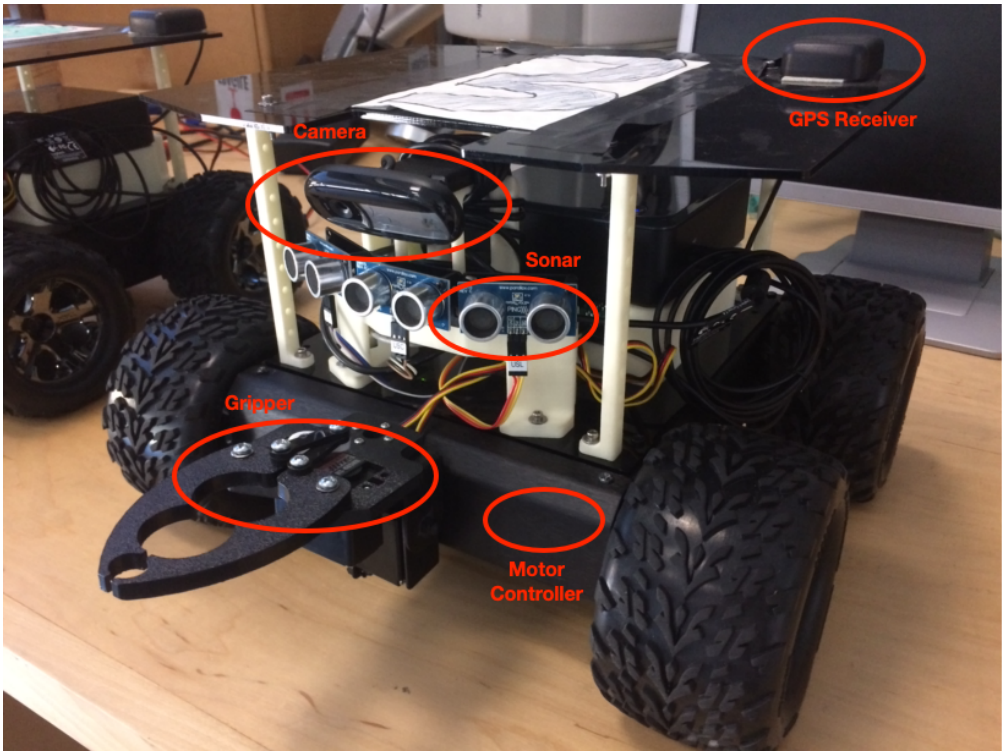


Figure 2.1: A *swarmie* mobile robot with some of its sensors and actuators highlighted.

This approach would lead to an architecture similar to the one depicted in Figure 2.2, where we further split the computation block into two nodes: one for image processing and one for decision-making. The oriented arrows between nodes illustrate how information flows between the different nodes. For example, the *GPS* node will be in charge of interacting with the GPS hardware receiver, i.e., exchanging data using its specific protocol, packaging the received data into a structure including latitude and longitude, and passing it along to other nodes that might need it. Likewise, the *motors* node will be in charge of interacting with the hardware running the motor controller and will translate high-level commands received from other nodes into low-level signals suitable for the specific hardware it interacts with. Importantly, each of these nodes is executed concurrently.

The modular decomposition into nodes facilitates code reuse, simplifies debugging, and increases robustness. To accomplish a shared objective, nodes exchange information and/or offer functionalities to other nodes. Although this greatly simplifies the idea, we could say that nodes perform some computation locally and then broadcast the results to other nodes. Accordingly, the input data for the computation carried out by a node is often the output produced by another node. ROS supports three forms of communication between nodes: *topics*, *services*, and *actions*. Each of these will be introduced in this chapter.

Nodes can be standalone processes, but they may also be separate threads inside the same process or even run on separate machines connected by a network. The decision on which approach to follow is left to the developer. Nodes can be written using various ROS client

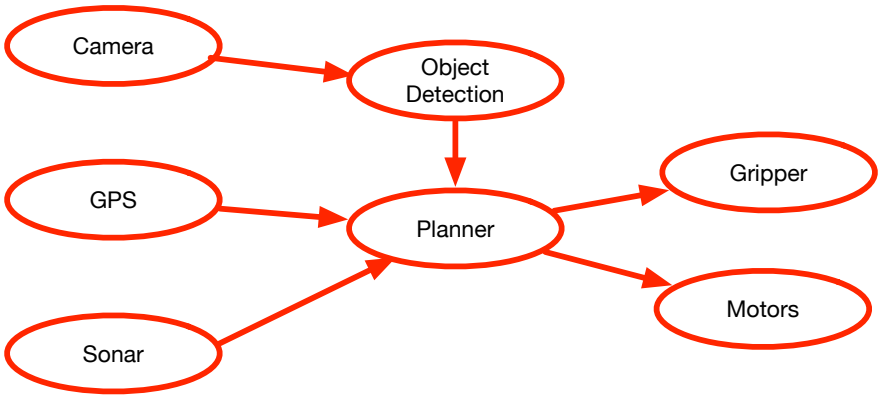


Figure 2.2: A hypothetical simplified ROS software architecture for the robot in Figure 2.1. Each oval represents a node while arrows between nodes show how information flows between them. Note that this architecture resembles a directed graph – see also Section 2.6.

libraries (RCL) available in different languages. The ROS team supports two languages, i.e., C++ and Python, but additional ones (Java, Matlab, and more) are also supported, thanks to RCL developed by the ROS 2 community. Importantly, a ROS application may include nodes written in different languages. In this book, we exclusively deal with the C++ client library, but all examples can be easily converted into other languages.

When a ROS node is run, it relies on various parameters that can have default values or be reconfigured. This topic will be further expanded in Chapter 5. For example, a robot equipped with two cameras may run two instances of the same node performing image processing to concurrently analyze the two streams of images coming from the two cameras. To distinguish the two input streams, each node should be assigned a unique name, and we will see how this can be done through remapping. Note that although, at the operating system level, nodes can be standalone programs, they are usually not directly invoked but are instead launched using a command-line tool called `ros2`. `ros2` will be extensively discussed in the remainder of this chapter.

2.3 Topics and Messages

A *topic* is an asynchronous, unidirectional stream of messages of a defined type. Topics implement one form of communication between nodes, where nodes can send or receive messages through the topic. A topic is *asynchronous*, meaning that senders and receivers do not rely on a shared timeline. A topic is *unidirectional*, i.e., messages always flow in one direction only, from senders to receivers. A topic is a *stream*, i.e., messages are sent sequentially, and the relative order is not changed. Finally, all messages sent through a topic are typed, and all messages exchanged through the same topic must have the same type. Therefore, what is exchanged are instances of messages, whereas a topic is the channel through which messages are exchanged. Because all messages exchanged through a topic must have the same type, with slight abuse of language, we can say that the topic has a type (e.g., if topic `A` is used to send messages of type `typeB`, we can say that topic `A` has type `typeB`.) A message is a data structure encapsulating either basic data types (integers, floats,

etc.) or complex structures with multiple fields. A complex message can also be obtained by composing other messages. The structure of a message will be discussed later, but for practical purposes, one can think of a message as a C++ `struct` built by combining primitive data types or other previously defined types. As with nodes, each topic and message must have a unique name, i.e., a string.

Data exchange through a topic follows a registration mechanism, i.e., before a node can start to send or receive messages to/from a certain topic, it must first explicitly associate with the topic. Each topic has a unique name (i.e., a string), and to send or receive a message to/from a topic, a node must identify the topic by name. The process of sending a message to a topic is called *publishing*. Similarly, if a node wants to receive messages from a certain topic, it needs to register, i.e., it needs to *subscribe* to the topic. Following this terminology, a node can be described as a *subscriber* or a *publisher* to a topic. A node can send/receive messages to/from a topic only after it has registered as a publisher or a subscriber for the topic. A node is not restricted to publishing or subscribing to a single topic, but can simultaneously subscribe to and publish on multiple topics. The handshake between publishers and subscribers connected to the same topic relies on the unique topic name. Care must be taken, as misspelling the topic name is a frequent source of bugs that result in messages not reaching their intended destination.

Topics are implemented through finite-length queues whose length is determined when the topic is created. Since it is a queue, if it fills up because the subscriber consumes messages slower than the publisher produces them, then older ones are dropped to make space for the new ones according to a first-in-first-out policy. It is also important to note that messages exchanged through a topic can be *one-to-one* (one publisher, one subscriber), *one-to-many* (one publisher, multiple subscribers), *many-to-one* (multiple publishers, one subscriber), or *many-to-many* (multiple publishers, multiple subscribers). Figure 2.3 illustrates these possibilities.

What happens when multiple nodes publish to the same topic? As can be imagined, their respective messages are queued to the same topic and passed to the receiver(s). If multiple receivers subscribe to the same topic, each receives a copy of all the messages sent through the topic (provided they can retrieve them fast enough). In Section 2.5, we will run a small example showing how multiple nodes can exchange messages through shared topics.

Finally, it is important to remember that while ROS comes with a large collection of predefined messages (i.e., data types), if needed, one can also define new message types that can be exchanged through topics. While this is possible, for most of the material covered in this book, it will not be necessary to define new messages, and we will rely on existing ones. Readers interested in exploring this possibility are referred to the ROS website or the references at the end of this chapter.

2.4 Packages and Workspaces

Packages and workspaces are two terms often used in ROS. A package is a container of ROS resources, such as source code, configuration files, compiled code, launch files, datasets, and more. A workspace, on the other hand, is a directory (sometimes referred to as a *location*) containing packages. Packages are a pervasive concept in ROS, and almost everything in

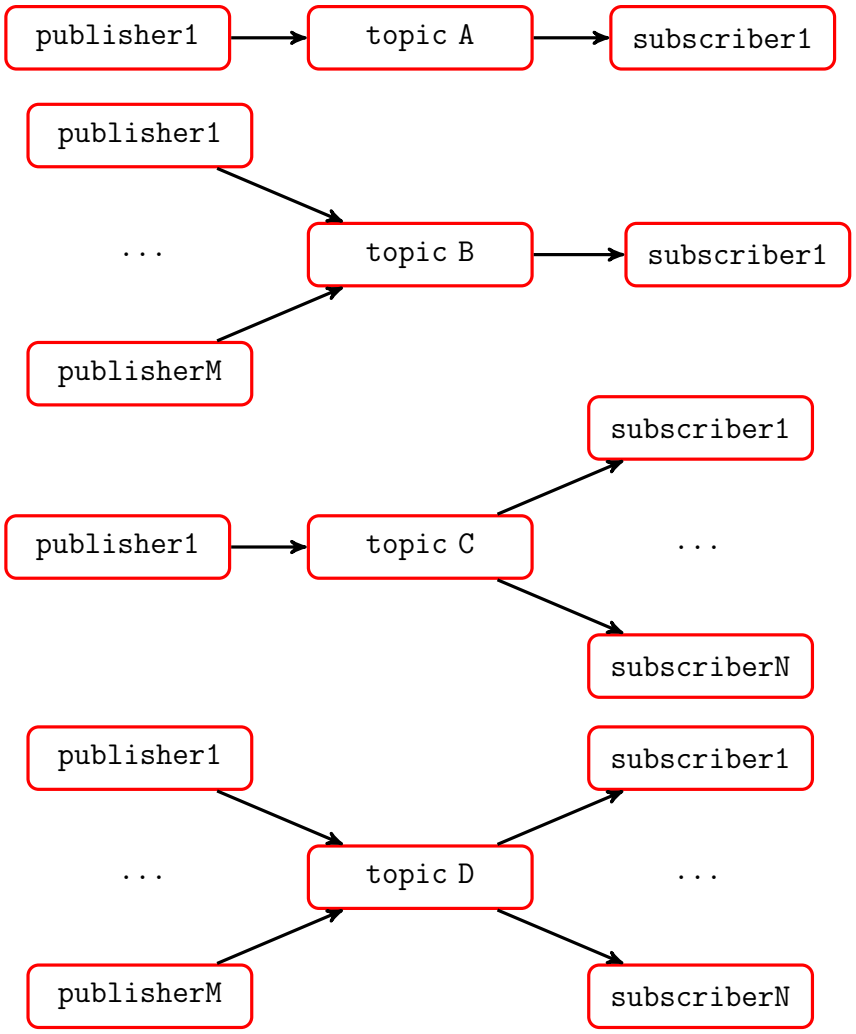


Figure 2.3: Topics in ROS can have all possible combinations of single/multiple publishers and subscribers.

ROS *lives* within a package. Conceptually, a package is similar to a folder in the file system. In a file system, it is up to the user to decide the name of a folder and its contents, but to keep things organized, it makes sense to place files with some logical association into the same folder. Similarly, in ROS, it is up to the developer to decide which packages should be created and what should go inside. The general rule of thumb is to include in the same package entities that are somehow related. Nevertheless, this is a subjective decision. A key difference between packages and folders, however, is that while it is possible to create folders inside other folders, it is not possible to create packages inside other packages. In addition to keeping things organized, packages also solve another problem: name clashes. By placing entities (nodes, messages, etc.) inside packages, we reduce the possibility of creating new entities that have the same names as existing ones, thereby avoiding ambiguities (think of namespaces in C++ or modules in Python). The *fully qualified name* of an entity is given by the name of the package where it is found, *and* its name (recall that almost everything must be inside a package). For example, the fully qualified name of a node is determined by

the name of the package containing the node and the node's name itself. Evidently, to avoid name clashes, it is necessary to ensure that each package has a unique name. At the operating system level, the contents of packages are stored inside directories. However, their internal structure in terms of files and subdirectories is not arbitrary but must follow a prescribed structure. To ensure that the required structure is followed, ROS provides utilities to create them. When executed, the command will create a directory with the name of the package and properly initialize it with subdirectories and required files. Details and examples will be given in section 3.2.

As mentioned above, a workspace is a directory containing packages. Packages inside a workspace can only be used after they are *made visible* by executing a script that adjusts the paths accordingly — a process we will describe shortly. When using ROS, you typically use at least two types of workspaces. The first is the *underlay*, i.e., the workspace that includes all the standard packages shipped with the ROS 2 distribution. Before you can start using ROS 2, you must make these standard packages available. Assuming you are using the `bash` shell, this is done by executing the following command:[3]:

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

This operation is also referred to as *sourcing the setup*. After executing this command, the standard packages shipped with ROS become usable, but only in the shell where you executed this command. If you open another shell, you will need to execute the command again in that shell. For this reason, it is convenient to add the `source` command to the shell startup scripts to ensure it is executed automatically every time you open a shell, without having to manually run it. If you use the `bash` shell, this can be done by adding the command to the file `.bashrc`. From now on, we will assume that the underlay has always been sourced in every shell. In addition to the underlay, you will typically use other workspaces containing packages with the code you are developing, or code provided by other parties. These additional workspaces are called *overlays*. Overlays can be added to the underlay as layers, i.e., you can add multiple overlays, with each newly added overlay being on top of those added previously. To add a workspace as an overlay, we follow a process similar to the one used to make the underlay visible, i.e., we source a file called `local_setup.bash`, which is automatically created when building a package. More details about creating workspaces and making them visible will be provided in Section 3.1. It is important to note that if the underlay or overlays are not sourced, the associated packages are not accessible and cannot be used by ROS. Especially for beginners, this is a frequent cause of problems.

2.5 The command line tool `ros2`

Before creating nodes from scratch, it is useful to gain preliminary experience by executing some of the simple applications included with ROS as part of the underlay.

To this end, in this section, we introduce the command-line tool `ros2`, which is used to perform many fundamental operations with nodes, topics, packages, and more. The `ros2`

[3]If you installed ROS 2 in a location other than `/opt/ros`, or if you are using a version other than `jazzy`, or if you use a different shell, you will need to adjust the command accordingly.

command line tool is generally executed as follows:

```
ros2 <command> <positional arguments>
```

Here, `command` represents one of a predefined set of commands. The possible positional arguments depend on the specific command used with `ros2`. For example, to list all available packages, we use `ros2` with the command `pkg` and the argument `list`:

```
ros2 pkg list
```

To obtain a list of all the possible commands accepted by `ros2` we can type

```
ros2 -h
```

and to get additional help about a specific command we type the name of the command followed by `-h`. For example, to obtain help about the `run` command (used to start ROS executables), type

```
ros2 run -h
```

and this will print

```
usage: ros2 run [-h] [--prefix PREFIX] package_name executable_name ...

Run a package specific executable

positional arguments:
package_name     Name of the ROS package
executable_name  Name of the executable
argv             Pass arbitrary arguments to the executable

optional arguments:
-h, --help       show this help message and exit
--prefix PREFIX  Prefix command, which should go before the executable.
Command must be wrapped in quotes if it contains spaces
(e.g. --prefix 'gdb -ex run --args').
```

So, from the help screen we see that the basic syntax is as follows:

```
ros2 run pakage_name executable_name
```

where `pakage_name` is the name of a package and `executable_name` is the name of a node in that package. All other components are optional, but the package and node name must always be given. To illustrate how to use `ros2`, we can run a set of nodes creating all the different scenarios displayed in figure 2.3. To do that, we use the package `demo_nodes_cpp` that provides the nodes `talker` and `listener`. `demo_nodes_cpp` is part of the underlay, so

no further action must be taken to make it visible to `ros2`. The node `talker` continuously publishes a string to a topic called `chatter`. The string is also echoed to the screen and includes a progressive integer to distinguish the successive messages being sent. The node `listener` instead subscribes to the topic and prints to the screen the messages it receives. To start the nodes, run the following commands in two separate terminal windows:

```
ros2 run demo_nodes_cpp talker
```

```
ros2 run demo_nodes_cpp listener
```

Note that the package name is the same, because both nodes are part of the same package. The output displayed in the terminal window running the `talker` node will be similar to the following

```
[INFO] [1671867496.759166671] [talker]: Publishing: 'Hello World: 1'
[INFO] [1671867497.759217757] [talker]: Publishing: 'Hello World: 2'
[INFO] [1671867498.759388264] [talker]: Publishing: 'Hello World: 3'
[INFO] [1671867499.758347144] [talker]: Publishing: 'Hello World: 4'
[INFO] [1671867500.758752912] [talker]: Publishing: 'Hello World: 5'
[INFO] [1671867501.758529495] [talker]: Publishing: 'Hello World: 6'
[INFO] [1671867502.759054233] [talker]: Publishing: 'Hello World: 7'
[INFO] [1671867503.758694266] [talker]: Publishing: 'Hello World: 8'
```

and the output for the `listener` will be something like

```
[INFO] [1671867581.507052479] [listener]: I heard: [Hello World: 20]
[INFO] [1671867582.506023418] [listener]: I heard: [Hello World: 21]
[INFO] [1671867583.500559769] [listener]: I heard: [Hello World: 22]
[INFO] [1671867584.499014372] [listener]: I heard: [Hello World: 23]
[INFO] [1671867585.496213494] [listener]: I heard: [Hello World: 24]
[INFO] [1671867586.494576127] [listener]: I heard: [Hello World: 25]
```

Each of the lines printed to the screen can be broken into four segments. The first (`[INFO]`) identifies the type of stream used to output the information (streams will be discussed in a later section.) In this case `[INFO]` indicates that what follows is standard information (i.e., it is not an error or a warning). The next numeric segment is the timestamp[4] for the message, followed by the node name and then followed by the message itself. This pattern for messages printed to the screen is pervasive in ROS and will appear repeatedly in subsequent examples.

It is instructive running the nodes multiple times, altering the order in which they are started, or stopping one while the other is still running and then restarting. The output printed shows the handshake between the two nodes. Then, it is also interesting starting multiple instances of `talker` in separate shells with a single `listener` running. The

[4]The timestamp is shown using the Unix Epoch, i.e., seconds and nanoseconds since Jan 1st, 1970.

`listener` output will be something like

```
[INFO] [1671867736.406355131] [listener]: I heard: [Hello World: 11]
[INFO] [1671867736.707731363] [listener]: I heard: [Hello World: 8]
[INFO] [1671867737.400471741] [listener]: I heard: [Hello World: 12]
[INFO] [1671867737.706266374] [listener]: I heard: [Hello World: 9]
[INFO] [1671867738.394589810] [listener]: I heard: [Hello World: 13]
[INFO] [1671867738.700404464] [listener]: I heard: [Hello World: 10]
[INFO] [1671867739.389693209] [listener]: I heard: [Hello World: 14]
[INFO] [1671867739.695336020] [listener]: I heard: [Hello World: 11]
[INFO] [1671867740.385000533] [listener]: I heard: [Hello World: 15]
```

confirming that the messages sent by the two instances of `talker` are queued together through the same topic and retrieved by the single subscribed node. Finally, running multiple instances of `listener` in separate windows will show that each node subsribed to a topic receives its own copy of the messages (the reader is strongly encouraged to run these simple examples.) These simple tests should be compared to Figure 2.3.

To further explore the functionalities of `ros2`, we will run `turtlesim`, a very simple application with a GUI that is shipped with ROS. While `turtlesim` is a toy application used to illustrate simple concepts, from a kinematic point of view the turtle is equivalent to many floor cleaning robots that are commercially available, so some ideas developed while interacting with the turtle have broader applicability. Like everything else, `turtlesim` is part of a package. The name of the package containing the `turtlesim` application is `turtlesim`. To see which executables are available inside the `turtlesim` package, we can use the following command:

```
ros2 pkg executables turtlesim
```

The output will be as follows:

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Note the format of the output. The name of each executable is preceded by the name of the package it belongs to. This is important, because, as we formerly stated, to run an executable we have to specify both its name and the name of the package it belongs to. At this point we can use the `run` command to start one of the executables included in the package `turtlesim`.

For example, to run the executable `turtlesim_node` we run the following command

```
ros2 run turtlesim turtlesim_node
```

When the command is executed, the following output is printed to the screen (each line starts with `INFO`; the output has been split in the following for display convenience):

```
[INFO] [1652849821.564382521] [turtlesim]: Starting turtlesim with node name
 /turtlesim
[INFO] [1652849821.615942206] [turtlesim]: Spawning turtle [turtle1] at
 x=[5.544445], y=[5.544445], theta=[0.000000]
```

and a window similar to the one shown in figure 2.4 appears. The first message indicates the name assigned to the node (`/turtlesim`), and the second communicates that a turtle with name `turtle1` was created (spawned) with a given location ($x = 5.544445, y = 5.544445$) and orientation ($\theta = 0$). The name of the turtle is important because we could instantiate multiple turtles in the same environment, and then control each of the separately by using the name of the turtle when issuing motion commands.

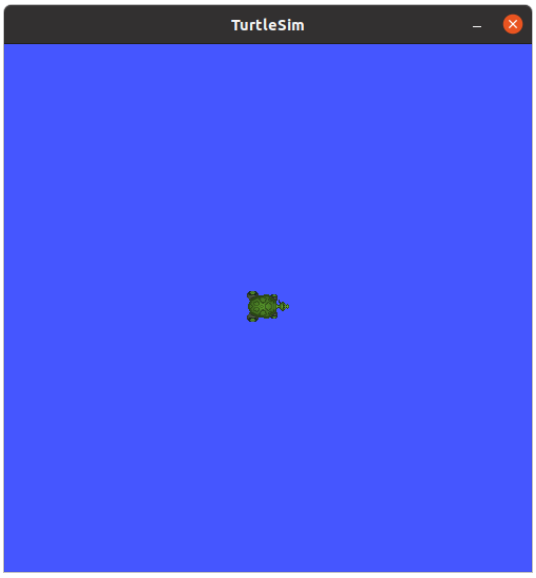


Figure 2.4: The turtlesim simulator. Different ROS distributions may display different colors for the background, or use a different icon for the turtle.

The turtle in the window can be moved around sending commands as we would do with a real robot. To this end, open a new shell and run the executable `turtle_teleop_key`. This can be started using the `run` command (note that we again must specify the name of the package):

```
ros2 run turtlesim turtle_teleop_key
```

Following the instructions printed on the screen, the turtle can be moved using the arrows on the keyboard, and also be rotated towards absolute orientations. At this point we have two running nodes, and it is evident that information must be flowing from one node to the other, because the keys stroke in the shell running `turtle_teleop_key` move the turtle in the GUI that is run by a separate node that was started in a different terminal. This information is indeed being passed through a topic. The commands `node` and `topic` can be used to get information about nodes and topics. First, we can use the command `node` with the option `list` to display all running nodes:

```
 ros2 node list
```

In this case it will print

```
/teleop_turtle
/turtlesim
```

confirming that the two nodes we started executing `ros2 run` are still running. More details about this command will be given in Section 2.8. Next, we can use the `topic` command with the option `list` to show all topics currently created in the system:

```
ros2 topic list
```

In this case the output will be

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

The list printed to the screen shows the names of the currently available topics. The `topic` command accepts various positional parameters in addition to `list`, and is extensively used when developing and debugging applications in ROS 2. For example, to determine the type of messages exchanged through a topic we can use the `type` option for the `topic` command, together with the name of the topic, e.g.,

```
ros2 topic type /turtle1/cmd_vel
```

prints the type of messages exchanged through the topic `/turtle1/cmd_vel` and will print the following output

```
geometry_msgs/msg/Twist
```

If we want to get additional information about a topic, we can use the `info` option, e.g.,

```
ros2 topic info /turtle1/cmd_vel
```

will print

```
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 1
```

In addition to the type information, this command shows that there is one publisher and one

subscriber to the topic. One may correctly guess that the publisher node is `/teleop_turtle` and the subscriber node is `/turtlesim`. Indeed, when a key is pressed in the shell running the `/teleop_turtle` node, a message of type `geometry_msgs/msg/Twist` is assembled and published to the topic `/turtle1/cmd_vel`. Another useful option for the `topic` command is `echo`. This command subscribes to a topic and prints to the screen all messages received. For example, if we type

```
ros2 topic echo /turtle1/cmd_vel
```

and then press some keys in the terminal running the `turtle_teleop_key` node, we will see an output similar to the following (the specific values vary depending on which key is pressed):

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

As we will learn later on, the output shows that the teloperation node is sending a command velocity with a linear velocity of 2 along the x axis and no angular velocity, i.e., a pure translation. Yet another option for `topic` is `hz` which determines and prints to the screen the average publishing rate of a topic. For example, to get the publishing rate of the topic `/turtle1/pose` we can run

```
ros2 topic hz /turtle1/pose
```

and the output will be like the following

```
average rate: 62.510
        min: 0.015s max: 0.017s std dev: 0.00030s window: 64
average rate: 62.490
        min: 0.015s max: 0.017s std dev: 0.00027s window: 127
average rate: 62.492
        min: 0.015s max: 0.017s std dev: 0.00027s window: 19
```

showing that the average frequency is 62 Hz. This is useful because as we said earlier topics are implemented with finite size queues, and when the queue is full older messages are dropped to make space for the new ones. Knowing the average publishing rate of a topic may help to make design decisions about the processing requested to a publisher or a subscriber.

`ros2` features numerous commands, and each of them can accept various parameters and options. Table 2.2 lists the commands accepted by `ros2` together with a brief description.

Options and parameters accepted by the commands can be obtained typing the name of the command followed by the option `--help`. Not all these commands will be used in the following.

Command	Scope
`action`	Command related to actions
`bag`	Command related to recording and replaying data
`component`	Command related to components
`daemon`	Interaction with daemons
`doctor`	Diagnostic functions
`interface`	Information about interfaces (aka messages)
`launch`	Runs a launch file
`lifecycle`	Information about nodes lifecycle
`multicast`	UDP packets management
`node`	Retrieves information about running nodes
`param`	Set/get parameters
`pkg`	Information about packages and package creation
`run`	Runs an executable from a package
`security`	Secure communication management
`service`	Command related to services
`topic`	Information and interaction with topics

Table 2.2: Commands accepted by `ros2`

.

Remark 2.1. *`ros2` can locate executables inside packages and run them. Therefore, at the operating system level paths have to be accordingly set. For nodes and packages part of the standard distribution (like `turtlesim`) this is achieved by sourcing the setup, as discussed while earlier on when we introduced the underlay. For packages and nodes developed by the user, however, this is not the case, and manual operations are necessary. This is the* overlapping *process we formerly mentioned.*

Remark 2.2. *For novices and seasoned users alike, it may be not easy to always remember the names of nodes or topics, or to recall which executables are provided by a certain package. To ease this task, `ros2` features tab-completion, i.e., after part of a command or parameter is typed, by pressing the TAB key it is possible to see all possible ways to complete a parameter needed by a command. For example, if in the above example one types `ros2 run turtlesim` and then hits TAB, the list of all executables in the `turtlesim` package is printed to the screen. The reader is encouraged to experiment with this feature while exploring the `ros2` command.*

2.5.1 Running distributed ROS applications

In the previous example, it was implicitly assumed that both nodes, `turtlesim_node` and `turtle_teleop_key`, were run on the same host. In most robotics applications, this is the

case, i.e., all ROS nodes are run on the same computer, which, in the case of mobile robots, is typically found on the robot itself. However, this does not have to be the case, and it is indeed possible to allocate the execution of different nodes on different hosts, as long as they are connected to the same network. This is implemented through the concept of domain IDs, which are associated with each host running ROS nodes. The domain ID is an integer, and if not explicitly set, it defaults to 0. When a node starts, it announces its presence to other nodes on the network and declares its domain ID. This announcement is automatically handled by the underlying communication middleware (DDS), and the user does not need to do anything about it. All nodes on the same network and sharing the same domain ID can interact with each other[5] as if they were on the same host. However, it is also possible to separate them by changing the domain ID to different values. This can be done by setting an environment variable called `ROS_DOMAIN_ID`.

2.6 The ROS Graph

As discussed earlier, a robot control system written in ROS 2 typically consists of multiple nodes interacting with each other. Information among nodes is exchanged through topics, actions, and services in a many-to-many fashion. Keeping track of these interactions is important for both design and debugging purposes. A natural way to model these interactions is by considering that nodes and topics are organized in a graph data structure, with nodes associated with vertices and topics, actions, and services associated with edges. This graph is called the *ROS graph*. ROS comes with a tool called `rqt_graph` that produces a graphical representation of this graph. To run it, simply type:

```
rqt_graph
```

When you execute `rqt_graph`, a GUI is launched, and by adjusting the controls, different details are included or omitted, resulting in different graphs. Figure 2.5 shows the graph obtained when only the `turtlesim_node` node is running.

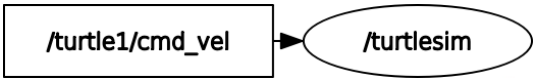


Figure 2.5: The ROS-oriented graph generated by `rqt_graph` when only the executable `turtlesim_node` is running.

Figure 2.5 is obtained by selecting the option *Nodes/Topics (active)* and hiding dead sinks, debug, and parameter topics. The figure shows one node (represented as an oval) and one topic (displayed as a rectangle). The strings shown inside the shapes are the names of the corresponding nodes and topics. The reader will note that the oval node displays a name (`/turtlesim`) different from the one used to start it with `ros2 run` (`turtlesim_node`). This happens because the name assigned to a node when it starts does not need to be the same

[5]To be precise, they must also share the same quality of service, but this is an advanced topic that will not be discussed here.

as the executable that is run to start the node. In fact, the latter is typically fixed, whereas the former can be arbitrarily changed in a variety of ways. The string inside the rectangle representing the topic is the name of the topic itself, as one can easily verify by running the command `ros2 topic list`. The oriented edge in the graph provides information about which topics a node subscribes to and which topics it publishes to. In the figure, the arrow from `/turtle1/cmd_vel` to `/turtlesim` indicates that the node `/turtlesim` subscribes to the topic `/turtle1/cmd_vel`. Consistent with the fact that topics are unidirectional streams, the ROS graph is oriented, i.e., its edges have a direction, as shown by the arrows between topics and nodes. If we now, in a separate shell, run the command `ros2 run turtlesim turtle_teleop_key`, the figure changes as shown in Figure 2.6 (note that the GUI will need to be refreshed after a new node is started).

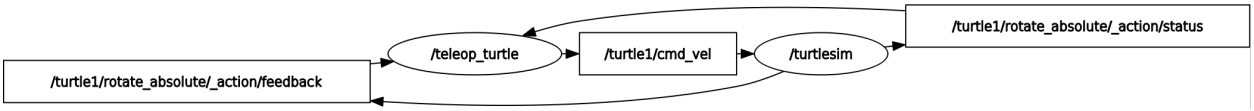


Figure 2.6: The oriented ROS graph generated by `rqt_graph` while both `turtlesim_node` and `turtle_teleop_key` are running.

We now see that there are two nodes running (`/turtlesim` and `/teleop_turtle`), as well as two new topics associated with an action (as indicated by their names—these will be ignored for now). The arrows connecting topics and nodes indicate the flow of messages. As we anticipated when we first ran the `turtlesim` demo, the arrows in the figure show that `/teleop_turtle` publishes to the topic `/turtle1/cmd_vel`, and `/turtlesim` subscribes to it.

`rqt_graph` can be configured to display a wide array of information, and the reader is encouraged to explore its features to experiment with its possibilities. If a problem arises when developing or running a ROS application, executing `rqt_graph` is usually one of the first steps to ensure that all necessary nodes are running and that the information flow is correctly set up between the various nodes, topics, actions, and services.

2.7 Inspecting topics and messages

Topics and messages are two core components of any ROS 2 application, and it is often useful to use `ros2` to obtain information about them. To this end, we run `ros2` in a separate shell after the relevant executables have already been started. As we discussed in a previous section, `ros2 topic list` prints to the screen the list of active topics. If the `turtlesim` example is running, `ros2 topic list` prints only the names of the topics. To gather additional information, two additional options can be added: `-t` (or, equivalently, `--show-types`) and `--include-hidden-topics`. For example, if after starting `turtlesim_node` and `turtle_teleop_key`, we run

```
ros2 topic list -t
```

we get the following output, where now, after each topic, we see its type.

```
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

If we instead run `ros2 topic list --include-hidden-topics` the output will be

```
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
/turtle1/rotate_absolute/_action/feedback
/turtle1/rotate_absolute/_action/status
```

The two new *hidden* topics listed at the end are associated with the action `rotate_absolute`, which we have not yet discussed (see also Figure 2.6). Actions are another method for exchanging information between nodes that, under the hood, rely on topics; hence the presence of the hidden topics. Alternatively, as we have previously seen in Section 2.5, if we want to determine the type of a single topic, we can use the command `ros2 topic info` followed by the name of the topic, e.g., `ros2 topic info /turtle1/cmd_vel`. This command will print the type of the topic, as well as the number of nodes publishing to and subscribed to the topic, but it does not indicate which nodes are connected to the topic. This information can be obtained by adding the `-v` (verbose) option. If we type

```
ros2 topic info /turtle1/cmd\_vel -v
```

the output will be (the numbers following `GID`, `Topic type hash`, and `QoS` will be different and can be ignored for the time being):

```
Type: geometry_msgs/msg/Twist

Publisher count: 1

Node name: teleop_turtle
Node namespace: /
Topic type: geometry_msgs/msg/Twist
Topic type hash: RIHS01_9c45bf16fe0983d80e3cfe750d6835843d265a9a6c46<truncated>
Endpoint type: PUBLISHER
GID: 01.0f.ff.89.c9.14.c0.0f.00.00.00.00.00.00.14.03
QoS profile:
Reliability: RELIABLE
History (Depth): UNKNOWN
Durability: VOLATILE
```

```
Lifespan: Infinite
Deadline: Infinite
Liveliness: AUTOMATIC
Liveliness lease duration: Infinite

Subscription count: 1

Node name: turtlesim
Node namespace: /
Topic type: geometry_msgs/msg/Twist
Topic type hash: RIHS01_9c45bf16fe0983d80e3cfe750d6835843d265a9a6c46<truncated>
Endpoint type: SUBSCRIPTION
GID: 01.0f.ff.89.97.14.2a.68.00.00.00.00.00.00.1d.04
QoS profile:
Reliability: RELIABLE
History (Depth): UNKNOWN
Durability: VOLATILE
Lifespan: Infinite
Deadline: Infinite
Liveliness: AUTOMATIC
Liveliness lease duration: Infinite
```

In this case, we see additional details about the nodes publishing to and subscribing to the topic, such as their node name as well as their namespace. The command `ros2 topic list` also accepts other options that can be explored by typing `ros2 topic list -h`.

To get more details about the structure of messages exchanged through a topic, the command `ros2 interface` can be used. In a previous example, we saw that the type of the topic `/turtle1/cmd_vel` is `geometry_msgs/msg/Twist`. To see the structure of this message, i.e., which information it carries, we can run the command

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

The output will be

```
# This expresses velocity in free space broken into its linear and angular parts

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

The first line starting with `#` is obviously a comment, while the rest shows that a `Twist` message includes a linear and an angular component, and that each of them is an instance of

the message `geometry_msgs/msg/Vector3`. (We know that `Vector3` comes from the same package as `Twist` because the name of the package is not provided before `Vector3`.) The output shows the components of a message of type `Vector3`, but this can be further inspected with the command

```
ros2 interface show geometry_msgs/msg/Vector3
```

and we get

```
# This represents a vector in free space.

# This is semantically different than a point.
# A vector is always anchored at the origin.
# When a transform is applied to a vector, only the rotational component is
                                                                        applied.

float64 x
float64 y
float64 z
```

where we see that `Vector3` has three components, each of which is a floating point number represented on 64 bits (`float64`). Like structures in C++, messages are ultimately compositions of elementary data types, and such compositions can be recursively combined to obtain more complex messages. In the example we just saw, `Twist` includes two components, i.e., `linear` and `angular`. Both are messages of type `Vector3` and each `Vector3` is made of three `float64`.

Observe that when we indicate a message, we must specify its fully qualified name (e.g., `geometry_msgs/msg/Vector3`) and not just `Vector3`. This is to avoid name clashes, as there could be multiple messages called `Vector3` defined in different packages. This approach, based on *namespaces*, will be further discussed later on. As previously suggested, since it may not always be easy to remember which package a message belongs to or which messages are provided by a package, pressing the TAB key while typing the command provides suggestions for possible completions.

Having discovered the structure of messages of type `geometry_msgs/msg/Vector3`, we can now use `ros2 topic pub` to manually publish a message to this topic. This command is useful during debugging to send specific messages through a topic from the command line.

For example, if we type (on a single line)

```
ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist
'{linear:  {x: 0.1,y: 0.0,z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

we will send messages of type `geometry_msgs/msg/Twist` to the topic `/turtle1/cmd_vel`. Each message will have the `linear` and `angular` values specified in the YAML string, i.e., a linear velocity of 0.1 along the x direction[6] and no angular velocities. The reader should

[6]The x axis in this case referes to the direction pointed by the turtle head. This will be clarified in chapter

observe the correspondence between the parameters passed to `ros2 topic pub` and the structure of the message, including its components `linear` and `angular`. The message will be repeatedly sent every second until the command is stopped. The command `pub` accepts many options, and the reader is referred to the documentation for more details. Among these, we mention `-1` (or equivalently `--once`) to publish just a single message and then exit, and `-t` (or `--times`) followed by an integer N to send the message N times.

2.7.1 Understanding the recursive structure of a message

It was previously stated that ROS messages can be thought of as a C++ struct. This is not just an analogy, because, as we will see later, messages are indeed stored in C++ ROS programs as instances of structures. Consequently, a message may include both elementary data types (integers, floats, etc.) and other messages. This is similar to what happens with user-defined structures in C++, where each field has a type that is either an elementary data type or a user-defined data type, such as another structure. These definitions can be recursive, allowing one to nest a structure inside another structure, and so on. When `ros2 interface show` is used to inspect the structure of a message, the type and name of each field are printed. If the type is not an elementary data type (i.e., it is a message), then we can call the command multiple times until we reach only elementary data types. In the previous subsection, we used `ros2 interface show` to explore and understand the structure of a message whose subcomponents were from the same package (`geometry_msgs`). In this section, we take a deeper dive and show how to recursively investigate the structure of a message in a more complex scenario. For example, if we run

```
ros2 interface show geometry_msgs/msg/PoseWithCovarianceStamped
```

we get the following output:

```
# This expresses an estimated pose with a reference coordinate frame and
                                                                    timestamp

std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id
PoseWithCovariance pose
  Pose pose
    Point position
      float64 x
      float64 y
      float64 z
  Quaternion orientation
      float64 x 0
```

4 when we will introduce robot kinematics.

```
      float64 y 0
      float64 z 0
      float64 w 1
  float64[36] covariance
```

We observe that the message contains two other messages called `header` and `pose`. `header` is a message of type `std_msgs/Header`, while `pose` is a message of type `PoseWithCovariance`. Note that `Header` is preceded by `std_msgs`, whereas `PoseWithCovariance` is not preceded by anything. This means that `Header` is from the package `std_msgs`, while `PoseWithCovariance` belongs to the same package as `PoseWithCovarianceStamped`, i.e., `geometry_msgs`. This is a general rule: if the name of the message is not preceded by the name of the package, it means that it belongs to the same package as the message containing it. Note that the output already shows the recursive structure of `header` and `pose`. However, we can use

```
ros2 interface show std_msgs/msg/Header
```

to inspect just the structure of a message of type `std_msgs/Header`, which is

```
# Standard metadata for higher-level stamped data types.
# This is generally used to communicate timestamped data
# in a particular coordinate frame.

# Two-integer timestamp that is expressed as seconds and nanoseconds.
builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec

# Transform frame with which this data is associated.
string frame_id
```

We see that a message of type `std_msgs/Header` has two fields called `stamp` and `frame_id`. `stamp` is a message of type `builtin_interfaces/Time` while `frame_id` is of type `string`. The type of `frame_id` is elementary, so no further recursive inspection is needed. Instead, we could use

```
ros2 interface show builtin_interfaces/msg/Time
```

for determining the structure of `stamp` and get

```
# Time indicates a specific point in time, relative to a clock's 0 point.

# The seconds component, valid over all int32 values.
int32 sec

# The nanoseconds component, valid in the range [0, 10e9).
uint32 nanosec
```

`sec` and `nanosec` are of types `int32` and `uint32`, so no further calls to `ros2 interface` are needed to determine the full structure of `header`. A similar process could be followed to unveil the structure of `PoseWithCovariance`. Observe that even though `ros2 interface show` already breaks down the recursive structure of a message (e.g., `PoseWithCovarianceStamped`), it is nevertheless useful to call `ros2 interface show` on each message type to print out the comments describing the meaning of each field.

If one wants to send or receive a message of type `PoseWithCovarianceStamped`, an instance of the struct `geometry_msgs::msg::PoseWithCovarianceStamped` should be used. The following code snippet shows how to hypothetically initialize part of it.

Listing 2.1: Accessing Message Subfields

```
1 geometry_msgs::msg::PoseWithCovarianceStamped pwcs;
2
3 pwcs.header.stamp.sec = 2;
4 pwcs.pose.pose.position.x = 3.4;
5 pwcs.pose.pose.orientation.w = 1;
6 pwcs.pose.covariance[3] = 0;
```

The important aspect is that the output of `ros2 interface show` is sufficient to properly instantiate and initialize the message, without having to read the header file where the corresponding C++ structure is defined. In fact, such header file is autogenerated by an intermediate program and not meant to be read by the programmer.

Remark 2.3. *The previous examples have shown a peculiar aspect of how message names are specified. For example, both* `PoseWithCovarianceStamped` *and* `PoseWithCovariance` *are from the* `geometry_msgs` *package. Therefore, when referring to* `PoseWithCovariance` *from within the definition of* `PoseWithCovarianceStamped`*, we do not have to specify the package name. Instead, since* `header` *is of type* `std_msgs/Header`*, i.e., it is a message from the package* `std_msgs`*, we need to specify both the name of the message and the name of the package. However, when we use* `ros2 interface` *to inspect the structure of a message, observe how messages are located in a* `msg` *folder within the package. That is, we must use the command* `ros2 interface show std_msgs/msg/Header` *and not* `ros2 interface show std_msgs/Header` *(the latter will result in an error).*

2.8 Inspecting nodes

Another command useful for debugging ROS applications is `ros2 node`. For example, if we run

```
ros2 node list
```

we obtain the list of all currently running nodes. Assuming the `turtlesim` example is still up and running, the output would be:

```
/teleop_turtle
/turtlesim
```

The command `ros2 node info` instead prints detailed information about a running node. For example the command

```
ros2 node info /turtlesim
```

produces the following output:

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically:
                                  rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

The output is divided into three sections: one for topics, one for services, and one for actions. Each section contains two subsections. `Subscribers` and `Publishers` list the topics to which the node subscribes or publishes. For each topic, both the name (e.g., `/turtle1/pose`) and the type (e.g., `turtlesim/msg/Pose`) are displayed. Similarly, the subsections `Service Servers` and `Service Clients` list the services that the node provides or uses. Finally, `Action Servers` and `Action Clients` list the actions provided by the node and those it uses. As with topics, for each service and action, the command also lists the

associated type.

2.9 Services

Besides message passing through topics, ROS provides two other ways for nodes to interact with each other: services and actions. As suggested by the name, *services* implement a *client/server* computation paradigm through a call/response communication model to perform remote procedure calls. A server node provides a service that can be requested by multiple client nodes, but each service can be provided by only one server node. When a client node initiates the service call, a function implementing the service is executed by a different server node. The server node typically receives some parameters associated with the request message and may return results to the client via a response message. In general, both the request and the response may also be empty if there are no parameters passed or values returned. Figure 2.7 illustrates the interaction between a client and a server. The request and response are separate messages exchanged between the client and the server. As with topics, the type of this structure is said to be the type of the service.

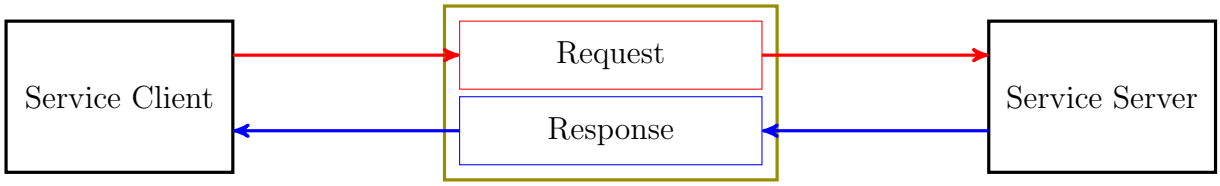


Figure 2.7: Information exchange between client and server in a ROS service. Note that request and response are two separate messages exchanged between client and server, and that either of them could be empty.

The response is provided to the client only after the server completes the requested computation, but the exchange is not blocking, i.e., after sending a request through a call, the client can wait for the response or perform other tasks before waiting for the response.[7] Information between clients and servers is exchanged only during this request/response process. Once this interaction is completed, i.e., when the server provides its response, the communication stops. Note that, as with topics, interactions between clients and servers are, in general, many-to-many. This means that a client may call a service multiple times and may request services provided by different servers. Likewise, a server may respond to requests from multiple clients. Services are generally well suited for operations that take a limited amount of time to complete, e.g., reconfiguring a sensor or processing an image. Similarly to what we have seen for topics, ROS comes with a set of predefined services, but programmers can also define new ones.

Simple examples of services and actions will be presented in this chapter, while more detailed examples will be discussed in greater depth in Chapter 5.

[7] This is a notable difference from ROS 1.

2.10 Interacting with services

The command-line tool `ros2` provides dedicated commands to retrieve information and interact with services provided by nodes. As in the previous examples, in the following we assume that the node `turtlesim_node` is running. The command

```
ros2 service list
```

prints to the screen the list of available service. Its output is

```
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
```

As for the command `ros2 topic`, if we add the option `-t` the output will include also the type of message associated with the service, i.e.,

```
ros2 service list -t
```

will produce

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
/turtlesim/describe_parameters [rcl_interfaces/srv/DescribeParameters]
/turtlesim/get_parameter_types [rcl_interfaces/srv/GetParameterTypes]
/turtlesim/get_parameters [rcl_interfaces/srv/GetParameters]
/turtlesim/list_parameters [rcl_interfaces/srv/ListParameters]
/turtlesim/set_parameters [rcl_interfaces/srv/SetParameters]
/turtlesim/set_parameters_atomically
                                    [rcl_interfaces/srv/SetParametersAtomically]
```

To get information about the type of a single service, one can run the command `ros2`

`service type <servicename>`, where `<servicename>` is the name of an available service. Similarly to what we have seen for topics, the command `ros2 interface` can be used to inspect the structure of the input and output of a service. The syntax is exactly the same. For example, if we type

```
ros2 interface show turtlesim/srv/Spawn
```

we get the output[8]

```
float32 x
float32 y
float32 theta
string name #Optional. A unique name will be created and returned if this is
                                                                          empty
---
string name
```

This service, when called, will spawn a new turtle in the GUI. The three dashes in the output `---` separate the request from the response (see Figure 2.7). Each of the two parts is interpreted similarly to what we saw in Section 2.7.1, where we analyzed the structure of messages. In this example, we see that the request message includes four fields. The first two are the coordinates where the turtle will be spawned, whereas the third is the orientation (heading). All of these are input parameters of type `float32`. The fourth optional parameter is the name to be given to the turtle being spawned, represented by a `string`. The structure of the response, starting after `---`, shows that the service returns a message consisting of a single string called `name`. This is the name assigned to the turtle being created. If the input parameter `name` is provided, then the output parameter `name` will be the same. However, if the input `name` is not given, the server will generate one and return it as part of the response. It is possible to call a service from the command line using the command `ros2 service call`. This command takes three additional positional parameters. The first is the name of the service to call, the second is the type of the service, and the last is the request encoded as a YAML string. Note that if a service accepts no input parameters, then the request is empty, and this last parameter is omitted. If we want to call the `Spawn` service, we can then give the following command (typed all in one line):

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 1, y: 1, theta: 0, name:
 'T1'}"
```

and the output on the shell will be

```
requester: making request: turtlesim.srv.Spawn_Request(x=1.0, y=1.0, theta=0.0,
name='T1')
```

[8]Note that the naming follows the same structure used for messages, i.e., the definition of services is found inside the `srv` folder in the package.

```
response:
turtlesim.srv.Spawn_Response(name='T1')
```

In this case since we provided the input parameter `name` as part of the request, the server included in the response the same name, i.e., `T1`. If we instead spawn another turtle without providing a name:

```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 5, theta: 0.6}"
```

the output will be

```
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=5.0, theta=0.6,
  name='')

response:
turtlesim.srv.Spawn_Response(name='turtle2')
```

where now we see that the server created a new unique name for the new turtle being spawned and returned it as part of the response. Figure 2.8 shows the turtlesim window after the two additional turtles have been spawned.

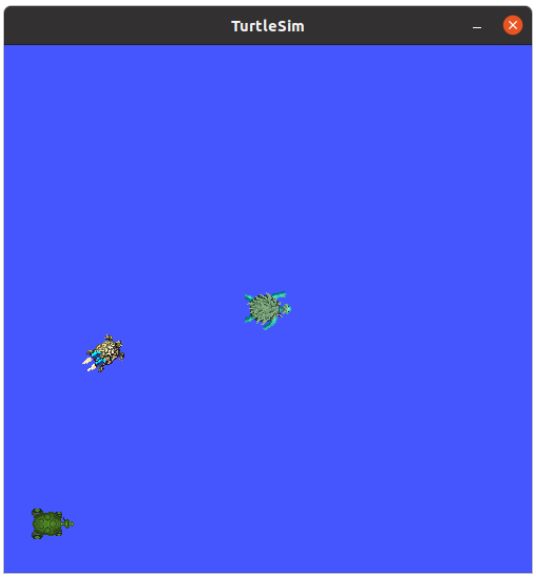


Figure 2.8: The turtlesim simulator after two more turtles have been spawned using the `/spawn` service.

If we now run the command `ros2 topic list` the output will be

```
/T1/cmd_vel
/T1/color_sensor
/T1/pose
/parameter_events
```

```
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
/turtle2/cmd_vel
/turtle2/color_sensor
/turtle2/pose
```

The meaning of this output is straightforward to interpret. Each turtle is associated with three topics, each prefixed with the name of the turtle. More specifically, the name assigned to the turtle defines its namespace to keep each turtle's set of entities separate from the others. As expected, each instance of the turtle subscribes to its own `cmd_vel` topic and publishes to its own `color_sensor` and `pose` topics. Even though we now have three turtles in the simulator, we still have just one node running, as can be verified by running `ros2 node list`. The node `/turtlesim` manages the three instances we have created. To move the turtle `T1`, we must publish to `/T1/cmd_vel`, whereas publishing to `/turtle2/cmd_vel` will move the turtle called `turtle2`. As a final exercise with services, we can use the service called `/kill` to remove one of the turtles. We have previously seen that the type of this service is `turtlesim/srv/Kill`, so to learn more about it, we use the command

```
ros2 interface show turtlesim/srv/Kill
```

we get the output

```
string name
---
```

This means that the request to the service accepts one string parameter called `name` and the response is empty. As it may be guessed, `name` is the name of the turtle we want to remove. To remove the turtle called `T1` we can then call the service as follows

```
ros2 service call /kill turtlesim/srv/Kill "{name: 'T1'}"
```

and the output will be

```
requester: making request: turtlesim.srv.Kill_Request(name='T1')

response:
turtlesim.srv.Kill_Response()
```

showing that the response is empty because the server did not return any value. However, you can verify that in response to the service call one of the turtles has been removed from the GUI.

2.11 Actions

Actions are the third and final interaction method offered by ROS and are built on top of topics and services. Informally speaking, actions can be seen as services that may take a long time to complete and therefore require a continuous flow of information to be passed back to the client to inform it about progress toward the goal. A classic example of a task that could be implemented as an action is navigation to a desired goal location. As it may take quite some time to complete the assignment, navigation could be implemented as an action, and the node in charge of moving the robot to the goal location would provide regular feedback to the node that initiated the action to inform it about its progress toward its objective—or lack thereof. Informed by the feedback, the node that initiated the action could let the action continue until it is completed or cancel it, i.e., request to stop it halfway through its execution. Similar to services, actions use a client/server approach whereby a server node may offer one or more actions that can be used by one or more client nodes. However, unlike services, actions can be preempted, i.e., the client that initiates an action can decide to interrupt its execution before it is completed. Figure 2.9 illustrates the information exchange involved in an action.

Figure 2.9: Information exchange between client and server in a ROS action. Goal, and results are service calls, while feedback is sent through a topic.

The three components involved in an action are a goal service call (from the client to the server), feedback messages from the server to the client through a topic, and a result service call (also from client to server) through which results are obtained. The interaction is as follows: The client submits a goal service call to the server, which responds with an acknowledgment that the request has been received. The server then initiates some computation to fulfill the goal request and generates a stream of intermediate feedback messages through a topic. The client can also submit a result service call, but its response may be delayed, as it will be communicated only when the action terminates. Based on the received feedback, the client may decide to continue to wait for the action to terminate and provide the response to the result service call, or it may decide to end the action, for example, if progress toward completion is too slow.

2.12 Interacting with actions

We now show how `ros2` can be used to get information about actions and initiate them. As for services, we rely on the `turtlesim` package for some very simple cases. To this end, in

the following we assume that the nodes `turtlesim` and `turtlesim_teleop` are running. To get a list of available actions we use the command

```
ros2 action list
```

which will produce the output

```
/turtle1/rotate_absolute
```

showing that there is just one action available. This action, when executed, will rotate the turtle towards a desired absolute orientation. As for services, if we add the option `-t` we can get more detailed information about the action:

```
ros2 action list -t
```

produces the output

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

Next, we can display the structure of the action using

```
ros2 interface show turtlesim/action/RotateAbsolute
```

which will produce the output

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

As for actions, the output is divided into segments, but this time, as shown in Figure 2.9, there are three segments. The first is the goal (the desired final heading), the second is the result (the actual displacement obtained from the starting heading), and the last is the feedback (the remaining rotation, which should decrease to 0 if the action is making progress). To initiate an action request using `ros2`, we can use the following command (all in one line):

```
ros2 action send_goal  /turtle1/rotate_absolute
                               turtlesim/action/RotateAbsolute '{theta: 0.5}'
```

whereby we use `send_goal` command followed by the action name, the action type, and a YAML string specifying the desired goal. This is essentially the same format used for initiate

a service request. The output will be similar to the following

```
Waiting for an action server to become available...
Sending goal:
theta: 0.5

Goal accepted with ID: d3f27d18f60f45b18e880f6c4d37a36e

Result:
delta: -0.46400007605552673

Goal finished with status: SUCCEEDED
```

and shows that the action server first acknowledges the goal request, and then provides both the result (`delta`) as well as a final status. In this case, the feedback, although provided by the action server, is not displayed. This can be changed by adding the option `-f`.

More examples will be introduced in Chapter 6. As with topics and services, one can rely on predefined actions or define new ones.

2.13 ROS Launch Files

As previously stated, a ROS application consists of multiple interacting nodes. So far, when we needed to run multiple nodes simultaneously, we manually started each one from a separate shell. However, this approach does not scale well as the application grows in complexity and requires multiple nodes to run, each potentially with its own set of parameters. In fact, a typical ROS application runs multiple executables and can easily initiate more than a dozen nodes. To address this issue, ROS uses launch files. A launch file is a script that, when executed by ROS, starts multiple nodes (hence the name). As soon as an application includes more than a couple of nodes, creating a launch file to manage them becomes almost essential. To start a complex application, we create a launch file containing the details of all the nodes to be executed, and then we simply use `ros2` to run the launch file and start all the nodes at once.

In ROS 2, launch files can be written in Python, XML, or YAML[9]. While Python-based launch files offer the highest flexibility and are often preferred for complex applications, they are also more complex to write. For now, we will use the simpler, albeit less flexible, XML format. Python launch files will be introduced in Section 5.9.

Typically, launch files are placed inside packages, but `ros2` can also execute a launch file directly by providing its full path. (This is one of the few cases where a ROS resource can be located outside a package.) To run a launch file inside a package, we use the following syntax, which is similar to running a node:

```
ros2 launch  <packagename> <launchfilename>
```

[9]This is in contrast to ROS 1, where launch files were only written in XML.

If a launch file is located inside a package, it must be placed in a folder called `launch` – more details will be provided in chapter 3. To instead run a launch file using the the full path, we use the following form

```
ros2 launch  <launchfilename>
```

where `launchfilename` is the path to the file. We start by creating a simple launch file that runs two `talker` nodes and one `listener` node, similarly to what we did in section 2.5. The XML code for the launch file is displayed in listing 2.2.

Listing 2.2: Simple launch file

```
1 <launch>
2   <node pkg="demo_nodes_cpp" exec="talker" name="talkerA" />
3   <node pkg="demo_nodes_cpp" exec="talker" name="talkerB" />
4   <node pkg="demo_nodes_cpp" exec="listener"  />
5 </launch>
```

As expected, the contents of the launch file must be enclosed within the `<launch>` section. Each `node` tag specifies a node to be executed. For each node, we provide the package name (`pkg`), the name of the executable (`exec`), and the name to assign to the node (`name`). Notice that each instance of `talker` is assigned a unique name. On the other hand, the `listener` node is not explicitly given a name, so ROS will automatically assign a default one. To execute the launch file we simply type

```
ros2 launch topics.launch.xml
```

where we assumed that the file is called `topics.launch.xml` and that the file is located[10] in the same folder where we executed `ros2`. Once run, the output streams produced by the three nodes all go to the same shell where `ros2` was run. The output will look similar to the following:

```
[talker-1] [INFO] [1694098997.952] [talkerA]: Publishing: 'Hello World: 1'
[talker-2] [INFO] [1694098997.952] [talkerB]: Publishing: 'Hello World: 1'
[listener-3] [INFO] [1694098997.953] [listener]: I heard: [Hello World: 1]
[listener-3] [INFO] [1694098997.953] [listener]: I heard: [Hello World: 1]
[talker-1] [INFO] [1694098998.952] [talkerA]: Publishing: 'Hello World: 2'
[talker-2] [INFO] [1694098998.952] [talkerB]: Publishing: 'Hello World: 2'
[listener-3] [INFO] [1694098998.952] [listener]: I heard: [Hello World: 2]
```

where we can see the different names assigned to the instances of `talker`.

The structure of a launch file can be quite complex, as it may include not only details about the nodes to be launched but also configurations for topic remapping, parameter settings, and other advanced features we have not yet introduced. For a comprehensive

[10] The file `topics.launch.xml` is found in the `unsorted` folder in the MRTP GitHub.

overview, the reader is encouraged to consult the official documentation. Here, we focus on providing the essential information needed to develop simple launch files.

As a further example, the example in listing 2.3 shows how to run both `turtlesim_node` and `turtle_teleop_key`. Recall that `turtle_teleop_key` controls the turtle in the GUI by reading the keys pressed on the keyboard. Therefore, it is convenient to start the two nodes in two separate shells; otherwise, the output of `turtlesim` ends up in the same shell where the input is provided. To separate the two, we start them in two separate terminal windows. This is accomplished[11] using the `launch-prefix` option in the `node` tag when launching `turtle_teleop_key`. This way, `turtlesim` is executed in the same terminal window where `ros2 launch` is run, while a new terminal window is started explicitly to run `turtle_teleop_key`.

Listing 2.3: Turtlesim launch file

```
1 <launch>
2   <node pkg="turtlesim" exec="turtlesim_node" name="turtlesim" />
3   <node pkg="turtlesim" exec="turtle_teleop_key" name="teleop_key"
4     launch-prefix="gnome-terminal --" />
5 </launch>
```

2.14 Interacting with ROS using `rqt`

In this chapter, we have seen how the `ros2` command-line tool can be used to interact with different components of the ROS ecosystem. `ros2`, with its numerous commands, can be used to run nodes, inspect data structures, interact with nodes, and much more. While the command-line interface is very flexible and allows for extensive customization of command execution, it may not always be easy to remember all commands and parameters. To mitigate these difficulties, ROS is distributed with an application called `rqt`, which allows many of these tasks to be performed using a graphical user interface. To start it, simply type

```
rqt
```

Figure 2.10 shows how the application appears if it is run together with the node `turtlesim_node`.

`rqt` is a graphical user interface that provides a collection of tools for visualization, debugging, and introspection of code written in ROS. `rqt` allows the user to monitor and interact with nodes, topics, services, and parameters similarly to what can be done with `ros2`, but in a much more intuitive way (albeit less flexible). `rqt` supports various functionalities such as plotting data, visualizing node graphs, inspecting logs, and managing configurations. `rqt` is built using a plugin architecture that also allows users to define new modules that can be connected to it.

[11] This example assumes you are using the GNOME desktop manager and the associated terminal emulator `gnome-terminal`. If you use a different desktop manager, you will need to change that.

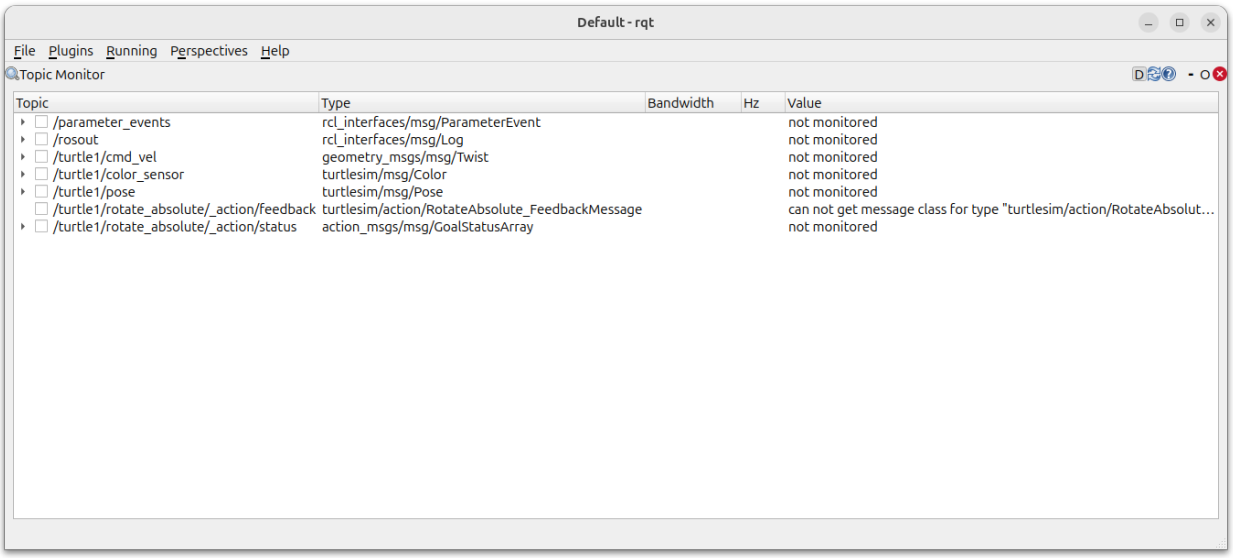


Figure 2.10: `rqt` window showing the topics found when `turtlesim_node` is run.

2.15 Plotting data with `plotjuggler`

When developing and debugging ROS code, it is often necessary to analyze the data coming from sensors or other subsystems. This data can often be thought as a time series and it is often useful to visualize multiple data series at the same time. To ease this task, ROS is includes a tool called `plotjuggler`. `plotjuggler` is a powerful and versatile data visualization tool that allows the user to efficiently plot and analyze time-series data in real-time or from recorded logs. `plotjuggler` uses a drag-and-drop interface, customizable layouts, and advanced filtering and transformation capabilities, allowing the user to manipulate and inspect data with precision. It supports multiple data formats, including ROS topics and bag files. To start it, simply type

```
ros2 run plotjuggler plotjuggler
```

Figure 2.11 shows an example of how `plotjuggler` can be used to visualize how the content of messages sent over a topic changes over time. More precisely, assuming that the `turtlesim_node` node is running and that it is being moved using `turtle_teleop_key`, the plots show how the x and y components of the turtle's position vary over time. This information is obtained from the topic `/turtle1/pose`, which is of type `turtlesim/msg/Pose`.

The reader is referred to the online documentation and to the numerous video tutorials that can be found on the web to learn how to use it.

Further Reading

A general overview of ROS 2 can be found in [33]. The ROS official website `docs.ros.org` also features numerous tutorials about the concepts presented in this chapter. As mentioned

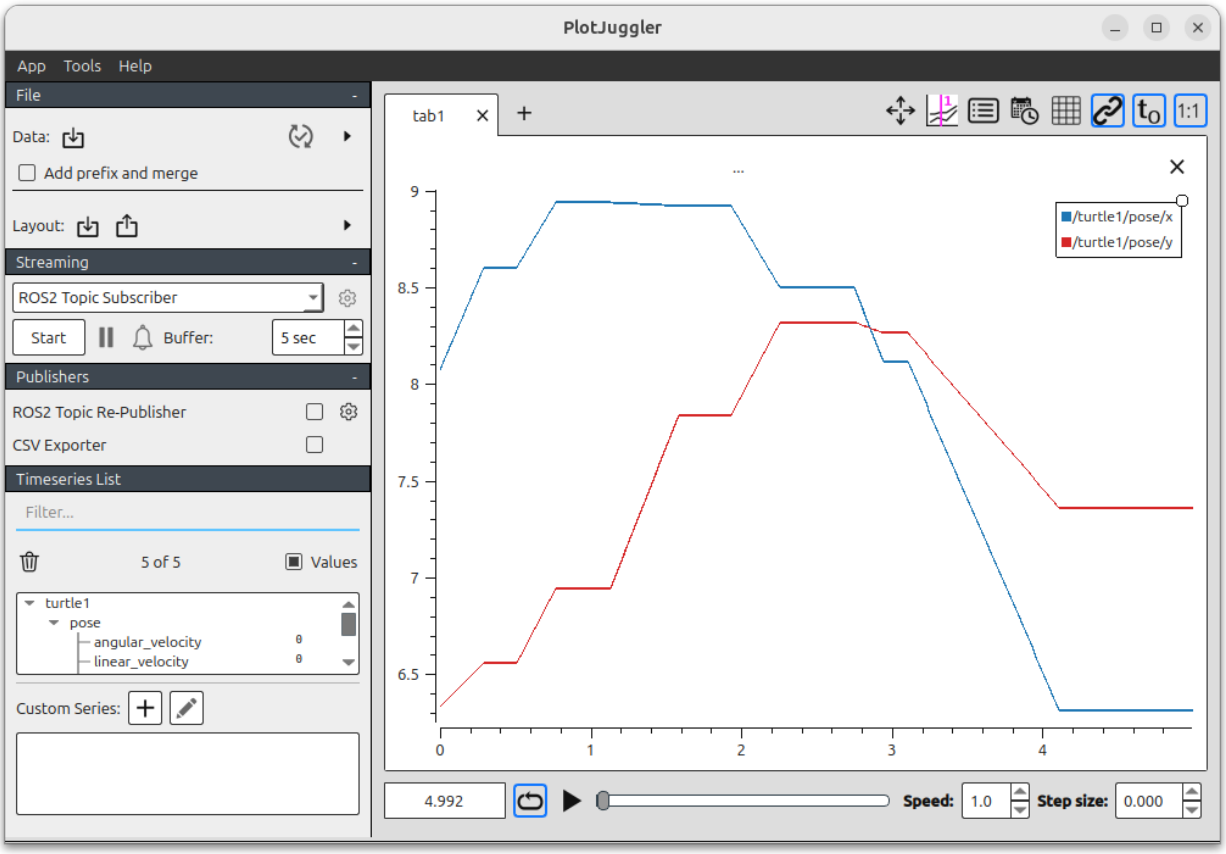


Figure 2.11: `plotjuggler` window showing how the x and y coordinates from the topic `/turtle1/pose` vary over time when the turtle is controlled from the keyboard.

in the foreword, there are numerous books about ROS2, but they often assume preliminary knowledge of robotics fundamentals and do not provide in depth coverage of theoretical concepts. Among these we find [45] and [26].

Chapter 3

Introduction to programming in ROS

3.1 Building a ROS 2 application

A ROS 2 application typically consists of multiple interacting nodes from different packages. Some of these may be part of the standard distribution (underlay), some may be provided by third parties, and others may be written by the developer. Each package, in turn, may include multiple source files. As with other complex software applications, there are often dependencies between packages that must be satisfied before a package can be built or used. When multiple packages and workspaces are iteratively modified during development, it is convenient to use tools to streamline some of these operations without having to manually check all the dependencies.

To this end, ROS 2 relies on a *build tool* and a *build system*. The build system builds a single package and processes all source files inside a package. While ROS 2 supports multiple build systems, in the following, we focus only on `ament_cmake`, an extension of the widely used `CMake` build system that has been explicitly customized for ROS 2. `ament_cmake` is, de facto, the most commonly used build system when developing ROS code in C++. In the following we will only discuss how to create packages using C++. The build tool, on the other hand, operates on packages. It determines the dependencies between packages, and when modifications are made, it calls the build system on the packages that must be rebuilt. This reduces the compilation time and relieves the programmer from having to manually determine which packages to rebuild. The build tool for ROS 2 is called `colcon`. `colcon` stands for *collective construction* and is a sophisticated tool with many advanced features that will not be discussed in this book. The reader is referred to its official website[1] for the complete documentation.

3.1.1 Creating and building a workspace

As stated in Chapter 2, a workspace is a container for packages. The name of the workspace is arbitrary, but its structure must follow a fixed pattern. For example, packages inside a workspace typically go into a subfolder called `src`. The following command creates the basic structure for a workspace called `CSE180`:

[1] `https://colcon.readthedocs.io/en/released/`

```
mkdir -p CSE180/src
```

It consists of a folder called `CSE180` together with its `src` folder, where packages will be located. So far, this is just a plain pair of nested folders. However, we can use `colcon` to turn it into a workspace by adding the necessary additional folders and files. To this end, we simply execute the following command:

```
colcon build
```

from the root of the workspace, i.e., from `CSE180`. The command will print that it built 0 packages, but most importantly, it will create some new folders in the workspace, namely `build`, `install`, and `log`. In the `install` folder, we now find the scripts to add this new workspace as an overlay. This can be done running the command[2]

```
. install/local_setup.bash
```

from the `CSE180` folder. Now, the contents of the workspace are visible and can be used by `ros2`.

Remark 3.1. *According to the ROS 2 official documentation, it is important to source the overlay from a shell different from the one where it was built. Failure to do so may result in errors. Therefore, before executing* `. install/local_setup.bash`*, open a new shell, move into the workspace, and run it from there. Accordingly, from now onward, it will be tacitly assumed that this operation is always performed from a new shell.*

Remark 3.2. *Forgetting to make the overlay visible to ROS is a very common error. Just building your packages is not sufficient to use them. If ROS complains that it cannot find the packages you built, in all probability, you have not run the above command from the shell where ROS is executed. Importantly, the overlay must be made visible from each shell from which you want to use its resources.*

In the following, for simplicity, we will always suggest running `colcon build`. This command will determine all packages inside the workspace and possibly rebuild all of them, though it will avoid recompiling code that has not changed since the last build. However, if one wants to build a single package or a subset of packages, the option `--packages-select` can be used. For example, to build just a hypothetical package called `first` in the current workspace, one would run

```
colcon build --packages-select first
```

This command would build only the package `first` while ignoring all other packages in the workspace. If you want to build multiple packages, you can list all their names separated by spaces.

[2]Note that there is a space between the dot `.` and `install`. The reason to use this syntax rather than `source` is that it is more portable.

3.2 Adding a package

A package is a collection of ROS 2 resources, such as source code, launch files, data files, and more. To automate the process of locating these files, compiling, etc., the structure of a package follows a prescribed format and must include some specific files. Packages can either be obtained from third parties or created using the command `ros2 pkg`. The following command, executed from the `src` folder in the workspace, will create a package called `first`:

```
ros2 pkg create first
```

The output will be similar to the following:

```
going to create a new package
package name: first
destination directory: /home/shamano/CSE180/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['username <username@maildomain.com>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
creating folder ./first
creating ./first/package.xml
creating source and include folder
creating folder ./first/src
creating folder ./first/include/first
creating ./first/CMakeLists.txt

[WARNING]: Unknown license 'TODO: License declaration'.
This has been set in the package.xml, but no LICENSE file has been created.
It is recommended to use one of the ament license identifiers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

Inspecting the output and the filesystem, we can see that the command created some folders and files. More specifically, the folder `first` was created, and inside it, the empty source and include folders were also created (`first/src` and `first/include/first`, respectively). As you may anticipate, the source code for the package will go in `first/src`, while include files will be placed in `first/include/first`. The files `package.xml` and `CMakeLists.txt`

were created as well and will be discussed shortly. The command also produced a warning because no license was specified. This can be done later by editing the file `package.xml`, but it is also possible to specify the license when the package is created. This can be done as follows by specifying one of the license identifiers given in the warning message:

```
ros2 pkg create first --license MIT
```

In this case, the warning message disappears, and an additional file called `LICENSE` is created with the associated license terms.

If we now run `colcon build` again from the workspace folder `CSE180`, we will see that one package was built (albeit empty). To delete a package, it is sufficient to remove its folder from the `src` folder. `ros2 pkg` can also be executed with additional parameters to not only create an empty package but also generate template sources for a node. This is done by running the following command[3]:

```
ros2 pkg create --build-type ament_cmake --node-name firstnode first
```

This will create not only a package called `first` but also the source for a template executable called `firstnode`. The additional parameter `--build-type ament_cmake` specifies that `ament_cmake` is the desired build system. The output of this command is similar to the previous one, but the last few lines are as follows (we omit the warning for brevity):

```
build type: ament_cmake
dependencies: []
node_name: firstnode
creating folder ./first
creating ./first/package.xml
creating source and include folder
creating folder ./first/src
creating folder ./first/include/first
creating ./first/CMakeLists.txt
creating ./first/src/firstnode.cpp
```

and show that now a source file `firstnode.cpp` has been created as well. At this point, the workspace can be rebuilt again with `colcon build`. After sourcing `install/setup.bash`, `ros2 run` can be used to run the node `firstnode` from the package `first`. To do so, we must specify the package name and the node name, as we did in Chapter 2

```
ros2 run first firstnode
```

Besides being created from scratch, packages can also be downloaded from the web, either as archives or via version control systems such as `git`. Either way, packages must be placed inside the `src` folder in the workspace. For example, one can download the con-

[3]Before running it, it is necessary to remove the previously created package because every new package must have a unique name in the workspace.

tents of the code associated with the book from the MRTP GitHub and place the folder `MRTP/src/talklisten` in the `CSE180/src` folder. That folder includes the source code for a package that can now be built by running `colcon build` from `CSE180`.

Figure 3.1 summarizes the structure obtained after the commands in the previous section have been executed. Note that additional files and folders are located inside `build`, `install`, and `log`, but they are not shown in the figure. For each package inside `src`, a similar structure is repeated.

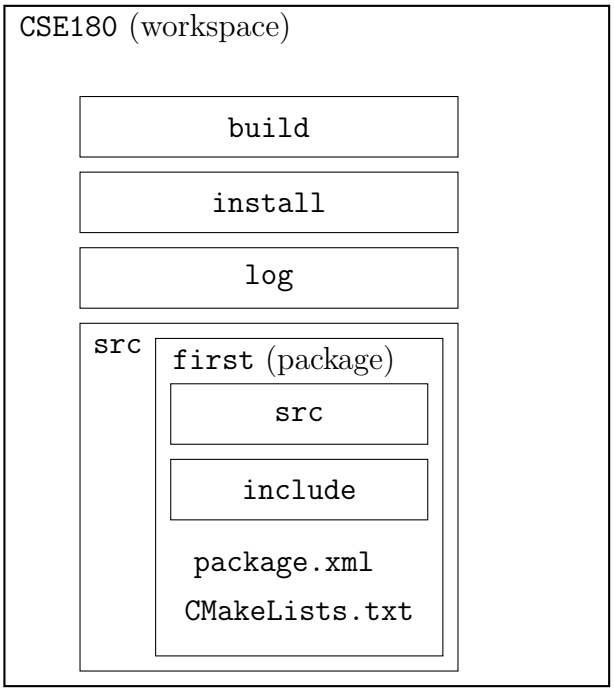


Figure 3.1: Internal structure for workspace and package `first`, with rectangles indicating folders.

3.2.1 `package.xml`: the manifest file

`package.xml` is an XML file providing the so-called package manifest. All packages have a manifest file, including information about licensing and dependencies on other packages. When a package is added with `ros2 pkg create`, a basic `package.xml` file is added to the package folder so that the programmer can customize it for the package being created. Each package has its own manifest file, so if you create multiple packages, you must correspondingly edit each manifest file for each package. The initial skeleton for the manifest file includes some meta-information about the package (author, license, version, etc.), and in addition, it can specify package dependencies, i.e., modules needed to build the package, execute the package, and more. The following listing shows the manifest file that was generated when creating the package `first`.

Listing 3.1: Default package file

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

```
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>first</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="username@todo.todo">username</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

The meaning of `version`, `description`, `maintainer`, and `license` is evident and can be filled with strings or left to their default values. The second part of the file is used to add dependency tags that must be included to specify the packages that `first` depends on. Examples of how to add dependencies to `package.xml` will be given later. Alternatively, dependencies can also be specified when running `ros2 pkg create`, though this approach will not be explained here. More details can be found on the official website or by running the command with the option `-h`.

Remark 3.3. *If you forget to add a dependency to the manifest file, but correctly edit the* ***`CMakeLists.txt`*** *file described next, your package can still be correctly built locally. However, your package will not be properly handled when using package management tools (e.g.,* ***`rosdep`****) that rely on the information included in the manifest file.*

3.2.2 `CMakeLists.txt`

Another file created by `ros2 pkg create` is `CMakeLists.txt`. `ament_make` is a frontend to the `CMake` development tool, and therefore `CMakeLists.txt` follows the syntax specified by `CMake`. However, it also includes some additional macros specific to ROS. The structure of this file can be quite complex, and since every time we add a node to the package we have to add additional lines to indicate how to build it, its size can quickly grow.

A complete description of this file is beyond the scope of these notes, and the reader is referred to the ROS website for a comprehensive discussion. As with the manifest file, `ros2 pkg create` generates a template that can then be customized and is sufficient for most users. In a subsequent section, where we create the first nodes in C++, we will discuss a set of minimal changes that can be made to customize the file so that `colcon` can correctly determine how to build the package.

3.3 Creating ROS Nodes

So far, we have just executed nodes shipped with the ROS distribution, but we now have all the elements to create the first ROS nodes from scratch. Before embarking on writing code, some high-level software considerations are in order. To tap into its overall software infrastructure, ROS provides the "ROS Client Library," called `rcl`. `rcl` is a low-level library written in C that supports the implementation of ROS client libraries in other languages by providing access to ROS concepts like nodes, topics, services, etc. Client libraries for C++ and Python are officially supported by the ROS project, but others have been developed by third parties as well. `rclcpp` is the ROS Client Library for C++ and is found in a package called `rclcpp`. Similarly, `rclpy` is the ROS Client Library for Python and is found in a package called `rclpy`. Being built on top of the same foundation, the two libraries offer similar functionalities and mostly differ in how the API is exposed in the specific language. In the following, we will exclusively write code using C++ and `rclcpp`.

As previously stated, a ROS program consists of a collection of interacting nodes exchanging information through topics, service calls, and action calls. Central to this programming approach is the presence of topics through which messages are asynchronously exchanged. Accordingly, a ROS node typically subscribes to one or more topics to receive data and publishes to one or more topics. A ROS program implementing a node usually includes the following steps:

- initializing the ROS system;
- instantiating a node and setting up communication with topics (plus services and actions, if needed);
- carrying out the necessary computation, often by exchanging messages via topics;
- shutting down the ROS system.

The above pattern is sufficient to characterize most of the code presented in the following, but, of course, more complex applications may deviate from this breakdown and/or implement it in sophisticated ways. An approach common to many ROS programs is to organize the node computation through *callbacks*. This means that functions are associated with certain events, and when the event occurs, the specific function is called (hence the name callback). For example:

- when a node subscribes to a topic, a callback function is registered with the topic, and every time a message is received through the topic, the associated callback function is called and the message is passed as a parameter for further processing;
- a program that must execute certain operations at a regular frequency (e.g., querying a sensor) may create a timer with the desired rate and associate a callback function with the timer. Every time the timer ticks, the associated callback function is executed.

One final concept central to ROS programs is *spinning*. Spinning a node means letting the communication subsystem take care of all pending events that trigger a callback function. Consequently, ROS programs subscribing to topics typically include calls to one of the various

spin functions provided by `rclcpp`. An additional aspect worth mentioning is that when writing ROS code, it is necessary to keep track of the node dependencies, i.e., the packages that the node depends on, because these need to be explicitly listed in `package.xml` and `CMakeLists.txt`. Failure to do so will result in build errors, even if the code is syntactically correct.

3.4 The first ROS nodes

In this section, we provide the full implementation of the first two ROS nodes that exchange string messages through a topic. These nodes essentially replicate the functionality of the `chatter` and `talker` nodes we ran in Section 2.5. This is a classic example, also provided in the tutorials section on the ROS website,[4] though the version we present here is even simpler.

The code examples we develop will be placed in various packages inside a new workspace called `MRTP`. In particular, these first two nodes will be part of a package called `talklisten`. The following commands create the workspace and the package.[5]

```
cd ~
mkdir -p MRTP/src
cd MRTP/src
ros2 pkg create --build-type ament_cmake talklisten
```

Listing 3.2 shows the code for the `talker` node. The file, called `talker.cpp`, should be placed in the `src` folder inside the package folder `talklisten`.

Listing 3.2: Talker Node

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/string.hpp> // needed to publish strings
3
4  int main(int argc,char **argv) {
5
6    rclcpp::init(argc,argv); // initialize the ROS subsystem
7
8    rclcpp::Node::SharedPtr nodeh;
9    rclcpp::Publisher<example_interfaces::msg::String>::SharedPtr pub;
10   rclcpp::Rate rate(1); // create rate object
11
12   nodeh = rclcpp::Node::make_shared("talker"); // create node instance
13   // create publisher to topic "message" of strings
14   pub = nodeh->create_publisher<example_interfaces::msg::String>("message",1);
15
16   int counter = 0;
17   while ( (counter++ < 100) && (rclcpp::ok())  ) {
```

[4] https://docs.ros.org/en/jazzy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html

[5] All code examples presented from now onward can be downloaded from https://github.com/stefanocarpin/MRTP.

```
18      RCLCPP_INFO(nodeh->get_logger(),"Sending message #%d",counter);
19      example_interfaces::msg::String stringtosend;
20      // prepare message to send
21      stringtosend.data = "Message #" + std::to_string(counter);
22      pub->publish(stringtosend); // publish message
23      rate.sleep(); // wait
24    }
25    rclcpp::shutdown(); // shutdown ROS
26    return 0;
27  }
```

As discussed in the previous section, the main function initializes the ROS system by calling the function `rclcpp::init()` at the beginning and shuts down ROS by calling the function `rclcpp::shutdown()` at the end. Next, we set up a node and a publisher to send strings. Nodes are created by instantiating `rclcpp::Node` via the function `rclcpp::Node::make_shared`. The function takes as a parameter the name to assign to the node (*talker* in this case) and returns a pointer to the instance. The node is accessed via a *handler* stored in the variable `nodeh` (node handler). The method `create_publisher` is a member of the class `rclcpp::Node` and returns a pointer to a publisher object, `pub`, which allows the node to send messages of type `example_interfaces::msg::String`. This message type serves as a wrapper for transmitting the classic elementary C++ string data type over a topic. The first parameter specifies the name of the topic (`message`), while the second parameter is an integer that specifies the depth of the message queue associated with the topic (1 in this case). If a topic with the specified name (`message`) already exists, the node will associate with it. This is consistent with the fact that multiple nodes may publish to the same topic. Otherwise, if no topic called `message` exists, a new one will be created. Note that the type of messages exchanged through the topic must be specified using one of the ROS message types (either one of the default ones or one created by the programmer). In this case, to send a string, we use `example_interfaces::msg::String`, which is specified as the type in the call to the template method `create_publisher`. More details about basic message types will be given in a later section. The specific structure of this message can be discovered using

```
ros2 interface show example_interfaces/msg/String
```

which produces the following output (comments omitted for brevity):

```
string data
```

Observe how the output produced by `ros2 interface show` matches the syntax we use in the main loop to store the string in the instance of the message we want to publish through the topic. The object `rate` is an instance of `rclcpp::Rate`, enabling the node to wait (sleep) for a certain amount of time. More precisely, the passed parameter specifies the desired frequency (1 Hz in this case), and when the method `sleep` is called, the program suspends its operation for an amount of time sufficient to match the desired frequency, as opposed to waiting for a fixed amount of time. After the initialization phase is completed, the rest of the code is rather straightforward. The while loop iterates 100 times or until the

ROS system receives a signal to terminate (e.g., if the user presses CTRL-C, `rclcpp::ok()` will return false and terminate the loop). Inside the loop, an instance of the message is declared and initialized. Note how the message to send is stored in the `data` field. To send the message, the method `publish` of the publisher object is used. At the end of the loop, the node sleeps for an amount of time sufficient to match the 1 Hz frequency, and the loop restarts. When the loop ends, ROS is shut down, and the program terminates. Inside the main loop, the function `RCLCPP_INFO` is used to print some diagnostic information to the screen. These functions will be discussed in detail later, but for the time being, it is sufficient to note that they take as parameters a logger object and the message to print (note the syntax similar to the `printf` function to format the message). The logger object is retrieved from the node handler using the function `get_logger`, as different nodes may have different loggers.

Finally, note that at the beginning, we included two header files located in two folders called `rclcpp` and `example_interfaces`. These indicate two packages this node depends on, namely `rclcpp` and `example_interfaces`. This is an important detail to consider because we will later need to indicate these dependencies in `package.xml` and `CMakeLists.txt`.

Remark 3.4. *Even though this node does little, its syntax may seem a bit intimidating. However, most nodes we will encounter in the following sections will be based on this or similar templates. The main differences will lie in the actual work performed in the main loop, while the initialization steps will always be very similar and can be easily copied and adapted. After a while, this process will become quite automatic.*

At this point, we can consider the listener node that will subscribe to the `message` topic and print to the screen whatever it receives. The source code is provided in Listing 3.3.

Listing 3.3: Listener Node

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/string.hpp> // needed to receive strings
3
4  rclcpp::Node::SharedPtr nodeh;
5
6  // callback function called every time a message is received from the
7  // topic "message"
8  void callback(const example_interfaces::msg::String::SharedPtr msg) {
9    // process the message: just print it to the screen
10   RCLCPP_INFO(nodeh->get_logger(),"Received: '%s'",msg->data.c_str());
11 }
12
13 int main(int argc,char **argv) {
14
15   rclcpp::init(argc,argv); // initialize ROS subsystem
16   rclcpp::Subscription<example_interfaces::msg::String>::SharedPtr sub;
17   nodeh = rclcpp::Node::make_shared("listener"); // create node instance
18   // subscribe to topic "message" and register the callback function
19   sub = nodeh->create_subscription<example_interfaces::msg::String>
20                                                ("message",10,&callback);
21   rclcpp::spin(nodeh);  // wait for messages and process them
22   rclcpp::shutdown();
23   return 0;
```

```
24
25 }
```

Most of the setup steps provided in the `main` function of the `listener` node mirror what we saw in the `talker` node, with the exception that in this case we create an instance of a `Subscription` object instead. The method `create_subscription` takes three parameters: the name of the topic (`message`), the length of the incoming queue (10), and a pointer to the callback function. As previously stated, the callback function is the function that will be called every time a message is received through this topic. After the node and the subscriber are created, the function `rclcpp::spin` is called to spin the node `nodeh`. When `spin` is called, the program enters an infinite loop during which it continuously checks for events that trigger callback functions. If such an event occurs, the corresponding callback function is executed. Since this loop never terminates, `spin` is a non-returning function. In this specific example, the only event that triggers a callback function is an incoming message on the subscribed topic. The callback function `callback` accepts as input an instance of the received message `msg` and prints its data to the screen using `RCLCPP_INFO`. Note that the callback function always receives as input just an instance of the message and no other parameters. This is why we store the `nodeh` pointer as a global variable— so that the logger can be retrieved from the function (since it cannot be passed as a parameter). Generally speaking, storing the handle as a global variable is not good programming practice, but this is done here for simplicity. Later on, we will show how this can be avoided.[6]

Remark 3.5. *When writing ROS programs, it is customary to use the fully qualified name when using ROS objects (e.g.,* ***`rclcpp::Node::SharedPtr`****) and messages. While it is legitimate to use the C++ directive* ***`using namespace`*** *at the beginning to avoid having to specify the namespace explicitly, this is usually not done when referring to ROS-provided components. However, this is a convention rather than a strict requirement.*

At this point, before compiling the source code of the two nodes, it is necessary to update `package.xml` and `CMakeLists.txt` to indicate the dependencies and specify the files we want to compile. Listing 3.4 shows the file manifest obtained by slightly modifying the template generated when the package was created.

Listing 3.4: Package file for the `talklisten` package

```
1  <?xml version="1.0"?>
2  <?xml-model href="http://download.ros.org/schema/package_format3.xsd"
3  schematypens="http://www.w3.org/2001/XMLSchema"?>
4  <package format="3">
5    <name>talklisten</name>
6    <version>0.0.2</version>
7    <description>Talker/Listener nodes</description>
8    <maintainer email="scarpin@ucmerced.edu">Stefano Carpin</maintainer>
9    <license>Apache License 2.0</license>
10
11   <buildtool_depend>ament_cmake</buildtool_depend>
12   <depend>rclcpp</depend>
```

[6] One could also use the function `rclcpp::get_logger` to create a standalone instance of the class `rclcpp::Logger`, but in such a case, the logger would not be associated with the node.

```
13   <depend>example_interfaces</depend>
14
15   <test_depend>ament_lint_auto</test_depend>
16   <test_depend>ament_lint_common</test_depend>
17
18   <export>
19     <build_type>ament_cmake</build_type>
20   </export>
21 </package>
```

Besides authorship and other information (which could also be left at their default values), the key change is adding the `depend` tags for the `rclcpp` and `example_interfaces` packages. These two tags should be placed immediately after the `buildtool_depend` tag and indicate that our package depends on them.

Similarly, we need to update the default `CMakeLists.txt` file generated when the package was created. The file is shown in listing 3.5, where for brevity we have removed the sections that are irrelevant to the project (although they could be left in as well).

Listing 3.5: CMake file for the `talklisten` package

```
1  cmake_minimum_required(VERSION 3.5)
2  project(talklisten)
3
4  # Default to C++14
5  if(NOT CMAKE_CXX_STANDARD)
6    set(CMAKE_CXX_STANDARD 14)
7  endif()
8
9  if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
10   add_compile_options(-Wall -Wextra -Wpedantic)
11 endif()
12
13 find_package(ament_cmake REQUIRED)
14 find_package(rclcpp REQUIRED)
15 find_package(example_interfaces REQUIRED)
16
17 add_executable(talker src/talker.cpp)
18 add_executable(listener src/listener.cpp)
19 ament_target_dependencies(talker rclcpp example_interfaces)
20 ament_target_dependencies(listener rclcpp example_interfaces)
21
22 install(TARGETS  talker DESTINATION lib/${PROJECT_NAME})
23 install(TARGETS listener DESTINATION lib/${PROJECT_NAME})
24
25 ament_package()
```

The key additions are the following:

- Specify the required packages using the `find_package` macro. These should be added immediately after the standard line `find_package(ament_cmake REQUIRED)`.
- Use the macro `add_executable` to specify the names of the executables to create and the source files needed to generate those executables. Note that since the package includes two executables, we add two of these macros.

- Use the macro `ament_target_dependencies` to specify the package dependencies for the executables.
- Use the macro `install` to specify where the executables should be installed.

As for `package.xml`, the listing we provided can be used as a template to start with. More complex features can be added, and these are documented on the official website. At this point, it is possible to build the executables in the package. To do so, we can move to the workspace folder `MRTP` and issue the command `colcon build`. If no errors are found, the output will look like the following

```
Starting >>> talklisten
Finished <<< talklisten [5.63s]

Summary: 1 package finished [6.04s]
```

confirming that the package has been built. Now, we can open two new shells, source the overlay, and run the new nodes using `ros2 run`. After sourcing the overlay, the nodes we just created are found and executed using the same syntax we used for the standard nodes shipped with ROS and part of the underlay. Specifically, we can run

```
ros2 run talklisten talker
```

to run the `talker` node and

```
ros2 run talklisten talker
```

to run the `listener` node. The output will look like the following for `talker`

```
[INFO] [1672482702.756594287] [talker]: Sending message #1
[INFO] [1672482703.744354838] [talker]: Sending message #2
[INFO] [1672482704.744626257] [talker]: Sending message #3
[INFO] [1672482705.743791683] [talker]: Sending message #4
[INFO] [1672482706.743500771] [talker]: Sending message #5
[INFO] [1672482707.745124286] [talker]: Sending message #6
[INFO] [1672482708.744923859] [talker]: Sending message #7
[INFO] [1672482709.743992184] [talker]: Sending message #8
[INFO] [1672482710.743906316] [talker]: Sending message #9
[INFO] [1672482711.744225130] [talker]: Sending message #10
[INFO] [1672482712.744533815] [talker]: Sending message #11
```

and

```
[INFO] [1672482706.745114892] [listener]: Received: Message # 5
[INFO] [1672482707.746839561] [listener]: Received: Message # 6
[INFO] [1672482708.746548545] [listener]: Received: Message # 7
[INFO] [1672482709.745019691] [listener]: Received: Message # 8
```

```
[INFO] [1672482710.744835135] [listener]: Received: Message # 9
[INFO] [1672482711.744877524] [listener]: Received: Message # 10
[INFO] [1672482712.745475959] [listener]: Received: Message # 11
```

for `listener`. At this point, it would be a good idea to add a launch file to run both nodes at once. This could be either a standalone file to be executed as we saw in Section 2.13, or a launch file integrated into the package. A standalone file would be similar to the one we provided in Section 2.13 and could be run with `ros2 launch filename.launch.xml`. To integrate the launch file into the package and run it with the command `ros2 launch packagename filename.launch.xml`, further changes to `package.xml` and `CMakeLists.txt` are necessary. This process will be illustrated in the next section.

3.5 More ROS examples

In this section, we develop a new package called `multipletopics`, where we include two nodes similar to `talker` and `listener` that publish and subscribe to two separate topics of different types (strings and integers). Moreover, we will develop a launch file and show how it can be integrated into the package to be launched with `ros2 launch`. We start by creating the package inside the `MRTP` workspace.

```
cd ~MRTP/src
ros2 pkg create --build-type ament_cmake multipletopics
```

Listing 3.6 shows the code for the node `multipublish` that publishes both strings and integers.

Listing 3.6: Node publishing to multiple topics

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/int32.hpp> // to publish integers
3  #include <example_interfaces/msg/string.hpp> // to publish strings
4
5  int main(int argc,char **argv) {
6
7    rclcpp::init(argc,argv); // initialize the ROS subsystem
8
9    rclcpp::Node::SharedPtr nodeh;
10   rclcpp::Publisher<example_interfaces::msg::String>::SharedPtr pubs;
11   rclcpp::Publisher<example_interfaces::msg::Int32>::SharedPtr pubi;
12   rclcpp::Rate rate(2);
13
14   nodeh = rclcpp::Node::make_shared("multipublish"); // create node
15   // create publisher to topic "strigm" of strings
16   pubs = nodeh->create_publisher<example_interfaces::msg::String>("stringm",1);
17     // create publisher to topic "intm" of integers
18   pubi = nodeh->create_publisher<example_interfaces::msg::Int32>("intm",1);
19
20   int value=0;
21   example_interfaces::msg::Int32 intToSend; // integer message to send
22   example_interfaces::msg::String stringToSend; // string message to send
```

```
23    stringToSend.data = "CSE180-Robotics"; // constant string to send
24
25    while (rclcpp::ok()) {
26      intToSend.data = value++; // update message to send
27      pubi->publish(intToSend); // publish the integer message
28      pubs->publish(stringToSend); // publish the string message
29      RCLCPP_INFO(nodeh->get_logger(),"Completed iteration #%d",value);
30      rate.sleep(); // wait
31    }
32    rclcpp::shutdown(); // unreachable in the current form
33    return 0;
34
35  }
```

The logic is similar to the one in listing 3.2, with just a few minor changes. First, since we want to publish messages of two different types, the node creates two separate publisher objects for integers and strings, called `pubi` and `pubs`, respectively. The topics are given the names `stringm` and `intm`. Note that to publish integers, we now include the file `example_interfaces/msg/int32.hpp` that defines the message to transmit an integer represented on 32 bits. Its structure can be shown with

```
ros2 interface show example_interfaces/msg/Int32
```

which produces the following output

```
int32 data
```

As in the previous example, observe how the output of `ros2 interface show` directly translates to how, in each cycle, we update the integer to send. The other minor change is that the `rate` object now targets a frequency of 2 Hz. The rest is pretty much the same, although the main loop in this case will run indefinitely and not just for a fixed number of iterations (the iteration can be interrupted with CTRL-C).

Listing 3.7 shows instead the code for the node `multisub` that subscribes to the two topics created by the `multipublish`.

Listing 3.7: Node subscribing to multiple topics

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/int32.hpp> // to receive integers
3  #include <example_interfaces/msg/string.hpp> // to receive strings
4
5  rclcpp::Node::SharedPtr nodeh;
6
7  // callback function called every time a message is received from the
8  // topic "stringm"
9  void stringCallback(const example_interfaces::msg::String::SharedPtr msg) {
10   // print received string to the screen
11   RCLCPP_INFO(nodeh->get_logger(),"Received string: %s",msg->data.c_str());
12 }
13
14 // callback function called every time a message is received from the
```

```
15 // topic "intm"
16 void intCallback(const example_interfaces::msg::Int32::SharedPtr msg) {
17   // print received integer to the screen
18   RCLCPP_INFO(nodeh->get_logger(),"Received integer: %d",msg->data);
19 }
20
21
22 int main(int argc,char **argv) {
23
24   rclcpp::init(argc,argv); // initialize ROS subsystem
25   rclcpp::Subscription<example_interfaces::msg::String>::SharedPtr subs;
26   rclcpp::Subscription<example_interfaces::msg::Int32>::SharedPtr subi;
27   nodeh = rclcpp::Node::make_shared("multisub"); // create node instance
28   // subscribe to topic "stringm" an register the callback function
29   subs = nodeh->create_subscription<example_interfaces::msg::String>
30                                          ("stringm",10,&stringCallback);
31     // subscribe to topic "intm" an register the callback function
32   subi = nodeh->create_subscription<example_interfaces::msg::Int32>
33                                          ("intm",10,&intCallback);
34   rclcpp::spin(nodeh); // wait for messages and process them
35
36   rclcpp::shutdown();
37   return 0;
38
39 }
```

For this program, too, the logic is similar to the corresponding program presented in listing 3.3. In this node, we subscribe to the two topics `stringm` and `intm`, and we correspondingly register two separate callbacks handling the two streams of messages. As in the previous example, the program calls `rclcpp::spin` and remains inside there, delegating to `rclcpp` the handling of incoming messages and the callbacks. `rclcpp::spin` will call the callback functions, and when running the example, we may observe that even though the publisher alternates sending messages to `intm` and `stringm`, on the subscriber side the callbacks are not necessarily called with the same alternating pattern. This is consistent with the overall communication ROS infrastructure, whereby the handling of communication across different topics is asynchronous, and one should not make assumptions about the relative timing of messages exchanged across different topics. However, the order of messages sent on the same topic is preserved, as we have seen in the previous example.

For this example, we now create a launch file `multipletl.launch.xml` to be integrated into the package. To this end, we create a folder called `launch` inside the package folder `multipletopics`, and we place the following launch file there, similar to the one we saw earlier in listing 2.3. The contents of the file are exactly the same, except for the package and executable names and some formatting to keep it more compact.

Listing 3.8: Launch file `multipletl.launch.xml`

```
1 <launch>
2   <node pkg="multipletopics" exec="multipublish" name="multipublish"
3         launch-prefix="gnome-terminal --" />
4   <node pkg="multipletopics" exec="multisub" name="multisub" />
5 </launch>
```

Next, we need to update `package.xml` and `CMakeLists.txt`. For the nodes, the changes are similar to those presented in the previous section, but for the launch file, we need to inform `colcon` that we are now including a launch file as well. Listing 3.9 shows the manifest file. The only new element is the tag `exec_depend`, which specifies that this package depends on `ros2launch` because we are integrating a launch file into the package.

Listing 3.9: Manifest file for the `multipletopics` package

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>multipletopics</name>
  <version>0.0.1</version>
  <description>multisub/multipublish nodes</description>
  <maintainer email="scarpin@ucmerced.edu">Stefano Carpin</maintainer>
  <license>Apache License 2.0</license>
  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>
  <depend>example_interfaces</depend>

  <exec_depend>ros2launch</exec_depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Finally, in listing 3.10, we show the `CMakeLists.txt` file for the `multipletopics` package. The only new element is the last macro, `install`, which indicates that the package includes a folder called `launch` that should be installed when the package is built.

Listing 3.10: CMake file for the `multipletopics` package

```
cmake_minimum_required(VERSION 3.5)
project(multipletopics)
# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)
add_executable(multipublish src/multipublish.cpp)
add_executable(multisub src/multisub.cpp)
ament_target_dependencies(multipublish rclcpp example_interfaces)
ament_target_dependencies(multisub rclcpp example_interfaces)
```

```
19
20 install(TARGETS  multipublish DESTINATION lib/${PROJECT_NAME})
21 install(TARGETS multisub DESTINATION lib/${PROJECT_NAME})
22 install(DIRECTORY launch DESTINATION share/${PROJECT_NAME})
23
24 ament_package()
```

It is now possible to build the package using `colcon build`, and after having sourced the overlay the launch file can be run as follows

```
ros2 launch multipletopics multipletl.launch.xml
```

where we now specify the name of the package and the name of the launch file, and not just the path to the launch file.

Troubleshooting

What if the code compiles correctly, but when the executables are run, nothing is printed to the screen? A very common error is using different names for the topics when advertising and subscribing. Since the matching is based on the topic name, a typo in either node will prevent the communication channel from being established. Evidently, this error cannot be caught at compile time. A quick way to verify whether a topic has been correctly connected between the publisher and subscriber is by executing `rqt_graph`. In this case, the output would be similar to Figure 3.2, showing the expected connection between the publisher and subscriber.

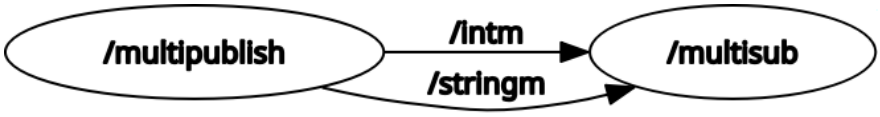


Figure 3.2: ROS graph obtained with `rqt_graph` when the previous example is run. The graph confirms that publisher and subscriber are connected to the correct topics.

A different way, and perhaps less immediate, to get the same information is by running `ros2 node info` to inspect either node. For example, running

```
ros2 node info /multisub
```

will produce the following output (long lines wrapped for clarity):

```
/multisub
  Subscribers:
    /intm: example_interfaces/msg/Int32
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /stringm: example_interfaces/msg/String
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
```

```
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /multisub/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /multisub/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /multisub/get_parameters: rcl_interfaces/srv/GetParameters
    /multisub/get_type_description: type_description_interfaces/srv/
                                                          GetTypeDescription
    /multisub/list_parameters: rcl_interfaces/srv/ListParameters
    /multisub/set_parameters: rcl_interfaces/srv/SetParameters
    /multisub/set_parameters_atomically: rcl_interfaces/srv/
                                                      SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

confirming that the node is correctly subscribed to the two topic `intm` and `stringm`. Note that the topic is also subscribed to another topic we did not create (`parameter_events`). Its nature will be discussed in a later section.

Finally, another useful tool for debugging is running `ros2 topic` with the option `echo`. This allows verifying whether information is being published to a topic or not. In the previous example, if `multipublish` is running and we type

```
ros2 topic echo /intm
```

we will get an output like the following:

```
data: 22
---
data: 23
---
data: 24
---
data: 25
---
data: 26
---
data: 27
---
data: 28
---
data: 29
```

This way it is easy to verify whether a publisher is correctly working even before writing the corresponding subscriber.

ROS Streams

ROS defines its own streams to output textual information from a node. When using `rclcpp`, these are exposed through a set of macros similar to the macro `RCLCPP_INFO` we have used in the previous examples. `rclcpp` provides five different macros that differ in verbosity level (amount of information printed and color) and in where the information is sent by default (although this is configurable). The five basic macros and their intended use are as follows:

RCLCPP_INFO : Used for printing information to the user.

RCLCPP_DEBUG : Used for debugging purposes.

RCLCPP_WARN : Diagnostic about unusual conditions (warnings).

RCLCPP_ERROR : Diagnostic about recoverable errors.

RCLCPP_FATAL : Diagnostic about unrecoverable errors.

Each of the macros receives as input an instance of `rclcpp::Logger`, which is obtained from an instance of `rclcpp::Node`, as we have seen in previous examples. After the logger, the macros accept a format string followed by a variable number of parameters, using the same formatting as the `printf` function. The above classification regarding the intended use of the macros is meant to ease interpretation, but it is up to the programmer to decide which stream to use. For example, whether something should be sent to the *WARN* or the *DEBUG* log is often subjective, and both options are accepted— i.e., ROS does not strictly enforce the above guidelines. The package `rclcpp` also provides more sophisticated versions with conditional printing, throttled output, and many other features. The reader is referred to the online documentation for these advanced features. Finally, note that it is generally not a good idea to use both these macros and the standard C++ `iostream` library, and in fact most ROS developers just stick to the ROS streams just described.

Publishing to a Topic

In listings 3.2 and 3.6, we have seen that a node declares (*advertises*) that it will publish to a certain topic by using the `create_publisher` method of the `rclcpp::Node` class. The simplest form, used in the examples, requires specifying just the name of the topic and the length of the queue (in addition to the type of the message through the template). However, topics can be configured to exhibit more sophisticated behaviors by using a quality of service (QoS) profile. With such a profile, it is, for example, possible to create a *latched* topic, i.e., a topic where the last message sent is saved and passed to all subscribers that subscribe to the topic even after the message has been published (recall that normally, subscribers receive only messages sent after they connect to a topic). Latched topics are particularly useful when sending large messages, such as maps. The reader is referred to the official documentation for a complete discussion of the many options available when setting up a topic.

3.6 Exchanging Elementary Data Types

In the nodes we just discussed, we saw that to transmit an elementary data type (e.g., an integer) between two nodes, it is necessary to pass through a topic that has been created using the appropriate message type. This is done by creating instances of `rclcpp::Publisher` and `rclcpp::Subscription` on the publisher and subscriber sides, respectively. To facilitate this task, ROS comes with a set number of predefined messages that are part of the package `example_interfaces`. These are wrappers for primitive data types, such as integers, strings, booleans, and so on. As can be shown using the `ros2 interface show`, each of these messages has just one field called `data`, used to store the corresponding information. From a C++ standpoint, these messages are C++ structures with a public field called, indeed, `data`. This is evident when analyzing the listings studied thus far, where the messages are sent and printed to the screen. Table 3.1 shows the correspondences between some of C++'s basic data types and messages in the `example_interfaces` package.

C++ data type	ROS message
`bool`	`example_interfaces::msg::Bool`
`char`	`example_interfaces::msg::Char`
`unsigned char`	`example_interfaces::msg::UInt8`
`int`	`example_interfaces::msg::Int32`
`unsigned int`	`example_interfaces::msg::UInt32`
`long int`	`example_interfaces::msg::Int64`
`unsigned long int`	`example_interfaces::msg::UInt64`
`float`	`example_interfaces::msg::Float32`
`double`	`example_interfaces::msg::Float64`

Table 3.1: Correspondences between C++ elementary data types and ROS standard messages.

In addition, C++ strings of type `std::string` can be sent using messages of type `example_interfaces::msg::String`. The package `example_interfaces` also allows to send multidimensional arrays of data, like vectors, matrices and even higher dimensional data structures, as well as other useful types, like time, duration, and others. These will be analyzed next.

Remark 3.6. *ROS also provides the package* ***`std_msgs`****, which offers messages and functionalities similar to the package* ***`example_interfaces`****. However, since the ROS* Foxy *distribution, this package has been deprecated, so in the following, we will not use it. Nevertheless, it is very common to find code on the web that still relies on it. For most purposes, the two packages are interchangeable.*

Remark 3.7. *When using* ***`ros2 interface show`*** *to inspect the structure of any of the above types, a text message is shown suggesting not to use these primitive data types, but rather to define semantically meaningful messages. That is to say, for example, if we are transmitting a temperature encoded as a floating point number on 32 bits, rather than creating a topic that transmits messages of type* ***`example_interfaces::msg::Float32`****, we should create a new*

message type (perhaps called `Temperature`*), put it into a package, and use this one instead. This is a valid suggestion that is certainly appropriate for production systems. However, for the sake of simplicity, in the following, we do not follow this route.*

3.7 Transmitting and Receiving Arrays of Data

As a further example, let us write two nodes that transmit and receive an array of integers. The overall structure of the code is similar to the listings we have seen so far, but in this case, we have to use a different message type, namely `example_interfaces/msg/Int32MultiArray`. As the name suggests, this message carries a multidimensional array, i.e., an array with an arbitrary number of dimensions. Because multidimensional arrays must ultimately be stored in a linear structure (i.e., a one-dimensional array), the message also includes information necessary to determine how the multidimensional array can be serialized into or deserialized from a one-dimensional structure. To gain insights on the internal structure of this message, we can run

```
ros2 interface show example_interfaces/msg/Int32MultiArray
```

This will generate the following output (comments removed for brevity):

```
MultiArrayLayout  layout        # specification of data layout
    MultiArrayDimension[] dim #
        string label   #
        uint32 size    #
        uint32 stride  #
    uint32 data_offset        #
int32[]           data          # array of data
```

Here, `data` is the one-dimensional array that holds the serialized multidimensional array, while `layout` stores information about how to reconstruct a multidimensional array from its serialized representation. `layout` includes an array called `dim` of type `MultyArrayDimension` with information about each dimension, as well as an offset. Following the same process discussed in Section 2.7.1, we can analyze the structure of `MultiArrayDimension` with the command

```
ros2 interface show example_interfaces/msg/MultiArrayDimension
```

which produces the output

```
string label   # label of given dimension
uint32 size    # size of given dimension (in type units)
uint32 stride  # stride of given dimension
```

In summary, a message of type `example_interfaces/msg/Int32MultiArray` includes a message of type `example_interfaces/msg/MultiArrayLayout` called `layout` to interpret

the serialization, an the field `data`. `example_interfaces/msg/MultiArrayLayout` has two fields: a vector of messages of type `example_interfaces/msg/MultiArrayDimension` and an integer called `data_offset`. `example_interfaces/msg/MultiArrayDimension` describes a single dimension. It includes a string `label` and two integers, `size` and `stride`. If we are transmitting an array with n dimensions, then `dim` will have n elements, each defining the properties of one dimension. When arrays are included in a message definition, they are instances of the `vector` class from the C++ Standard Template Library (STL). This means that `data` and `dim` offer the usual methods found in the STL `vector` class, such as `size`, `push_back`, and so on.

The code showing how a message of type `example_interfaces/msg/Int32MultiArray` can be used to send a vector of integers is shown in Listing 3.11. The important point to remember is that since we are sending a unidimensional array, the vector `layout.dim` has just one element to describe the single dimension of the structure we are sending.[7]

Listing 3.11: Node publishing an array of integers

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/int32_multi_array.hpp> //for arrays of ints
3  #include <example_interfaces/msg/multi_array_dimension.hpp> // for dimensions
4
5  #define SIZE 10 // size of the array we are sending
6
7  int main(int argc,char **argv) {
8
9    rclcpp::init(argc,argv);
10   rclcpp::Node::SharedPtr nodeh;
11   rclcpp::Rate rate(1);
12
13   nodeh = rclcpp::Node::make_shared("sendarray"); // create node
14   // create publisher
15   auto pubA = nodeh->create_publisher<example_interfaces::msg::Int32MultiArray>
16       ("arrayint",10);
17
18   int value = 0;
19   example_interfaces::msg::Int32MultiArray toSend; // message to send
20
21   // setup data structure to send one dimension
22   toSend.layout.dim.push_back(example_interfaces::msg::MultiArrayDimension());
23   toSend.layout.dim[0].size = SIZE; // first dimension size
24   toSend.layout.dim[0].stride = 1; // 1 because unidimensional
25   toSend.layout.dim[0].label = "row"; // arbitrary label
26   // make space for the serialized array
27   toSend.data.resize(toSend.layout.dim[0].size);
28
29   while (rclcpp::ok()) {
30     // store some values
31     for (int i = 0; i < SIZE ; i++)
32       toSend.data[i] = i+value;
33     // publish
34     pubA->publish(toSend);
```

[7] The code for these and subsequent examples can be found on the MRTP GitHub in the package `examples`.

```
35      value++;
36      rate.sleep();
37    }
38  }
```

Observe that we initialize the layout of the array by defining a single dimension. Next, we resize the `data` field using the `resize` method provided by the STL `vector` class. In this example, the field `stride` is equal to 1 because we have only one dimension and is irrelevant. Later, we will discuss how to set it when dealing with multiple dimensions. Note that, in general, `stride` is not equal to the number of dimensions. Then, in the main loop, we fill the data structure and publish to the appropriate topic, exactly as we did for the previous nodes. In this example, we also demonstrate that complex syntax in the definition of the subscriber can be avoided by using the keyword `auto` to detect the type. At this point, the structure of a node that subscribes to a topic containing messages of type `example_interfaces/msg/Int32MultiArray` should be straightforward to deduce. Listing 3.12 shows the code.

Listing 3.12: Node subscribing to a topic with an array of integers

```
1  #include <rclcpp/rclcpp.hpp> // needed for basic functions
2  #include <example_interfaces/msg/int32_multi_array.hpp> // for arrays of ints
3
4  rclcpp::Node::SharedPtr nodeh;
5
6  // callback function to process incoming messages
7  void arrayCallback(const example_interfaces::msg::Int32MultiArray::SharedPtr
8                                                                          msg) {
9    RCLCPP_INFO(nodeh->get_logger(),"Received new message");
10   // just print everything to the screen
11   for(unsigned int i = 0 ; i <msg->data.size() ; i++)
12     RCLCPP_INFO(nodeh->get_logger(),"%d",msg->data[i]);
13 }
14
15 int main(int argc,char ** argv) {
16
17   rclcpp::init(argc,argv);
18   nodeh = rclcpp::Node::make_shared("arraysubscriber"); // create node
19
20   // create subscriber and register callback function
21   auto sub = nodeh->create_subscription
22                              <example_interfaces::msg::Int32MultiArray>
23     ("arrayint",10,&arrayCallback);
24
25   // receive all messages
26   rclcpp::spin(nodeh);
27 }
```

The handler function in this case retrieves the size of the array directly using the `size` method in `data` because we are receiving a unidimensional array, making the rest of the layout information unnecessary. This will instead be explained and expanded upon in the next subsection.

3.7.1 Sending and Receiving a Matrix

The next two examples demonstrate how `example_interfaces/msg/Int32MultiArray` can be used to send and receive a multidimensional array—in this case, a two-dimensional matrix. As previously stated, regardless of the number of dimensions, all elements in the multidimensional array are ultimately stored in the unidimensional structure `data`. The key lies in initializing the `layout` with the appropriate information on how to map between multidimensional indices and the linear structure `data`. Listing 3.13 illustrates this process. Since we are sending a matrix (i.e., a 2-dimensional structure), `dim` must be initialized to store two elements. For each dimension, we must specify the size and the stride, i.e., the linear separation between two successive elements. Additionally, we set the `data_offset` field to 0, as the data starts from the beginning. However, in general, this value could be set differently[8]. If the multidimensional structure has n dimensions, say $A_1 \times A_2 \times A_3 \times \dots A_n$, then the stride S_i of the i-th dimension is defined as:

$$S_i = A_i \times A_{i+1} \times \cdots \times A_n.$$

In this case, an element is identified by n indices, say $(i_1, i_2, \dots, i_n)$. Its position in the linear data structure (assuming an offset of 0) is given by:

$$i_1 \cdot S_2 + i_2 \cdot S_3 + \cdots + i_{n-1} \cdot S_n + i_n.$$

For the specific case where $n = 2$, this formula is used inside the two nested `for` loops to populate the `data` field. It is also applied in the callback function in Listing 3.14. Another important aspect to consider is the order in which dimensions are declared, i.e., from the outermost to the innermost.

Listing 3.13: Node publishing a matrix of integers

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <example_interfaces/msg/int32_multi_array.hpp>
3  #include <example_interfaces/msg/multi_array_dimension.hpp>
4
5  #define ROWS 4 // matrix dimensions
6  #define COLS 5
7
8  int main(int argc,char **argv) {
9    rclcpp::init(argc,argv);
10   rclcpp::Node::SharedPtr nodeh;
11   rclcpp::Rate rate(1);
12
13   nodeh = rclcpp::Node::make_shared("sendmatrix"); // create node
14   // create publisher
15   auto pubA = nodeh->create_publisher<example_interfaces::msg::Int32MultiArray>
16       ("matrixint",10);
17
18   // instance of message to send
19   example_interfaces::msg::Int32MultiArray toSend;
20   int value=0;
```

[8]By setting `data_offset` to a value other than 0, it is possible to allocate a buffer at the beginning of `data` to store auxiliary information.

```
21
22    // setup layout for a matrix of size ROWS * COLS
23    toSend.layout.dim.resize(2);  // dimensions
24    toSend.layout.dim[0].size = ROWS; // size of the first dimension
25    toSend.layout.dim[0].stride = ROWS * COLS; // not necessarily needed
26    toSend.layout.dim[0].label = "row"; // label for first dimension
27    toSend.layout.dim[1].size = COLS; // size of second dimension
28    toSend.layout.dim[1].stride = COLS; // separation between columns
29    toSend.layout.dim[1].label = "col"; // label for second dimension
30    toSend.layout.data_offset = 0; // no offset
31    toSend.data.resize(toSend.layout.dim[0].stride); // number of elements
32    while (rclcpp::ok()) {
33      // fills entry (i,j) with i*j+value
34      for (int i = 0; i < ROWS ; i++) {
35        for ( int j = 0 ; j < COLS ; j++ ) {
36          // note how access (i,j) in data
37          toSend.data[i*toSend.layout.dim[1].stride + j] = i+j+value;
38        }
39      }
40      value++;
41      pubA->publish(toSend); // publish
42      rate.sleep();
43    }
44  }
```

Listing 3.14 illustrates how a node subscribing to the topic can extract the sizes of the different dimensions directly from the message and use this information to reconstruct the data in matrix form. In this case, the callback function retrieves the data from the message and assembles the matrix as a vector of vectors using the STL `vector` class. However, it does not print the matrix to the screen.

Listing 3.14: Node subscribing to a matrix of integers

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <example_interfaces/msg/int32_multi_array.hpp>
3  #include <vector>
4
5  rclcpp::Node::SharedPtr nodeh;
6
7  // callback function called when a matrix is received
8  void matrixCallback(const example_interfaces::msg::Int32MultiArray::SharedPtr
9                                                                          msg) {
10   // first extracts dimensions from message
11   int nrows =  msg->layout.dim[0].size;
12   int ncols =  msg->layout.dim[1].size;
13   RCLCPP_INFO(nodeh->get_logger(),"Received %dx%d matrix",nrows,ncols);
14   // allocate local matrix to copy received data
15   std::vector<std::vector<int> > matrix(nrows, std::vector<int>(ncols));
16   for(int i = 0 ; i < nrows ; i++)
17     for (int j = 0 ; j < ncols ; j++)
18       // note how element (i,j) is extracted and copied in matrix[i][j]
19       matrix[i][j] = msg->data[i*msg->layout.dim[1].stride + j];
20 }
21
```

```
22
23 int main(int argc,char ** argv) {
24
25   rclcpp::init(argc,argv);
26   nodeh = rclcpp::Node::make_shared("matrixsubscriber"); // create node
27   // create subscriber and register callback function
28   auto sub =
29     nodeh->create_subscription<example_interfaces::msg::Int32MultiArray>
30     ("matrixint",10,&matrixCallback);
31
32   rclcpp::spin(nodeh);
33
34 }
```

The output will look like the following

```
[INFO] [1672674595.288773746] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674596.288823136] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674597.288452699] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674598.289269984] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674599.288628307] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674600.290741247] [matrixsubscriber]: Received 4x5 matrix
[INFO] [1672674601.292297280] [matrixsubscriber]: Received 4x5 matrix
```

However, if we run

```
ros2 topic echo matrixint
```

we can visualize the matrices being sent over the topic directly on the screen. The output will resemble the following, confirming that the matrix is transmitted according to the format we discussed.

```
---
layout:
dim:
- label: row
size: 4
stride: 20
- label: col
size: 5
stride: 5
data_offset: 0
data:
- 9
- 10
- 11
- 12
- 13
- 10
```

```
- 11
- 12
- 13
- 14
- 11
- 12
- 13
- 14
- 15
- 12
- 13
- 14
- 15
- 16
---
```

3.8 Publishing and subscribing from the same node

In many cases, a node is both a publisher and a subscriber. Typically, a node receives some information (e.g., an image from a camera), performs some processing (e.g., runs a computer vision algorithm to determine if there is an object of interest), and then passes the result to another node (e.g., if the object of interest is found, it estimates its position and passes this information to a navigation node that will move the robot towards the object). In other instances, the operation performed on the data can be much simpler, such as finding the smallest number in a series of distance readings from a proximity range finder. If the computation done in response to the received message is limited, it is possible to include the code for publishing to a topic inside the callback function handling incoming messages. However, it is not advisable to include time-consuming code in the callback function because, in such cases, the callback function may not be able to keep up with the incoming messages. Therefore, depending on the computation to be performed in response to an incoming message, different solutions may be implemented.

An important consideration is that the publisher object should not be declared and initialized inside the handler function for a variety of reasons (e.g., it can be relatively slow). As an example, we write a node that receives messages of type `sensor_msg/msg/LaserScan` and publishes to a topic of floats. The `sensor_msg/msg/LaserScan` message is used to transmit data produced by a laser scanner, a sensor commonly used in robotics, which will be introduced later on (Chapter 7.) Among other things, the sensor produces an array of distances (called `ranges`) to nearby obstacles. The node we write analyzes each message, determines the smallest value, and publishes it to a topic called `closest`. This behavior could, for example, be used to implement a safety feature, whereby another node could subscribe to `closest` and decide whether the robot should continue moving, depending on the distance to the closest obstacle. As usual, we use `ros2 interface` to determine the structure of the message of type `sensor_msg/msg/LaserScan`.

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

std_msgs/Header header # timestamp in the header is the acquisition time of
    builtin_interfaces/Time stamp
        int32 sec
        uint32 nanosec
    string frame_id
# the first ray in the scan.
#
# in frame frame_id, angles are measured around
# the positive Z axis (counterclockwise, if Z is up)
# with zero angle being forward along the x axis

float32 angle_min            # start angle of the scan [rad]
float32 angle_max            # end angle of the scan [rad]
float32 angle_increment      # angular distance between measurements [rad]

float32 time_increment       # time between measurements [seconds] - if your
# scanner is moving, this will be used in interpolating position
# of 3d points
float32 scan_time            # time between scans [seconds]

float32 range_min            # minimum range value [m]
float32 range_max            # maximum range value [m]

float32[] ranges             # range data [m]
# (Note: values < range_min or > range_max should be discarded)
float32[] intensities        # intensity data [device-specific units].  If your
# device does not provide intensities, please leave
# the array empty.
```

The details of all the various fields will be discussed in Chapter 7, where we will introduce various sensors. For the time being, we will just use `ranges`, which is an array of non-negative floats indicating a set of measured distances. Listing 3.15 shows a node implementing the functionality we just described.

Listing 3.15: Publisher/subscriber node

```
1 #include <rclcpp/rclcpp.hpp>
2 #include <sensor_msgs/msg/laser_scan.hpp> // to receive laser scans
3 #include <example_interfaces/msg/float32.hpp> // to send floating point numbers
4
5 // publisher object to send the result
6 rclcpp::Publisher<example_interfaces::msg::Float32>::SharedPtr pubf;
7
```

```
8  // callback function called when a laser scan is received
9  void processScan(const sensor_msgs::msg::LaserScan::SharedPtr msg) {
10   example_interfaces::msg::Float32 out; // message with closest distance
11   out.data = msg->ranges[0]; // initialize result
12   // iterate over all readings and update result if necessary
13   for(unsigned int i = 1 ; i < msg->ranges.size() ; i++ ) {
14     if ( msg->ranges[i] < out.data )
15       out.data = msg->ranges[i];
16   }
17   pubf->publish(out); // publish result
18 }
19
20
21 int main(int argc,char ** argv) {
22
23   rclcpp::init(argc,argv);
24   rclcpp::Node::SharedPtr nodeh;
25
26   nodeh = rclcpp::Node::make_shared("pubsub"); // create node
27
28   // create publisher (global variable)
29   pubf = nodeh->create_publisher<example_interfaces::msg::Float32>
30                                                    ("closest",1000);
31   // create subscriber and register callback function
32   auto sub = nodeh->create_subscription<sensor_msgs::msg::LaserScan>
33     ("scan",10,&processScan);
34
35   rclcpp::spin(nodeh);
36 }
```

In this case, the node subscribes to a topic `scan` from which it will receive messages of type `sensor_msgs/msg/LaserScan`. The node analyzes the entire `ranges` array and publishes the smallest distance it finds. The logic is straightforward, and the only part that requires some thought is where to place and initialize the instance of `rclcpp::Publisher` used to publish to the topic. Since we want to publish from the handler function `processScan`, we cannot declare and initialize the publisher in the `main` function because it would not be visible from the callback function. In this case, therefore, the publisher is declared globally and initialized in the `main` function. Of course, global variables are generally to be avoided, but this is the most straightforward way to initialize the publisher outside the handler function. The official ROS documentation describes more principled (and complex) approaches to avoid using a global variable. Once we will introduce object oriented programming in ROS (Chapter 5) there will no longer be the need to declare global variables. Another aspect to note is that this node depends on a new package, `sensor_msgs`, as can be seen from the initial `include`. Therefore, `CMakeLists.txt` and `package.xml` must be correspondingly updated (see files in the MRTP GitHub for details).

For completeness, we also provide the listing of the node that publishes to the `scan` topic so that we can test the whole application. Its code is provided in listing 3.16, where we see how the `ranges` array is resized to hold 181 values (a typical value for this type of sensor) and then filled with random numbers between 0 and 2.

Listing 3.16: Node publishing laser scans

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/laser_scan.hpp>
#include <cstdlib>

#define NDATA 181 // number of readings

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

    rclcpp::init(argc,argv);
    rclcpp::Node::SharedPtr nodeh;
    rclcpp::Rate rate(1);

    nodeh = rclcpp::Node::make_shared("pubscan"); // create node
    // create publisher
    auto pubs = nodeh->create_publisher<sensor_msgs::msg::LaserScan>
      ("scan",10);
    int iteration = 1; // just for output purposes
    // instance of message to be sent
    sensor_msgs::msg::LaserScan toSend;
    // setup data structure to send
    toSend.ranges.resize(NDATA);
    // other fields in toSend should be initialized, too...

    while (rclcpp::ok()) {
      // generate random distances in range 0-2
      for (int i = 0; i < NDATA ; i++)
        toSend.ranges[i] = (2*float(rand()))/RAND_MAX;
      RCLCPP_INFO(nodeh->get_logger(),"Publishing scan #%d",iteration++);
      pubs->publish(toSend); // publish
      rate.sleep();
    }

}
```

If we now build the package, run both nodes and then run

```
ros2 topic echo /closest
```

from a separate shell, we will get an output similar to the following

```
data: 0.002367734909057617
---
data: 0.0030525801703333855
---
data: 0.000956712756305933
---
data: 0.007958965376019478
---
data: 0.00041060103103518486
---
```

```
data: 0.003463807050138712
---
data: 0.0008788560517132282
---
```

thus confirming that `pubsub` is indeed working as expected.

Further reading

While there are numerous textbooks devoted to ROS 1 programming, due to the fact that ROS 2 is still relatively new, there are fewer printed resources available for it. Two recent books focusing on ROS 2 programming are [45, 46]. The ROS 2 official website also includes various tutorials about programming nodes in C++.

Previously published books about ROS 1 programming, such as [20,21,25,42,43], may also be useful for exploring different approaches to robot programming, but their code examples will not work "as is" in ROS 2 and will need to be adjusted. To this end, `https://docs.ros.org/en/jazzy/The-ROS2-Project/Contributing/Migration-Guide.html` provides a guide on how to migrate code from ROS 1 to ROS 2.

Chapter 4

Geometric Representations and Kinematics

4.1 Introduction

Kinematics is the study of motion without considering its causes. It is a fundamental topic in physics, and the reader should already be familiar with its principles. The study of kinematics relies on the ability to describe the position and orientation of objects in space. In particular, velocity is defined as the time derivative of pose. Mathematically, if $\mathbf{P}(t)$ represents the position and orientation of an object in space as a function of time t, then velocity $\mathbf{V}(t)$ is given by:

$$\mathbf{V}(t) = \frac{d\mathbf{P}(t)}{dt} \tag{4.1}$$

Thus, to study kinematics, we must first establish how to represent the position and orientation of objects in space—that is, their geometric representations. In this chapter, we will explore a subset of geometric representations and kinematics from two perspectives. First, we will introduce the mathematical notation used to describe problems relevant to robotics. Then, we will examine how these concepts are implemented in ROS.

Before getting into the mathematical details of geometric representations and kinematics, we start with some examples showing why these topics are important when studying robotics. Robots operate in the physical three-dimensional world, and it is therefore necessary to develop an appropriate set of tools to reason and make decisions about the space in which they move. For example, a robot scouting an orchard and capable of detecting anomalies (e.g., detecting a pest) should determine its location in the world and use this information to dispatch a human operator who can assess the situation. Without loss of generality, we assume the existence of a so-called *world* frame[1], i.e., a coordinate system with respect to which positions and angles are expressed. The location of the robot will, in general, be expressed with respect to the world frame. Abstracting the robot as a rigid object, its location in the world is defined by its position and orientation. In most instances, a robot is not a rigid body but is rather composed of various interconnected parts that can

[1]The formal definition of frame will be given in the next section. For now a frame can just be assumed to be a coordinate system.

move with respect to each other. Nevertheless, the rigid body assumption is a reasonable approximation to get started. Position and orientation combined will be collectively called *pose* in the following. This corresponds to $\mathbf{P}$ in Eq.(4.1). The position is given by a vector in $\mathbb{R}^3$, and the orientation is defined by three angles (details will be provided in the following). This approach can be conveniently modeled by assuming that a rigid frame is attached to the robot. *Rigidly attached* means that whenever the robot moves or rotates, the frame makes the same move or rotation (and vice versa), and there is no mutual motion or rotation between the attached frame and the robot. Hence, one can simply reason about the pose of the frame as a proxy for the pose of the robot. Figure 4.1 illustrates this idea. The red frame represents the world frame, whereas the orange frame, rigidly attached to the quadrotor, defines its pose. To describe the location of the quadrotor or its motion with respect to the world frame, one can then just describe the location of the orange frame or its motion with respect to the world frame.

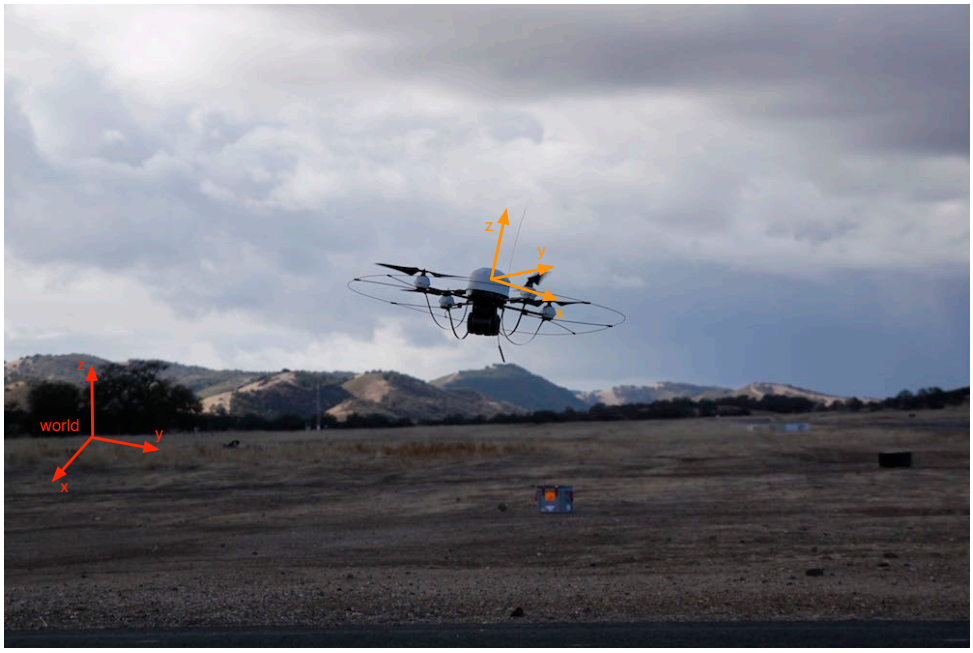


Figure 4.1: The orange frame rigidly attached to the quadrotor defines its pose with respect to the red world frame.

Another scenario worth considering is when there is more than one robot operating in the same environment. In this case, each of them will be associated with its rigidly attached frame. Therefore, it will be necessary to develop appropriate notation to consider multiple frames at once. Moreover, it will often be necessary to compute coordinate transformations between these multiple frames. This is most often the case because many sensors provide *local* or *relative* measurements. For example, a stereo camera estimating the pose of an object will most likely return a measurement expressed relative to the camera itself and not in the world frame. So if robot A determines the pose of an object through its onboard sensors and this pose is expressed with respect to a frame rigidly attached to A, it may be useful to determine the pose of the object with respect to a frame rigidly attached to a different

robot, say B, or in the world frame. This case is depicted in Figure 4.2. The robot on the left (say A) observes the purple diamond, and its position is expressed relative to robot A. Assuming that the pose of both robots A and B is known with respect to the world frame (blue frame at the top), how do we determine the pose of the purple diamond relative to the robot B on the right?

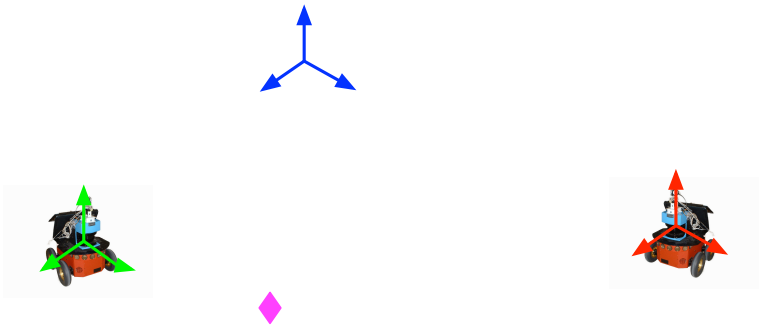


Figure 4.2: Two robots observing the same object in their local coordinate frames.

Questions like these are extremely important when multiple robots cooperate to solve the same task and therefore need to exchange information extracted by their sensors, which is expressed in local coordinates. Finally, changes of coordinates are necessary even when only a single robot is involved. Consider, for example, the case shown in Figure 4.3.

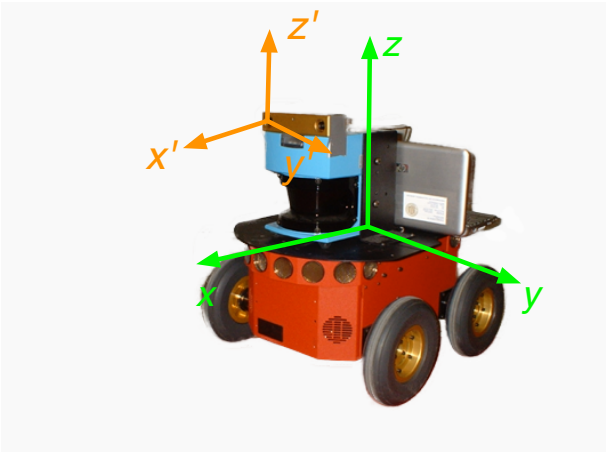


Figure 4.3: Multiple frames can be assigned to the same robot, e.g., attaching frames to sensors.

The robot uses a stereocamera for depth estimation. Algorithms processing data from the camera will most likely return information referred to the orange frame attached to the camera[2], e.g., the estimated distance of an object will be referred to the orange frame. If the robot needs to move to approach this object, this information must be referred to the green frame attached to the robot, assuming that the planning system computes motion

[2]In practice, the frame is likely to be oriented differently, i.e., with the $\mathbf{z}'$ axis pointing forward. However, this technical detail is immaterial for the rest of the discussion.

commands relative to the green frame. In many practical scenarios, a single robot does not feature just a couple of frames but a very large number. Figure 4.4, for example, shows the frames associated with the PR2 robot, a research platform consisting of two arms mounted on a mobile base (`https://robots.ieee.org/robots/pr2/`).

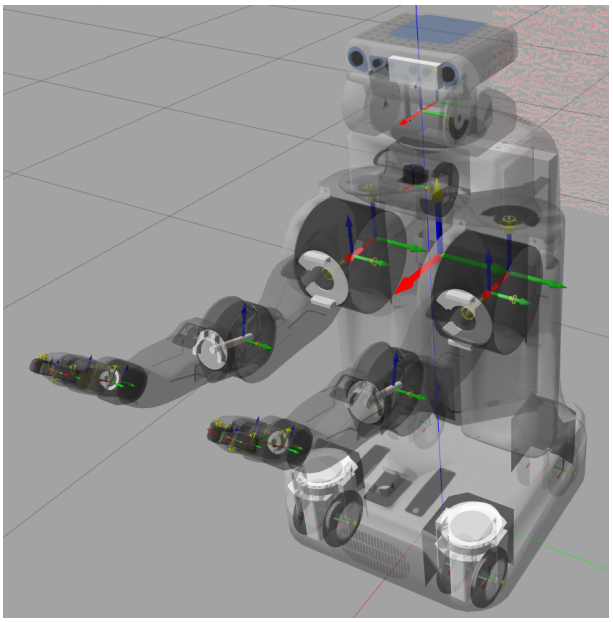


Figure 4.4: Frames attached to the PR2 robot.

From these examples, it should be clear that geometric representations, coordinate transformations and kinematics are concepts pervasive in robotics, and we need to develop the tools to reason about them and perform relevant operations in a principled and efficient way. In this chapter, we first present the mathematical formalism to solve these problems, and we then illustrate how these concepts are implemented in ROS.

4.2 Background and Notation

We start by providing some definitions and recalling basic concepts that should already be familiar to the reader. In the following, we will often refer to rigid bodies, or collections of (possibly interconnected) rigid bodies.

Definition 4.1. *A body $\mathcal{B}$ is said to be a* rigid body *if the Euclidean distance between any two points in $\mathcal{B}$ is constant.*

According to this definition, a body is rigid if it cannot be compressed, stretched, twisted, etc., i.e., if its shape does not change.

Definition 4.2. *A* frame of reference *(or simply* frame*) is given by an origin point O and three orthonormal vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ called axes satisfying the so-called right-hand rule, i.e.,*

$$\mathbf{x} \times \mathbf{y} = \mathbf{z} \qquad \mathbf{y} \times \mathbf{z} = \mathbf{x} \qquad \mathbf{z} \times \mathbf{x} = \mathbf{y} \tag{4.2}$$

where $\times$ is the vector (cross) product.

Recall that the orthonormal requirement stipulates that each of the three vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ has Euclidean norm 1, and that they are mutually orthogonal, i.e.,

$$\mathbf{x} \cdot \mathbf{x} = 1 \qquad \mathbf{y} \cdot \mathbf{y} = 1 \qquad \mathbf{z} \cdot \mathbf{z} = 1 \qquad \mathbf{x} \cdot \mathbf{y} = 0 \qquad \mathbf{y} \cdot \mathbf{z} = 0 \qquad \mathbf{z} \cdot \mathbf{x} = 0 \tag{4.3}$$

where $\cdot$ is the dot product.

It shall be noted that one could also consider left-handed reference frames, but these are rarely used in robotics and will therefore not be considered. In the following, we will often indicate a frame as $O - \mathbf{xyz}$, and we will give symbolic names to frames, like A, B, etc. Figure 4.5 shows the typical representation for a frame.

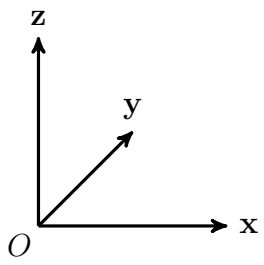


Figure 4.5: Representation of a frame

A point $\mathbf{p}$ in $\mathbb{R}^n$ is represented by n values[3] called coordinates, i.e., $\mathbf{p} = [p_1\ p_2\ \ldots p_n]^T$ where the superscript ^{T} indicates the transpose, and we have, as usually assumed, that elements in $\mathbb{R}^n$ are column vectors. The values of the *coordinates* are always expressed with reference to a frame. In many cases, there is just one frame, so no confusion will arise. This is likely the case already seen by most readers. But in most robotics applications, there will be many frames, so it is important to be precise and specify to which frame the coordinates are referred. To this end, it is customary to introduce a leading superscript, like ${}^A\mathbf{p}$ and ${}^B\mathbf{p}$. With this notation, ${}^A\mathbf{p}$ is the vector with the coordinates of point $\mathbf{p}$ referred to reference frame A, whereas ${}^B\mathbf{p}$ indicates the coordinates of the same point but referred to reference frame B. Figure 4.6 illustrates this concept. In this case, ${}^A\mathbf{p} = [0.6\ 2\ 2]^T$ and ${}^B\mathbf{p} = [0\ -1.4\ 0]^T$

4.3 Representing a frame

We now turn to the problem of representing a frame with respect to another frame. Recall that a frame is defined by its origin (a point) and its three orthonormal axes. Since we have just learned how to represent the coordinates of a point with respect to different frames, the problem of representing the origin of a frame is therefore solved, and we can focus on how to represent its axes with respect to a frame.

To be specific, let $A = O - \mathbf{xyz}$ be the first frame and $B = O' - \mathbf{x'y'z'}$ be the second frame. Our objective is to represent frame B with respect to frame A (see Figure 4.7).

[3]Depending on the situation, in robotics problems we may have $n = 2$ or $n = 3$. These are the only two cases we will have to consider.

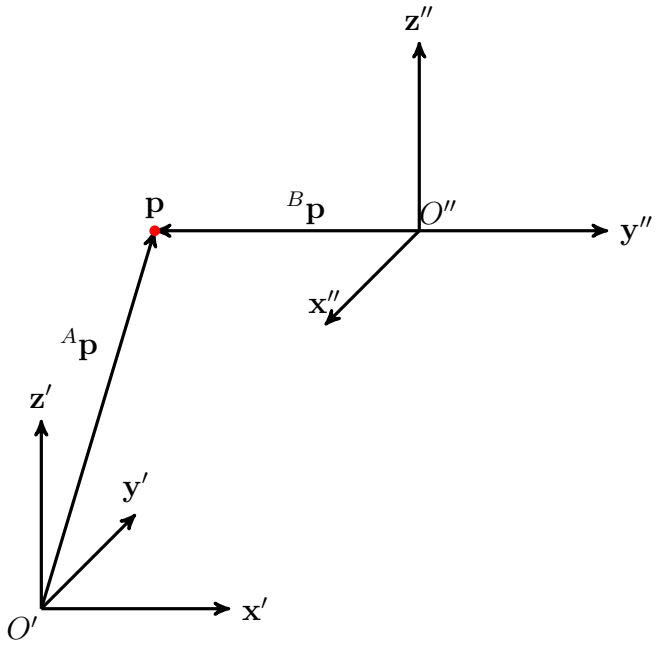


Figure 4.6: Given two frames $A = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ and $B = O'' - \mathbf{x}''\mathbf{y}''\mathbf{z}''$, the coordinates of point $\mathbf{p}$ can be expressed with respect to either frame. In the first case, the coordinates are indicated as ${}^A\mathbf{p}$, whereas in the latter they are ${}^B\mathbf{p}$. The vectors ${}^A\mathbf{p}$ and ${}^B\mathbf{p}$ have different coordinate values.

The origin O' is a point and is therefore described in frame A as a three-dimensional vector. In particular, let $\mathbf{p}$ be a point coincident with O'. Its coordinates with respect to frame A are then given by the vector ${}^A\mathbf{p}$ (alternatively, we could write ${}^AO'$). Next, we want to describe the three vectors $\mathbf{x}', \mathbf{y}', \mathbf{z}'$ with respect to frame A. This is done by considering the projections of each of these three axes along the axes $\mathbf{x}, \mathbf{y}, \mathbf{z}$. Conceptually, one can think of translating (without rotating) frame B so that its origin coincides with the origin of frame A, and then taking the coordinates of the endpoints of the three unitary axes $\mathbf{x}', \mathbf{y}', \mathbf{z}'$ with respect to frame A (see Figure 4.8).

The coordinates of the endpoints are, by definition, the projections along the axes. This leads to the following expressions for the three coordinates of the axis $\mathbf{x}'$ expressed with respect to frame A:

$$ {}^A\mathbf{x}' = \begin{bmatrix} \mathbf{x}' \cdot \mathbf{x} \\ \mathbf{x}' \cdot \mathbf{y} \\ \mathbf{x}' \cdot \mathbf{z} \end{bmatrix} $$

where $\mathbf{x}' \cdot \mathbf{x}$ is the dot product between $\mathbf{x}'$ and $\mathbf{x}$, $\mathbf{x}' \cdot \mathbf{y}$ is the dot product between $\mathbf{x}'$ and $\mathbf{y}$, and so on. Similarly, we can obtain analogous expressions for $\mathbf{y}'$ and $\mathbf{z}'$, i.e.,

$$ {}^A\mathbf{y}' = \begin{bmatrix} \mathbf{y}' \cdot \mathbf{x} \\ \mathbf{y}' \cdot \mathbf{y} \\ \mathbf{y}' \cdot \mathbf{z} \end{bmatrix} \qquad {}^A\mathbf{z}' = \begin{bmatrix} \mathbf{z}' \cdot \mathbf{x} \\ \mathbf{z}' \cdot \mathbf{y} \\ \mathbf{z}' \cdot \mathbf{z} \end{bmatrix}. $$

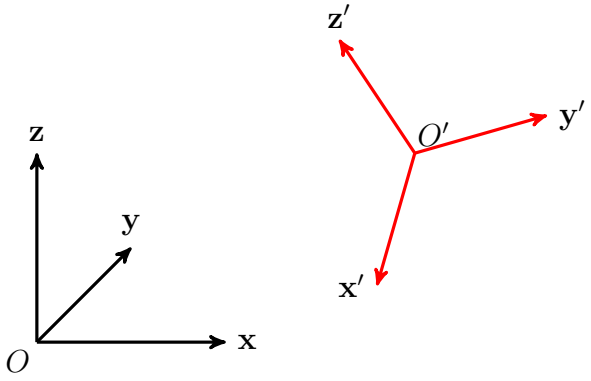


Figure 4.7: Given two frames $A = O - \mathbf{xyz}$ and $B = O' - \mathbf{x'y'z'}$, we want to represent frame B with respect to frame A.

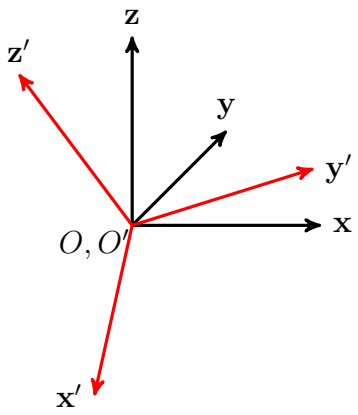


Figure 4.8: Frames A and B with coincident origins. Note that B has not been rotated (compare with Figure 4.7.)

These three vectors are compactly represented as the three columns of a 3×3 matrix called the *rotation matrix*.

$$ {}^A_B\mathbf{R} = \begin{bmatrix} \mathbf{x}' \cdot \mathbf{x} & \mathbf{y}' \cdot \mathbf{x} & \mathbf{z}' \cdot \mathbf{x} \\ \mathbf{x}' \cdot \mathbf{y} & \mathbf{y}' \cdot \mathbf{y} & \mathbf{z}' \cdot \mathbf{y} \\ \mathbf{x}' \cdot \mathbf{z} & \mathbf{y}' \cdot \mathbf{z} & \mathbf{z}' \cdot \mathbf{z} \end{bmatrix} \tag{4.4} $$

It is important to stress the notation used. The subscript B and superscript A indicate that this matrix describes the rotation of frame B with respect to frame A. The order is important because in general ${}^A_B\mathbf{R} \neq {}^B_A\mathbf{R}$. Summarizing, frame B can be described with respect to frame A through a three-dimensional vector for its origin and a 3×3 rotation matrix for its orientation. Moreover, consistently with the notation we used for points, the leading superscript A indicates that the rotation matrix is referred to frame A.

Example 4.1. *Let $A = O - \mathbf{xyz}$ and $B = O' - \mathbf{x'y'z'}$ be two aligned frames, i.e., two frames with parallel axes (see figure 4.9). Then ${}^A_B\mathbf{R}$ is the 3×3 identity matrix $\mathbf{I}$. This result is easy to derive considering Eq. (4.4). The first column is the projection of $\mathbf{x}'$ along the axes $\mathbf{xyz}$, i.e., the vector $[1\ 0\ 0]^T$ since $\mathbf{x}'$ coincides with $\mathbf{x}$ and is orthogonal to both $\mathbf{y}$ and $\mathbf{z}$ (and*

therefore the length of its projections along these axes is 0). Following a similar reasoning, the second column is the projection of $\mathbf{y}'$ *along the axes* $\mathbf{xyz}$*, i.e., the vector* $[0\ 1\ 0]^T$*, and the third column is* $[0\ 0\ 1]^T$*. Putting the three columns together, we then obtain* ${}^A_B\mathbf{R} = \mathbf{I}$*. Following an identical reasoning, we also verify that* ${}^B_A\mathbf{R} = \mathbf{I}$.

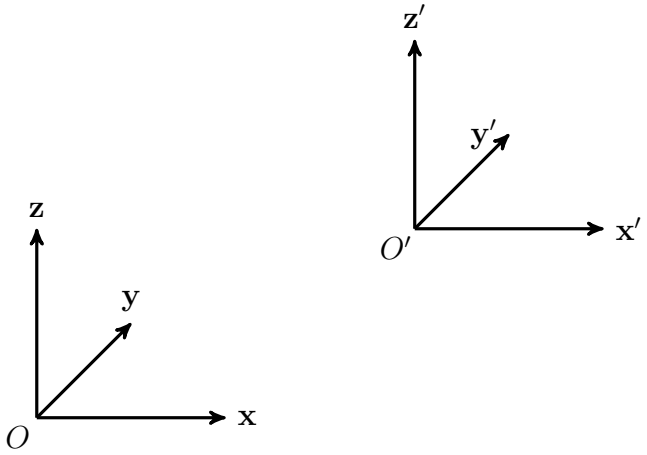


Figure 4.9: Two frames with parallel axis and different origins.

We next determine ${}^B_A\mathbf{R}$, i.e., the rotation matrix describing frame A with respect to frame B. Note how in this case the subscript and superscript have been flipped. Similarly to what we did before, we consider the projections of the axes $\mathbf{xyz}$ along the axes $\mathbf{x'y'z'}$. Following the same reasoning that led to Eq. (4.4), we can write

$${}^B_A\mathbf{R} = \begin{bmatrix} \mathbf{x}\cdot\mathbf{x}' & \mathbf{y}\cdot\mathbf{x}' & \mathbf{z}\cdot\mathbf{x}' \\ \mathbf{x}\cdot\mathbf{y}' & \mathbf{y}\cdot\mathbf{y}' & \mathbf{z}\cdot\mathbf{y}' \\ \mathbf{x}\cdot\mathbf{z}' & \mathbf{y}\cdot\mathbf{z}' & \mathbf{z}\cdot\mathbf{z}' \end{bmatrix}.$$

Recalling that the dot product is commutative, and the previous matrix is therefore nothing but the transpose of (4.4), i.e.,

$${}^B_A\mathbf{R} = {}^A_B\mathbf{R}^T. \tag{4.5}$$

4.4 Change of coordinates

A change of coordinates problem emerges in situations like the one depicted in Figure 4.2. A robot[4] B determines the position of an object in its local frame and wants to pass this information to another robot A. This requires expressing the pose of the same object in two different frames, each attached to a robot. This problem is formulated as follows. Let $A = O - \mathbf{xyz}$ and $B = O' - \mathbf{x'y'z'}$ be two frames, and let ${}^B\mathbf{p} = [p_x\ p_y\ p_z]^T$ be the coordinates of point $\mathbf{p}$ expressed with respect to frame B. We want to determine ${}^A\mathbf{p}$, i.e., the coordinates of $\mathbf{p}$ with respect to frame A. We assume the availability of the description of frame B with respect to A, i.e., ${}^AO'$ and ${}^A_B\mathbf{R}$.

[4]We use the same symbols for robots and frames because, as we previously stated, frames and robots are interchangeable in this discussion.

To solve the general problem, we first start with two simpler special cases. First, assume that the two frames are aligned (see Figure 4.10). This means that $\mathbf{x}$ is parallel to $\mathbf{x}'$, $\mathbf{y}$ is parallel to $\mathbf{y}'$, and $\mathbf{z}$ is parallel to $\mathbf{z}'$, and therefore ${}^A_B\mathbf{R} = \mathbf{I}$.

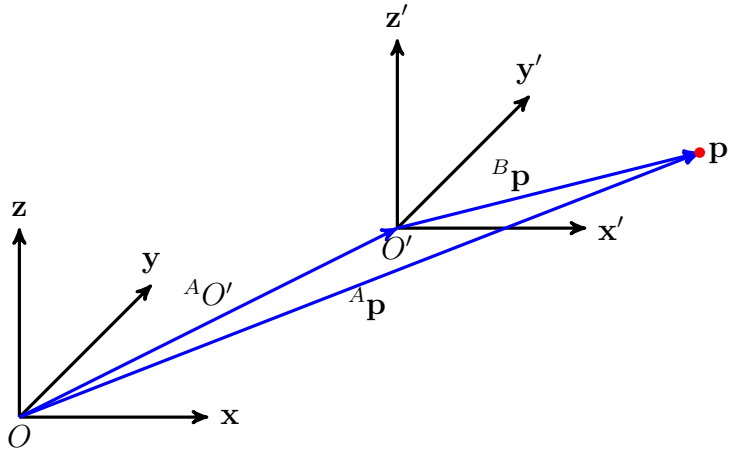


Figure 4.10: Change of coordinates for the case where frames A and B are aligned.

Let ${}^AO' = [\Delta x\ \Delta y\ \Delta z]^T$. Then it trivially follows that the vector of coordinates of $\mathbf{p}$ in frame A, i.e., ${}^A\mathbf{p}$ is the sum of the two vectors ${}^AO'$ and ${}^B\mathbf{p}$:

$$ {}^A\mathbf{p} = {}^B\mathbf{p} + {}^AO' = \begin{bmatrix} p_x + \Delta x \\ p_y + \Delta y \\ p_z + \Delta z \end{bmatrix}. $$

We next consider the other special case where ${}^AO' = [0\ 0\ 0]^T$ but ${}^A_B\mathbf{R} \neq \mathbf{I}$. In this case, the two frames share the same origin but are not aligned (see Figure 4.8). Since ${}^B\mathbf{p} = [p_x\ p_y\ p_z]^T$ is the vector of coordinates with respect to frame $B = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ this means that

$$ {}^B\mathbf{p} = p_x\mathbf{x}' + p_y\mathbf{y}' + p_z\mathbf{z}' \tag{4.6} $$

This expression holds because ${}^B\mathbf{p}$ by definition has the coordinates with respect to to frame B, and frame B is defined by the axes $\mathbf{x}'$, $\mathbf{y}'$, $\mathbf{z}'$.

Next, recalling how we defined the rotation matrix ${}^A_B\mathbf{R}$ in Eq. (4.4), each of the vectors $\mathbf{x}'$, $\mathbf{y}'$, and $\mathbf{z}'$ appearing in Eq. (4.6) can be expressed with respect to the frame $A = O - \mathbf{xyz}$. Recall that the first column of ${}^A_B\mathbf{R}$ gives the projections of $\mathbf{x}'$ along the three axes $\mathbf{xyz}$, the second gives the projections of $\mathbf{y}'$ along the three axes $\mathbf{xyz}$, and the third gives the projections of $\mathbf{z}'$ along the three axes $\mathbf{xyz}$. Writing out the elements of the rotation matrix as follows:

$$ {}^A_B\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} $$

we can then write

$$
\begin{aligned}
{}^{A}\mathbf{x}' &= r_{11}\mathbf{x} + r_{21}\mathbf{y} + r_{31}\mathbf{z} \\
{}^{A}\mathbf{y}' &= r_{12}\mathbf{x} + r_{22}\mathbf{y} + r_{32}\mathbf{z} \\
{}^{A}\mathbf{z}' &= r_{13}\mathbf{x} + r_{23}\mathbf{y} + r_{33}\mathbf{z}
\end{aligned}
$$

Substituting these relationships into Eq. (4.6) we can then determine ${}^{A}\mathbf{p}$:

$$
\begin{aligned}
{}^{A}\mathbf{p} &= p_x(r_{11}\mathbf{x} + r_{21}\mathbf{y} + r_{31}\mathbf{z}) + p_y(r_{12}\mathbf{x} + r_{22}\mathbf{y} + r_{32}\mathbf{z}) + p_z(r_{13}\mathbf{x} + r_{23}\mathbf{y} + r_{33}\mathbf{z}) \\
&= (p_x r_{11} + p_y r_{12} + p_z r_{13})\mathbf{x} + (p_x r_{21} + p_y r_{22} + p_z r_{23})\mathbf{y} + (p_x r_{31} + p_y r_{32} + p_z r_{33})\mathbf{z}.
\end{aligned}
$$

Recalling the definition of matrix-vector multiplication, this last expression can be compactly written as the following matrix-vector product

$$
{}^{A}\mathbf{p} = {}^{A}_{B}\mathbf{R}\,{}^{B}\mathbf{p}. \tag{4.7}
$$

This expression can be easily recalled by noting that the B superscript and subscript *cancel out diagonally*, thus giving a vector with the superscript A.

Example 4.2. *Let* ${}^{B}\mathbf{p} = [0\ 2\ 1]^T$ *the coordinates of point* $\mathbf{p}$ *expressed in frame* B *and let*

$$
{}^{A}_{B}\mathbf{R} = \begin{bmatrix} 0.70710678 & 0 & 0.70710678 \\ 0.61237244 & 0.5 & -0.61237244 \\ -0.35355339 & 0.8660254 & 0.35355339 \end{bmatrix}
$$

be the rotation matrix of B *with respect to* A*. The coordinates of* $\mathbf{p}$ *with respect to* A *can therefore be obtained using Eq.* (4.7) *and the result is* ${}^{A}\mathbf{p} = [0.70710678\ 0.38762756\ 2.0856042]^T$*. The reader should verify this result by doing the matrix vector multiplication.*

The last case we need to consider is the general one, i.e., the case when $B = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ is both translated and rotated with respect to $A = O - \mathbf{xyz}$ (see Figure 4.11).

In this case, B's pose with respect to A is described by both ${}^{A}O'$ and a matrix ${}^{B}_{A}\mathbf{R} \neq \mathbf{I}$. Combining the previous two results, we can then write

$$
{}^{A}\mathbf{p} = {}^{A}_{B}\mathbf{R}\,{}^{B}\mathbf{p} + {}^{A}O'. \tag{4.8}
$$

This last expression shows that for a generic change of coordinates, it is necessary to consider both a matrix and a vector. This motivates the use of a compact representation combining both of them into just one object. This is achieved with homogeneous coordinates and transformation matrices, as described in Sections 4.6 and 4.7.

Example 4.3. *With reference to Figure 4.11, let us consider the case where we have the coordinates of point* $\mathbf{p}$ *with respect to frame* B*,* ${}^{B}\mathbf{p} = [1\ 4\ 2]^T$*. Let us furthermore assume that we have the description of frame* B *with respect to frame* A*, i.e.,* ${}^{A}O' = [7\ -2\ 1]^T$ *and*

$$
{}^{A}_{B}\mathbf{R} = \begin{bmatrix} 0.70710678 & 0 & 0.70710678 \\ 0.61237244 & 0.5 & -0.61237244 \\ -0.35355339 & 0.8660254 & 0.35355339 \end{bmatrix}
$$

To determine the coordinates of point $\mathbf{p}$ *with respect to frame* A *we just need to apply Eq.* (4.8)*, obtaining* ${}^{A}\mathbf{p} = [9.12132034\ -0.61237244\ 4.81765501]^T$.

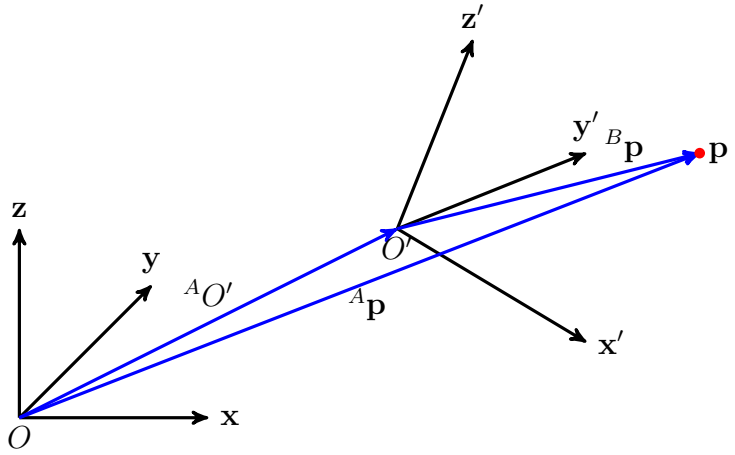


Figure 4.11: General case in which frame B is both rotated and translated with respect to A.

4.5 Rotation matrices

Rotation matrices play a very important role in many robot modeling problems, and we therefore investigate their structure in more detail. Although ${}^A_B\mathbf{R}$ features nine elements, these are subject to the six constraints given in Eq. (4.3). Consequently, the matrix is fully determined by only three independent parameters. To be a valid rotation matrix representing the orientation of a frame (or rigid body), a 3×3 matrix must satisfy the following three conditions:

1. Each of its columns has length 1.

2. Its columns are mutually orthogonal.

3. Its determinant is 1.

The first two conditions are those expressed by Eq. (4.3), whereas the last one imposes that the frame is a right-hand frame, i.e., the condition given by Eq. (4.2). A matrix satisfying the first and second properties is said to be *orthogonal*, whereas a matrix satisfying all three properties is said to be *special orthogonal*. Hence, all rotation matrices are special orthogonal.

Example 4.4. *The matrix ${}^A_B\mathbf{R}$ given in Example 4.2 is a valid rotation matrix, i.e., it is a special orthogonal matrix. Verifying the three properties given above is a simple exercise (albeit boring if done by hand). Note that when considering the dot product between the columns, the result is not exactly 0 because ${}^A_B\mathbf{R}$ has been given in approximate form.*

The following theorem states an important property of rotation matrices.

Theorem 4.1. *The transpose of a rotation matrix is equal to its inverse, i.e., if $\mathbf{R}$ is a rotation matrix, then $\mathbf{R}^T = \mathbf{R}^{-1}$.*

Proof. Since $\mathbf{R}$ is a rotation matrix, its determinant is equal to 1, and therefore it is invertible. By definition, the inverse of $\mathbf{R}$ is a matrix $\mathbf{R}^{-1}$ such that

$$\mathbf{R}^{-1}\mathbf{R} = \mathbf{R}\mathbf{R}^{-1} = \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix. Exploiting the first and second properties listed above, it is immediate to verify that

$$\mathbf{R}^T\mathbf{R} = \mathbf{I}.$$

This can be verified as follows. The generic (i, i) element of $\mathbf{R}^T\mathbf{R}$ is the dot product between the i-th row of $\mathbf{R}^T$ and the i-th column of $\mathbf{R}$. By definition of the transpose, the i-th row of $\mathbf{R}^T$ is the i-th column of $\mathbf{R}$, and therefore their dot product is 1 because of Property 1. Thus, all elements on the main diagonal are equal to 1. Similarly, the generic off-diagonal element at position (i, j) is the dot product between the i-th column of $\mathbf{R}$ and the j-th column, which is equal to 0 because of Property 2. Hence, $\mathbf{R}^T\mathbf{R}$ is the identity matrix. Similarly, one can show that

$$\mathbf{R}\mathbf{R}^T = \mathbf{I}$$

and so $\mathbf{R}^T$ is the inverse of $\mathbf{R}$. □

Combining Theorem 4.1 with Eq. (4.5), we therefore obtain

$${}^B_A\mathbf{R} = {}^A_B\mathbf{R}^T = [{}^A_B\mathbf{R}]^{-1}$$

which means that ${}^B_A\mathbf{R}$ is the inverse of the matrix ${}^A_B\mathbf{R}$.

Example 4.5. *Consider the following rotation matrix expressing the rotation of frame B with respect to A:*

$${}^A_B\mathbf{R} = \begin{bmatrix} 0.9211 & -0.3894 & 0 \\ 0.3894 & 0.9211 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The matrix expressing the rotation of A with respect to B is its inverse, i.e., ${}^B_A\mathbf{R} = [{}^A_B\mathbf{R}]^{-1}$. According to Theorem 4.1, this can be obtained by considering the transpose of ${}^A_B\mathbf{R}$, i.e.,

$${}^B_A\mathbf{R} = \begin{bmatrix} 0.9211 & 0.3894 & 0 \\ -0.3894 & 0.9211 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Remark 4.1. *With reference to Example 4.3, one may think that if we are given ${}^A\mathbf{p}$, ${}^A\mathbf{O}'$, and ${}^A_B\mathbf{R}$, we can recover ${}^B\mathbf{p}$ by applying formula* (4.8)*, using the result we just saw to set ${}^B_A\mathbf{R} = {}^A_B\mathbf{R}^T$. This is correct, but to apply the formula, we also need ${}^B\mathbf{O}$. A common mistake is assuming ${}^B\mathbf{O} = -{}^A\mathbf{O}'$. This is generally incorrect, as the reader can easily verify with a simple two-dimensional example. The formula to derive ${}^B\mathbf{O}$ from ${}^A\mathbf{O}'$ and ${}^A_B\mathbf{R}$ will be discussed in Section 4.7.5.*

4.5.1 Elementary Rotation Matrices

We begin by outlining the structure of so-called *elementary rotation* matrices, i.e., matrices representing a frame rotated about a single axis. These simple matrices can be combined to represent more complex orientations and provide insight into their structure. Consider two frames, $A = O - \mathbf{xyz}$ and $B = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$, and assume they are initially coincident, i.e., they share the same origin and have aligned axes. Next, assume we rotate frame B about its $\mathbf{x}'$ axis by an angle α (note that this is equivalent to rotating about the $\mathbf{x}$ axis because they are initially aligned). Figure 4.12 illustrates this situation, and by inspection, we can determine ${}^{A}_{B}\mathbf{R}$, i.e., the rotation matrix describing frame B after a rotation of angle α about $\mathbf{x}$.

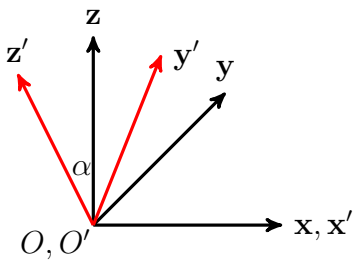


Figure 4.12: The red frame is initially coincident with the black frame and then rotates by an angle α about the $\mathbf{x}$ axis. Note that $\mathbf{x}$ coincides with $\mathbf{x}'$ throughout the motion.

Following the reasoning presented in Section 4.3, it is straightforward to determine that the coordinates of the endpoint of axis $\mathbf{x}'$ in frame A are $[1 \quad 0 \quad 0]^T$ since axes $\mathbf{x}$ and $\mathbf{x}'$ remain aligned after frame B rotates about $\mathbf{x}'$. Similarly, we can determine the coordinates of the endpoint of axis $\mathbf{y}'$ in frame A. Its projection along axis $\mathbf{x}$ is 0, whereas, from elementary trigonometry, its projection along axis $\mathbf{y}$ is $\cos\alpha$ and its projection along axis $\mathbf{z}$ is $\sin\alpha$. Hence, its coordinates in frame A are $[0 \quad \cos\alpha \quad \sin\alpha]^T$. Applying the same reasoning to axis $\mathbf{z}'$ shows that its coordinates in frame A after the rotation are $[0 \quad -\sin\alpha \quad \cos\alpha]^T$. Combining these three columns, as in Eq. (4.4), we obtain the matrix $\mathbf{R}_x(\alpha)$, where the notation indicates that it is the matrix obtained by rotating about the $\mathbf{x}$ axis by an angle α:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}. \tag{4.9}$$

Following the same reasoning, we can derive expressions for the elementary rotation matrices about the $\mathbf{y}$ and $\mathbf{z}$ axes. Figure 4.13 illustrates these rotations, while Eqs. (4.10)-(4.11) provide the expressions for the associated matrices:

$$\mathbf{R}_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{4.10}$$

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{4.11}$$

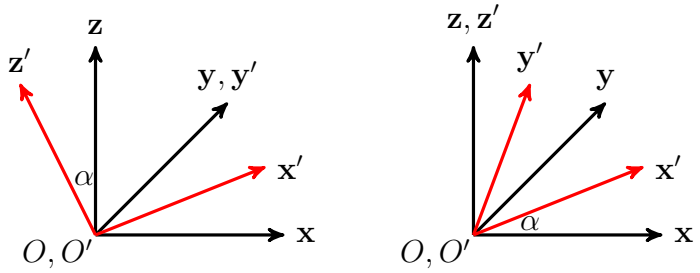


Figure 4.13: Left: Rotation about the **y** axis. Right: Rotation about the **z** axis.

The reader can verify that these are indeed rotation matrices, i.e., they satisfy the three properties outlined at the beginning of this section.

4.5.2 Composite rotations

We have just derived expressions for elementary rotation matrices obtained through a single rotation about one of the three axes. A rigid body is, however, not restricted to rotate about the main axes only, and it is therefore of interest to derive rotation matrices to describe generic rotations. To do so, we start by considering the rotation matrix describing a frame that undergoes two successive rotations. The first fundamental aspect to observe is that rotations do not commute. This means the following: consider a rigid body and assume its attached frame B is initially aligned with a frame $A = O - \mathbf{xyz}$. Next, consider the following two scenarios. First, rotate B by an angle α about $\mathbf{x}$ and then by an angle β about $\mathbf{z}$. Call ${}^A_B\mathbf{R}$ the rotation matrix obtained after these two rotations. Next, assume B and A are again aligned, but now rotate B first by an angle β about $\mathbf{z}$ and then by an angle α about $\mathbf{x}$, i.e., invert the order of the rotations. Call ${}^A_B\mathbf{R}'$ the rotation matrix obtained after these two rotations. In general, ${}^A_B\mathbf{R} \neq {}^A_B\mathbf{R}'$. This is what is meant by the expression *rotations do not commute*, i.e., the order matters and cannot be inverted (recall the commutative property of the operations sum or product.) Figure 4.14 illustrates this fact.

The top row shows a frame that undergoes two rotations. The leftmost figure shows the frame in its initial pose. The second frame (middle) shows the frame obtained after a 90-degree rotation about the **z** axis. The last frame is obtained from the second one after a 90-degree rotation about the **y** axis. Note that in this last rotation, we consider the **y** axis of the original frame on the left. In this case, we say that rotations are performed about the fixed initial frame. The second row shows what happens if we swap the order of rotations, i.e., we first rotate 90 degrees about the (fixed) **y** axis and then 90 degrees about the fixed **z** axis. The two frames at the end are different. When considering composite rotations, one could also use rotations about the moving axis. In this case, too, the order matters. This is illustrated in Figure 4.15. This is in contrast to translations, which instead commute (e.g., translating first along **x** and then along **y** is the same as translating first along **y** and then along **x**.)

We next consider the problem of explicitly computing the rotation matrix representing two successive rotations. We start with the case where all rotations are performed about the fixed frame. As in the previous example, assume we have two initially aligned frames A

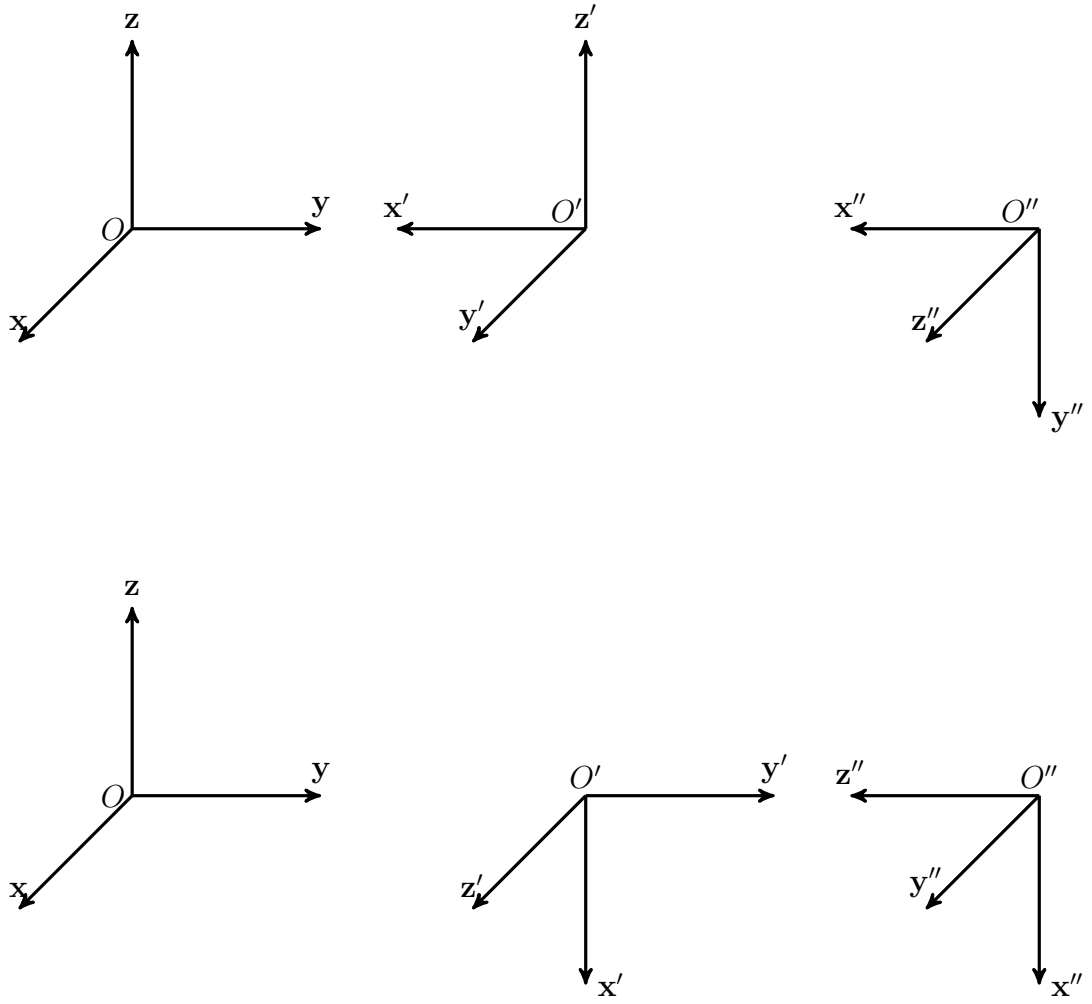


Figure 4.14: Two initially overlapping frames undergoing two sequences of rotations about the fixed axis. The top frame rotates first 90 degrees about **z** and then 90 degrees about **y**. The bottom frame rotates first 90 degrees about **y** and then 90 degrees about **z**.

and B, with A being the fixed frame and B being the moving frame. Next, assume that B rotates by an angle α about $\mathbf{x}$ and then by an angle β about $\mathbf{z}$, where $\mathbf{x}$ and $\mathbf{z}$ are the axes of the fixed frame A. What is the expression for ${}^A_B\mathbf{R}$? Let us consider the two steps separately. Since initially A and B are aligned, after the first rotation we obtain an intermediate rotation matrix ${}^A_B\mathbf{R}'$, which is by definition $\mathbf{R}_x(\alpha)$ as per Eq. (4.9). To determine the expression of the final result, it is useful to recall that the columns of a rotation matrix expressed with respect to a given frame give the coordinates of the axes of the frame associated with the matrix expressed in the reference frame (see Eq. (4.4)). Applying a rotation to the frame is equivalent to rotating each of its axes. Therefore, to compute how a generic rotation matrix $\mathbf{R}'$ changes when a rotation $\mathbf{R}$ is applied, it is sufficient to apply $\mathbf{R}$ to the axes of $\mathbf{R}'$. But this is something we have determined already, and it is given by Eq. (4.7). Therefore, the expression for the final matrix is given by

$$ {}^A_B\mathbf{R} = \mathbf{R}_z(\beta)\mathbf{R}_x(\alpha). $$

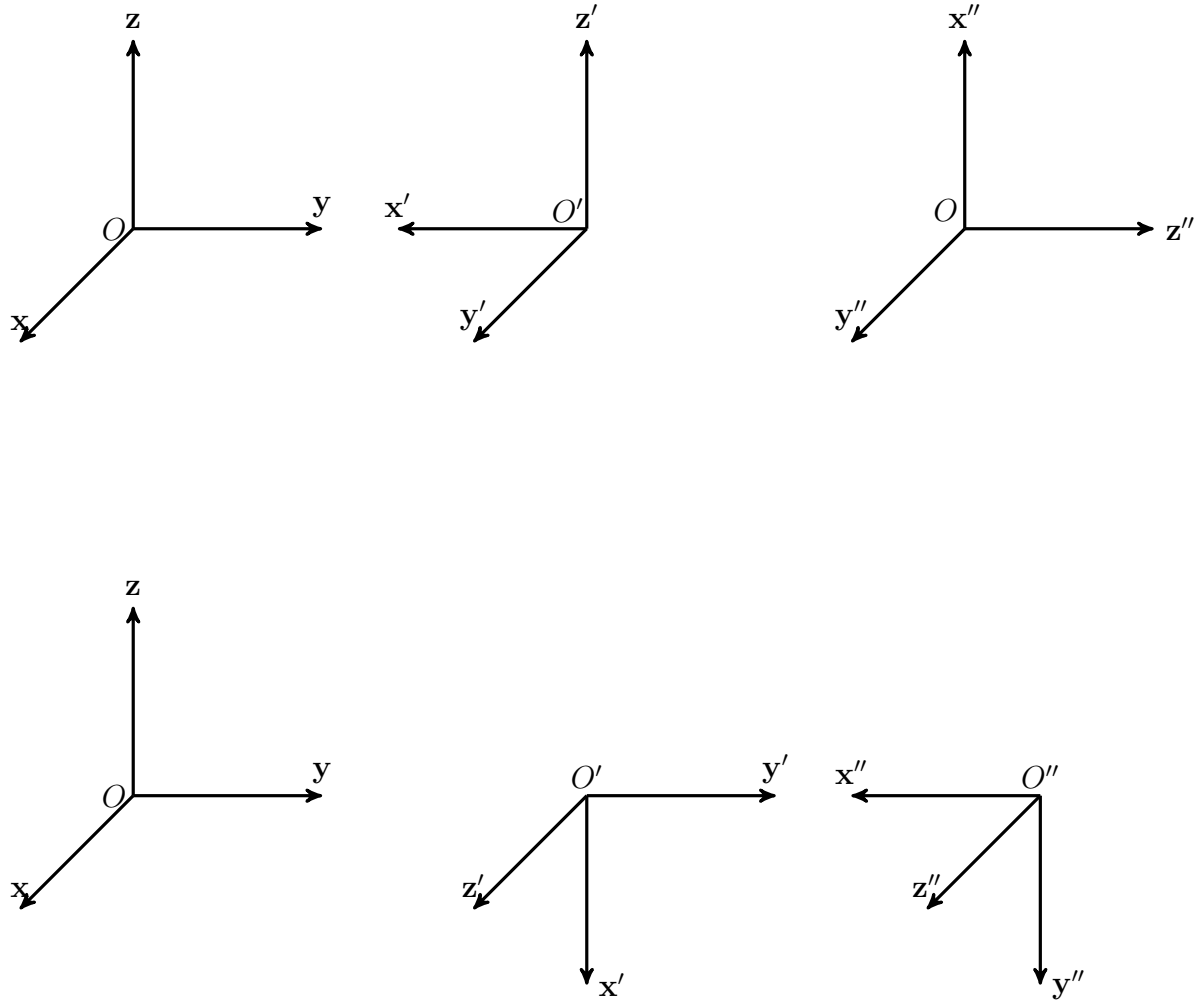


Figure 4.15: Two initially overlapping frames undergoing two sequences of rotations about the moving axis. The top frame rotates first 90 degrees about $\mathbf{z} = \mathbf{z}'$ and then 90 degrees about the moving axis $\mathbf{y}'$. The bottom frame rotates first 90 degrees about $\mathbf{y} = \mathbf{y}'$ and then 90 degrees about $\mathbf{z}'$.

This expression can be generalized as follows. Let A and B be two initially coincident frames, and let B undergo a sequence of n rotations $\mathbf{R}_1, \mathbf{R}_2 \dots \mathbf{R}_n$, all expressed about the fixed frame A. Note that the order is important, i.e., the first rotation is represented by $\mathbf{R}_1$, the second by $\mathbf{R}_2$, and so on. Then the final expression for B expressed in reference A is

$$ {}^A_B\mathbf{R} = \mathbf{R}_n \dots \mathbf{R}_2 \mathbf{R}_1. \tag{4.12} $$

Observe the swapped order in the product, i.e., the rightmost rotation in the multiplication is the first one, whereas the leftmost rotation is the last one. Consistent with our previous observations, the order matters both when considering composite rotations and when considering the associated matrix multiplication (which is not commutative.) Following a slightly more complicated reasoning, it is similarly possible to determine the expression of ${}^A_B\mathbf{R}$ when performing n successive rotations $\mathbf{R}_1$, $\mathbf{R}_2, \dots \mathbf{R}_n$ about the moving axis. In this

case, the final result is

$$ {}^{A}_{B}\mathbf{R} = \mathbf{R}_1\mathbf{R}_2 \dots \mathbf{R}_n. \tag{4.13} $$

Comparing Eq. (4.12) with Eq. (4.13), note that the order is swapped, i.e., when rotations are about the moving axes, the matrices are post-multiplied.

Example 4.6. *Let A and B be two initially coincident frames. Frame B rotates first by an angle α about the fixed axis $\mathbf{x}$, then by an angle β about the moving axis $\mathbf{z}$, and finally by an angle γ about the fixed axis $\mathbf{y}$. We want to determine the final rotation matrix describing B with respect to frame A.*

This case is only slightly more complicated because the rotations alternate between fixed and moving axes. Initially, A and B are coincident, so the initial transformation matrix is the identity. Let ${}^{A}_{B}\mathbf{B}'$, ${}^{A}_{B}\mathbf{B}''$, and ${}^{A}_{B}\mathbf{B}'''$ be the three orientation matrices obtained after the successive rotations. Trivially, the first intermediate transformation matrix is:

$$ {}^{A}_{B}\mathbf{B}' = \mathbf{R}_x(\alpha). $$

This matrix is then rotated about the moving axis $\mathbf{z}$, and therefore we post-multiply by the rotation matrix $\mathbf{R}_z(\beta)$:

$$ {}^{A}_{B}\mathbf{B}'' = {}^{A}_{B}\mathbf{B}'\mathbf{R}_z(\beta) = \mathbf{R}_x(\alpha)\mathbf{R}_z(\beta). $$

Finally, we perform one more rotation about the fixed axis $\mathbf{y}$, and thus we pre-multiply by the rotation matrix $\mathbf{R}_y(\gamma)$:

$$ {}^{A}_{B}\mathbf{B}''' = \mathbf{R}_y(\gamma) \cdot {}^{A}_{B}\mathbf{B}'' = \mathbf{R}_y(\gamma)\mathbf{R}_x(\alpha)\mathbf{R}_z(\beta). $$

The example discussed above shows how to compose an arbitrary number of rotations about either fixed or moving axes. Each time a new rotation is introduced, it either pre-multiplies or post-multiplies the previously accumulated result, depending on whether the rotation is about a fixed or moving axis.

This example also illustrates an important point: the order of rotations and the axes about which they are performed (fixed vs. moving) significantly affect the outcome. Since matrix multiplication is not commutative, the order of rotation matrices in the product matters and ultimately determines the final orientation.

We now conclude this section with an important theorem about the multiplication of rotation matrices:

Theorem 4.2. *Let ${}^{A}_{B}\mathbf{R}$ and ${}^{B}_{C}\mathbf{R}$ be rotation matrices describing the orientation of frame B with respect to frame A, and frame C with respect to frame B, respectively. Then,*

$$ {}^{A}_{C}\mathbf{R} = {}^{A}_{B}\mathbf{R} \cdot {}^{B}_{C}\mathbf{R}, \tag{4.14} $$

i.e., their product gives the orientation of frame C with respect to frame A.

Proof. Each column of ${}^{B}_{C}\mathbf{R}$ gives the coordinates of the unit vectors of frame C expressed in frame B, assuming that the origins of the two frames coincide. This is precisely how rotation

matrices were defined in Eq. (4.4). Now, observe that each column of ${}^A_C\mathbf{R}$ is obtained by multiplying ${}^A_B\mathbf{R}$ by the corresponding column of ${}^B_C\mathbf{R}$. This operation, by Eq. (4.7), corresponds to changing coordinates from frame B to frame A, thus correctly producing the rotation of frame C with respect to frame A. □

This theorem is easy to remember by noting that the common frame (in this case B) cancels diagonally when chaining multiple rotations. This is the same principle introduced in Eq. (4.7).

4.5.3 Rotations parametrization

We previously observed that although a rotation is defined by nine numbers, due to the six constraints expressed in Eq. (4.2) and Eq. (4.3), these numbers cannot be arbitrarily chosen. In fact, only three of them can be independently selected. It therefore makes sense to try to parameterize a generic rotation matrix using three parameters related to simpler rotations, like the elementary rotation matrices about the $\mathbf{x}, \mathbf{y}, \mathbf{z}$ axes. Using the composition rules we just defined, it is straightforward to study this problem.

First, let us assume that all rotations are about the moving axes. Therefore, one can come up with twelve different combinations of rotations. The number twelve arises from the fact that the first axis can be any of the three axes, but the second one can only be chosen from two, as it cannot be the same as the first; otherwise, the second rotation would be along the same axis as the first rotation, making it equivalent to just one rotation rather than two.[5] For the same reason, the third rotation must be picked from the two remaining axes (different from the second rotation's axis), resulting in a total of $3 \times 2 \times 2 = 12$ possible combinations. Specifically, these are XYZ, XYX, XZY, XZX, YXZ, YXY, YZX, YZY, ZXZ, ZXY, ZYZ, and ZYX, where, for example, XYX means that we first rotate about $\mathbf{x}$, then about $\mathbf{y}$ and then again about $\mathbf{x}$. The other combinations can be interpreted in a similar way.

These rotations about the moving axes are all valid, but in practice, only a few are commonly used. In particular, the ZYZ combination is the most frequently used. The corresponding triplet of angles is also known as the Euler angles. Assuming that the rotations are of an angle α about Z, angle β about Y, and angle γ about Z (always about the moving axes), the final expression is immediately obtained by applying Eq. (4.13):

$$\mathbf{R} = \mathbf{R}_z(\alpha)\mathbf{R}_y(\beta)\mathbf{R}_z(\gamma). \tag{4.15}$$

Given the three angles, it is then immediate to compute the matrix $\mathbf{R}$. From this expression, it is natural to ask the inverse question, i.e., given a valid rotation matrix $\mathbf{R}$, how can we determine the angles α, β, and γ so that $\mathbf{R}$ can be obtained through a ZYZ transformation as per Eq. (4.15)? More precisely, let the given matrix $\mathbf{R}$ be

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{4.16}$$

[5] For example, a rotation about $\mathbf{x}$ of α followed by a rotation about $\mathbf{x}$ of β is equivalent to just one rotation about $\mathbf{x}$ of $\alpha + \beta$.

Through a somewhat tedious trigonometric exercise, the answer can be determined, and below, we provide only the result (see the references at the end of the chapter, particularly [49] for details and the full derivation).

The first angle, α, is

$$\alpha = \text{atan2}(r_{23}, r_{13}),$$

where atan2 is the inverse of the tangent function that takes into account the signs of its two arguments. The second angle is

$$\beta = \text{atan2}\left(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right),$$

and the third one is

$$\gamma = \text{atan2}(r_{32}, -r_{31}).$$

It is immediately apparent that the solution to this problem is not unique, as we arbitrarily chose the positive square root for β, thereby restricting β to the range $[0, \pi)$. If, instead, we want β to be in the range $(-\pi, 0]$, then the following formulas can be used (note that the formula for α remains unchanged):

$$\beta = \text{atan2}\left(-\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right) \qquad \gamma = \text{atan2}(-r_{32}, r_{31}).$$

It is easy to verify that the two solutions are equivalent, meaning they produce the same rotation matrix. This result is unsurprising, given the periodic nature of the trigonometric functions involved in defining the elementary rotation matrices.

Example 4.7. *Determine the ZYZ Euler angles for the following rotation matrix:*

$$\mathbf{R} = \begin{bmatrix} -0.0474 & -0.7891 & 0.6124 \\ 0.6597 & 0.4356 & 0.6124 \\ -0.7500 & 0.4330 & 0.5000 \end{bmatrix}$$

We simply need to apply the formulas we just derived. First, we compute α:

$$\alpha = \text{atan2}(r_{23}, r_{13}) = \text{atan2}(0.6124, 0.6124) = \pi/4.$$

Next, we determine β in the range $[0, \pi)$:

$$\beta = \text{atan2}\left(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right) = \text{atan2}\left(\sqrt{0.6124^2 + 0.6124^2}, 0.5000\right) = \pi/3.$$

Finally, we determine γ:

$$\gamma = \text{atan2}(r_{32}, -r_{31}) = \text{atan2}(0.4330, 0.7500) = \pi/6.$$

The alternative solution, with β in the range $(-\pi, 0]$, is

$$\beta = \text{atan2}\left(-\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right) = \text{atan2}\left(-\sqrt{0.6124^2 + 0.6124^2}, 0.5000\right) = -\pi/3.$$

and

$$\gamma = \text{atan2}(-r_{32}, r_{31}) = \text{atan2}(-0.4330, 0.7500) = -5\pi/6.$$

When defining the Euler angles, we assumed that all rotations were about the moving axes. Equivalently, one could assume that all rotations are about the fixed axes, in which case twelve combinations could also be obtained. Among them, the ZYX sequence is the most commonly used in practice and is related to the *roll-pitch-yaw* angles often used in aeronautics—see Figure 4.16. Note that when considering mobile robots, the attached frame is typically assigned as shown in Figure 4.16, i.e., the $\mathbf{x}$ axis points forward, $\mathbf{y}$ points to the left, and $\mathbf{z}$ points upwards (see Figure 4.17).

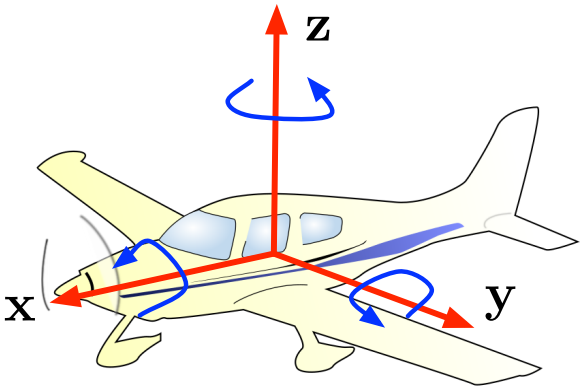


Figure 4.16: Axes orientation for the roll-pitch-yaw angles. (plane clipart from openclipart.org).

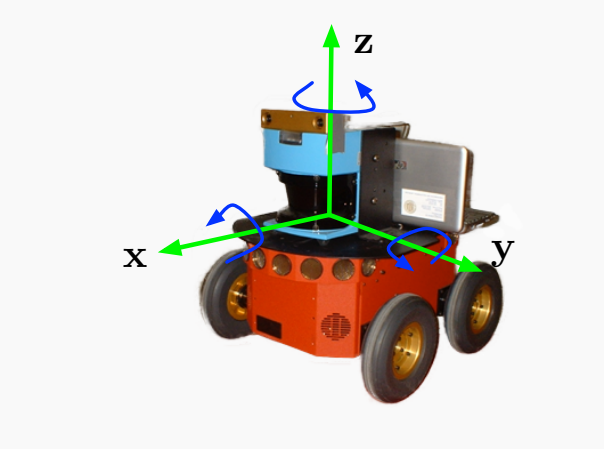


Figure 4.17: Axes orientation for the roll-pitch-yaw angles for a mobile robot.

To be precise, the roll-pitch-yaw sequence consists of a rotation α about the $\mathbf{x}$ axis (roll), followed by a rotation β about the $\mathbf{y}$ axis (pitch), and then a rotation γ about the $\mathbf{z}$ axis (yaw). Since all rotations are about the fixed frame, we apply Eq. (4.12), and we therefore obtain (note the order):

$$\mathbf{R} = \mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha).$$

As in the ZYZ case, it is interesting to solve the inverse problem, i.e., given a rotation matrix $\mathbf{R}$, determine the roll-pitch-yaw angles that produce the given matrix. Once again,

the solution can be obtained through trigonometric manipulations, and it is not unique (see [49] for the complete derivation.). The first solution is

$$\alpha = \text{atan2}(r_{32}, r_{33}) \qquad \beta = \text{atan2}\left(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}\right) \qquad \gamma = \text{atan2}(r_{21}, r_{11}),$$

which limits β to the range $(-\pi/2, \pi/2)$. The second equivalent solution is again obtained by considering the negative square root for β, given by

$$\alpha = \text{atan2}(-r_{32}, -r_{33}) \qquad \beta = \text{atan2}\left(-r_{31}, -\sqrt{r_{32}^2 + r_{33}^2}\right) \qquad \gamma = \text{atan2}(-r_{21}, -r_{11}).$$

In this case, the angle β is in the range $(\pi/2, 3\pi/2)$.

Example 4.8. *Determine the roll-pitch-yaw angles for the following rotation matrix:*

$$\mathbf{R} = \begin{bmatrix} 0.5721 & 0.0064 & 0.8202 \\ 0.5721 & 0.7135 & -0.4046 \\ -0.5878 & 0.7006 & 0.4045 \end{bmatrix}$$

As in the previous example, all we need to do is apply the formulas we just provided. The first solution is

$$\alpha = \text{atan2}(r_{32}, r_{33}) = \text{atan2}(0.7006, 0.4045) = \pi/3,$$

$$\beta = \text{atan2}\left(-r_{31}, -\sqrt{r_{32}^2 + r_{33}^2}\right) = \text{atan2}\left(0.5878, -\sqrt{0.7006^2 + 0.4045^2}\right) = \pi/5,$$

$$\gamma = \text{atan2}(r_{21}, r_{11}) = \text{atan2}(0.5721, 0.5721) = \pi/4.$$

The second solution is obtained using the other set of formulas and yields

$$\alpha = \pi/3, \qquad \beta = 4\pi/5, \qquad \gamma = -3\pi/4.$$

Gimbal Lock

The rotation parametrizations presented in the previous subsection share a common problem that may go unnoticed at first. Let us consider the ZYZ case and explicitly write the expression in Eq. (4.15). Plugging in Eqs. (4.10) and (4.11) and working out the product, we obtain:

$$\mathbf{R} = \begin{bmatrix} \cos\alpha\cos\beta\cos\gamma - \sin\alpha\sin\gamma & -\cos\alpha\cos\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta \\ \sin\alpha\cos\beta\cos\gamma + \cos\alpha\sin\gamma & -\sin\alpha\cos\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta \\ -\sin\beta\cos\gamma & \sin\beta\sin\gamma & \cos\beta \end{bmatrix}. \quad (4.17)$$

Next, compare Eq. (4.16) with Eq. (4.17), and consider the formulas to determine the ZYZ Euler angles (say, the first solution). If $\sin\beta = 0$, then $r_{13} = r_{23} = r_{31} = 0$. $\sin\beta = 0$ happens when $\beta = k\pi$, with k being an arbitrary integer. In this case, the solution to the inverse problem cannot be found; that is, it is not possible to determine both α and γ, but only their sum. This is somewhat expected, because if $\beta = 0$ (or $\beta = k\pi$ for an arbitrary

integer), then the first and third rotations both occur about the same axis $\mathbf{z}$, since the second rotation did not change the orientation of the Z axis. Therefore, the final result is indeed the result of an overall rotation of $\alpha+\gamma$ about Z. A similar situation can occur when considering the roll-pitch-yaw triplet of angles. More generally, this problem arises whenever a triplet of angles is used to parametrize a rotation in three dimensions. This issue is known as *gimbal lock* or *singularity*. It occurs when the axis of the second rotation becomes parallel to either the axis of the first or the third rotation. This problem was already observed by Euler,[6] who stated a theorem asserting that any rotation matrix can be obtained as a sequence of no more than three rotations about three axes, where no two successive rotations are about the same axis. To overcome this problem, a non-minimal representation relying on four parameters (rather than three) is discussed next.

4.5.4 Representing rotations with quaternions

Quaternions offer an alternative way to represent rotations in space that overcomes the singularity problem and many other issues arising when dealing with rotations parametrized by three angles. The price to pay is the use of a non-minimal representation, i.e., relying on four rather than three parameters. The representation is also not unique, i.e., there is no one-to-one correspondence between quaternions and rotation matrices. Both of these issues, however, are minor when compared with the advantages. It should be noted that quaternions are a rich algebraic structure that was introduced for other purposes, and there exists a body of literature much broader than what we will cover here. In a sense, using quaternions to represent three-dimensional rotations can be seen as a useful side effect, but not the primary objective. The starting point is Euler's rotation theorem, which can be stated as follows.

Theorem 4.3 (Euler's rotation theorem)**.** *In three dimensions, every rotation is equivalent to a single rotation about an axis through the origin.*

This result has profound consequences, and it is useful to compare it with the previous rotation parametrizations obtained through three successive rotations about three axes. This way of representing rotations is also known as the *axis-angle* representation, and Figure 4.18 illustrates the idea. The rotation angle is θ, and the rotation axis is $\mathbf{r} = [r_1\ r_2\ r_3]^T$, where its coordinates are expressed with respect to the world frame $O - \mathbf{xyz}$.

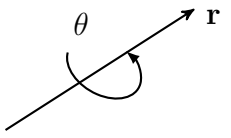


Figure 4.18: Interpretation of Euler's theorem (every rotation is equivalent to a single rotation about an axis through the origin).

A quaternion can be conveniently thought of as an extension of a complex number, featuring one real part and three imaginary parts. Without getting into algebraic technicalities,

[6] This theorem is sometimes referred to as Euler's theorem, but this name makes little sense because multiple original results were proved by Euler, and so there are many *Euler's theorems*. Its correct name would be Euler's rotation theorem.

a quaternion $\mathbf{q}$ is represented as $\mathbf{q} = a+bi+cj+dk$, where a, b, c, d are real numbers and i, j, k are imaginary components subject to the following constraints: $i^2 = j^2 = k^2 = ijk = -1$. Note that a real number can be seen as a quaternion for which $b = c = d = 0$, and a complex number can be seen as a quaternion in which $c = d = 0$. Therefore, in a sense, quaternions generalize both real and complex numbers. In the following, we will focus on *unit-length* quaternions, where the length (or norm) of a quaternion is $|\mathbf{q}| = \sqrt{a^2 + b^2 + c^2 + d^2}$. As the name suggests, a unit quaternion is a quaternion whose length is one. Without getting into the details of its derivation, there exists a mapping between unit quaternions and rotation matrices. To be precise, let $\mathbf{q} = a + bi + cj + dk$ be a unit quaternion. Its associated rotation matrix is

$$\mathbf{R}(\mathbf{q}) = \begin{bmatrix} 2(a^2+b^2)-1 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & 2(a^2+c^2)-1 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & 2(a^2+d^2)-1 \end{bmatrix} \tag{4.18}$$

In the following, it is convenient to write $\mathbf{R}(\mathbf{q})$ for the matrix obtained from $\mathbf{q}$ by applying Eq. (4.18). It is easy (albeit perhaps tedious) to verify that this is indeed a rotation matrix, i.e., it is special orthogonal. Therefore, unit quaternions represent rotations, and Eq. (4.18) shows how to map a unit quaternion into a rotation matrix. Given a unit quaternion $\mathbf{q} = a + bi + cj + dk$, it is interesting to determine its parameters according to Euler's theorem, i.e., the axis of rotation and the angle (see Figure 4.18). The angle θ is given by

$$\theta = 2 \arccos a. \tag{4.19}$$

The rotation axis $\mathbf{r} = [r_1 \; r_2 \; r_3]^T$ is:

$$r_1 = \frac{b}{\sin\frac{\theta}{2}} \qquad r_2 = \frac{c}{\sin\frac{\theta}{2}} \qquad r_3 = \frac{d}{\sin\frac{\theta}{2}} \tag{4.20}$$

Note that these expressions are not defined when $\sin\frac{\theta}{2} = 0$, i.e., when $\theta = 0$ or $\theta = 2\pi$. This actually makes sense, because for these two values there is no rotation at all. The opposite transformation can therefore be defined as well, i.e., given θ and a unit vector $\mathbf{r} = [r_1 \; r_2 \; r_3]^T$, the associated quaternion is

$$\mathbf{q} = \cos\frac{\theta}{2} + i \sin\frac{\theta}{2} r_1 + j \sin\frac{\theta}{2} r_2 + k \sin\frac{\theta}{2} r_3 \tag{4.21}$$

Representing a rotation with quaternions has one slight problem, i.e., the mapping between quaternions and rotations is not unique. In fact, it is an easy exercise to verify that if $\mathbf{q}$ is a unit quaternion, then $-\mathbf{q}$ is also a unit quaternion, and Eq. (4.18) maps them to the same rotation matrix $\mathbf{R}$. According to Euler's theorem, the interpretation is that a rotation of θ about an axis $\mathbf{r}$ is equivalent to a rotation of $2\pi - \theta$ about the axis $-\mathbf{r}$. One last question we may ask when considering the mapping between quaternions and rotation matrices is the inverse of Eq. (4.18), i.e., given a rotation matrix $\mathbf{R}$, determine a unit quaternion $\mathbf{q}$ such that $\mathbf{R}(\mathbf{q}) = \mathbf{R}$. This is obtained with the following formulas [29]:

$$a = \frac{\sqrt{r_{11} + r_{22} + r_{33} + 1}}{2} \qquad b = \frac{r_{32} - r_{23}}{4a} \qquad c = \frac{r_{13} - r_{31}}{4a} \qquad d = \frac{r_{21} - r_{12}}{4a} \tag{4.22}$$

Given two quaternions $\mathbf{q}_1$ and $\mathbf{q}_2$, it is immediate to define the sum and product operations by treating them as polynomials in i, j, k and applying the rules defined earlier for terms like i^2, and so on. Moreover, given a quaternion $\mathbf{q} = a + bi + cj + dk$, we define its *conjugate* as $\mathbf{q}^* = a - bi - cj - dk$, i.e., the quaternion obtained by inverting the signs of the imaginary coefficients. The conjugate of a quaternion is useful for defining the operation of rotating a point, similarly to what we did for rotation matrices (see Eq. (4.7)). Let $\mathbf{q}$ be a unit quaternion defining a rotation of θ about the axis $\mathbf{r}$, and let $\mathbf{p} = [p_x \ p_y \ p_z]^T$ be a point. The point obtained by rotating $\mathbf{p}$ using the rotation defined by $\mathbf{q}$ can be computed as follows. Let $\mathbf{p}' = 0 + p_x i + p_y j + p_z k$ be a quaternion defined using the components of $\mathbf{p}$. Then we compute

$$\mathbf{p}'' = \mathbf{q}\mathbf{p}'\mathbf{q}^* = p'' + p''_x i + p''_y j + p''_z k \tag{4.23}$$

The rotated point is obtained by taking the imaginary components of $\mathbf{p}''$, i.e., the rotated point is $\mathbf{p}_r = [p''_x \ p''_y \ p''_z]^T$. Note, moreover, that it can be shown that in this product, the first coefficient p'' is always zero because of the way we defined $\mathbf{p}'$.

Example 4.9. *Let* $\mathbf{p} = [1 \ 2 \ 3]^T$*, and let* $\mathbf{R}$ *be the following rotation matrix*

$$\mathbf{R} = \begin{bmatrix} 0.23859519 & -0.67245146 & 0.70062927 \\ 0.49326748 & 0.7053854 & 0.50903696 \\ -0.8365163 & 0.22414387 & 0.5 \end{bmatrix}$$

Use quaternions to determine the point obtained applying the given rotation to $\mathbf{p}$*.*

We start by using Eq. (4.21) *to determine the quaternion* $\mathbf{q}$ *associated with* $\mathbf{R}$*. This gives* $\mathbf{q} = 0.782 - 0.091i + 0.492j + 0.373k$*, and its conjugate is* $\mathbf{q}^* = 0.782 + 0.091i - 0.492j - 0.373k$*. The quaternion associated with the point* $\mathbf{p}$ *is* $\mathbf{p}' = 0 + 1i + 2j + 3k$*. We then apply Eq.* (4.23) *and obtain the quaternion* $0 + 0.996i + 3.431j + 1.112k$*, which means that the coordinates of the rotated point are* $[0.996 \ 3.431 \ 1.112]^T$*. It is easy to verify (and a useful exercise, too) that the same result is obtained using the matrix-vector multiplication* $\mathbf{R}\mathbf{p}$*.*

4.6 Homogeneous coordinates

In section 4.3, we have shown how a frame can be represented by a vector in $\mathbb{R}^3$ and a rotation matrix. Together, they identify the location of the origin of the frame and its orientation. Using two separate mathematical objects to represent a single entity is error-prone from a practical perspective. For example, when computing changes of coordinates, it is easy to make mistakes by mixing together the origins and orientations of different frames. One way to overcome this problem is by introducing so-called *homogeneous coordinates*. Homogeneous coordinates offer a unified representation for points and directions and lead to the definition of transformation matrices, i.e., a single matrix to represent both the origin and orientation of a frame (see section 4.7). A point in $\mathbb{R}^3$ can be represented in homogeneous coordinates as a point in $\mathbb{R}^4$ whose last coordinate is 1. That is to say, if $\mathbf{p} = [x \ y \ z]^T$ is a point in three dimensions, then its representation in homogeneous coordinates is given by the four-dimensional vector $\mathbf{p} = [x \ y \ z \ 1]^T$. Note that we use the same symbol $\mathbf{p}$ when referring to both the canonical representation in $\mathbb{R}^3$ and the representation using homogeneous coordinates.

Example 4.10. *Let* $\mathbf{p} = [3\ 1\ 5]^T$. *Its representation in homogeneous coordinates is* $\mathbf{p} = [3\ 1\ 5\ 1]^T$.

Next, let us consider how to represent directions using homogeneous coordinates, and in particular, let us examine how to represent the directions that express the orientation of a frame with respect to another one. At the beginning of the chapter, we saw that these axes can be expressed by three orthonormal vectors in $\mathbb{R}^3$. In this case, when switching to homogeneous coordinates, we set the fourth coordinate to 0. This is clarified in the following example.

Example 4.11. *The orientation of frame* $B - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ *with respect to frame* $A - \mathbf{xyz}$ *is given by the following rotation matrix:*

$$\mathbf{R} = \begin{bmatrix} 0.5721 & 0.0064 & 0.8202 \\ 0.5721 & 0.7135 & -0.4046 \\ -0.5878 & 0.7006 & 0.4045 \end{bmatrix}$$

If we want to express the three directions in homogeneous coordinates, we must recall that the first column represents the coordinates of a unit vector along the direction $\mathbf{x}'$*, expressed in frame* $A - \mathbf{xyz}$*. The second column represents the coordinates of a unit vector along the direction* $\mathbf{y}'$*, and the third column represents the coordinates of a unit vector along the direction* $\mathbf{z}'$ *(both expressed in frame* $A - \mathbf{xyz}$*). Therefore, the direction* $\mathbf{x}'$ *expressed in homogeneous coordinates is* $\mathbf{x}' = [0.5721\ 0.5721\ -0.5878\ 0]^T$*. Similarly, the direction* $\mathbf{y}'$ *expressed in homogeneous coordinates is* $\mathbf{y}' = [0.0064\ 0.7135\ 0.7006\ 0]^T$*, and the direction* $\mathbf{z}'$ *expressed in homogeneous coordinates is* $\mathbf{z}' = [0.8202\ -0.4046\ 0.4045\ 0]^T$*. In matrix form, the result is the following* 4×3 *matrix*

$$\mathbf{R} = \begin{bmatrix} 0.5721 & 0.0064 & 0.8202 \\ 0.5721 & 0.7135 & -0.4046 \\ -0.5878 & 0.7006 & 0.4045 \\ 0 & 0 & 0 \end{bmatrix}$$

where we again use the same symbol $\mathbf{R}$ *for both the canonical representation and the representation using homogeneous coordinates.*

In summary, when using homogeneous coordinates, if the last coordinate is 1, we are representing a point, whereas if the last coordinate is 0, we are representing a direction. Note that in other contexts (e.g., computer graphics), other values for the last coordinate are also used, which can be interpreted as *scaling*. However, we will not need this feature.

4.7 Transformation matrices

We start by defining transformation matrices from a formal standpoint, and we will later study their numerous interpretations.

Definition 4.3. *Let* $A - \mathbf{xyz}$ *and* $B - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ *be two frames. Let* ${}^A\mathbf{p} = [p_x\ p_y\ p_z]^T$ *be the coordinates of the origin of* $B - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ *expressed in frame* A*, and let* ${}^A_B\mathbf{R}$ *be the rotation*

matrix expressing the orientation of frame B with respect to A. Then, the transformation matrix *expressing frame $B-\mathbf{x'y'z'}$ with respect to frame $A-\mathbf{xyz}$ is the 4×4 matrix defined as follows:*

$$ {}^{A}_{B}\mathbf{T} = \begin{bmatrix} & & & p_x \\ & {}^{A}_{B}\mathbf{R} & & p_y \\ & & & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

The above definition simply describes how a 4×4 matrix can be built starting from a three-dimensional vector and a rotation matrix. Observe that, for a transformation matrix, the elements of the last row are always the same, i.e., [0 0 0 1]. If we analyze the structure of ${}^{A}_{B}\mathbf{T}$ in the context of homogeneous coordinates, a meaningful interpretation immediately emerges. The last column is nothing but ${}^{A}\mathbf{p}$ expressed in homogeneous coordinates. Similarly, the first three columns represent the directions of ${}^{A}_{B}\mathbf{R}$ in homogeneous coordinates as well. That is to say, ${}^{A}_{B}\mathbf{T}$ is obtained by combining ${}^{A}_{B}\mathbf{R}$ and ${}^{A}\mathbf{p}$ after converting them to homogeneous coordinates. From a conceptual standpoint, this is convenient because we now use a single object (${}^{A}_{B}\mathbf{T}$) instead of two (${}^{A}_{B}\mathbf{R}$ and ${}^{A}\mathbf{p}$) to represent frame B with respect to A. However, this is not the main advantage. As we will see in the following, transformation matrices can be associated with both operands and operators that can be used to solve various problems arising in spatial representations. Each of these interpretations is presented in the following.

4.7.1 Transformation matrices represent frames

Based on the previous discussion, it is immediate to see that a transformation matrix can be used to represent a frame because it incorporates both the origin and the orientation.

4.7.2 Transformation matrices are operators to transform points and directions

Let $\mathbf{T}$ be a transformation matrix, and let $\mathbf{p}$ be a point expressed in homogeneous coordinates. From an algebraic standpoint, we can consider the product $\mathbf{Tp}$ between the 4×4 matrix $\mathbf{T}$ and the 4×1 vector $\mathbf{p}$. By explicitly working out the math, an interesting interpretation emerges:

$$ \mathbf{Tp} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11}p_x + r_{12}p_y + r_{13}p_z + t_x \\ r_{21}p_x + r_{22}p_y + r_{23}p_z + t_y \\ r_{31}p_x + r_{32}p_y + r_{33}p_z + t_z \\ 1 \end{bmatrix} \tag{4.24} $$

Now, recall Eq. (4.8). We can see that multiplying $\mathbf{p}$ by $\mathbf{T}$ is equivalent to rotating $\mathbf{p}$ using the rotation matrix $\mathbf{R}$ embedded in $\mathbf{T}$ (the top-left 3×3 submatrix), and then translating the result by the vector stored in the first three elements of the last column of $\mathbf{T}$.

Similarly, assume $\mathbf{d} = [d_x\ d_y\ d_z\ 0]^T$ is a direction expressed in homogeneous coordinates. Let us consider the expression $\mathbf{Td}$ in this case and work out the math:

$$\mathbf{Td} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} d_x \\ d_y \\ d_z \\ 0 \end{bmatrix} = \begin{bmatrix} r_{11}d_x + r_{12}d_y + r_{13}d_z \\ r_{21}d_x + r_{22}d_y + r_{23}d_z \\ r_{31}d_x + r_{32}d_y + r_{33}d_z \\ 0 \end{bmatrix}$$

In this case, the translation components do not affect the result. This makes sense, as a vector representing a direction can be rotated but not translated. Additionally, observe that the result is still a direction expressed in homogeneous coordinates. Therefore, $\mathbf{T}$ can be used to transform either a point or a direction expressed in homogeneous coordinates. It is often convenient to consider simplified transformation matrices that encode basic transformations such as translation-only or rotation-only. For example, the transformation matrix associated with a translation by Δ_x, Δ_y, Δ_z is commonly denoted as $\mathbf{T}(\Delta_x, \Delta_y, \Delta_z)$ and is given by:

$$\mathbf{T}(\Delta_x, \Delta_y, \Delta_z) = \begin{bmatrix} 1 & 0 & 0 & \Delta_x \\ 0 & 1 & 0 & \Delta_y \\ 0 & 0 & 1 & \Delta_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Given a point $\mathbf{p} = [p_x\ p_y\ p_z\ 1]^T$, applying Eq. (4.24) to compute $\mathbf{T}(\Delta_x, \Delta_y, \Delta_z)\mathbf{p}$ yields a translated point:

$$\mathbf{T}(\Delta_x, \Delta_y, \Delta_z)\mathbf{p} = \begin{bmatrix} \Delta_x + p_x \\ \Delta_y + p_y \\ \Delta_z + p_z \\ 1 \end{bmatrix}$$

Similarly, we can define transformation matrices for rotation-only transformations about one of the three axes. For example, the transformation matrix associated with a rotation of angle θ about the z-axis is denoted as $\mathbf{T}(\mathbf{z}, \theta)$ and is given by (recall Eq. (4.11)):

$$\mathbf{T}(\mathbf{z}, \theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Recalling Eq. (4.9) and Eq. (4.10), we can similarly define $\mathbf{T}(\mathbf{x}, \theta)$ and $\mathbf{T}(\mathbf{y}, \theta)$:

$$\mathbf{T}(\mathbf{x}, \theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T}(\mathbf{y}, \theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By applying Eq. (4.24), we can easily verify that $\mathbf{T}(\mathbf{z}, \theta)\mathbf{p}$ yields the point obtained by rotating $\mathbf{p}$ about the z-axis by an angle θ. Equivalent interpretations hold for $\mathbf{T}(\mathbf{x}, \theta)\mathbf{p}$ and $\mathbf{T}(\mathbf{y}, \theta)\mathbf{p}$.

4.7.3 Transformation matrices are operators to change coordinates

Considering the previous two interpretations, it is immediate to observe that if ${}^A_B\mathbf{T}$ is the transformation matrix representing frame B with respect to frame A, and ${}^B\mathbf{p}$ is the vector expressing the coordinates of a point $\mathbf{p}$ in frame B, then

$$ {}^A\mathbf{p} = {}^A_B\mathbf{T}\,{}^B\mathbf{p} \tag{4.25} $$

This result follows directly by comparing Eq. (4.24) with Eq. (4.8). Therefore, transformation matrices can also be seen as operators for coordinate change.

4.7.4 Transformation matrices are operators to transform transformation matrices

We next consider the product of two transformation matrices. From an algebraic standpoint, this is a legitimate operation since these are square 4×4 matrices, and therefore their product is defined and results in another 4×4 matrix. To be specific, let us consider two transformation matrices: ${}^A_B\mathbf{T}$, which represents frame $B - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ with respect to frame $A - \mathbf{xyz}$, and ${}^B_C\mathbf{T}$, which represents frame $C - \mathbf{x}''\mathbf{y}''\mathbf{z}''$ with respect to frame B. Let ${}^A\mathbf{O}' = [{}^AO'_x\ {}^AO'_y\ {}^AO'_z]^T$ be the coordinates of the origin of frame B with respect to frame A, and let ${}^B\mathbf{O}'' = [{}^BO''_x\ {}^BO''_y\ {}^BO''_z]^T$ be the coordinates of the origin of frame C with respect to frame B. The product of these two transformation matrices is:

$$ \begin{aligned} {}^A_B\mathbf{T}\,{}^B_C\mathbf{T} &= \begin{bmatrix} & {}^A_B\mathbf{R} & & \begin{matrix} {}^AO'_x \\ {}^AO'_y \\ {}^AO'_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} & {}^B_C\mathbf{R} & & \begin{matrix} {}^BO''_x \\ {}^BO''_y \\ {}^BO''_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} & {}^A_B\mathbf{R}\,{}^B_C\mathbf{R} & & {}^A_B\mathbf{R}\,{}^B\mathbf{O}'' + {}^A\mathbf{O}' \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} & {}^A_C\mathbf{R} & & {}^A\mathbf{O}'' \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^A_C\mathbf{T} \end{aligned} \tag{4.26} $$

Here we used the relationships introduced previously, in particular Eq. (4.14) from Theorem 4.2 and Eq. (4.8). Exploiting the associative property of matrix multiplication, it is of course possible to consider a chain of multiplications. In this case, the previous relationship can be repeatedly applied, leading to the following notable result:

$$ {}^0_n\mathbf{T} = {}^0_1\mathbf{T}\,{}^1_2\mathbf{T}\,{}^2_3\mathbf{T} \ldots {}^{n-1}_n\mathbf{T} $$

4.7.5 Inverse of a transformation matrix

Since transformation matrices can be used to perform changes of coordinates or to transform other transformation matrices, it is immediate to ask: what is the inverse of a transformation

matrix? That is, given ${}^A_B\mathbf{T}$, we are interested in ${}^B_A\mathbf{T}$. By definition, this is a matrix such that ${}^A_B\mathbf{T}{}^B_A\mathbf{T} = {}^B_A\mathbf{T}{}^A_B\mathbf{T} = \mathbf{I}$. Through elementary algebra, it is immediate to determine that if

$$ {}^A_B\mathbf{T} = \begin{bmatrix} & {}^A_B\mathbf{R} & & {}^AO' \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

then

$$ {}^B_A\mathbf{T} = \begin{bmatrix} & {}^B_A\mathbf{R} & & -{}^B_A\mathbf{R}{}^AO' \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.27} $$

where, in Eq. (4.27), it is useful to recall that ${}^B_A\mathbf{R} = {}^A_B\mathbf{R}^T$.

Example 4.12. *Let us consider again the situation shown in Figure 4.2, where we have two robots operating in the same environment. Let A be the blue frame (world frame), B be the green frame attached to the robot on the left, and C be the red frame attached to the robot on the right. Let us assume that we know ${}^A_B\mathbf{T}$ and ${}^A_C\mathbf{T}$, i.e., the pose of the two robots with respect to the world frame. Assume that the robot on the left detects the purple diamond with its onboard sensors, and let ${}^B\mathbf{p}$ be the position of the diamond with respect to the robot on the left. We want to determine ${}^C\mathbf{p}$, i.e., we want to determine the position of the purple diamond in the frame attached to the robot on the right.*

This question can be answered in two different, but equivalent, ways. First, we can determine the position of the diamond in the world frame using Eq. (4.25)*. Once we have ${}^A\mathbf{p}$, to determine ${}^C\mathbf{p}$, we need ${}^C_A\mathbf{T}$. This transformation matrix can be obtained from ${}^A_C\mathbf{T}$ using Eq.* (4.27)*, given that we know its inverse ${}^A_C\mathbf{T}$. Stacking these products together, we therefore have*

$$ {}^C\mathbf{p} = {}^C_A\mathbf{T}{}^A_B\mathbf{T}{}^B\mathbf{p} $$

The second way to solve this problem is to first compute ${}^C_B\mathbf{T}$ and then determine ${}^C\mathbf{p}$ using again Eq. (4.25)*. Given that we know ${}^A_B\mathbf{T}$ and ${}^A_C\mathbf{T}$, we can determine ${}^C_A\mathbf{T}$ with Eq.* (4.27) *and then apply Eq.* (4.26) *to obtain ${}^C_B\mathbf{T}$. Therefore,*

$$ {}^C\mathbf{p} = {}^C_B\mathbf{T}{}^B\mathbf{p} $$

and since ${}^C_B\mathbf{T} = {}^C_A\mathbf{T}{}^A_B\mathbf{T}$ (Eq. (4.25)*), we obtain the same result (as expected) as in the first case.*

4.8 Transformation trees

Example 4.12 illustrates a problem very common in robotics, i.e., change of coordinates. In such a case, the necessary matrix ${}^C_B\mathbf{T}$ was not directly provided, but could be recovered from other matrices. In the example, the problem was particularly simple because there were only a handful of transformation matrices and frames to be considered. In real-world applications, one may have tens of frames and transformation matrices capturing the spatial relationships

between them. This is particularly evident when considering robot manipulators or mobile manipulators, where a frame is attached to every link (moving part), and there could be many of them (see, for example, the PR2 robot in Figure 4.4). In these cases, a principled method is necessary to determine whether a change of coordinates is possible and how it can be obtained. Transformation trees are used to solve these problems in an algorithmic way. A transformation tree captures the relationships between a set of provided frames and transformation matrices relating them. The definition of a transformation tree is given in the following, after the more general definition of a transformation graph.

Definition 4.4 (Transformation graph)**.** *Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of n frames and let $\mathcal{T} = \{{}^{A_{i(1)}}_{A_{j(1)}}\mathbf{T} \ldots {}^{A_{i(k)}}_{A_{j(k)}}\mathbf{T}\}$ be a set of k transformation matrices between the given frames. The* transformation graph *defined by these frames and transformation matrices is a graph $G = (V, E)$ with n vertices and k edges. Each vertex is labeled with one of the frames, i.e., v_1 is labeled A_1, v_2 is labeled A_2, and so on. Edge $(v_l, v_m) \in E$ if and only if ${}^{A_l}_{A_m}\mathbf{T} \in \mathcal{T}$.*

A transformation graph models the availability of transformation matrices between any two frames in $\mathcal{A}$. An edge between two vertices indicates that the transformation between the two associated frames is given. The definition does not define an oriented graph, i.e., edges do not have an origin and a destination. This is due to the fact that transformation matrices are always invertible, and therefore, if there is a transformation between A_l and A_m, the transformation between A_m and A_l is also available, as per section 4.7.5.

Definition 4.5 (Transformation tree)**.** *A* transformation tree *is a transformation graph with no cycles.*

Although we defined both transformation trees and transformation graphs, we usually prefer working with transformation trees rather than transformation graphs. The reason is that in a graph, cycles are present, and therefore, as we will see in the following, this means that the same problem (e.g., change of coordinates) can be solved in different ways. While this is in principle not an issue, this may lead to inconsistencies, i.e., different results are obtained when the same problem is solved in different ways. These inconsistencies are due to the fact that, oftentimes, transformation matrices are the result of noisy estimation processes involving approximations. While this problem can be addressed and managed, the topic is beyond the scope of these notes, and we therefore assume that no cycles are present. Transformation trees are illustrated in the following example.

Example 4.13. *Consider the situation depicted in Figure 4.19, where 7 frames are given (A, B, C, D, E, F, G). Assume that the following set of transformation matrices is given: $\mathcal{T} = \{{}^A_B\mathbf{T}, {}^B_C\mathbf{T}, {}^A_D\mathbf{T}, {}^D_E\mathbf{T}, {}^A_F\mathbf{T}, {}^F_G\mathbf{T}\}$.*

The associated transformation graph in this case is indeed a transformation tree featuring 7 vertices and 6 edges, and it is displayed in Figure 4.20. Note that the edges in the tree are undirected because for each of the given transformation matrices, the inverse can be determined as well. For example, the edge between A and D exists because ${}^A_D\mathbf{T} \in \mathcal{T}$. However, from ${}^A_D\mathbf{T}$ we can determine ${}^D_A\mathbf{T}$ too, from Eq. (4.27)*, and therefore both directed edges (v_A, v_D) and (v_D, v_A) exist and are displayed as a single undirected edge between the vertices associated with frames A and D.*

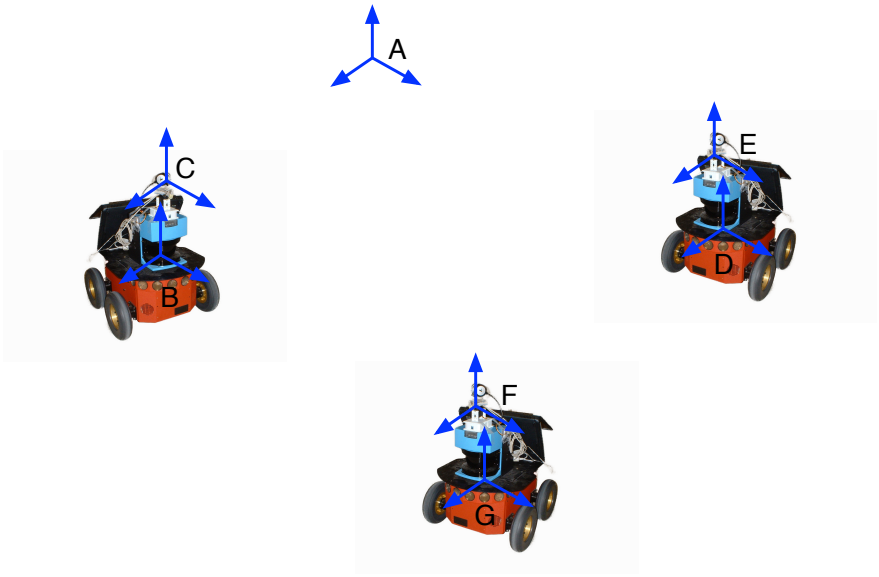


Figure 4.19: Three robots with seven frames.

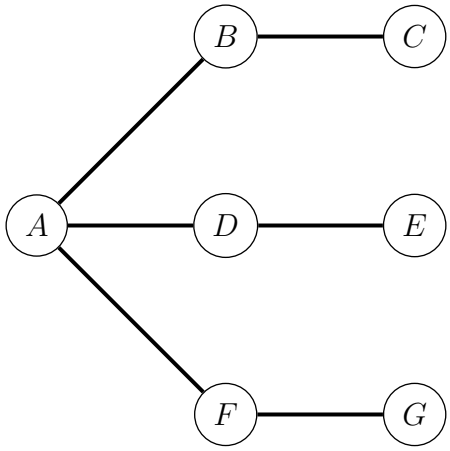


Figure 4.20: Transformation tree associated with the example in Figure 4.19.

Building upon the definition of a transformation tree, we can algorithmically answer various interesting questions. For example, for a given set of frames $\mathcal{A}$ and a set of transformation matrices $\mathcal{T}$, assume we are given ${}^{A}\mathbf{p}$ and want to determine if ${}^{B}\mathbf{p}$ can be computed, for $A, B \in \mathcal{A}$. If ${}^{B}_{A}\mathbf{T} \in \mathcal{T}$, the answer is obviously positive, and is ${}^{B}\mathbf{p} = {}^{B}_{A}\mathbf{T}\,{}^{A}\mathbf{p}$. However, even if ${}^{B}_{A}\mathbf{T} \notin \mathcal{T}$, it may still be possible to determine the answer by indirectly recovering ${}^{B}_{A}\mathbf{T}$. This can be done as follows: First, determine the transformation tree as per Definition 4.5. Note that the result can be either a tree or a forest, i.e., a set of trees. Next, run a graph search algorithm to determine if A and B are in the same connected component, i.e., if there is a path connecting them. If such a path does not exist, the change of coordinates cannot be performed using the provided information. If instead they are in the same connected component, it means that there is a unique path between A and B in the transformation tree, and each edge along the path is associated with a transformation matrix in $\mathcal{T}$. The transformation matrix needed to perform the change of coordinates is obtained by multiply-

ing together the transformation matrices along the path from B to A. Note that this path is *oriented*, i.e., the transformation matrix associated with the edge from v_i to v_j is ${}^{A_i}_{A_j}\mathbf{T}$. This matrix is either part of $\mathcal{T}$ or it can be determined by inverting one of the matrices in $\mathcal{T}$. Additionally, the path is unique because we are considering a tree and not a graph.

The previous example also elucidates why inconsistencies may emerge if a graph is considered rather than a tree. If A and B are part of the same connected component of a graph, there can potentially be more than one path between frame A and frame B. Each of them, in principle, provides a legitimate method to compute the matrix for the change of coordinates. However, if the transformation matrices are computed as the result of noisy perceptual processes, possibly involving approximations, the results may be different, i.e., inconsistent.

Finally, as frames are rigidly attached to robots or, more generally, to moving objects, the transformation matrices associated with edges in the transformation tree vary over time. Hence, even if the structure (e.g., nodes and vertices) of the tree does not change, the transformation matrices associated with each edge may change. Moreover, in some circumstances, frames may be added or removed from the system, and therefore the structure of the transformation tree may change as well. Either way, the transformation tree is a dynamic data structure that in most cases must be queried and updated at runtime.

Example 4.14. *Let us refer again to the situation depicted in Example 4.13. Assume that besides $\mathcal{A}$ and $\mathcal{T}$, we are given ${}^{E}\mathbf{p}$ and we want to compute ${}^{B}\mathbf{p}$. First, notice that the transformation matrix ${}^{B}_{E}\mathbf{T}$ is not in $\mathcal{T}$, and therefore we need to see if this information can be retrieved from the available data. The transformation tree is shown in Figure 4.20, and we notice that frames E and B are in the same component of the forest (in fact, there is just one tree.) Therefore, it is possible to recover ${}^{B}_{E}\mathbf{T}$ from the graph. The matrix is obtained by considering the path from B to E in the tree, i.e., $(v_B, v_A), (v_A, v_D)(v_D, v_E)$. Each of these edges is associated with a transformation matrix, namely ${}^{B}_{A}\mathbf{T}$, ${}^{A}_{D}\mathbf{T}$, and ${}^{D}_{E}\mathbf{T}$ (note how the direction of the edge defines which transformation matrix we need.) While ${}^{B}_{A}\mathbf{T}$ is not in $\mathcal{T}$, it can be obtained by inverting the matrix ${}^{A}_{B}\mathbf{T}$ that is part of $\mathcal{T}$. Therefore, the matrix needed to perform the change of coordinates is*

$$ {}^{B}_{E}\mathbf{T} = {}^{B}_{A}\mathbf{T}\,{}^{A}_{D}\mathbf{T}\,{}^{D}_{E}\mathbf{T} $$

Note that this whole process is easy to code in an algorithm that builds the transformation graph, determines its connected components, and extracts the relevant matrices from paths in the graph.

4.9 Kinematic motion models

In this section, we present the kinematic models governing some of the most commonly used mobile robotic platforms. Kinematic models are useful for both *analysis* and *synthesis*. In the analysis problem, we are interested in predicting the kinematic effects of specific inputs (e.g., velocities) applied to the robot. In the synthesis problem, on the other hand, we address the inverse problem, i.e., we want to determine which inputs we should apply to obtain a desired motion. Both problems are pervasive in robotics applications. As we

focus on wheeled vehicles, the kinematic models discussed in the following predict the motion resulting from applying angular velocities to the wheels. Kinematic models are (by definition) approximations that ignore dynamics, but the relationships we provide are useful for understanding some of the limitations that will affect the motion abilities of these vehicles, and also for gaining a high-level understanding of the input/output relationships from a higher-level standpoint (e.g., programming API). However, it should be clear that kinematic models alone are insufficient, and in many practical scenarios, dynamics should be considered as well.

4.9.1 Differential Drive

Robots using differential drive arrangements are pervasive in research and practical applications. Figure 4.21 shows a sample of robots that use differential drive for locomotion.

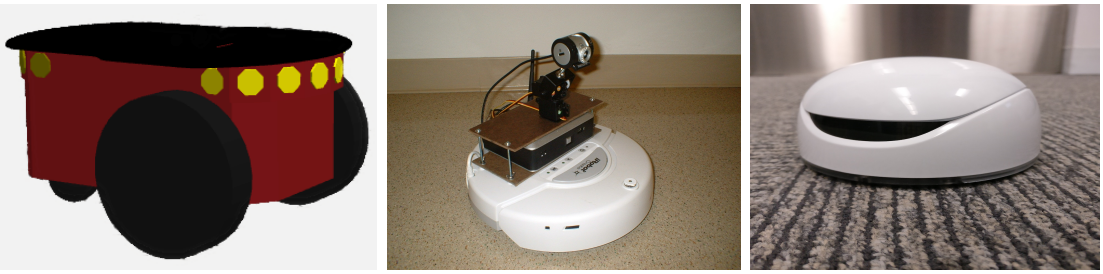


Figure 4.21: Three differential drive platforms: P2DX rendered in simulation (left), iRobot Create (center) and Vortex (right).

A differential drive robot has two wheels located on opposite sides of the robot chassis and uses two independent motors to control each of them. Usually, the robot also has a third passive (i.e., non-actuated) wheel, placed at the back. This is the case, for example, with the iRobot Create, the Vortex (Figure 4.21), the DukieBot (Figure A.1a), and many others. Since the left and right wheels can be rotated at different velocities, by appropriately setting these velocities, it is possible to implement a variety of maneuvers, including straight motion (forward/backward), turning in place (clockwise or counterclockwise), or moving along curved arcs. In the following, we derive these relationships for the differential drive. According to a kinematic perspective, we assume the motors can be controlled in velocity, which is consistent with the API provided by ROS.

The parameters governing the motion of a differential drive robot are the following:

- R: the radius of the wheels. For simplicity, we assume that both wheels have the same radii.
- ω_R and ω_L: the angular velocities for the right and left wheels, respectively.
- L: the distance (separation) between the wheels.

Figure 4.22 illustrates these quantities and the moving frame associated with a differential drive.

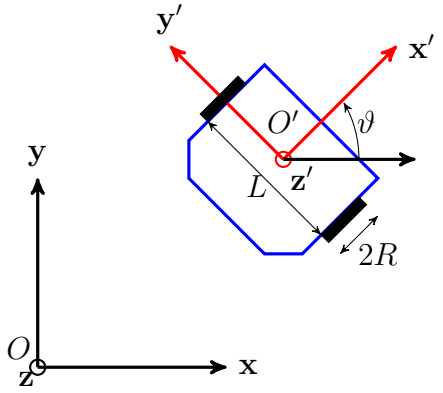


Figure 4.22: Moving frame associated with a differential drive and meaning of the parameters L and R.

As customary for mobile robots, a moving frame $B = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ is attached to the differential drive robot, with its $\mathbf{x}'$ axis pointing forward and its $\mathbf{y}'$ axis pointing to the left. Since the robot is constrained to move on a plane, the moving frame is described by the following transformation matrix (see Definition 4.3):

$$ {}^{A}_{B}\mathbf{T} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta & 0 & x \\ \sin\vartheta & \cos\vartheta & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

This matrix is fully specified by the three parameters x, y, and ϑ, where x and y are the coordinates of the origin O', and ϑ is the yaw angle. The point O' is placed at the midpoint of the axis connecting the two wheels. For differential drive robots, it is common—when context allows—to use the triplet (x, y, ϑ) as a compact representation of the robot's pose, rather than the full transformation matrix ${}^{A}_{B}\mathbf{T}$. With a bit of math (see e.g., [29, 53]), we can derive the relationships between the parameters defined above and the evolution of the robot's pose:

$$ \begin{aligned} \dot{x} &= \frac{R}{2}(\omega_R + \omega_L)\cos\vartheta \\ \dot{y} &= \frac{R}{2}(\omega_R + \omega_L)\sin\vartheta \\ \dot{\vartheta} &= \frac{R}{L}(\omega_R - \omega_L). \end{aligned} \tag{4.28} $$

Equation (4.28) is the *state transition equation* for a differential drive robot, and it can be compared with Equation (1.5). Note that the differential drive robot has two control inputs; therefore, the input vector is $\mathbf{u} = [\omega_R \ \omega_L]^T$, and its dimension is $p = 2$.

Equation (4.28) highlights an important detail with significant practical implications. Specifically, we have only two control inputs (ω_R and ω_L) to influence three state derivatives ($\dot{x}$, $\dot{y}$, and $\dot{\vartheta}$). This implies that these three components cannot be independently assigned

arbitrary values, and as a consequence, the robot's possible motions are inherently constrained. According to Equation (4.28), if $\omega_R = \omega_L \neq 0$, the robot moves in a straight line, i.e., forward if $\omega_R = \omega_L > 0$ and backward if $\omega_R = \omega_L < 0$. Conversely, if $\omega_R = -\omega_L \neq 0$, the robot rotates in place, clockwise when $\omega_R > 0$ and counterclockwise when $\omega_R < 0$. From a programming standpoint, this means that the API for controlling a differential drive robot typically accepts two parameters: the angular speeds of the left and right wheels. However, from a higher-level perspective, it is often more intuitive to control the robot using its translational and rotational velocities. The rotational velocity corresponds to $\dot{\vartheta}$, while the translational speed is given by $\sqrt{\dot{x}^2 + \dot{y}^2}$ for forward motion and $-\sqrt{\dot{x}^2 + \dot{y}^2}$ for backward motion.

Indeed, many control APIs for differential drive robots, such as those used in ROS, accept these two parameters: translational and rotational speed. Often, such APIs include an additional constraint allowing only one of these values to be nonzero at a time. This enforces simplified control modes in which the robot either moves in a straight line or turns in place, but not both simultaneously. We will see this design choice in more detail when we explore the ROS control stack later in this chapter.

Example 4.15. *Let the radius of the wheel be $R = 0.05\,m$ and the separation between the wheels be $L = 0.15\,m$. Determine ω_L and ω_R such that the translational speed is $0.1\,m/s$ and the rotational speed is $0.05\,rad/s$.*

From the above relationships, the (positive) translational speed can be written as

$$\begin{aligned}\sqrt{\dot{x}^2 + \dot{y}^2} &= \sqrt{\left(\frac{R}{2}(\omega_R + \omega_L)\cos\vartheta\right)^2 + \left(\frac{R}{2}(\omega_R + \omega_L)\sin\vartheta\right)^2} \\ &= \sqrt{\frac{R^2}{4}(\omega_R + \omega_L)^2} = \frac{R}{2}(\omega_R + \omega_L)\end{aligned}$$

We therefore obtain a linear system of two equations in two unknowns:

$$\begin{aligned}\frac{R}{2}(\omega_R + \omega_L) &= 0.1 \\ \frac{R}{L}(\omega_R - \omega_L) &= 0.05\end{aligned}$$

Substituting the given values for R and L, the system solves to $\omega_R = 2.075\,rad/s$ and $\omega_L = 1.925\,rad/s$.

The above example confirms that, from an API standpoint, it is indeed possible to specify translational and rotational velocities directly and then map these to the (less intuitive) angular velocities of the left and right wheels. In various scenarios (e.g., in estimation or prediction problems), it becomes necessary to solve the *forward kinematics* problem, that is, to predict the *next* pose of the robot given its current pose and current velocity inputs. If the control input is specified as the angular velocity of the left and right wheels, we can use Eq. (4.28) to derive a first-order, approximate, discrete-time model by discretizing with a time step Δt. This yields:

$$
\begin{aligned}
x(t+\Delta t) &\approx x(t) + \dot{x}\Delta t = x(t) + \frac{R}{2}(\omega_R + \omega_L)\cos\vartheta(t)\Delta t \\
y(t+\Delta t) &\approx y(t) + \dot{y}\Delta t = y(t) + \frac{R}{2}(\omega_R + \omega_L)\sin\vartheta(t)\Delta t \\
\vartheta(t+\Delta t) &\approx \vartheta(t) + \dot{\vartheta}\Delta t = \vartheta(t) + \frac{R}{L}(\omega_R - \omega_L)\Delta t
\end{aligned} \tag{4.29}
$$

Naturally, the accuracy of this approximation deteriorates as Δt increases. Alternatively, if the input is provided as translational velocity v_t and rotational velocity v_r, we can use the following approximate expressions derived in [53]. If $v_r \neq 0$:

$$
\begin{aligned}
x(t+\Delta t) &\approx x(t) + \left[-\frac{v_t(t)}{v_r(t)}\sin\vartheta(t) + \frac{v_t(t)}{v_r(t)}\sin\left(\vartheta(t) + v_r(t)\Delta t\right)\right] \\
y(t+\Delta t) &\approx y(t) + \left[\frac{v_t(t)}{v_r(t)}\cos\vartheta(t) - \frac{v_t(t)}{v_r(t)}\cos\left(\vartheta(t) + v_r(t)\Delta t\right)\right] \\
\vartheta(t+\Delta t) &\approx \vartheta(t) + v_r(t)\Delta t
\end{aligned} \tag{4.30}
$$

If instead $v_r = 0$, these expressions simplify to describe exact straight-line motion (i.e., no approximation is involved):

$$
\begin{aligned}
x(t+\Delta t) &= x(t) + v_t \sin\vartheta(t) \\
y(t+\Delta t) &= y(t) + v_t \cos\vartheta(t) \\
\vartheta(t+\Delta t) &= \vartheta(t)
\end{aligned} \tag{4.31}
$$

Equations (4.29) and (4.30) provide the discrete-time state transition equations for the differential drive robot and can be compared with the general form in Eq. (1.7).

4.9.2 Skid steer drive

Alternatively, the robot may have four[7] wheels, but still be driven by just two motors. In this setup, the wheels on the same side are mechanically coupled, typically via a belt or chain, and are actuated by the same motor. Robots of this type include, for example, the P3AT (Figure 4.23) and the Husky (Figure 1.1). Due to the mechanical coupling, it is not possible to independently control the speed of wheels on the same side of the robot. To achieve turning in place, the motors are driven at the same speed but in opposite directions. As a result, the wheels skid, hence the term *skid steer*. More generally, when the two motors operate at different speeds, the robot follows curved trajectories. However, this typically induces skidding in all cases except for straight-line motion, where both sides are driven at equal velocities.

From a programming perspective, both differential drive and skid-steer robots are controlled using the same type of commands. A notable evolution of this concept is the *Amiga*

[7] Although four wheels is the most common configuration, the same considerations apply to any even number of wheels greater than four.

Figure 4.23: P3AT robot

Figure 4.24: The Amiga robot by Farm-Ng

by Farm-Ng, which is equipped with four independent motors, each controlling one of its wheels (see Figure 4.24). Despite this more complex hardware setup, the ROS API abstraction ensures that controlling the Amiga is functionally equivalent to operating platforms like the Husky or the P3AT. This equivalence will become clearer later in this chapter.

4.9.3 Ackerman Steer

The Ackerman steer is the configuration used in cars and comparable vehicles with steering wheels. In this section, we present an abstraction useful for planning purposes, but we omit some of the mechanical details used in physical implementations. For this reason, the model presented herein is also called the *simple car* [29]. As we all know from everyday experience, driving a car can be abstracted to providing two inputs, i.e., the velocity (positive or negative) and the steering. These two inputs will be indicated as v_t (translational speed, as for the differential drive robot) and ϕ. Figure 4.25 illustrates the abstraction we will use when studying this type of vehicle, together with the moving frame $B = O' - \mathbf{x}'\mathbf{y}'\mathbf{z}'$ associated

with the vehicle. As for the differential drive, the frame has the $\mathbf{x}'$ axis pointing forward and the $\mathbf{y}'$ axis pointing to the left. Similarly, ϑ is the rotation about $\mathbf{z}$. As for the differential drive, the transformation matrix ${}^{A}_{B}\mathbf{T}$ can be summarized by the triplet pose (x, y, ϑ).

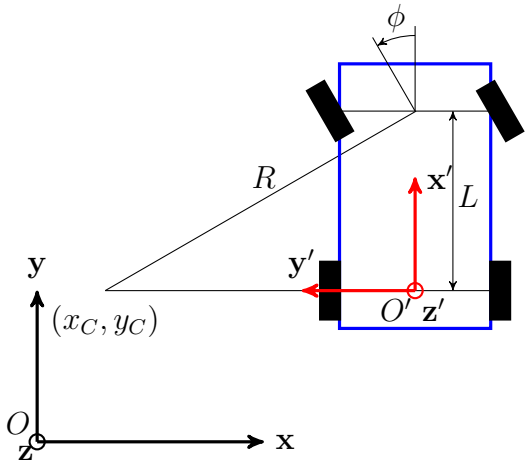


Figure 4.25: Moving frame associated with a differential drive and meaning of the parameters L and R.

Note that there is a physical constraint on the possible values for the steering angle, i.e., $|\phi| < \pi/2$. Next, assume that both v_t and ϕ are constant and both different from 0. In this case, the car moves along a circle whose radius is determined by ϕ and L, i.e., the separation between the front and rear wheels, and whose center (x_C, y_C) is determined by ϕ, L, and the pose of the moving frame B. With some math [29], it is possible to show that the radius of the center of rotation is

$$R = \frac{L}{\tan\phi}$$

From this value and (x, y, ϑ), it is possible to determine the center of rotation (x_C, y_C). The following differential relationships provide the *state transition equation* for the Ackerman steer vehicle and show how the three coordinates change as a function of the two inputs v_t and ϕ:

$$\begin{aligned} \dot{x} &= v_t \cos\vartheta \\ \dot{y} &= v_t \sin\vartheta \\ \dot{\vartheta} &= \frac{v_t}{L}\tan\phi. \end{aligned} \tag{4.32}$$

Eq. (4.32) is the state transition equation for the Ackerman steer drive, and it shows that $\mathbf{u} = [v_r \ \phi]^T$, i.e., $p = 2$ also for the Ackerman steer drive. The three coordinates x, y, ϑ change according to two inputs, v_r and ϕ, and this leads to various motion restrictions. Through a first-order approximation, it is possible to predict how the state will evolve in time, thus obtaining the discrete-time state transition equation:

$$\begin{aligned} x(t+\Delta t) &\approx x(t) + v_t \Delta t \cos\vartheta \\ y(t+\Delta t) &\approx y(t) + v_t \Delta t \sin\vartheta \\ \vartheta(t+\Delta t) &\approx \vartheta(t) + \frac{v_t \Delta t}{L} \tan\phi \end{aligned} \tag{4.33}$$

4.10 Velocity

As a robot moves in space, its pose changes over time. Obviously, the first derivative of the pose with respect to time yields velocity, and this was informally introduced in the previous section with the discussion of kinematic motion models. A principled treatise of velocities is beyond the scope of these notes. Since the pose of the robot is defined by a transformation matrix, it would be necessary to view the transformation matrix as a function of time and then consider its derivative. A transformation matrix includes both the position, i.e., an element in $\mathbb{R}^3$, and the orientation. The orientation can be equivalently represented as a 3×3 rotation matrix $\mathbf{R}$ or a quaternion $\mathbf{q}$. The derivative with respect to time for the position is straightforward to consider and is most likely known to the reader, but things are slightly more complex when considering the derivative of a rotation matrix (or a quaternion). Still, recalling that it is possible to parametrize a rotation matrix with three angles, like roll, pitch, and yaw, we can intuitively guess that there will be three angular velocities associated with rotations about the frame axes.

4.11 Kinematics in ROS

Due to its practical importance, ROS provides various packages and messages to support kinematics and geometric processing. Moreover, the package `tf2` handles multiple frames and their relationships, and can be used to track how they evolve over time (see Section 4.13).

4.11.1 The `geometry_msgs` Package

The package `geometry_msgs` defines messages to represent geometric quantities commonly used in robotics, such as points, orientations, and their derivatives with respect to time. All quantities previously introduced in this chapter are implemented in ROS, and table 4.1 displays the correspondences between some of the concepts introduced in this chapter and the associated ROS messages.

The package `geometry_msgs` does not define a message for rotation matrices, because rotations are represented as is defined as

```
# This represents the transform between two coordinate frames in free space.

Vector3 translation
Quaternion rotation
```

Message Type	Quantity
`geometry_msgs::msg::Point`	Point in $\mathbb{R}^3$
`geometry_msgs::msg::PointStamped`	Point with time stamp
`geometry_msgs::msg::Quaternion`	Orientation
`geometry_msgs::msg::QuaternionStamped`	Orientation with time stamp
`geometry_msgs::msg::Pose`	Position and orientation (as quaternion)
`geometry_msgs::msg::PoseStamped`	Pose with time stamp
`geometry_msgs::msg::Pose2D`	Pose in the plane, i.e., x, y, θ
`geometry_msgs::msg::Transform`	Transformation matrix
`geometry_msgs::msg::TransformStamped`	Transformation matrix with time stamp
`geometry_msgs::msg::Vector3`	Direction in space
`geometry_msgs::msg::Vector3Stamped`	Direction in space with time stamp
`geometry_msgs::msg::Twist`	Velocity (linear and angular)
`geometry_msgs::msg::TwistStamped`	Velocity with time stamp

Table 4.1: ROS messages to represent geometric data.

of type `geometry_msgs::msg::Quaternion`. However, there exists a data type to represent both rotation and transformation matrices, but this is provided by the `tf2` library discussed in Section 4.13.1. Some of these messages have different names but identical components. `geometry_msgs::msg::Pose` includes a message of type `geometry_msgs::msg::Point`, called `position`, and a message of type `geometry_msgs::msg::Quaternion`, called `orientation`; `geometry_msgs::msg::Transform` includes a field of type `geometry_msgs::msg::Point`, called `translation`, and one of type `geometry_msgs::msg::Quaternion`, called `rotation`. This is consistent with our discussion about transformation matrices, and the fact that they can be used to represent frames, and therefore, once rigidly attached to a body (robot), they can represent its pose, too. To represent velocities, the `geometry_msgs` package provides the message `geometry_msgs::msg::Twist`, which includes linear velocities along the three axes and angular velocities about the three axes. This message is commonly used to send velocity commands to mobile robots with a differential or skid-steer drive, and will be further analyzed later on.

Time stamps

As many geometric quantities change over time, ROS provides *stamped* versions for many messages. The structure of the various stamped messages is similar, i.e., in addition to the message itself, they include a message of type `Header` from the package `builtin_interfaces`. For example, the output of the command

```
ros2 interface show geometry_msgs/msg/Vector3
```

is

```
float64 x
```

```
float64 y
float64 z
```

while

```
ros2 interface show geometry\_msgs/msg/Vector3Stamped
```

produces

```
std_msgs/Header header
 builtin_interfaces/Time stamp
  int32 sec
  uint32 nanosec
 string frame_id
Vector3 vector
 float64 x
 float64 y
 float64 z
```

The structure of `builtin_interfaces/Time` was already explored in Section 2.7.1 and consists of two integers to measure time in seconds and nanoseconds relative to a starting time. This may be either the system clock or a custom time source if `use_sim_time` is set (see Section 5.4.3 for more details on this topic.) A stamped `Vector3` therefore includes a header providing the timestamp as well as the name of the frame with which the vector is associated, i.e., `frame_id`. This structure is common to all *stamped* messages and is aligned with the theory we discussed earlier in this chapter, where we outlined the importance of referring quantities to a specific frame.

4.11.2 Pose2D

When dealing with robots moving in the plane, the pose is normally defined by three parameters only, i.e., x, y, and the yaw angle θ, as per our discussion in Section 4.9.1. For this reason, it is possible to use messages of type `geometry_msgs::msg::Pose2D`, which are defined as follows:

```
 float64 x
 float64 y
 float64 theta
```

However, it should be noted that oftentimes mobile robots nevertheless broadcast their pose using messages of type `geometry_msgs::msg::Pose`.

4.12 Controlling a differential/skid steer robot in ROS

Most differential drive and skid-steer[8] robots can be controlled in ROS by publishing messages of type `geometry_msgs::msg::Twist` to an appropriate topic. The structure of this message is the following:

```
#This expresses velocity in free space broken into its linear and angular parts.

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

Owing to the kinematic structure of the differential drive we discussed in Section 4.9.1, the field `linear.x` is used for the translational speed and `angular.z` for the rotational speed. The other fields should be set to 0, as a differential drive robot cannot translate along the **y** and **z** axes of its local frame, nor can it rotate about its **x** and **y** axes. The frame is attached to the robot as shown in Figure 4.22. In this section, we show a simple example of differential drive control using the `turtlesim` package introduced in Chapter 2. Despite providing an overly simplified environment, `turtlesim` is useful because the simulated turtle is a differential drive robot and can be controlled by sending messages of type `geometry_msgs::msg::Twist`. Various commercial mobile robot platforms, like the Husky robot shown in Figure 1.1, can be controlled using exactly the same code (possibly adjusting the name of the topic). Listing 4.1 provides the code of a node that moves the turtle to draw a square on the screen.

is defined as

```
# This represents the transform between two coordinate frames in free space.

Vector3 translation
Quaternion rotation
```

Listing 4.1: Drawsquare node

```
1 #include <rclcpp/rclcpp.hpp>
2 #include <geometry_msgs/msg/twist.hpp>
3
4 int main(int argc,char **argv) {
5
6   rclcpp::init(argc,argv);
7   rclcpp::Node::SharedPtr nodeh;
```

[8] In the following, for brevity, we refer only to the differential drive, but the same concepts also apply to the skid-steer drive.

```
8   rclcpp::Rate rate(1);
9
10  nodeh = rclcpp::Node::make_shared("drawsquare");
11  auto pub = nodeh->create_publisher<geometry_msgs::msg::Twist>
12      ("turtle1/cmd_vel",100);
13
14  geometry_msgs::msg::Twist msg;
15  while (rclcpp::ok()) {
16    msg.linear.x = 1;
17    msg.angular.z = 0;
18    pub->publish(msg);
19    rate.sleep();
20    msg.linear.x = 0;
21    msg.angular.z = M_PI/2;
22    pub->publish(msg);
23    rate.sleep();
24  }
25 }
```

The source code features familiar lines, such as including the header file `rclcpp.hpp` and initializing the node and the publisher. However, in this case, we also need to include the header file that provides the definition of messages of type `geometry_msgs::msg::Twist.h`. This implies that the node depends on the package `geometry_msgs`, and therefore, the files `package.xml` and `CMakeLists.txt` should be updated accordingly.

The name of the topic (which must be matched by the publisher) can be obtained using the command `ros2 topic list` after starting the node `turtlesim_node` from the `turtlesim` package. Observe that, while in general the name of the topic may vary, ground robots typically accept twist commands on a topic called `cmd_vel`. In this case, the name of the command is prefixed by the name of the robot, i.e., `turtle1`, so that the full topic name is `turtle1/cmd_vel`. Section 5.2 will provide additional details on the process of including the robot name in the topic name. In the while loop, we alternate between moving the robot forward and turning it counterclockwise by 90 degrees. This is done by alternately setting `linear.x` and `angular.z` to 1 m/s or $\frac{\pi}{2}$ rad/s, respectively. Since the `rate` object is initialized with a target frequency of 1 Hz, every time `rate.sleep()` is called, the program waits for approximately one second, and the robot will therefore alternately move forward by 1 meter and rotate counterclockwise by $\pi/2$. To run and test this node, you first need to start the `turtlesim_node` node typing

```
ros2 run turtlesim turtlesim_node
```

and then start the node `drawsquare`.

```
ros2 run examples drawsquare
```

If you let the code run for a while, you will observe that the path followed by the turtle eventually diverges from a square pattern (see Figure 4.26).

This happens because the algorithm implements an open-loop controller (recall Figure 1.6) and does not integrate any information coming from the robot itself. This is almost

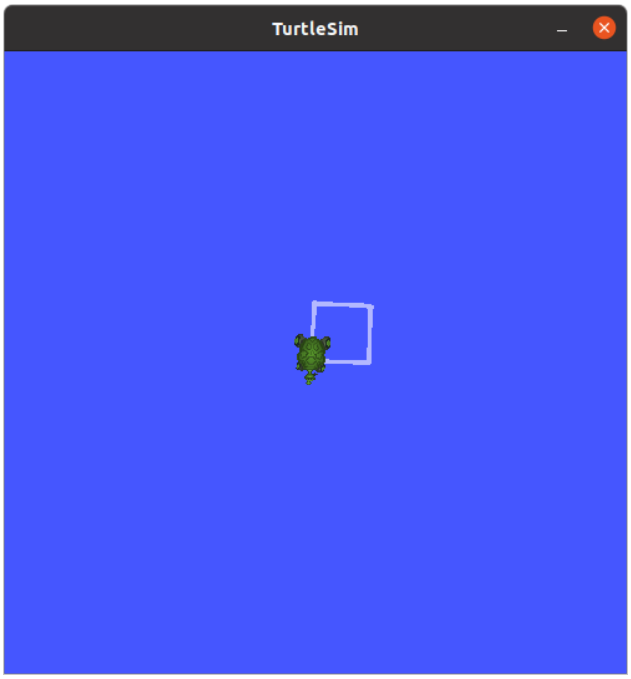


Figure 4.26: After a few iterations, the shape drawn by the turtle diverges from a square.

is defined as

```
# This represents the transform between two coordinate frames in free space.

Vector3 translation
Quaternion rotation
```

invariably a bad idea, and this simple example shows how this strategy is prone to errors. In this case, `rate.sleep()` tries to match the desired 1 Hz frequency, but there is no guarantee that this will be done accurately. Moreover, part of the time within the cycle is spent executing other operations, such as assignments, and so on. Therefore, even if `rate.sleep()` exactly achieves the target frequency, the time spent translating or rotating will be slightly different from one second. These errors accumulate over time and eventually the orientation of the traced square tends to rotate and diverge from the initial one, where the edges are aligned along the xy directions.

How can we fix this problem? If, after starting the `turtlesim_node`, we run `ros2 topic list`, we see that the node publishes a topic called `/turtle1/pose`, and with `ros2 topic info` we can verify that this specific message is of type `turtlesim::msg::Pose`, which is defined as:

```
float32 x
float32 y
float32 theta
```

```
float32 linear_velocity
float32 angular_velocity
```

As can be guessed, the message includes the pose of the robot (x, y, ϑ) as well as its linear and angular velocity. In particular, the provided orientation `theta` is the absolute orientation of the turtle. This message abstracts a sensor, i.e., a system providing information about the robot, as discussed in Section 1.2. This sensor could then be used to implement a simple motion strategy (closed loop) where we use information about the actual pose to decide the command to execute (see also Figure 1.4). More precisely, to match a desired orientation (a multiple of $\pi/2$ to move along perpendicular lines), we do not stop turning after a certain amount of time has passed, but rather when the actual orientation matches the desired one. Similarly, the robot stops translating after it has moved a certain distance from the point where it switched from turning to moving straight. Both the actual orientation and the actual traveled distance can be obtained from `/turtle1/pose`, either directly or through some simple computations. Note that one could devise much more sophisticated closed loop solutions, but for the time being this is sufficient and addresses the problem we identified. This revised solution is shown in Listing 4.2. As this node both publishes velocity commands and retrieves sensor information, it implements both a publisher and a subscriber, following the same approach presented in Listing 3.15.

Listing 4.2: Drawsquare node with feedback

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <geometry_msgs/msg/twist.hpp>
3  #include <turtlesim/msg/pose.hpp>
4
5  float x, y, theta;
6  bool init;
7
8  void poseCallback(const turtlesim::msg::Pose::SharedPtr msg) {
9    x = msg->x;
10   y = msg->y;
11   theta = msg->theta;
12   init = true;
13 }
14
15 int main(int argc,char **argv) {
16
17   rclcpp::init(argc,argv);
18   rclcpp::Node::SharedPtr nodeh;
19   rclcpp::Rate rate(1);
20
21   nodeh = rclcpp::Node::make_shared("drawsquarefb");
22   auto pub = nodeh->create_publisher<geometry_msgs::msg::Twist>
23     ("turtle1/cmd_vel",1000);
24   auto sub = nodeh->create_subscription<turtlesim::msg::Pose>
25     ("turtle1/pose",1000,&poseCallback);
26
27   geometry_msgs::msg::Twist msg;
28   init = false;
29
```

```
30    bool rotate = false;
31    float start_x = x;
32    float start_y = y;
33    double DIRS[] = {0,M_PI/2,-M_PI,-M_PI/2}; // desired directions
34    int direction = 0; // current direction
35
36    while (!init) // wait for at least one sensor reading
37      rclcpp::spin_some(nodeh);
38
39    while (rclcpp::ok()) {
40      rclcpp::spin_some(nodeh); //get sensor reading, if available
41      if (rotate) { // rotating?
42        // reached desired heading?
43        if ( ( ( direction == 0  ) && ( theta < DIRS[0] ) ) ||
44             ( ( direction == 1  ) && ( theta < DIRS[1] ) ) ||
45             ( ( direction == 2  ) && ( theta  > 0 ) ) ||
46             ( ( direction == 3  ) && ( theta < DIRS[3] ) ) ) {
47          msg.linear.x = 0; msg.angular.z = M_PI/8; // no; keep rotating
48        }
49        else { // yes
50          msg.linear.x = 0; msg.angular.z = 0; //stop the robot
51          rotate = false; // switch to translating
52          start_x = x; start_y = y; // record current location
53        }
54      }
55      else { // translating?
56        if (hypotf((x-start_x),(y-start_y))<2) { // moved less than 2 units?
57          msg.linear.x = 0.5; msg.angular.z = 0;  // no, keep moving forward
58        }
59        else { // moved 2 units
60          rotate = true; // switch to rotate
61          msg.linear.x = 0; msg.angular.z = 0; // stop the robot
62          ++direction %= 4; // track next direction
63        }
64      }
65      pub->publish(msg); // send motion command
66    }
67  }
```

The controller goes through two different stages controlled by the variable `rotate`. When `rotate` is true, the robot turns in place, and when it is false, it moves straight ahead. To decide when to switch from one stage to the other, the robot checks if it has reached one of the desired orientations (line 43) or if it has moved a prescribed distance from the point when it switched to translating (line 56). The test in line 43 considers the fact that the orientation returned by the `pose` message is in the $[-\pi, \pi]$ range[9]. Therefore, if the robot is trying to align itself along the directions 0, $\pi/2$, and $-\pi/2$, it keeps turning as long as it has not reached that orientation (first, second, and fourth cases in the `if` condition). If instead the robot is aligning itself to the π direction, it keeps turning as long as the orientation does not switch from positive heading to negative heading (i.e., it goes from π to $-\pi$). This is

[9]Note that in different ROS versions this may be different, e.g., in some cases the direction is in the $[0, 2\pi]$ range. If that is the case, this code will not work as-is and must be changed.

the third case in the `if` statement.

Also observe that, in the beginning, it performs a busy waiting cycle (line 36) to wait for the first `pose` message to be received. This is necessary because the following code relies on having received at least one sensor reading to initialize the position and heading of the turtle. Figure 4.27 shows the result obtained by the revised version of the code. After many iterations, the turtle is still moving along a well-aligned square, and the drift problem formerly identified no longer appears.

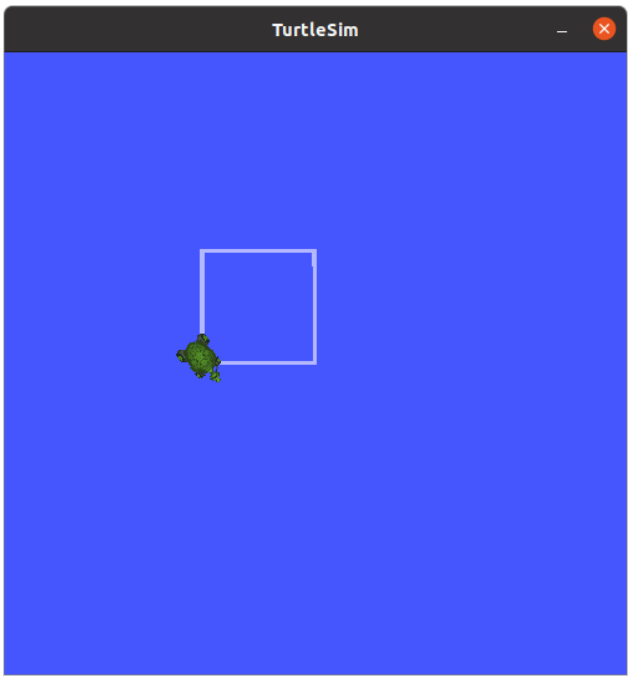


Figure 4.27: Shape obtained running the revised controller in listing 4.2.

The simple examples presented above can do more than just control the turtle in the `turtlesim` environment. In fact, the same code can be used to control real robots as well. To experiment with this, we launch *Gazebo*, a high-fidelity robot simulator that can work seamlessly with ROS. One of the nice features of this pairing is that one can run exactly the same code in simulation and on the real robot. A complete discussion of Gazebo is beyond our goals, and here we just use some launch files and examples provided in the MRTP GitHub. We begin by starting the simulator with an empty world and a model of the TurtleBot robot – a differential drive:

```
ros2 launch gazeboenvs tb4_simulation.launch.py
```

Observe how, in this case, the launch file is written in Python rather than XML. This is an advanced feature we will explore further in Section 5.9. This command will start[10] the simulator and instantiate the robot, as shown in Figure 4.28. With `ros2 topic list`, we can see that, among others, the TurtleBot is subscribed to a topic called `/cmd_vel` of type

[10]The first time you run this launch file, there will be a delay at the beginning because the simulator retrieves some files from a remote repository. These will be cached, and subsequent executions will be faster.

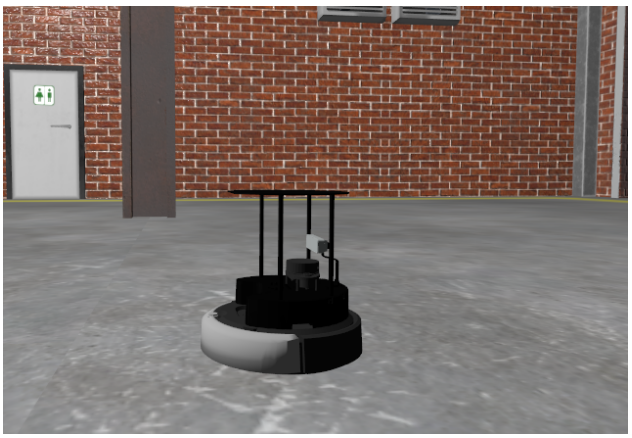


Figure 4.28: Gazebo simulation of the TurtleBot robot in a warehouse environment.

`geometry_msgs::msg::Twist`. As can be guessed, this is the topic used to send velocity commands to the robot. We would like to use the `drawsquare` example in Listing 4.1 to move the robot around, but this cannot be used as-is because there is a mismatch in the topic names. The robot is subscribed to `/cmd_vel`, but the node publishes to `turtle1/cmd_vel`; hence, no messages would flow from the node to the robot. This can be fixed in two ways. One, less flexible, consists of editing the source code and changing the name of the topic, or perhaps adding it as a command-line parameter. The other approach, which we follow, is to use a ROS feature called *remapping*, whereby we remap topics, i.e., we reassign topic names when the node is started. This is achieved with the following syntax:

```
ros2 run examples drawsquare --ros-args --remap
turtle1/cmd_vel:=/cmd_vel
```

Conceptually, this can be thought of as creating a connection between `turtle1/cmd_vel` and `/cmd_vel`, so that messages sent by `drawsquare` to `turtle1/cmd_vel` end up in the topic `/cmd_vel`. Remapping is a convenient ROS mechanism that allows easy reuse of code, and it will be further discussed in Section 5.1. After starting the node, if you observe Gazebo, you will see that the robot moves along a path that resembles a square, though it is not a perfect square. This is due to the fact that Gazebo integrates dynamics into the simulation, introducing numerous sources of inaccuracy that did not appear with the turtle. Still, this shows how, with a few lines of code, it is possible to move a mobile robot providing a ROS interface.

At this point, we might be tempted to try using remapping to run `drawsquarefb` (Listing 4.2) to control the TurtleBot. However, this cannot be done because the simulated robot does not publish messages of type `turtlesim::msg::Pose`, so it is not possible to remap and pass that information to `drawsquarefb`. As a side note, though, the robots publish to a topic `odom` of type `nav_msgs::msg::Odometry`, which simulates a sensor (called *odometry*) that includes the pose of the robot. So, with some changes, one could apply the same approach. However, the results will not be as good as with the turtle, because in addition to the issues mentioned above, the information provided by the odometry sensor includes errors that grow over time (more details will be provided in Chapter 7).

When using `geometry_msgs::msg::Twist` messages to control a robot, some platform-specific parameters must be considered. For example, for safety reasons, many robots include a *timeout* feature that will stop the robot if no twist command is received for a certain amount of time. In such cases, it will be necessary to keep sending commands to the robot to keep it moving.

4.13 The transform library

ROS provides the *transform library*[11] to easily process geometric data and solve transformation-related problems, such as changes of coordinates. The library is called `tf2`, as it is the second iteration that replaced the original one called `tf`. `tf2` is a sophisticated set of components offering many functionalities. To simplify, we could say that it offers: 1) numerous classes supporting most of the geometric concepts discussed earlier in this chapter; 2) static functions to perform geometric operations; 3) tools and datatypes to track multiple coordinate frames changing over time.

4.13.1 `tf2` classes, messages and functions

Table 4.2 shows some of the classes provided by `tf2` together with a brief description of their purpose.

Class	Description
`tf2::Matrix3x3`	Rotation Matrix
`tf2::Quaternion`	Quaternion
`tf2::Vector3`	Point or vector
`tf2::Transform`	Rigid transformation, i.e., rotation and translation

Table 4.2: `tf2` classes to represent geometric data.

`tf2` provides overloaded versions of various arithmetic operators, making it possible to perform operations in an intuitive way, such as multiplying a matrix and a vector, two quaternions, and so on. The package `tf2_msgs`, also part of the library, defines two messages, namely `tf2_msgs::msg::TF2Error` and `tf2_msgs::msg::TFMessage`, whose structures are as follows.

```
geometry_msgs/TransformStamped[] transforms
 std_msgs/Header header
  builtin_interfaces/Time stamp
   int32 sec
   uint32 nanosec
  string frame_id
 string child_frame_id
 Transform transform
```

[11] Although it is called *library*, owing to the fact that everything in ROS belongs to a package, these functionalities are offered through packages.

```
  Vector3 translation
   float64 x
   float64 y
   float64 z
  Quaternion rotation
   float64 x 0
   float64 y 0
   float64 z 0
   float64 w 1
```

i.e., a it includes vector of messages of type `geometry_msgs::msg::TransformStamped`. As outlined earlier, *stamped* messages include both a structure and a header. The message `std_msgs::msg::Header` has already been discussed in Section 4.11.1, and the message `geometry_msgs::msg::Transform` includes a translation and a rotation, consistent with what we learned when transformation matrices were introduced in Section 4.7. The fields `translation` and `rotation` define the origin and orientation of the frame, whereas `child_frame_id` in `geometry_msgs::msg::TransformStamped` is a string identifying the frame (i.e., the frame name). The `header` section of the message defines the timestamp of the frame (useful for moving frames), as well as the string `frame_id` identifying the frame with respect to which this frame is referenced. Therefore, ${}^{B}_{A}\mathbf{T}$ will be represented by a message `TransformStamped` with `child_frame_id` set to A and `frame_id` set to B.

As the name implies, messages of type `tf2_msgs::msg::TF2Error` are instead used to exchange information about errors occurred in the `tf2` system. The structure of the message is the following:

```
uint8 NO_ERROR = 0
uint8 LOOKUP_ERROR = 1
uint8 CONNECTIVITY_ERROR = 2
uint8 EXTRAPOLATION_ERROR = 3
uint8 INVALID_ARGUMENT_ERROR = 4
uint8 TIMEOUT_ERROR = 5
uint8 TRANSFORM_ERROR = 6

uint8 error
string error_string
```

The field `error_string` stores a message describing the error, while `error` stores an integer whose value should be interpreted according to the constants defined in the message itself. Finally, the package includes the definition of a large number of functions that can be used to perform various geometric operations, such as combining two transformations. Some of these are discussed in the following.

4.13.2 Quaternions and rotations

The package `geometry_msgs` does not provide a message for rotation matrices, because rotations are instead exchanged as quaternions. However, the package `tf2` includes classes

and methods that can be used to model quaternions and easily convert between quaternions and rotation matrices. The class `tf2::Quaternion` represents a quaternion. It may seem redundant that quaternions are represented in two packages, but there is a reason for this. `geometry_msgs::msg::Quaternion` is just a message to be transmitted and it includes just data, but it does not include auxiliary methods to operate on the data. These additional methods are available only in `tf2::Quaternion`. While the ROS documentation is the ultimate reference, we outline the following methods that may be useful:

- `getAngle` returns the angle associated with the quaternion (see Eq.(4.19));
- `getAxis` returns the axis associated with the quaternion (see Eq.(4.20));
- `setEulerZYZ` sets the quaternion to the rotation associated with a given triplet of Euler angles (see Section 4.5.3);
- `setRPY` sets the quaternion to the rotation associated with a given triplet of roll-pitch-yaw angles (see Section 4.5.3).

In addition, the package `tf2` defines numerous static functions that can be used on quaternions to perform a variety of useful operations. Among these, the following are often used in practice:

- `tf2::getYaw`: accepts as a parameter an instance of `tf2::Quaternion` and returns the associated yaw angle. This is extremely useful for mobile robots moving in the plane.
- `tf2::toMsg`: converts an instance of `tf2::Quaternion` into a message of type `geometry_msgs::Quaternion`, so that it can be published (see also Listing 4.4 for more details).
- `tf2::fromMsg`: converts a message of type `geometry_msgs::Quaternion` into the equivalent class of the `tf2` package.

`tf2` also provides a class called `tf2::Matrix3x3` that represents a rotation matrix. Its method `setRotation` can be used to convert a quaternion into a rotation matrix (see Eq. (4.18)), whereas the method `getRotation` determines the quaternion equivalent to the given rotation matrix, as per Eq. (4.22). The class `tf2::Matrix3x3` also includes various other methods to directly operate on the rotation matrix, such as computing the inverse, transpose, etc. Listing 4.3 shows how to use some of the classes and methods we have just introduced.

Listing 4.3: Geom Node

```
1 #include <rclcpp/rclcpp.hpp>
2 #include <tf2/LinearMath/Quaternion.h>
3 #include <tf2/LinearMath/Matrix3x3.h>
4 #include <tf2/LinearMath/Transform.h>
5 #include <tf2/LinearMath/Vector3.h>
6 #include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
7
```

```
8
9  int main(int argc,char ** argv) {
10
11   rclcpp::init(argc,argv);
12   rclcpp::Node::SharedPtr nodeh;
13   nodeh = rclcpp::Node::make_shared("geom");
14
15   tf2::Quaternion q1,q2,q3;
16   q1.setRPY(0,0,M_PI/2);
17   q2.setRPY(0,M_PI/4,0);
18   q3 = q1*q2;
19   RCLCPP_INFO(nodeh->get_logger(), "%f-%f-%f-%f",
20               q3.x(), q3.y(), q3.z(), q3.w());
21   tf2::Matrix3x3 r;
22   r.setRotation(q3);
23   tf2::Vector3 o1(0,1,3),o2(1,4,0);
24   tf2::Transform t1(q2,o1);
25   tf2::Transform t2(r,o2);
26   tf2::Transform t3= t1*t2;
27   tf2::Vector3 ori = t3.getOrigin();
28   RCLCPP_INFO(nodeh->get_logger(),"%f-%f-%f",ori.x(),ori.y(),ori.z());
29
30 }
```

Two quaternions, `q1` and `q2`, are created and initialized by specifying the associated roll-pitch-yaw angles. A third quaternion, `q3`, is then obtained using the overloaded multiplication operator, and therefore represents the composite rotation. Next, a 3×3 rotation matrix `r` is initialized to store the same rotation associated with `q3`. Finally, two 4×4 transformation matrices, `t1` and `t2`, are created by specifying their rotation and translation components, and a third transformation matrix, `t3`, is obtained by combining them. Observe that when creating a transformation matrix, one can specify the rotation using either a quaternion or a rotation matrix, thus offering great flexibility. The output of the program is the following.

```
[INFO] [1673573523.415627261] [geom]: -0.270598 0.270598 0.653281 0.653281
[INFO] [1673573523.415738210] [geom]: 0.707107 5.000000 2.292893
```

4.13.3 Conversions between different representations

In Listing 4.4, we show a node that subscribes to a topic, receives data in one representation, and republishes it in a different format. As in the case of Listing 4.2, the node acts as both a publisher and a subscriber, and it makes use of the `ros::spin_some` function. The node receives the pose of the turtle as a message of type `turtlesim::msg::Pose`, which includes only two coordinates and the orientation, and then republishes it as a message of type `geometry_msgs::msg::Pose`, which includes the three-dimensional position and the orientation represented as a quaternion.

Listing 4.4: RepublishPose Node

```
1 #include <rclcpp/rclcpp.hpp>
```

```
2  #include <turtlesim/msg/pose.hpp>
3  #include <geometry_msgs/msg/pose.hpp>
4  #include <tf2/LinearMath/Quaternion.h>
5  #include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
6
7  float x;
8  float y;
9  float theta;
10 bool valid;
11
12 void poseReceived(const turtlesim::msg::Pose::SharedPtr msg) {
13     x = msg->x;
14     y = msg->y;
15     theta = msg->theta;
16     valid = true;
17 }
18
19 int main(int argc,char ** argv) {
20
21   rclcpp::init(argc,argv);
22   rclcpp::Node::SharedPtr nodeh;
23   nodeh = rclcpp::Node::make_shared("republishpose");
24
25   auto sub = nodeh->create_subscription<turtlesim::msg::Pose>
26                                        ("turtle1/pose",10,&poseReceived);
27   auto pub = nodeh->create_publisher<geometry_msgs::msg::Pose>("pose",1000);
28
29   geometry_msgs::msg::Pose poseToPublish;
30   tf2::Quaternion q;
31   valid = false;
32
33   while (rclcpp::ok()) {
34     rclcpp::spin_some(nodeh);
35     if (valid) {
36       poseToPublish.position.x = x;
37       poseToPublish.position.y = y;
38       poseToPublish.position.z = 0;
39       q.setRPY(0,0,theta);
40       poseToPublish.orientation = tf2::toMsg(q);
41       pub->publish(poseToPublish);
42       valid = false;
43     }
44   }
45 }
```

The `main` function starts with the usual initializations and creates both a subscriber and a publisher object. The subscriber is associated with the handler function `poseReceived`, which extracts the `x`, `y`, and `theta` components of the message and stores them in three global variables with matching names. In the `main` function, we run an infinite loop controlled by `rclcpp::ok()`. Within this loop, we check for messages from the topic `turtle1/pose` using the function `rclcpp::spin_some()`. If a message is received, the handler function copies the relevant values into the global variables and marks the data as valid. In the main loop, if a

message is marked as valid, we prepare an object of type `geometry_msgs::msg::Pose` by first copying the position values and then setting up the quaternion component `orientation`. To this end, we use an intermediate object of type `tf2::Quaternion` called `q`, and initialize it using the method `setRPY` with a given triplet of roll/pitch/yaw angles. Finally, we convert the object to an instance of `geometry_msgs::msg::Quaternion` and publish the message to the topic `pose`, which we previously created. After publishing the message, it is marked as invalid, to ensure that each incoming message is reposted only once. Note that we use the intermediate object because `geometry_msgs::msg::Quaternion` does not offer any method to easily create a quaternion from other parameterizations. We could have set those values manually using the formulas given in Section 4.5.4; however, that approach is error-prone, whereas the method we use here is shorter and relies on well-tested code. This node depends on the packages `turtlesim`, `geometry_msgs`, and `tf2`, and therefore the configuration files must be updated accordingly. Moreover, to run and test this code, the `turtlesim` environment must be up and running.

4.13.4 Transform tree

The transform library implements many of the ideas discussed when presenting transformation trees (see Section 4.8). `tf2` handles the transformation tree *buffered in time*, i.e., it is possible to look up spatial relationships either at the current time or in the past, e.g., 3 seconds ago. By default, the length of the temporal buffer is 10 seconds, but this can be changed. The ability to locate transformations back in time is particularly useful, considering that robots move around their environment and therefore many mutual relationships change over time. `tf2` also provides a set of tools (in the form of nodes) to debug applications, such as visualizing the transformation tree, performing coordinate transformations from the command line, and so on. Moreover, `tf2` standardizes how frames should be represented and named, and it defines some standard frames that are pervasively used in ROS. `tf2` is a rather sophisticated system, and it can operate in a fully distributed setup, with nodes dispersed over a network. In the following, we discuss just the fundamental principles and refer the reader to the official documentation for the full picture.

To take advantage of the `tf2` infrastructure, one needs to listen for transforms and/or broadcast transforms. By listening to transforms, one can receive relevant transforms broadcast in the system and then take advantage of the provided utilities to perform geometric operations. Conversely, by broadcasting transforms, one can communicate transforms to other nodes. In many instances, substantial benefits can be gained from `tf2` simply by listening to transforms broadcast by other nodes in the system. In fact, many standard ROS nodes are set up to automatically broadcast this information. Because the transformation tree is buffered in time, exchanged transforms are time-stamped, i.e., they are instances of `geometry_msgs::msg::TransformStamped`, which we analyzed earlier. Frames in ROS are given symbolic names expressed as strings, e.g., `map`, `base_link`, etc. Accordingly, transformations exchanged with `tf2` refer to frames using these symbolic names.

Frames are exchanged through two topics: `/tf` and `/tf_static`. The difference between the two is that, as the name suggests, `/tf_static` is used for transformations that do not change over time. This is useful, for example, to communicate the geometry of a robot where the transformation relating two objects rigidly connected to each other re-

mains constant. For example, referring to Image 4.3, the mutual position between the orange and green frames does not change over time and is therefore a good candidate for being exchanged through `/tf_static`. Both `/tf` and `/tf_static` exchange messages of type `tf2_msgs::TFMessage`. However, nodes do not usually publish or subscribe to these topics directly. Instead, they receive or send transformations using instances of the classes `TransformListener` and `TransformBroadcaster`. Listing 4.5 shows how a node can listen for various transformations using `tf2`. To run this example and observe the output, you must start a Gazebo simulation with frame names matching those used in the example:

```
ros2 launch gazeboenvs gazebo_husky.launch.py
```

The node instead must be run as follows:

```
ros2 run examples tflistener --ros-args  -r /tf:=/a200_0000/tf
```

This is another instance of remapping, first seen in section Section 4.12. The reason is that by default `TransformListener` listens to a topic called `/tf` but the Gazebo simulation uses a different topic name, i.e., `/a200_0000/tf`. Through remapping, we adjust this topic name mismatch.

The logic is rather simple. First, an instance of `tf2_ros::Buffer` is created. This is the storage area to buffer transforms up to 10 seconds in the past, as per the default configuration. Then, an instance of type `tf2_ros::TransformListener` is created and linked to the buffer. Inside the main loop, the method `lookupTransform` is used to return the transformation between two frames whose names are given as the first and second parameters. The third parameter specifies the desired *point in time*, and by specifying `tf2::TimePointZero` we indicate that we are requesting the most recent transformation. Note that the function returns the transform from the first parameter to the second parameter, i.e., in our example, it returns the coordinates of the frame `base_link` expressed in the frame `odom`. If we swap the order of the parameters, the code obviously still works and returns the inverse transformation matrix. This is consistent with our discussion in Section 4.8, where we outlined that if two frames are part of the same connected component, then it is possible to compute the transformation in either direction. This node depends on the packages `tf2` and `tf2_ros`, and the package and make files must be correspondingly updated.

Listing 4.5: Transformation Listener

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <tf2_ros/transform_listener.h>
3  #include <tf2_ros/buffer.h>
4  #include <geometry_msgs/msg/transform_stamped.hpp>
5
6  int main(int argc, char** argv){
7
8    rclcpp::init(argc,argv);
9    rclcpp::Node::SharedPtr nodeh;
10   nodeh = rclcpp::Node::make_shared("tflistener");
11
12   tf2_ros::Buffer buffer(nodeh->get_clock());
```

```
13    tf2_ros::TransformListener listener(buffer);
14    geometry_msgs::msg::TransformStamped transformStamped;
15
16    while (rclcpp::ok()){
17
18      try{
19        transformStamped = buffer.lookupTransform(
20                    "odom", "base_link", tf2::TimePointZero);
21
22      }
23      catch (tf2::TransformException &ex) {
24        RCLCPP_WARN(nodeh->get_logger(),"%s",ex.what());
25        rclcpp::Rate(1.0).sleep();
26        continue;
27      }
28      RCLCPP_INFO(nodeh->get_logger(),"Obtained transformation");
29      RCLCPP_INFO(nodeh->get_logger(),"Translation: %f %f %f",
30                    transformStamped.transform.translation.x,
31                    transformStamped.transform.translation.y,
32                    transformStamped.transform.translation.z);
33      RCLCPP_INFO(nodeh->get_logger(),"Rotation: %f %f %f %f",
34                    transformStamped.transform.rotation.x,
35                    transformStamped.transform.rotation.y,
36                    transformStamped.transform.rotation.z,
37                    transformStamped.transform.rotation.w);
38    }
39    return 0;
40  }
```

If, instead of the most recent transformation, we are looking for a transformation in the past, we can change the last parameter to indicate how far in the past we are looking. This can be done by initializing an instance of an object of type `rclcpp::Time` as follows:

```
rclcpp::Time pastlookup=nodeh->get_clock()->now()-tf2::durationFromSec(3);
```

In this case, `pastlookup` is initialized to refer to three seconds into the past, and if it is passed as the third parameter to the `lookupTransform`, the function will accordingly retrieve the past transformation from the time buffer[12] Since, by default, transformations are buffered in time for ten seconds only, one should not attempt to retrieve transformations more than ten seconds in the past (unless the default behavior is changed).

Finally, it is worth observing that `lookupTransform` does not look up just transforms that are directly connected to each other, but can traverse the tree and determine compounded transformations. For example, Figure 4.29 shows the frames attached to a Husky robot, while Figure 4.30 displays a part of the corresponding tree. If the above code is executed in a situation like this, the code can be modified to return the transform between `odom` and `front_left_wheel` even if are not directly connected.

Similarly, to broadcast a transformation to other nodes in the system, an object of type

[12] If you try making this change in Listing 4.5, it will not work because you need to tell ROS to retrieve time from the simulator and not from the system time. This will be explained in Chapter 5.

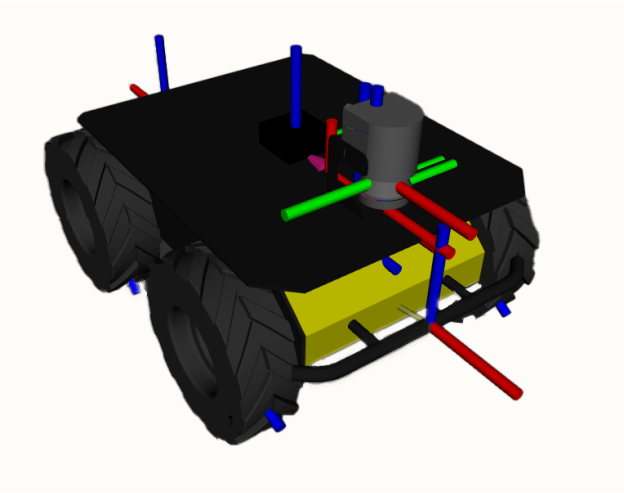


Figure 4.29: Frames associated with a Husky robot (see Figure 1.1).

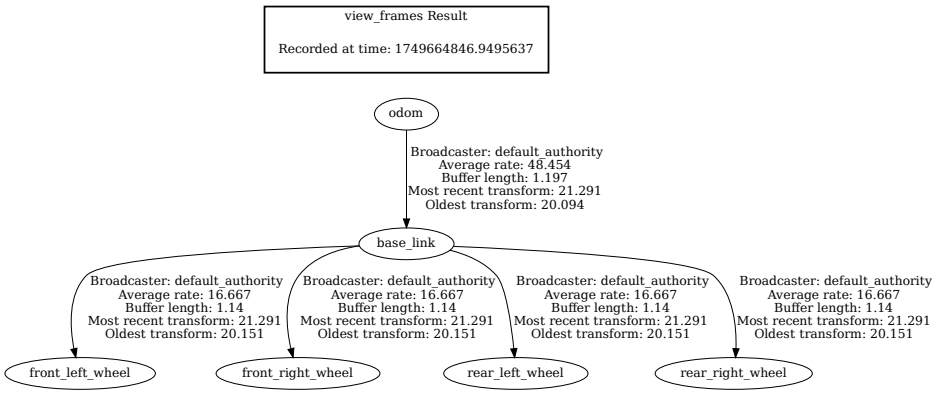


Figure 4.30: Tree showing the relationships for the frames attached to a Husky robot as returned by the Gazebo simulation.

`TransformBroadcaster` can be used. The mechanism is symmetric to the one we saw in Listing 4.5. An instance of `TransformStamped` is initialized as we described above, and then broadcast to the other nodes using the method `sendTransform`. Listing 4.6 shows this process.

Listing 4.6: Transformation Broadcaster

```
#include <rclcpp/rclcpp.hpp>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/msg/transform_stamped.hpp>
#include <tf2/LinearMath/Quaternion.h>

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

  rclcpp::init(argc, argv);
  rclcpp::Node::SharedPtr nodeh;
  nodeh = rclcpp::Node::make_shared("tfbroadcaster");

  tf2_ros::TransformBroadcaster broadcaster(nodeh);
  geometry_msgs::msg::TransformStamped transformStamped;
  tf2::Quaternion q;

  while (rclcpp::ok()){
    transformStamped.header.stamp = nodeh->get_clock()->now();
    transformStamped.header.frame_id = "base_link";
    transformStamped.child_frame_id = "myframe";
    transformStamped.transform.translation.x = 4.0;
    transformStamped.transform.translation.y = 0.0;
    transformStamped.transform.translation.z = 2.0;
    q.setRPY(0,0,0);

    transformStamped.transform.rotation.x = q.x();
    transformStamped.transform.rotation.y = q.y();
    transformStamped.transform.rotation.z = q.z();
    transformStamped.transform.rotation.w = q.w();
    broadcaster.sendTransform(transformStamped);
  }
  return 0;
}
```

If we execute this node

```
ros2 run examples tfbroadcaster
```

and then run `ros2 topic echo /tf`, we get and output like the following

```
---
transforms:
- header:
stamp:
sec: 1673894532
nanosec: 133698111
```

```
frame_id: base_link
child_frame_id: myframe
transform:
translation:
x: 4.0
y: 0.0
z: 2.0
rotation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0
```

thus confirming that the node is now broadcasting transformations that are managed by the `tf2` system.

Finally, `tf2` provides also some tools to inspect and debug the structure of the transformation tree. Among these, the package `tf2_tools` includes a node that listens for the frames broadcast in the system and builds the graph for the transformation tree. This utility can be run as follows:

```
ros2 run tf2_tools view_frames --ros-args  -r /tf:=/a200_0000/tf
```

and will generate and save a file with the transform tree structure. The tree shown in figure 4.30 was built using this utility. Note that in this case, too, we had to use topic remapping because `view_frames` looks for a topic called `/tf`.

4.13.5 Standard Frames

ROS has defined some conventions for naming frames. This facilitates code reuse and sharing, as long as these guidelines are consistently followed. The following frames are extensively used to name frames in ROS applications.

- `base_link` is a frame rigidly attached to the robot base. For mobile robots, this frame follows the conventions described in Section 4.9, i.e., the **x** axis points forward, **y** is to the left, and **z** points upwards. Being rigidly attached to the robot, `base_link` varies over time, for example when expressed with respect to the frames described later. If there are multiple robots in an application, each of them will have its own `base_link`, and care must be taken to avoid confusion. Importantly, there is no mandated rule for where `base_link` should be placed, but for differential drive and skid-steer robots it is very often placed at the center, so that a rotation in place by the robot corresponds to a rotation about the **z** axis with no change in the origin of the frame.

- `odom` is a world-fixed frame, i.e., it does not change over time. The pose of a robot expressed in this frame can drift over time, but remains continuous (as it would if the pose came from odometry—hence the name). Typically, the `odom` frame is placed where the robot starts. Sensors like encoders return the transformation between the

`base_link` frame and the `odom` frame. This will become clearer when discussing sensors in Chapter 7 and localization algorithms in Chapter 9.

- `map` is also a world fixed frame with the **z** axis pointing upwards. Differently from the `odom` frame, the pose of a robot expressed in this frame should not drift over time and is not supposed to be continuous.

- `earth` is another world fixed frame whose origin is at the center of the earth. This frame becomes useful if there are multiple robots operating using different map frames. In such case, `earth` is a common ancestor frame that can be used to related quantities expressed in different maps.

Both `odom` and `map` are world-fixed frames, meaning that they do not change over time. On the contrary, `base_link` is not a world-fixed frame because it is rigidly attached to the robot and therefore can vary over time as the robot moves. The different requirements for the `odom` and `map` frames suggest that the robot pose in the `map` frame is typically computed by localization algorithms like the particle filter (see Section 8.5) that match the frame specification. Instead, the robot pose in the `odom` frame should be computed through methods like encoders, inertial navigation units, and similar approaches that guarantee a continuous evolution over time. In many instances, the two frames are coincident, but this is not a requirement.

Figure 4.30 shows another important feature of how transform trees and standard frames are defined in ROS. Each frame can have multiple child frames, but exactly one parent. According to the previous discussion, `base_link` could be in principle expressed either in the `odom` frame or in the `map` frame, but this is not allowed because of the constraint of having a single parent. In the figure just `odom` is established, so there is no ambiguity. However, if both `map` and `odom` are present, then `map` is the parent frame for `odom` (see Figure 4.31).

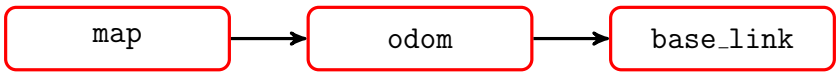


Figure 4.31: Arrangement of standard frames in ROS.

Finally, if there are multiple robots, it is possible that each of them is started with its own `map` and `odom` frames. In this situation, the frame `earth` is used as well, and the arrangement of frames is shown below (compare this with figure 4.19).

Further reading

Kinematics is a fundamental topic discussed in most robotics textbooks, like for example [16, 27, 49, 51], just to name a few. A discussion about the design of the `tf` package is given in [22].

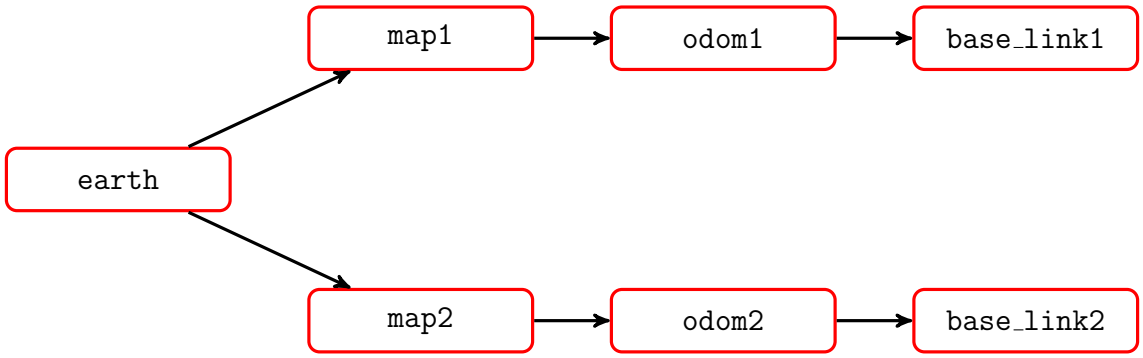


Figure 4.32: Arrangement of standard frames in ROS for a multiple robot application.

Chapter 5

Additional ROS concepts

In this chapter, we introduce some more advanced ROS concepts that can be useful as you start developing complex ROS applications.

5.1 Remapping

We saw in Chapter 2 that `ros2 run` is used to run executables. The help information provided by `ros2 run -h` shows that everything coming after the executable name becomes an argument passed to the executable. While an executable can, in general, accept arbitrary arguments, there are a few that are standard and are associated with each ROS executable. One of these is the name of the node, which can be altered when the executable is started as follows (note that instead of `--remap`, one can also use the shorter `-r`):

```
ros2 run talklisten talker --ros-args --remap __name:=newtalker
```

We can verify that the name of the node has changed running `ros2 node list` which will produce

```
/newtalker
```

and the same information is observed in the output produced by the node:

```
[INFO] [1673935144.419936102] [newtalker]: Sending message #1
[INFO] [1673935145.409056027] [newtalker]: Sending message #2
[INFO] [1673935146.409073411] [newtalker]: Sending message #3
[INFO] [1673935147.409061516] [newtalker]: Sending message #4
[INFO] [1673935148.408931236] [newtalker]: Sending message #5
```

In general, an executable can initiate more than one node through a process called composition. When that is the case, a slightly different syntax should be used to specify which node we are renaming. This last example is an instance of *remapping*, i.e., the process of altering some default properties of a node (the name in this case). We have already seen a remapping example in Section 4.12, where we remapped the name of a topic to redirect

the messages of `drawsquare` to the simulated Turtlebot robot, which expected the twist commands on a topic with a different name. This use of remapping is extremely common, as it allows repurposing existing nodes without having to recompile them. The general form is the following:

```
ros2 run packagename executablename --ros-args --remap
                                        originaltopicname:=newtopicname
```

where, in this case, the remapping involves the name of a topic. In general, multiple topics can be remapped if needed. Remapping can be applied to the node name, topic names, and namespaces, a feature that will be introduced in the next section. Remappings can also be specified in launch files, as we will see later.

5.2 Namespaces

Namespaces in ROS are introduced to avoid name clashes, similarly to the concept used in programming languages like C++. For example, if multiple instances of the same node are run, they would use the same names for resources like topics. We have already seen an example of this behavior when running the different talker/listener nodes. In some instances, this may be acceptable, but in others, it may be necessary to keep separate the resources associated with each node. For example, we may desire that the talker sends its message to a specific listener, and not to every listener. Similarly, two nodes developed by different developers could use the same name for a service or a topic. These name clashes can be avoided by assigning a different namespace to each node. Namespaces can be specified in different ways. First, we can remap the namespace from the command line, similarly to what we have just done for the node name or the topic names:

```
ros2 run talklisten talker --ros-args --remap __ns:=/t1
```

With this syntax we are starting the executable `talker` and its resources will be associated with the namespace `t1`. For example, if we run `ros2 node list` we get

```
/t1/talker
```

and if we run `ros2 topic list` we get

```
/parameter_events
/rosout
/t1/message
```

Moreover, the output produced by the node now looks like the following

```
[INFO] [1674254684.393268432] [t1.talker]: Sending message #1
[INFO] [1674254685.378026668] [t1.talker]: Sending message #2
```

```
[INFO] [1674254686.378038624] [t1.talker]: Sending message #3
[INFO] [1674254687.377929448] [t1.talker]: Sending message #4
[INFO] [1674254688.378053505] [t1.talker]: Sending message #5
[INFO] [1674254689.377921921] [t1.talker]: Sending message #6
[INFO] [1674254690.377917991] [t1.talker]: Sending message #7
```

where we see that the name of the node is now prefixed by the assigned namespace (i.e., the name is `t1.talker`). Hence, by having started the node with a namespace called `/t1`, the resources (topics, names, services, etc.) associated with `talker` now have names prefixed by `/t1`. It should be clear that if we now run `listener`, the node will not print anything on the screen, because it subscribes to a topic called `message`, but `talker` is now publishing to `/t1/message`. Of course, this can be fixed by remapping the topic for `listener`, too:

```
ros2 run talklisten listener --ros-args --remap message:=/t1/message
```

As it may be cumbersome to specify namespaces, remappings, and so on from the command line, these can be specified in a launch file, as shown in Listing 5.1.

Listing 5.1: Launch file with namespaces and remapping

```
1  <launch>
2    <node pkg="talklisten"
3          exec="talker"
4          name="newtalker"
5          namespace = "/t1"
6          launch-prefix="gnome-terminal --" />
7    <node pkg="talklisten"
8          exec="listener"
9          name="listener">
10     <remap from="/message" to="/t1/message" />
11   </node>
12 </launch>
```

The key elements to note are that the namespace can be specified as an attribute of the tag `node`, while the topic remapping is an element of the node, with attributes `from` and `to` (with their obvious meanings). The attribute `name`, which we already encountered, specifies the name for the node in the executable.

5.3 ROS names

Now that we have introduced namespaces, we can revisit the naming conventions for ROS resources such as topics, services, and nodes to better understand how to organize and name resources. As one can imagine, having two resources with exactly the same name may sometimes be desired but may also lead to unexpected consequences. For example, if from two different consoles we start the executable `talker` from the `demo_nodes_cpp` package, the two executables start and each spawns an instance of a node called `talker`. Both nodes publish strings to the same topic named `chatter`. With regard to publishing to the same topic, this is not a problem, as we indeed discussed that topics are many-to-many

communication channels identified by their names, so having multiple nodes publishing to a topic with a shared name is a feature and not a bug. But in some instances, we may desire to keep things separate. This was, for example, the case when we spawned multiple turtles in the `turtlesim` environment. In that case, it was essential for each turtle to have its own dedicated topic to receive velocity commands. Before moving forward, it is useful to run `ros2 node list` while the two executables are running. The output is:

```
WARNING: Be aware that there are nodes in the graph that share an exact name,
this can have unintended side effects.
/talker
/talker
```

This shows that, in general, it is possible to have two nodes with the same name, but the warning message reminds us that we should carefully consider whether this is what we really want. At this point, if we prefer the nodes to have different names to avoid ambiguity, we can use one of the remapping strategies discussed in the previous sections, either changing the node names or starting the nodes in different namespaces. Both approaches assign different names to the nodes, thus eliminating the warning, but the overall effects differ. If we start the nodes[1] by remapping the name using `--remap __name`, they will both continue to publish to the same topic called `/chatter`. Indeed, what we did was just change the name of the node. `ros2 node list` can be used to confirm that the nodes have different names, but the name of the topic they publish to remains unchanged. If instead we assign a different namespace to the executables when they start, we end up with nodes that not only have different names but also publish to different topics. For example, we can start one of the two nodes in a different namespace as follows:

```
ros2 run demo_nodes_cpp talker --ros-args --remap __ns:=/t2
```

If we now run `ros2 node list`, the output will be:

```
/t2/talker
/talker
```

and the output of `ros2 topic list` will be (omitting topics unrelated to the discussion):

```
/chatter
/t2/chatter
```

As we now know, the node started in the `/t2` namespace is publishing to `/t2/chatter`, i.e., the new namespace has been used to remap the name of the topic it publishes to. If we now start the `listener` node from the same package, it will only receive messages from one of the nodes, because it subscribes to the topic `/chatter`, and not to `/t2/chatter`.

At this point, we can introduce the concept of *fully qualified names* in ROS. Topics and services have both fully qualified names and relative names. A fully qualified name starts

[1] As a matter of fact, to get different names you need to remap just one of the two.

with the / character and explicitly identifies the namespace in which the resource exists. In contrast, relative names do not start with / and are interpreted with respect to the namespace of the node that creates or accesses the resource. If a resource is created without setting a namespace, its fully qualified name is simply the name of the resource preceded by a /. If instead the resource is created within a namespace, its fully qualified name is constructed by prefixing the namespace name followed by a /, then the resource name.

This behavior explains some of the patterns we observed earlier in our examples. For instance, if we revisit Listing 3.2, we see that the code publishes to a topic called `message`. If we start the executable without remapping the namespace and then run `ros2 topic list`, we will see the topic listed as `/message`, which is its fully qualified name. However, if we start the same node with the command `--ros-args --remap __ns:=/test`, we will find that `ros2 topic list` now shows `/test/message` as the topic name. This is because the publisher was created using a relative name (`message`), and the namespace `/test` is automatically prepended to form the fully qualified name. As we will see in a later example, it is also possible to access resources using their fully qualified names (starting with /) directly from within a ROS node. In some cases, doing so is not only possible but also necessary to ensure the correct behavior of the system. Finally, this is why when we remap a namespace using `--ros-args --remap __ns`, the namespace must be specified as an absolute path—i.e., it must start with the / character.

5.4 Parameters

While node names and namespaces are associated with all ROS nodes, it is also possible to introduce parameters that are specific to a particular node being developed. These parameters can be specified in various ways: directly in the launch file, from the command line, or via a configuration file. For example, a node that interfaces with a laser range finder may require a parameter to set its resolution, or a node that receives data from a GPS sensor may need to operate at different frequencies depending on the application. Such values can be declared as parameters with sensible default values to be used when no specific values are provided by the user. However, they can also be dynamically changed from the command line or passed when launching the executable, either using `ros2 run` or within a launch file. Regardless of the method used for setting the parameters, if a node relies on parameters in its source code, developers can use classes and functions from the `rclcpp` package to: declare parameters, retrieve their values, and modify them as needed during runtime. Additionally, the `ros2` command-line tool provides the `param` command, which allows users to interact with node parameters from the terminal. For example, if we start both the `talker` and `listener` nodes from the `demo_nodes_cpp` package, and we run the following command:

```
ros2 param list
```

we obtain the following output:

```
/listener:
qos_overrides./parameter_events.publisher.depth
```

```
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
start_type_description_service
use_sim_time
/talker:
qos_overrides./parameter_events.publisher.depth
qos_overrides./parameter_events.publisher.durability
qos_overrides./parameter_events.publisher.history
qos_overrides./parameter_events.publisher.reliability
start_type_description_service
use_sim_time
```

`ros2 param list` lists all parameters associated with the nodes currently running. The output shows that, among others, the nodes are associated with a parameter called `use_sim_time`. In fact, this parameter is included by default in all ROS nodes. Its purpose will be explained in more detail later on, but in brief, it allows a node to choose whether to use the system's real-time clock or a simulated clock. This is an essential feature when running simulations, for example in Gazebo or RViz. If we want to learn more about `use_sim_time` we can use

```
ros2 param describe talker use_sim_time
```

where we specify the name of node and the name of a parameter associated with it. In this case the output will be

```
Parameter name: use_sim_time
  Type: boolean
  Constraints:
```

showing that `use_sim_time` is boolean parameter not associated with any constraint. If we want to retrieve the value of the parameter we execute

```
ros2 param get talker use_sim_time
```

end we get

```
Boolean value is: False
```

These interactions from the command line reveal that parameters are local to nodes and not global. Both `talker` and `listener` have a parameter named `use_sim_time`, but each node maintains its own copy, which is not shared with the other. This is why we must specify the name of the node when using commands such as `describe`, `get`, and others we will encounter later.

Having established the basics of interacting with parameters from the command line, we now turn our attention to how parameters can be accessed programmatically from within a node. Listing 5.2 shows how a node can declare a parameter and retrieve its value.

Listing 5.2: Parameter Client

```
1  #include <rclcpp/rclcpp.hpp>
2
3  int main(int argc,char **argv) {
4
5    rclcpp::init(argc,argv);
6    rclcpp::Node::SharedPtr nodeh;
7    nodeh = rclcpp::Node::make_shared("paramclient");
8    // declare a parameter of type string called sensorport
9    nodeh->declare_parameter<std::string>("sensorport","/dev/tty0");
10   std::string port;
11   while (rclcpp::ok()) {
12     // retrieve the parameter as a string
13     port = nodeh->get_parameter("sensorport").get_parameter_value().
14                                                    get<std::string>();
15     RCLCPP_INFO(nodeh->get_logger(), "Parameter value: %s", port.c_str());
16     rclcpp::spin_some(nodeh);
17   }
18
19 }
```

Parameters have a name (a string) and are typed, and the type must be specified when the parameter is declared (a string in this case, but other types can be used, too). Importantly, when the parameter is declared, a default value must be specified for when the user does not specify a value for the parameter. In this case, we declare a parameter called `sensorport` of type string and assign `/dev/tty0` as the default value. In the main loop, we retrieve the parameter using its name, get its value, and convert it to the correct type (string) before printing it to the screen. If you run this node, which is part of the `examples` package, it will continuously print to the screen the default value that was hard-coded in the source:

```
[INFO] [1674454638.194856406] [paramclient]: Parameter value: /dev/tty0
```

Obviously, greater flexibility is achieved if the parameter is changed from outside. The first option is to specify the value of the parameter when the node is started using `ros2 run`. This can be done as follows, where we use the option `-p`, followed by the name of the parameter and its value:

```
ros2 run examples paramclient --ros-args -p "sensorport":="/dev/USB0"
```

In this case the output will be

```
[INFO] [1674454718.161467518] [paramclient]: Parameter value: /dev/USB0
```

where we can see that now the parameter specified when starting the executable has superseded the default value. Another option is to change the value of a parameter on the fly, after the node has already started. This can be done using the command `ros2 param set`. To set the value of the parameter `sensorport` we use the following syntax

```
ros2 param set paramclient sensorport /dev/USB1
```

where we specify the node name, the parameter name, and the new value. This command must be given after the node is already running; otherwise, it will return an error message. In this case, if we first start `paramclient` and then execute this command, we will see that the value of the parameter changes at runtime. The final way to change a parameter at start time is by including its desired value in the launch file, as shown in Listing 5.3.

Listing 5.3: Launch file with parameters

```
1 <launch>
2   <node pkg="examples"
3         exec="paramclient"
4         name="param">
5     <param name="sensorport" value="/dev/USB0" />
6   </node>
7 </launch>
```

Parameters can also be changed by a node using the function `set_parameter_value` that mirrors the function `get_parameter_value` we saw in Listing 5.2.

The methods described thus far become cumbersome when a node depends on multiple parameters. In such cases, it is more practical to group all parameters in a text file that can be easily edited and then pass the parameters when the node is run. To this end, ROS provides the command `dump`. To show all of the parameters of a node in YAML format, one can run the following command:

```
ros2 param dump <nodename>
```

which will print to the screen the parameters of the named node. Importantly, these parameters can be saved to a file using redirection, then edited and reused at a later time. To load the parameters saved in a YAML file into a node, there are two methods. The first is to use the command `load` as follows:

```
ros2 param load <nodename> <parameterfile>
```

where `parameterfile` is the name of a YAML file with the parameters and values (either previously generated with `dump` and possibly edited, or created from scratch). To use this method, the node must already be up and running. Another method consists of passing the name of the parameter file when the node is started. For example, assuming that `parameters.yaml` is the name of the file with the parameters we want to pass to the node `listener` in the package `talklisten`, we can start it as follows:

```
ros2 run talklisten listener --ros-args --params-file parameters.yaml
```

The advantage is that, in this case, the node is started and the parameters are loaded with a single command. Finally, it is also possible to specify the name of the parameter file in the launch file. This is illustrated in listing 5.4.

Listing 5.4: Launch file with parameters in YAML file

```
<launch>
  <node pkg="examples"
        exec="paramclient">
        <param from="$(find-pkg-share examples)/launch/parameters.yaml"/>
  </node>
</launch>
```

Note that in this case we use the macro `$(find-pkg-share examples)` to specify the relative location of the parameters file rather than the absolute path. With this setup, the `parameters.yaml` file is found in the `launch` folder of the `examples` package. These files are often placed there, or, perhaps more commonly, in a separate dedicated folder typically called `config`.

5.4.1 YAML configuration files for ROS

YAML files can be generated with the `dump` command or created from scratch using a text editor. Listing 5.5 shows the content of `parameters.yaml`, which can be used as a template.

Listing 5.5: YAML configuration file

```
/paramclient:
  ros__parameters:
    sensorport: /dev/tty5
    use_sim_time: false
```

In ROS, YAML parameter files follow a specific structure. The first entry is the name of the node receiving the parameters, followed by `ros__parameters`. Then, according to the YAML format, a sequence of key–value pairs follows. It is important to remember that the YAML file format requires indentation to separate different sections, and this requirement must be strictly followed. Notably, a single file can specify parameters for multiple nodes, allowing the same configuration file to be passed to each node. To achieve this, simply list the name of each node, followed by `ros__parameters` and then the parameters for that node.

5.4.2 Runtime parameters changes

In the previous discussion, we have seen that a parameter can be set when the node starts, but it can also be changed while the node is running by using `ros2 param set`. This raises the question of how a node can be notified on the fly that a parameter has been changed during its execution, and how it can appropriately react to it. Parameter changes can be intercepted by a node by subscribing to a topic called `parameter_events`, which is created every time a ROS executable starts. In fact, we have already seen this topic when practicing the command `ros2 topic list` in Section 2.5. When a parameter is changed, a message is published to `parameter_events`, and nodes subscribed to the topic will receive a message with details about the change. `ros2 topic info` shows that `parameter_change` has type `rcl_interfaces/msg/ParameterEvent`, and `ros2 interface show` displays the following structure:

```
# The time stamp when this parameter event occurred.
builtin_interfaces/Time stamp
  int32 sec
  uint32 nanosec

# Fully qualified ROS path to node.
string node

# New parameters that have been set for this node.
Parameter[] new_parameters
  string name
  ParameterValue value
    uint8 type
    bool bool_value
    int64 integer_value
    float64 double_value
    string string_value
    byte[] byte_array_value
    bool[] bool_array_value
    int64[] integer_array_value
    float64[] double_array_value
    string[] string_array_value

# Parameters that have been changed during this event.
Parameter[] changed_parameters
  string name
  ParameterValue value
    uint8 type
    bool bool_value
    int64 integer_value
    float64 double_value
    string string_value
    byte[] byte_array_value
    bool[] bool_array_value
    int64[] integer_array_value
    float64[] double_array_value
    string[] string_array_value

# Parameters that have been deleted during this event.
Parameter[] deleted_parameters
  string name
  ParameterValue value
    uint8 type
    bool bool_value
    int64 integer_value
    float64 double_value
```

```
    string string_value
    byte[] byte_array_value
    bool[] bool_array_value
    int64[] integer_array_value
    float64[] double_array_value
    string[] string_array_value
```

The interpretation is straightforward. An event has a timestamp `stamp` and is associated with a node named `node`. In addition, an event can include new parameters, changed parameters, and deleted parameters, each of which is stored in the field with the corresponding name and is an array of elements of type `rcl_interfaces/msg/Parameter`. A parameter is composed of a string `name` and a `value` of type `ParameterValue`. A key field in this message is `type`, which indicates the type of the parameter (recall that all parameters are typed). The value stored in `type` indicates which field should be read. Values for `type` are defined in `ParameterType`, which is as follows:

```
# These types correspond to the value that is set in the ParameterValue message.

# Default value, which implies this is not a valid parameter.
uint8 PARAMETER_NOT_SET=0

uint8 PARAMETER_BOOL=1
uint8 PARAMETER_INTEGER=2
uint8 PARAMETER_DOUBLE=3
uint8 PARAMETER_STRING=4
uint8 PARAMETER_BYTE_ARRAY=5
uint8 PARAMETER_BOOL_ARRAY=6
uint8 PARAMETER_INTEGER_ARRAY=7
uint8 PARAMETER_DOUBLE_ARRAY=8
uint8 PARAMETER_STRING_ARRAY=9
```

Therefore, based on the integer value assigned to `type`, we know which field in an instance of `ParameterValue` should be read. For example, if `type` is 2, the value is stored in `integer_value`, and so on. With this information, we can now write a new node that declares a parameter `order` of type integer and subscribes to `parameter_event` to take action when the parameter is changed (in this case, for simplicity, we simply print a message to the screen). The complete program is given in Listing 5.6.

Listing 5.6: Node reacting to parameter events

```
1 #include <rclcpp/rclcpp.hpp>
2 #include <rcl_interfaces/msg/parameter_event.hpp>
3 #include <rcl_interfaces/msg/parameter_type.hpp>
4
5 rclcpp::Node::SharedPtr nodeh;
6
7 // callback function called when a parameter event is received
8 void processEvent(const rcl_interfaces::msg::ParameterEvent::SharedPtr msg) {
9   if (msg->node == "/paramevent") {   // is this event for the current node?
```

```
10      if ( msg->changed_parameters.size() > 0 ) { // any parameter changed?
11        // scan all changed parameters
12        for ( unsigned int i = 0 ; i < msg->changed_parameters.size() ; i++ )
13          if (msg->changed_parameters[i].name == "order"){ // changed order?
14            if(msg->changed_parameters[i].value.type == // is it an integer?
15               rcl_interfaces::msg::ParameterType::PARAMETER_INTEGER) {
16              int order;  // get it
17              order = nodeh->get_parameter("order").get_parameter_value().
18                get<int>();
19              // print it
20              RCLCPP_INFO(nodeh->get_logger(), "Parameter order has changed");
21              RCLCPP_INFO(nodeh->get_logger(), "New value value: %d", order);
22            }
23        }
24      }
25    }
26  }
27
28  int main(int argc,char **argv) {
29
30    rclcpp::init(argc,argv);
31    nodeh = rclcpp::Node::make_shared("paramevent");
32    // subscriber to be notified of changes to parameters
33     auto sub = nodeh->create_subscription<rcl_interfaces::msg::ParameterEvent>
34       ("parameter_events",10,&processEvent);
35    // declare a parameter of type integer called order
36    nodeh->declare_parameter<int>("order",5);
37    int order;
38    order = nodeh->get_parameter("order").get_parameter_value().get<int>();
39    RCLCPP_INFO(nodeh->get_logger(), "Initial value: %d", order);
40    RCLCPP_INFO(nodeh->get_logger(), "Waiting...");
41    // just wait for events to happen...
42    rclcpp::spin(nodeh);
43
44
45  }
```

The main function subscribes to the topic `parameter_events`, declares an integer parameter called `order`, prints the initial default value, and then cedes control to the non-returning `spin` function that handles messages. The core logic is found in the callback function `processEvent`. For each received parameter event, the function first checks if the event refers to the current node by comparing the names. If this is the case, it scans all changed parameters, and if it determines that the parameter `order` has been modified, it retrieves its value as an integer and prints it to the screen. This example also demonstrates something new: how we can access symbolic constants declared in messages of type `rcl_interfaces/msg/ParameterType`. It is also worthwhile to observe that this node depends on the package `rcl_interfaces`, and therefore the manifest file and `CMakeLists.txt` must be updated accordingly (see files in the GitHub repository for details). If we run the node `paramevent` (part of the `examples` package) and then from a separate shell we give the command

```
ros2 param set paramevent order 11
```

the output will be

```
[INFO] [1674956136.149450904] [paramevent]: Initial value: 5
[INFO] [1674956136.149729008] [paramevent]: Waiting...
[INFO] [1674956142.647169868] [paramevent]: Parameter order has changed
[INFO] [1674956142.647300391] [paramevent]: New value value: 11
```

thus showing that the node has correctly identified at run time that the value of the parameter `order` associated with itself has been altered by an entity outside the node.

5.4.3 The parameter `use_sim_time`

As formerly stated, all nodes in ROS are associated with a parameter called `use_sim_time` that can be set or retrieved as discussed above. `use_sim_time` is a boolean parameter telling ROS whether it should retrieve the time from the system clock or from a special topic called `/clock`. This latter option is meaningful when a node is interacting with a simulator (e.g., Gazebo, discussed later in this chapter) rather than with the real world. In this case, setting `use_sim_time` to true tells the node to retrieve the time (clock) from the simulator and not from the system clock. This is useful when the robot must reason about time, for example, when retrieving transformations from the past using `tf2`. Note that when a node is started, `use_sim_time` is set by default to `false`, so one should set `use_sim_time` to true when an executable running in simulation is started.

5.5 Calling Services

As we anticipated in Chapter 2, in addition to topics, ROS provides *services*, i.e., a method to implement procedure calls between nodes. In this paradigm, a *server* node offers a service, and a *client* node uses the service. In essence, services provide a mechanism to perform remote procedure calls between nodes, where input parameters and results are exchanged through messages. It is also worth noting that although this approach resembles the client/server paradigm, nodes can act as both clients and servers in the same application. As with topics and messages, in addition to using standard services provided by ROS, it is possible to create new services, although this will not be covered in this book. We saw an example of services in action in Section 2.10, where we used `ros2 service call` to call a service from the command line. In this section, we now show how we can write nodes that can call services. To this end, we use a Gazebo simulated environment as we did in Section 4.12:

```
ros2 launch gazeboenvs tb4_simulation.launch.py
```

This will start Gazebo and spawn a simulated TurtleBot[2] robot equipped, among other

[2]See `https://www.turtlebot.com/` for more information about this platform.

things, with a range scanner that can be used to build a map of the environment (more details about sensors will be provided in Chapter 7). Most importantly, the launch command will also start various nodes, including a `slam_toolbox` node that builds a map and publishes it on the topic `map`, and a node `map_saver` that offers a service `save_map` which can save the map to a file. This information about topics and services can be obtained using the various `ros2` commands we previously discussed and that should be familiar to the reader by now. `ros2 service list -t` shows that the type of `save_map` is `nav2_msgs/srv/SaveMap` and with

```
ros2 interface show nav2_msgs/srv/SaveMap
```

we see that its structure is

```
# URL of map resource
# Can be an absolute path to a file: file:///path/to/maps/floor1.yaml
# Or, relative to a ROS package: package://my_ros_package/maps/floor2.yaml
string map_topic
string map_url
# Constants for image_format. Supported formats: pgm, png, bmp
string image_format
# Map modes: trinary, scale or raw
string map_mode
# Thresholds. Values in range of [0.0 .. 1.0]
float32 free_thresh
float32 occupied_thresh
---
bool result
```

The service accepts as parameters four strings specifying the name of the topic where the map is published (`map_topic`), the name of the file where it should be saved (`map_url`), the image format (`image_format`), and the type (mode) of the map (`map_mode`). The last two parameters, of type float, specify the thresholds used to classify a map cell as free or occupied (mapping will be discussed in detail in Chapter 9). The service returns a boolean result indicating whether the service succeeded or not, i.e., whether the map was saved. In Listing 5.7, we show how a service call is performed from inside a node rather than from the command line.

Listing 5.7: Service Client

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <nav2_msgs/srv/save_map.hpp>
3
4  int main(int argc,char **argv) {
5
6      rclcpp::init(argc,argv);
7      rclcpp::Node::SharedPtr nodeh;
8      nodeh = rclcpp::Node::make_shared("servicecall"); // create node
9
10     // create client for service
```

```
11      rclcpp::Client<nav2_msgs::srv::SaveMap>::SharedPtr client =
12      nodeh->create_client<nav2_msgs::srv::SaveMap>("/map_saver/save_map");
13
14      // wait indefinitely for service to become available
15      while (! client->wait_for_service())
16        RCLCPP_INFO(nodeh->get_logger(),"Waiting for service to be available");
17
18      // create a request object for the SetCameraInfo service
19      auto request = std::make_shared<nav2_msgs::srv::SaveMap::Request>();
20      request->map_topic = "/map";
21      request->map_url = "./mymap";
22      request->image_format = "png";
23      request->map_mode = "trinary";
24      request->free_thresh = 0.2f;
25      request->occupied_thresh = 0.8f;
26      // send request to server
27      auto response = client->async_send_request(request);
28      if (rclcpp::spin_until_future_complete(nodeh, response) ==
29          rclcpp::FutureReturnCode::SUCCESS) {  // waited and got success?
30        // print result
31        RCLCPP_INFO(nodeh->get_logger(), "Success? %d",
32                    response.get()->result);
33      }
34      else // Error:
35        RCLCPP_ERROR(nodeh->get_logger(),
36                     "Error calling service save_map");
37
38      rclcpp::shutdown();
39      return 0;
40
41  }
```

The code starts with the usual node initialization. Next, to call a service from within the node, we create an instance of the class `rclcpp::Client`, where, through templates, we specify the type of service (`nav2_msgs::srv::SaveMap`) as well as the name of the service we want to call (`/map_saver/save_map`). In this case, we use the fully qualified name of the service and not a relative name, because we want our code to work correctly even if its namespace is remapped at startup. Note that `create_client` is a member function of the class `rclcpp::Node`. Next, before calling the service, we have to ensure that the server node is ready to provide the service. This is achieved with the `wait_for_service` function of the class `rclcpp::Client`. In this example, since we do not pass any parameters, the client waits indefinitely; however, it is also possible to pass a timeout to abort the wait after a specified deadline. After having ascertained that the server is ready, we create an instance of the request message for `nav2_msgs::srv::SaveMap`, where we set the parameters described earlier (topic name, file name, etc.). At this point, we can send the request to the server by calling the method `async_send_request` of the `client` object. To wait for the response message, we use a new version of the spin function, namely `spin_until_future_complete`, which takes two parameters[3], namely a node and a so-called *future*. This function blocks until it receives a response from the server. The response from the server is stored in the

[3]The function may actually accept a timeout as well, but we do not discuss it here.

`response` object, and the function returns a code indicating success or failure of the spin. Then, the client accesses the fields in the response object and prints the result to the screen.

5.6 OOP in ROS

In all examples seen so far, we created instances of the class `rclcpp::Node`, used some of its methods to create publishers and subscribers, and then spun when needed. This approach is legitimate and viable for small applications, but it is not the recommended approach for larger software systems. In particular, we have seen that having certain functionalities such as callback functions external to the node required storing some data globally, with all the associated drawbacks (see, e.g., the examples in Chapter 3). Additionally, some of the logic governing how nodes work was not encapsulated within the node itself and in our examples it was always placed in the `main` function or in an external callback function.

These issues can be overcome by embracing object-oriented programming (OOP) in ROS. This is done by creating nodes as instances of newly defined classes inheriting from the class `rclcpp::Node`. This new class encapsulates all the data and processing naturally associated with a node. In the following, we first revisit the simplest examples from Chapter 3 and then introduce some new features. Listing 5.8 shows how the example in Listing 3.2 can be rewritten using OOP.

Listing 5.8: Talker using classes

```
#include <rclcpp/rclcpp.hpp> // needed for basic functions
#include <std_msgs/msg/string.hpp> // needed because we publish strings

class Talker : public rclcpp::Node {
public:
  Talker() : Node("talkeroop") {
    // create publisher
    pub = this->create_publisher<std_msgs::msg::String>("message",1);
    // create rate at 1Hz
    rate = std::make_shared<rclcpp::Rate>(1);
  }

  void run() {
    std_msgs::msg::String stringtosend;
    int counter = 0;
    while ( (counter++ < 100) && (rclcpp::ok())  ) {
      RCLCPP_INFO(this->get_logger(),"Sending message #%d",counter);
      // prepare message to send
      stringtosend.data = "Message #" + std::to_string(counter);
      pub->publish(stringtosend); // publish message
      rate->sleep(); // wait
    }
  }

private:
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr pub;
  rclcpp::Rate::SharedPtr rate;
};
```

```
29
30 int main(int argc,char **argv) {
31
32   rclcpp::init(argc,argv); // initialize the ROS subsystem
33   Talker node; // create node
34   node.run();
35   rclcpp::shutdown(); // shutdown ROS
36   return 0;
37 }
```

We declare a new class `Talker` that inherits from `rclcpp::Node`, and inside its constructor we create the resources needed by the node, namely the publisher and the `Rate` object. These are *private* data members and therefore not accessible from the outside. The main logic of the node (iterating 100 times and publishing corresponding messages to the topic) is now included in a public `run` method inside `Talker`. In the main function, we simply create an instance[4] of `Talker` and call its `run` method.

Before moving to the listener node, it is worthwhile observing that `Talker` is a *producer*, i.e., a node that generates data (strings) to be consumed by other nodes without receiving its data from other nodes through topics. This paradigm is typical of nodes interfacing with sensors, whereby the node polls the sensor at regular intervals. For this reason, as we will see later, it makes more sense to declare a new callback function that will be called at regular intervals by a pre-initialized timer. Such callback function will be dealt with by the `spin` function, like the callback functions associated with incoming messages. It follows that the approach we presented in listing 5.8, where we embedded the logic in a function explicitly called, is not very common, albeit correct.

In listing 5.9 we instead present the OOP version of the listener node formerly presented in listing 3.3. Like in the previous example, we declare a class that inherits from `rclcpp::Node`, and inside its constructor we create the subscriber. The main difference with listing 3.3 is that the callback function is now a member function, and so there is no need to declare a global variable to have a reference to the node and retrieve its logger. This approach eliminates the need for most global variables we saw in earlier examples. In the main function we observe that the `spin` function receives, as usual, a pointer to an instance of a node, and in this case the instance is generated on the fly when the function is called.

Listing 5.9: Listener using classes

```
1 #include <rclcpp/rclcpp.hpp> // needed for basic functions
2 #include <std_msgs/msg/string.hpp> // needed because we receive strings
3
4
5 class Listener : public rclcpp::Node  {
6 public:
7   Listener():Node("listeneroop") {
8     sub = this->create_subscription<std_msgs::msg::String>
9       ("message",10,std::bind(&Listener::callback,this,std::placeholders::_1));
```

[4]The reader may observe that in this case we do not use a pointer, but rather create an instance of the class. This is due to some C++ technicalities and makes the code shorter when it comes to calling `run`. As explained, in the future we will most often use callbacks, so this way of directly calling methods will not be frequently used.

```
10    }
11
12  private:
13    // callback function called every time a message is received from the
14    // topic "message"
15    void callback(const std_msgs::msg::String::SharedPtr msg) {
16      // process the message: just print it to the screen
17      RCLCPP_INFO(this->get_logger(),"Received: %s",msg->data.c_str());
18    }
19
20    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub;
21
22  };
23
24  int main(int argc,char **argv) {
25
26    rclcpp::init(argc,argv); // initialize ROS subsystem
27    rclcpp::spin(std::make_shared<Listener>());  // create and spin
28    rclcpp::shutdown();
29    return 0;
30
31  }
```

We conclude this section presenting an alternative version of the talker using a callback function triggered by a timer. The source code is provided in listing 5.10.

Listing 5.10: Talker with timer

```
1  #include <chrono>
2  #include <rclcpp/rclcpp.hpp> // needed for basic functions
3  #include <std_msgs/msg/string.hpp> // needed because we publish strings
4
5
6  using namespace std::chrono_literals;
7
8  class TalkerTimer : public rclcpp::Node {
9  public:
10   TalkerTimer() : Node("talkerooptimer") {
11     // create publisher
12     pub = this->create_publisher<std_msgs::msg::String>("message",1);
13     // create timer
14     timer = this->create_wall_timer(1s,std::bind(&TalkerTimer::callback,this));
15     counter = 0;
16   }
17
18 private:
19   void callback() {
20     std_msgs::msg::String stringtosend;
21     RCLCPP_INFO(this->get_logger(),"Sending message #%d",counter);
22     stringtosend.data = "Message #" + std::to_string(counter);
23     pub->publish(stringtosend); // publish message
24     counter++;
25   }
26
27   int counter;
```

```
28    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr pub;
29    rclcpp::TimerBase::SharedPtr timer;
30  };
31
32  int main(int argc,char **argv) {
33
34    rclcpp::init(argc,argv); // initialize the ROS subsystem
35    rclcpp::spin(std::make_shared<TalkerTimer>());
36    rclcpp::shutdown(); // shutdown ROS
37    return 0;
38  }
```

The main feature in this alternative version of the talker is the `timer` object, initialized inside the constructor using the member function `create_wall_timer` of `rclcpp::Node`. This function takes two parameters: the length of the interval between trigger events (1 second in this case), and a pointer to a callback function to be called every time the timer is triggered. In this case, as with the listener in listing 5.9, the callback function is a member function of the class `TalkerTimer`. The `main` function now resembles the one we saw in listing 5.9, where an instance of the class is created on the fly and passed to the `rclcpp::spin` function. This timer-triggered approach is useful when a producer node needs to poll a resource at a regular frequency.

5.7 rviz2

`rviz2` is an extremely useful tool for debugging and testing ROS applications. It is the ROS2 port of `rviz`, a visualization tool capable of displaying a wide variety of data exchanged through topics (the name `rviz` stands for *ROS visualization*).

`rviz2` provides real-time visualization of sensor data, including laser scans, point clouds, and camera feeds, and can also display robot models and coordinate frames using the `tf2` library. Additionally, `rviz2` can visualize outputs from various algorithms such as planning and localization, as long as the data is published through topics.

To start `rviz2`, run the command `ros2 run rviz2 rviz2` or simply `rviz2`. Once launched, a graphical user interface appears (see Figure 5.1), allowing users to add *displays* for graphical visualization of sensor data and related information. Moreover, `rviz2` can be used to send messages or call services. For a complete overview of its features, the reader is referred to the online documentation.

Alternatively, `rviz2` can accept a configuration file (with extension `.rviz`) as a parameter, which specifies the panels to open automatically and the topics to display. Such configuration files are often provided by ROS packages. For example, the `nav2_bringup` package (discussed in Chapter 6) includes its own `rviz` configuration file, which can be passed to `rviz2` as follows:

```
ros2 run rviz2 rviz2
          -d /opt/ros/jazzy/share/nav2_bringup/rviz/nav2_default_view.rviz
```

In this case `rviz2` starts displaying something similar to what is shown in figure 5.2,

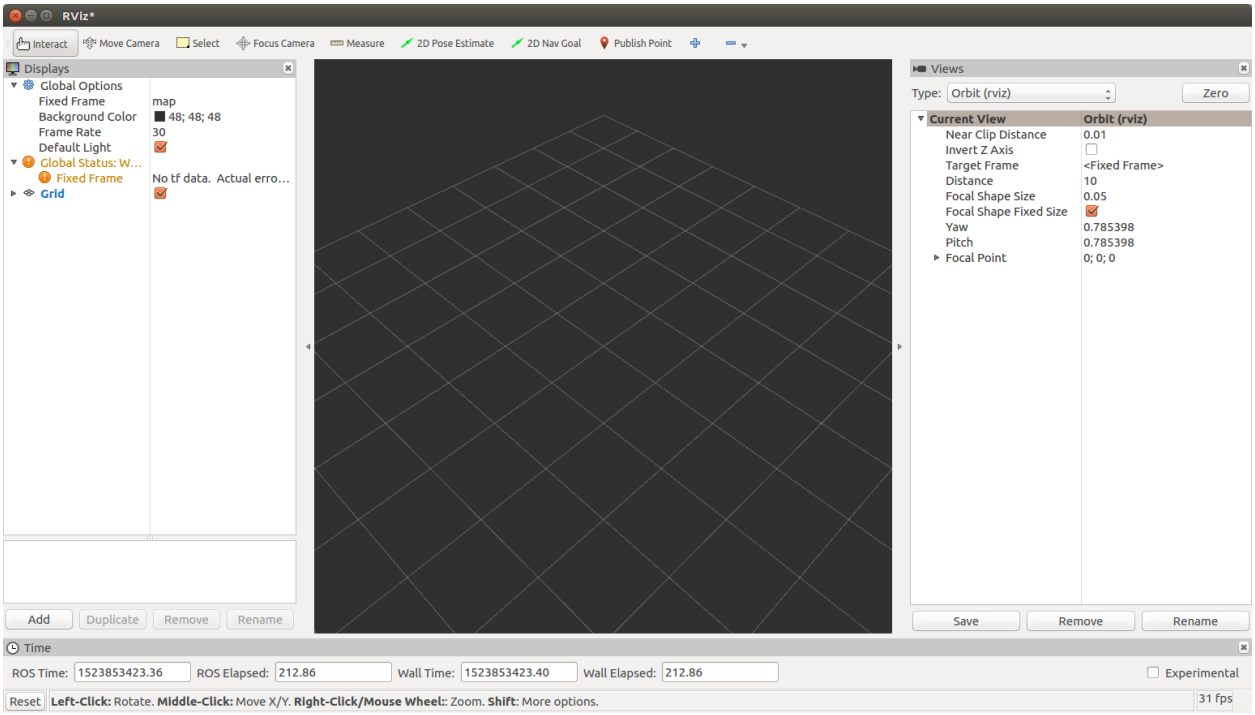


Figure 5.1: `rviz2` starting window.

where the tool shows different topics and panels.

5.8 `ros2 bag`

`ros2 bag` is a tool used to record and replay messages exchanged through ROS topics. This functionality is particularly useful for data collection, testing, and debugging. Recorded data is saved into files known as *ROS bags*, which can be played back later. To create a bag file, the command `ros2 bag record <topic>` is used, where `<topic>` is a list of one or more topics to be recorded. For example:

```
ros2 bag record /turtle1/cmd_vel /turtle1/pose
```

creates a bag file that stores messages transmitted through the topics `/turtle1/cmd_vel` and `/turtle1/pose`. Alternatively, the command `ros2 bag record -a` can be used to record all active topics. `ros2 bag` creates a directory that contains the recorded data alongside a YAML file that describes the structure and metadata of the bag. By default, the bag is stored in a folder named `rosbag2_YYYY_MM_DD-HH_MM_SS`, where the suffix corresponds to the timestamp when recording begins. To stop recording, simply press `CTRL+C`. To specify a custom name for the output folder, the `-o` option can be used, as in:

```
ros2 bag record -o my_file_name /turtle1/cmd_vel /turtle1/pose
```

The command `record` accepts a large number of options, including duration, specific

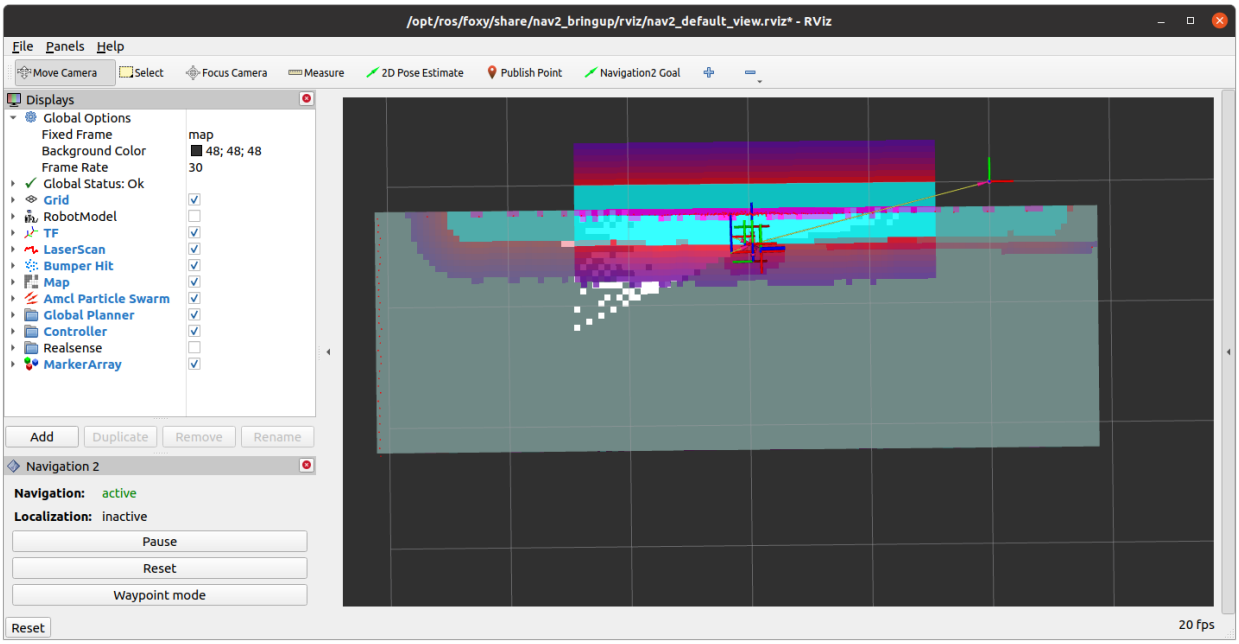


Figure 5.2: `rviz2` starting window when run with a configuration file specifying the panels and topics to display.

start time, topic remapping, compression, and more. To replay a recorded bag, use the command `ros2 bag play <bagfile>` where `<bagfile>` is a list of one or more previously recorded bag filenames. Like `record`, the `play` command also supports various options that modify its behavior. Among these, a particularly important one is the `--clock` flag. If `ros2 bag play` is started without `--clock` (which is the default), the bag data is played using the system clock, ignoring the original message timing. If instead `--clock` is provided, then `ros2 bag` additionally publishes a `/clock` topic that replicates the timestamps from the original recording. In this case, messages are replayed according to their original timing, and all subscribing nodes must have the parameter `use_sim_time` set to `true` (see also Section 5.4.3). Both playback modes are useful depending on the use case. However, it is crucial to ensure consistency. If `--clock` is enabled, all nodes subscribing to the replayed topics must also use simulated time, otherwise, the results may be unpredictable.

To inspect the contents of a bag file, use `ros2 bag info <bagfile>` to retrieve metadata about the topics and message types stored in the bag. Note that `ros2 bag` can generate large files quickly, especially when recording high-frequency or high-bandwidth topics. Care should therefore be taken during extended or extensive recording sessions.

5.9 Launch files in Python

In Section 2.13 we introduced XML launch files. The XML format is simple and may suffice for many basic applications. However, as the complexity of a system increases, it may become necessary to introduce more sophisticated flow control when launching a set of nodes. For example, it may be desirable to launch different nodes depending on user input, system

configuration, hardware availability, or other runtime conditions. In such cases, XML launch files may no longer be adequate, and it is often more convenient to write the launch logic in Python. A Python launch file is a standard Python script that uses a set of ROS-specific modules to define and start nodes and processes. These files have the extension `.launch.py` and typically rely on the `launch` and `launch_ros` modules provided by ROS. As for launch files written in XML, launch files written in Python shall be placed in a folder called `launch`.

The core of a Python launch file is a function named `generate_launch_description`, which must return an instance of the `LaunchDescription` class. This object encapsulates the list of nodes and actions that should be executed when the launch file is run. Listing 5.11 shows how to launch the nodes for one of the turtlesim examples we presented in Chapter 2 (compare with listing 2.3.)

Listing 5.11: Python launch file

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='turtlesim'
        ),
        Node(
            package='turtlesim',
            executable='turtle_teleop_key',
            name='teleop_key',
            prefix=['gnome-terminal --']
            )
    ])
```

As we can see, when the instance of `LaunchDescription` is created, we pass a list of instances of the class `Node`, with one instance for node to be started. Note that we only need to define the function `generate_launch_description` but it is not necessary to call it because it will be called when the launch file is processed by `ros2`. To run a lunch file written in Python we use exactly the same syntax used when starting launch files written in XML:

```
ros2 launch examples talkerlistener.launch.py
```

Each feature previously described for launching nodes (such as remapping, setting parameters, and more) can be incorporated when creating instances of the `Node` class. For example, listing 5.12 is equivalent to listing 5.4 and shows how a node can receive parameters from a YAML file.

Listing 5.12: Python launch file with parameters from a YAML file

```
from launch import LaunchDescription
from launch_ros.actions import Node
```

```
3  from launch.substitutions import PathJoinSubstitution
4  from ament_index_python.packages import get_package_share_directory
5
6  def generate_launch_description():
7
8      pkg_examples = get_package_share_directory('examples')
9      params_file_path = PathJoinSubstitution(
10         [pkg_examples, 'launch', 'parameters.yaml'])
11
12     return LaunchDescription([
13         Node(
14             package='examples',
15             executable='paramclient',
16             parameters=[params_file_path]
17         )
18     ])
```

For a comprehensive overview of these features and additional capabilities, the reader is referred to the official documentation of the `launch` and `launch_ros` libraries.

Chapter 6

Planning

6.1 Introduction

As pointed out in Steve LaValle's book *Planning Algorithms* [27], the terms *planning* and *problem solving* have often been used interchangeably, and the distinction between the two is somewhat blurry. Moreover, these terms may mean different things to different people. Informally speaking, *planning* refers to the task of selecting a sequence of actions to achieve a given goal. Although the goal is often simply *reaching a certain place*, its abstract formulation allows, in general, to pursue much more complex objectives. Examples include finding an object, exploring an area, or tracking a moving intruder, just to name a few. We will formalize this concept more precisely later, when specific planning problems are defined and addressed. Throughout this chapter, we focus exclusively on planning problems in discrete time. At least three distinct types of planning problems can be identified.

Open Loop Planning (aka Deterministic Planning): Given a start state $\mathbf{x}_s$, a goal state $\mathbf{x}_g$ (or a set of goal states), and a state transition equation $f(\mathbf{x}, \mathbf{u})$, determine a sequence of inputs $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n$ to transform the start state into the goal state. This sequence of actions is also called a *plan*. The following diagram, initially introduced in Section 1.3, illustrates this type of planning problem:

$$\mathbf{x_s} = \mathbf{x_0} \overset{\mathbf{u}_1}{\rightarrow} \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) \overset{\mathbf{u}_2}{\rightarrow} \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) \overset{\mathbf{u}_3}{\rightarrow} \ldots \qquad \mathbf{x}_n = f(\mathbf{x}_{n-1}, \mathbf{u}_n) = \mathbf{x}_g$$

In this chapter, consistently with the literature, we use symbols like x_i to indicate states and u_j for inputs. These symbols are chosen solely to simplify the notation and should not be interpreted as implying that states and inputs are scalar quantities. In fact, the state is typically a vector or a more complex object (e.g., a transformation matrix representing the position and orientation of a robot). Similarly, the input often includes multiple components, such as the left and right angular velocities for a differential drive robot, or its translational and rotational velocities. The function f can, for example, represent one of the kinematic models introduced in Chapter 4. However, from a planning perspective, the specific nature of this function is not critical, since the algorithms will be formulated in a more abstract manner that is largely independent of such details. All that is required is a method to compute the next state given a current state and an action. From the planning perspective, this can be treated as a *black box*. This level of abstraction is advantageous, as it allows for

broader applicability, as pointed out earlier.

In general, open loop planning is not an ideal approach because it relies on the assumption that the next state is a deterministic function of the current state and input. This assumption rarely holds true in practice due to the many disturbances affecting a robot's operation. Nevertheless, algorithms that solve the open loop planning problem remain useful for two main reasons. First, they introduce foundational concepts that can be extended to address more complex planning problems. Second, they may produce an *ideal* reference trajectory (i.e., a sequence of states) that can be used as an input for a controller tasked with trajectory tracking. It is important to note that open loop planning does not explicitly consider state observability (or estimation), i.e., the problem of determining the current state, because the state can be deterministically computed from the initial state and the sequence of inputs applied. In this idealized framework, if the start state x_s is known and the plan is given, sensors to determine the following states are not necessary, because these states can be computed by repeated applications of the function f. As a result, perception plays no role in open loop planning algorithms.

Feedback Planning: In feedback planning, one is given a start state x_s and a goal state x_g, and the objective is to determine a feedback function (also called a *policy*) $\pi : X \to U$ that specifies, for each state, the action to take. Feedback planning relies on two key assumptions. First, the state is fully observable, that is, at any given moment, the current state x_t can be accurately determined, typically via a suitable sensor,. Importantly, this sensor is assumed to be error-free, meaning it provides an exact measurement of the state. Second, feedback planning accounts for uncertainty in the state transitions. After applying an action u, the resulting next state is not deterministically predictable. If transitions were deterministic, a simpler open-loop (deterministic) planner, as described earlier, would suffice. For this reason, feedback planning uses a policy π that maps every possible state to an action, allowing the robot to act appropriately regardless of how the state evolves. This contrasts with deterministic planning, where the computed plan specifies actions only along the predicted trajectory and provides no guidance for states outside that path.

Planning in Belief Spaces: In the most general case, one is still given x_s and x_g, but the assumption of state observability no longer holds. That is to say, the current state cannot be determined with certainty, but only probabilistically. This is for example the case when the sensor is not error-free, like in most practical scenarios. Therefore, a feedback function like π can no longer be defined because the state is not known. Indicating with $\mathbb{P}(X)$ the set of all possible probability distributions (beliefs) over X, a planner in belief spaces will produce a function $\beta : \mathbb{P}(X) \to U$ associating to each belief the next action to take. This formulation would lead us to study partially observable Markov decision processes (POMDPs). However, we will not consider POMDPs in the following because their *basic* solution is provably inefficient, and more efficient methods are beyond the scope of these notes.

In all the methods described above there is an additional, common underlying hypothesis, i.e., that the set of states is finite. This will be instrumental to develop the graph-based

planning methods we introduce in the remainder of this chapter.

We conclude this introduction with a concrete example of the type of problems we are interested in. Figure 6.1 shows a map of an environment where each pixel represents a square patch of space in the physical world. White grid cells indicate free space, black grid cells indicate non-traversable space (i.e., obstacles such as walls or pieces of furniture), and gray grid cells mark areas that are unknown, i.e., neither known to be free nor occupied. In some regions, the map may appear messy and imprecise, which is indeed the case because it was autonomously built by a robot exploring the environment (mapping algorithms will be described in Chapter 9.) Such a map can be stored as a matrix of numerical values, such as unsigned integers stored on 8 bits, or floating-point numbers in the range $[0, 1]$. While the map shown in the figure classifies each grid cell into three categories (free, occupied, unknown), it is also possible, and very common, to use a variety of values. For example, each cell may be assigned a value representing the probability that the cell is occupied. In this case, assigning each cell a value between 0 and 1 is a natural and convenient choice. With this representation, free cells have a value of 0, occupied cells have a value of 1, and values in between represent the level of confidence that the cell is occupied. Alternatively, a cell may store a *cost to traverse*, i.e., a measure of the effort required to cross the cell. In this case, free cells would be assigned a value of 0, while cells closer to obstacles could have higher values to reflect the fact that, although traversable, it may be preferable to choose a different path to avoid proximity to obstacles. Two locations are marked on the map, namely a starting location (labeled as S) and a goal location (labeled as G). In this case, we are not specifying orientations, so S and G are simply represented as pairs of (x, y) values. A typical problem we face in planning is computing a sequence of actions to move a robot from S to G without colliding with any obstacles. In this chapter, we tackle some of the challenges related to this task, although, as will be discussed later, a plan alone is not sufficient to successfully move the robot from the start location to the goal location due to unavoidable disturbances. The figure, however, provides a general motivation for the problems we study later on.

6.2 Discrete Models

In the remainder of this chapter, we present various planning algorithms that share the following features. A finite, discrete state space set is defined: $S = \{x_1, x_2, \ldots, x_n\}$. The hypothesis of finiteness for the state space is essential to ensure the correctness of some of the algorithms presented in the following. For example, the DFS algorithm discussed in Section 6.3.3 is not guaranteed to operate correctly if the state space has infinite cardinality. In the following, we will exclusively consider problems with finite state spaces.

For each state $x_i \in S$, a finite set of actions (or inputs) $U(x)$ is defined. $U(x)$ is the set of actions that can be executed in state x. From these two sets, the set of all actions can be defined as $U = \bigcup_{x \in S} U(x)$. From the above assumptions, it follows that U is also discrete and finite. According to these definitions, different action sets may be available in different states (or, stated differently, not all actions in U are available in all states in S). Throughout the remainder of the discussion, when referring to the action to be executed in a state $x \in S$, we tacitly assume that the action belongs to $U(x)$, although, with slight abuse of notation,

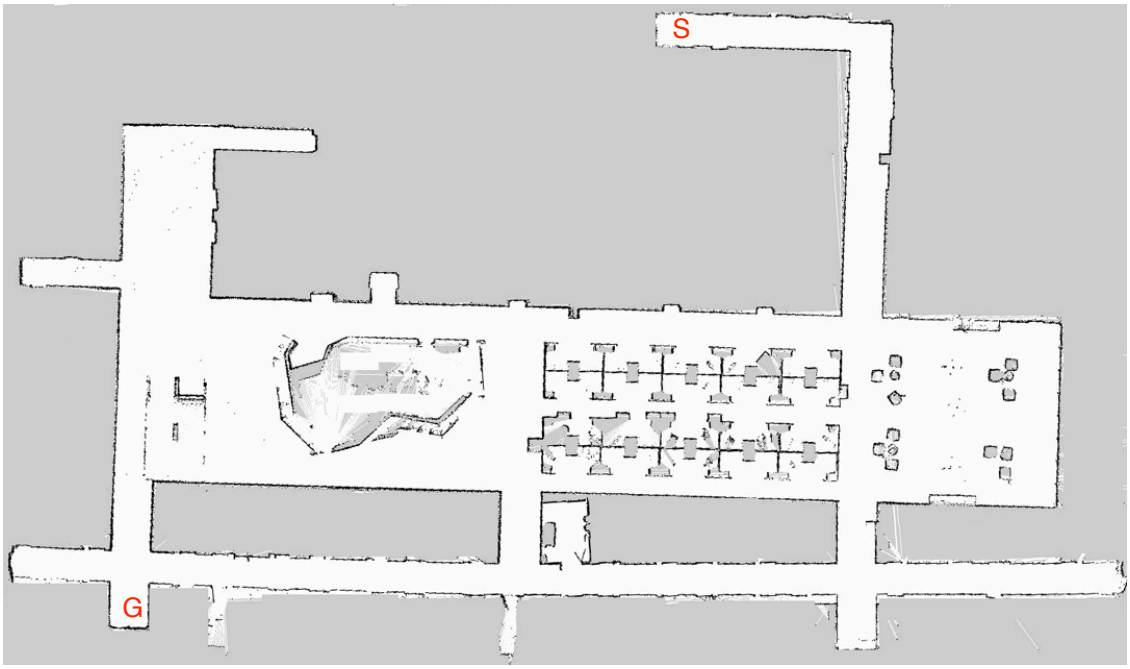


Figure 6.1: A typical planning problem for a mobile robot. Given a map of an environment and a start (S) and goal (G) location, determine a sequence of actions to move the robot from S to G.

we may write for brevity that the action belongs to U.

When studying planning algorithms, we will consider an important property, i.e., *completeness*. A planning algorithm is *complete* if it will always find a solution when one exists. This property is relatively easy to check and enforce in the case of discrete algorithms, but will be much more problematic when considering other types of planners, like sampling-based motion planning algorithms.

Another important characterization is whether the search is *uninformed* or *informed*. In the uninformed case, the structure of the problem is the only input available to the planner. In essence, this is the structure of an underlying graph, as it will be defined in the following (start, goal, edges, etc.). In informed search, the planner also has access to some heuristic identifying more promising search directions to consider. Examples of both informed and uninformed search will be given in the following. Since informed algorithms rely on more information than uninformed ones, they are in general more efficient. A *good* heuristic requires some domain-specific knowledge and is typically problem dependent. However, it may not always be possible to determine a good heuristic to guide the search of an informed algorithm, and therefore both types of approaches are used in practice.

6.2.1 On Abstractions

Most of the algorithms we discuss in this section start from the hypothesis that the state space S and the set of actions available for each state $U(x)$ are provided upfront. Where do these sets come from, and is this hypothesis consistent with real-world problems? There

are indeed automatic ways to build these sets, but S and U are problem dependent, and therefore algorithms or systems aiming at creating these two sets must be tailored to the specific problem being tackled. For example, consider again the situation we discussed in Figure 6.1, and assume the robot we are dealing with is a differential drive like the ones displayed in Figure 4.21. If we discard the orientation of the robot, it is relatively simple (albeit perhaps tedious) to write a program that pre-processes the map and builds a grid of equally sized square cells covering the free space part of the environment, i.e., the white region (in fact, grid cells do not even need to be of equal size or square, but this is a useful simplification).

How big should each grid cell be? The answer is not unique and there is a tradeoff between accuracy and size of the state space. For example, we could assume that each grid cell is large enough that the robot can be fully contained in it. Each such grid cell could then be one state in S and we would say that the robot is in state $x \in S$ if it is entirely inside cell x.[1]

For what concerns the set of actions $U(x)$ available in state x, one has to consider the motion capabilities of the robot. As we assumed to deal with a differential drive, it is appropriate to hypothesize that from each grid cell it is possible to move to any neighboring grid cell. Such an action may imply some maneuvering, e.g., turning in place, then moving forward, then possibly turning in place again. But from a planning perspective, this is just one action that, once successfully applied, will bring the robot from one grid cell (i.e., one state) to another one.

For simplicity, we could assume that for each state/grid cell x we only consider actions that move the robot to a neighboring cell, but if a library of appropriate maneuvers is available, nothing prevents considering actions that, when applied, would cause the robot to move between two far-away grid cells. In fact, such *complex* maneuvers could be obtained by solving multiple smaller planning problems. However, this is a route we will not follow. Either way, assuming that S and U are available to the planning algorithm is an assumption that is not inconsistent from the implementation standpoint.

6.3 Open Loop Planning

Open loop planning builds upon the discrete model described above by adding a deterministic, time-invariant state transition function:

$$x_t = f(x_{t-1}, u_t) \qquad x_t \in S,\ u_t \in U(x_{t-1}). \tag{6.1}$$

The meaning is the same as introduced in Eq. (1.7), i.e., if at time $t-1$ we are in state x_{t-1} and apply input u_t, then at time t we are in state x_t. The requirement that $u_t \in U(x_{t-1})$ can be enforced through the following definition:

$$K = \{(x, u) \in S \times U \mid x \in S \wedge u \in U(x)\}. \tag{6.2}$$

Then, the function f can be defined as $f : K \to S$. In essence, K defines the set of all legitimate state/input pairs.

[1] To keep things simple, let us for the time being ignore the fact that a robot could be sitting at the boundary of two neighboring cells x_i and x_j, and therefore be neither in x_i nor in x_j.

Given S and U, a *directed* graph $G = (V, E)$ is defined as follows. The set of vertices is $V = S$. The set of edges is defined by f as follows: if there exists $u \in U(x_i)$ such that $x_j = f(x_i, u)$, then we add a directed edge from x_i to x_j. Note that the set of edges is in one-to-one correspondence with the set K, i.e., for each element $(x, u) \in K$ there is an edge in E. Stated differently, each edge is in one-to-one correspondence with an input, so a path in the graph corresponds to a sequence of inputs applies to the corresponding traversed edges. This graph is called the *planning graph*, and it is the essential concept for turning a planning problem into a graph search problem. Throughout this chapter, unless otherwise stated, planning graphs will be directed. This means that if there exists an action $u_i \in U(x_j)$ such that $x_k = f(x_j, u_i)$, it does not necessarily mean that there exists an action $u_z \in U(x_k)$ such that $x_j = f(x_k, u_z)$. Stated in graph terms, this means that $(x_j, x_k) \in E$ does not necessarily imply that $(x_k, x_j) \in E$.

Example 6.1. *A classic problem considered when introducing planning problems is the* grid world*. It consists of an environment divided into cells arranged in a grid (hence the name). Figure 6.2 shows one such world. White grid cells are traversable, i.e., they represent a location where the robot can be. For this reason, they are also called* free cells*. Black cells are instead occluded, i.e., they model locations that the robot cannot traverse or occupy. Each free (white) cell is associated with a state in the corresponding planning problem. No states are associated with the occluded (black) cells. In the figure, x_1, x_2, x_3, and x_4 are four of the 33 states in the problem (the grid is 6×6, but the three occluded cells are not associated with any state, therefore there are 33 states in total).*

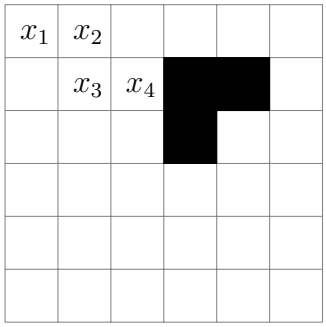


Figure 6.2: Grid world

Four actions are defined, namely up/down/left/right (indicated as U/D/L/R in the following). Not all actions are available in all states due to motion constraints. For example, in state x_1 we have $U(x_1) = \{D, R\}$ because it is possible to move down and right, but not up or left. Similarly, $U(x_2) = \{D, L, R\}$ and $U(x_4) = \{D, L, U\}$. In state x_3, all actions are available, and therefore $U(x_3) = \{D, R, L, U\}$. The transition function corresponds to the deterministic motion from one cell to another when an action is executed, e.g., $f(x_1, R) = x_2$, $f(x_2, D) = x_3$, and so on. From this information, it is straightforward to build the associated planning graph. The graph includes, for example, edges (x_1, x_2), (x_2, x_1), (x_2, x_3), etc. The sequence of edges $(x_1, x_2)(x_2, x_3)(x_3, x_4)$ forms a path in the planning graph and is associated with the sequence of actions (plan) R, D, R. This means that to go from x_1 to x_4, a valid plan is R, D, R.

Given a start state $x_s \in S$ and a goal state $x_g \in S$, the objective is to determine a plan (or sequence of actions) $u_1, u_2, \ldots, u_k$ transforming the start state into the goal state, as per the state transition function. *Transforming* the state means that if we start with the state x_s and we sequentially apply all actions in the plan, we end up in the goal state. It should at this point be evident that this problem is related to graph search, since it is equivalent to finding a path in the planning graph (G, V) induced by S, K and f. For sake of completeness, we recall the definition of path in a graph.

Definition 6.1. *Path Let $G = (V, E)$ be a graph, and $x_1 \in V$, $x_n \in V$ be two vertices. A* path *p between x_1 and x_n is a sequence of vertices $x_1, x_2, \ldots, x_n$ such that $(x_i, x_{i+1}) \in E$ for $1 \leq i \leq n-1$.*

In the following we will consider two different search problems. The first one is applicable to any graph.

> **Directed Graph Search Problem:** Given a directed graph $G = (V, E)$, and vertices $x_s \in V$, $x_g \in V$, determine a path from x_s to x_g or return *failure* if no path can be found.

The above formulation does not merely ask whether a path between x_s and x_g exists (the so-called feasibility problem), but rather requires returning the full path, if it exists. The second graph search problem is instead defined over weighted graphs. A graph $G = (V, E)$ is weighted if there exists a *cost* function $c : E \to \mathbb{R}$ associating a cost to every edge. Given a weighted graph, the cost of a path in the graph is defined as follows.

Definition 6.2 (Cost of a path)**.** *Let $G = (V, E)$ be a weighted graph, and let $c : E \to \mathbb{R}$ be its cost function. Let $p = x_1, x_2, \ldots, x_n$ be a path in G. The* cost of path *p is the sum of the costs of its edges, i.e.,*

$$c(p) = \sum_{i=1}^{n-1} c(x_i, x_{i+1}). \tag{6.3}$$

In the definition of the cost of a path, we have not imposed any restriction on the cost function c, but in the following problem definition we require that the cost function is non-negative.

> **Weighted Directed Graph Search Problem:** Given a directed weighted graph $G = (V, E)$ with a non-negative cost function $c : E \to \mathbb{R}_{\geq 0}$, and vertices $x_s \in V$, $x_g \in V$, determine a path of minimum cost from x_s to x_g, or return *failure* if no path can be found.

When comparing the directed graph search problem with the weighted version, it is immediate to recognize that the latter is an optimization problem, whereas the former is not. That is to say, in general, there exist multiple paths between x_s and x_g. The weighted directed graph search problem asks to return one path of smallest cost, whereas the directed graph search problem requires just returning any feasible path. As is usual in optimization problems, the smallest cost is unique, but there generally exist multiple paths achieving the minimal cost.

6.3.1 Common Traits in Graph Search Algorithms

The four graph search algorithms we present in the following share a few commonalities, which we describe before discussing each algorithm individually.[2] First, all algorithms begin by processing the starting vertex x_s and iteratively analyze more and more vertices. Vertices to be processed are stored in a data structure commonly referred to as the *OPEN* list or queue. *OPEN* is an abstract data type supporting the operations *insert* and *remove*. In some instances, *OPEN* will be prioritized using a key, in which case we rely on a *rebalance* operation to update the structure if the key of one of its elements is modified. All algorithms also rely on the concepts of *expansion* and *visited* vertices. A vertex is said to be *expanded* when it is removed from *OPEN* and all of its neighbors are processed. The specific operations executed during the expansion step vary depending on the algorithm. A vertex y is *visited* when a vertex x is expanded and there exists an edge $(x, y) \in E$. To determine the neighbors of a vertex x, we look at all edges (x, y). Due to the way edges are defined in the planning graph, this is equivalent to evaluating the effects of the various actions available in $U(x)$. After a node is removed from the *OPEN* data structure, it is typically moved into a container called *CLOSED*. From an implementation standpoint, *CLOSED* is often optional and its primary role is analytical, helping us understand algorithmic behavior and verify properties such as completeness. Nonetheless, to remain consistent with standard literature, we include it in the pseudocode.

Since we are dealing with a generic graph, a vertex can be visited multiple times if it can be reached from x_s via different paths. The algorithms we present include mechanisms to handle vertex re-visitation correctly and prevent infinite loops. To simplify the implementation, it is useful to associate attributes with the vertices. One attribute common to all algorithms is the *parent* attribute. The parent of a node x is *null* if no path from x_s to x has been discovered yet. This attribute is also called *back pointer* in the literature. Once a path p from x_s to x is found, the parent of x is a pointer to the vertex that precedes x on path p. Consequently, if a path from x_s to x_g is found by the algorithm, it can be reconstructed by recursively following the *parent* pointers backward from x_g to x_s. Some of the algorithms we discuss are informed, and others are uninformed; however, all are *complete*. Finally, all the algorithms we will present construct a spanning tree $\mathcal{T}$ rooted at x_s and extending to various vertices in the graph (the coverage depends on the specific algorithm). As we will show, the spanning tree $\mathcal{T}$ encodes plans for moving from x_s to any vertex included in the tree. These plans can be recovered by navigating from any vertex in the tree back to the root using the *parent* pointers.

6.3.2 Breadth First Search

Breadth First Search (BFS from now onwards) is a classic graph search algorithm to solve the Directed Graph Search Problem. BFS can be used to implement a complete, uninformed planning algorithm. Its defining feature is the policy followed to manage the *OPEN* list. In particular, in BFS, *OPEN* is a first-in-first-out data structure. In addition to the *parent* attribute, BFS associates a binary attribute *visited* to every vertex. This attribute is initially set to `false` and changed to `true` after the vertex has been discovered for the first time.

[2] This description mostly follows the approach described in [29].

BFS searches the graph starting from x_s and iteratively processes all vertices in G sorted by their *distance* from x_s, i.e., first all vertices one hop away from x_s, then all vertices two hops away, and so on. The search terminates when x_g is visited, or it fails if no more vertices can be explored and x_g has not been reached. Since G may include cycles, the algorithm uses the *visited* attribute to avoid revisiting the same vertices[3]. If a path between x_s and x_g exists, BFS returns the path with the smallest number of edges between x_s and x_g. This statement will be formalized with a theorem later on.

Algorithm 1 sketches the pseudocode for the algorithm. Note that many BFS implementations also store, for each node, the distance from x_s as they are discovered. Algorithm 1 does not record this information, but it is straightforward to add it. The algorithm starts by setting all parents to *null* and marking all vertices as not visited (loop at line 1). Then, it initializes the empty containers *OPEN* (line 4) and *CLOSED* (line 5), inserts the start node x_s into *OPEN* (line 6), and marks it as visited (line 7). In the main loop (lines 8–18), the algorithm removes one vertex x from *OPEN* (line 9), inserts it into *CLOSED* (line 10), and expands it (line 11). If a node x' is visited for the first time, the *visited* field is updated (line 13), and the *parent* is set to x (line 14). If x' is equal to the goal vertex x_g, the search terminates with success, and a path can be extracted by traversing the sequence of *parent* pointers from x_g back to x_s (line 16). Otherwise, the node is inserted into *OPEN* (line 18). This operation continues as long as there are more vertices in *OPEN*. If the main loop terminates, it means that x_g has not been discovered yet and there are no more vertices to process. Therefore, the algorithm reports failure (line 19).

[3]A node whose *visited* attribute is `true` is either in the *OPEN* or *CLOSED* containers. However, storing this attribute with the nodes themselves is more efficient than checking if the node is included in these data structures.

```
   Data: G = (V, E), x_s ∈ V, x_g ∈ V
   Result: Path from x_s to x_g if it exists, or FAILURE
 1 foreach x ∈ V do
 2 |   x.parent ← null;
 3 |   x.visited ← false;
 4 OPEN.initializeEmpty();
 5 CLOSED.initializeEmpty();
 6 OPEN.insert(x_s);
 7 x_s.visited ← true;
 8 while not OPEN.empty() do
 9 |   x ← OPEN.remove();
10 |   CLOSED.insert(x);
11 |   foreach x' ∈ V such that (x, x') ∈ E do
12 |   |   if x'.visited =false then
13 |   |   |   x'.visited ← true;
14 |   |   |   x'.parent ← x ;
15 |   |   |   if x' = x_g then
16 |   |   |   |   return ExtractPath(x_s, x_g);
17 |   |   |   else
18 |   |   |   |   OPEN.insert(x');
19 return FAILURE;
```

Algorithm 1: BFS/DFS algorithm

Because of how we defined the set of edges E, each edge is associated with an action u, so once a path is returned, one can determine a sequence of actions consistent with Eq. (6.1) that transforms the start state x_s into the goal state x_g. It is important to note that in BFS, the parent of a node is initially set to *null* and is changed at most once during the execution of the algorithm, specifically when the node is discovered for the first time. If a path between x_s and x_g is found, the path can be recovered by traversing the *parent* pointers from x_g back to x_s (this is done by the function ExtractPath). As this path is traced, the corresponding actions can also be determined due to the one-to-one correspondence between edges and actions. Therefore, the plan is built by backtracking from x_g to x_s via the parent pointers.

At the end of the computation, BFS produces a tree rooted at x_s. The tree does not span all vertices in G, but only those visited during the algorithm's execution. More precisely, the tree spans all nodes that have at some point been inserted into the $OPEN$ queue. According to our implementation, a vertex may not be included in the tree for two reasons. First, the vertex is not reachable from x_s. Second, the vertex is reachable from x_s, but its distance (measured as the number of hops) is greater than the distance from x_s to x_g, and the computation terminates before the vertex is visited.

The following example illustrates how BFS works and how the various attributes change during execution.

Example 6.2. *Let us consider the graph shown in Figure 6.3 and analyze how BFS processes it. To this end, it is convenient to examine each iteration of the main while loop in*

Algorithm 1, showing the attributes (e.g., visited *and* parent*) for each vertex, as well as the status of the* OPEN *queue.*

In what follows, Step 0 refers to the state before the first iteration of the main loop. It is assumed that when a node has multiple neighbors, they are processed in alphabetical order. For example, when x_s is expanded, node A is added to the queue before node C. Table 6.1 illustrates the progression of the algorithm. For each node, we display two fields: the parent *(the name of the parent node, or* N *for null) and the* visited *flag (*T *for true,* F *for false). The queue* OPEN *is also shown, with the leftmost vertex representing the head of the queue. At Step 5, node B is extracted from the queue, which becomes empty, and is then expanded. The goal vertex x_g is discovered, and the algorithm terminates. The path between x_s and x_g is reconstructed by following the* parent *fields (back pointers) stored in the nodes. The* parent *of x_g is B, the* parent *of B is C, and the* parent *of C is x_s, thus forming the desired path. The* parent *of x_s is* null*, indicating the root of the search.*

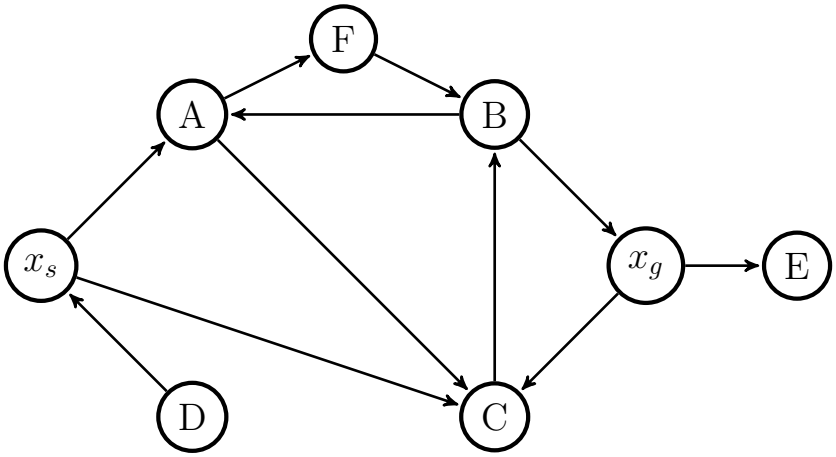


Figure 6.3: A simple directed graph.

Step	*OPEN*	x_s	A	B	C	D	E	F	x_g
0	x_s	N/T	N/F	N/F	N/F	N/F	N/F	N/F	N/F
1	A,C	N/T	x_s/T	N/F	x_s/T	N/F	N/F	N/F	N/F
2	C,F	N/T	x_s/T	N/F	x_s/T	N/F	N/F	A/T	N/F
3	F,B	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	N/F
4	B	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	N/F
5	$\varnothing$	N/T	x_s/T	C/T	x_s/T	N/F	N/F	A/T	B/T

Table 6.1: BFS progress while processing the graph in Figure 6.3.

Figure 6.4 shows the corresponding tree rooted at x_s obtained by running the BFS algorithm to find a path between x_s and x_g. The tree does not include node D because there is no path from x_s to D, and it also does not include E because the search terminates as soon as x_g is discovered, i.e., before E is visited. Observe that the tree also encodes a path between x_s and F, even though F is not part of the path from the start to the goal vertex.

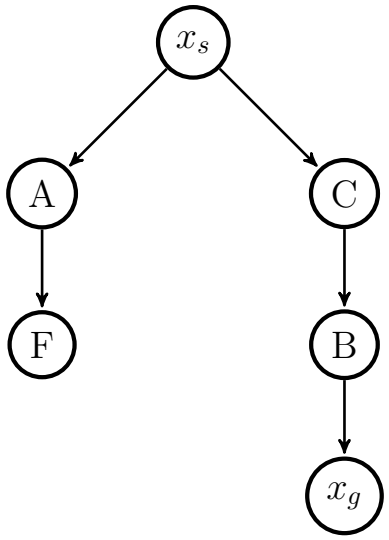


Figure 6.4: Tree produced by BFS.

Computational Complexity. It is straightforward to determine the time complexity of BFS. The initialization step (for loop at line 1) takes time $\mathcal{O}(|V|)$. Each operation on $OPEN$ and $CLOSED$ (insert, remove) takes time $\mathcal{O}(1)$ and each vertex is inserted or removed at most once. Therefore $\mathcal{O}(|V|)$ is also an upper bound for the time complexity of the operations on $OPEN$ and $CLOSED$. Finally, each edge $(x, y) \in E$ is processed at most once when x is removed from $OPEN$, and for each edge a constant number of operations is performed (expansion step, line 11). Therefore the compounded complexity of all expansions steps is $\mathcal{O}(|E|)$ and the overall complexity of the algorithm is $\mathcal{O}(|V| + |E|)$.

Theorem 6.1. *Algorithm BFS is complete, i.e., if a path between x_s and x_g exists, then BFS will find it and return the path between x_s and x_g.*

Theorem 6.2. *If algorithm BFS returns a path between x_s and x_g, then it returns a path with the smallest number of edges.*

The proofs of Theorems 6.1 and 6.2 can be found in any algorithms textbook (e.g., [13]). Note that Theorem 6.2 states that the algorithm returns *a* path with the smallest number of edges, and not *the* path with the smallest number of edges because in general there could be more than one.

Remark 6.1. *The graph shown in Figure 6.3 may seem overly simple and disconnected from practical robotic applications. This is true, and the example is simple on purpose to allow its complete analysis in a few steps. However, the same algorithm can be used "as is" for cases where the graph models more complex scenarios. To see how, the reader should revisit the grid world example (Example 6.1) and Figure 6.1.*

6.3.3 Depth First Search

Depth First Search (DFS) works almost exactly as BFS, and the pseudocode given in Algorithm 1 does not need any change. The only difference is the policy used to manage the

OPEN data structure. While BFS uses a first-in-first-out approach, DFS uses a last-in-first-out policy. Differently from BFS, the path returned by DFS is not guaranteed to be a path with the least number of vertices. If the graph is finite, though, DFS is guaranteed to return a valid path, if it exists.

Example 6.3. *We solve again Example 6.2, but this time using DFS. As in the previous example, we show the progress of the algorithm using the same tabular form (Table 6.2).*

Step	*OPEN*	x_s	A	B	C	D	E	F	x_g
0	x_s	N/T	N/F	N/F	N/F	N/F	N/F	N/F	N/F
1	A,C	N/T	x_s/T	N/F	x_s/T	N/F	N/F	N/F	N/F
2	B,A	N/T	x_s/T	C/T	x_s/T	N/F	N/F	N/F	N/F
3	A	N/T	x_s/T	C/T	x_s/T	N/F	N/F	N/F	B/T

Table 6.2: BFS progress while processing the graph in Figure 6.3.

Figure 6.5 shows the tree produced by DFS when solving the same planning search problem depicted in Figure 6.3 (compare this tree with Figure 6.4)

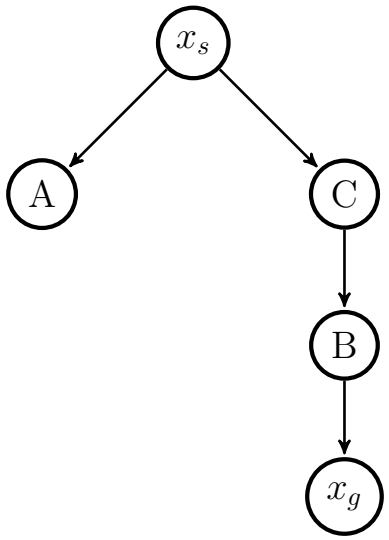


Figure 6.5: Tree produced by DFS.

In this case the algorithm does not discover a path to F because C is expanded before A because of the last-in-first-out policy. Moreover, after expanding C the algorithm discovers B and this in turn is again expanded before A, thus terminating the execution of the algorithm as soon as x_g is discovered.

Computational Complexity. The only difference between DFS and BFS is in the policy governing *OPEN*. In both cases insertion and removal take $\mathcal{O}(1)$. Hence the complexity of the algorithms is the same, i.e., $\mathcal{O}(|V| + |E|)$.

Theorem 6.3. *Algorithm DFS is complete, i.e., if a path between x_s and x_g exists, then DFS will find it and return the path between x_s and x_g.*

The proof of theorem 6.3 can be found in any algorithm textbook (e.g., [13]). Note that in this case we cannot infer other properties with regard to the properties of the returned path (e.g., number of edges in the path). Moreover, recall that throughout this chapter we are dealing with finite graphs, and this is an essential pre-requisite for Theorem 6.3.

6.3.4 Dijkstra's Algorithm

We next introduce a graph algorithm to solve the more complex (and realistic) Weighted Directed Graph Search Problem, where costs are associated with edges. Recalling that each edge represents an action, the cost of an edge is therefore the cost to execute the action associated with the edge, such as the time it takes to execute it, the amount of energy consumed, or the distance traveled. Starting from the edge costs, Dijkstra's algorithm computes a cost for every node x. The cost of a node x is also called "cost to come," and it is the cost along the shortest path from x_s to x discovered so far. In the following, this cost is indicated with the letter g. The key words in this definition of g are "so far." That is because this cost is iteratively refined (i.e., lowered) as better paths are discovered while the algorithm processes the graph. Note that in Dijkstra, we do not need the *discovered* attribute because a vertex may be discovered multiple times along different paths. Consequently, the parent of a node can be repeatedly modified during the execution of the algorithm, i.e., every time the cost of a node is lowered. Figure 6.6 shows some of the quantities characterizing Dijkstra's algorithm. The cost to come g is the cost between x_s and x along the path from x_s to x_g passing through x Note that g is well defined irrespective of whether the path between x_s and x_g is the shortest path or not.

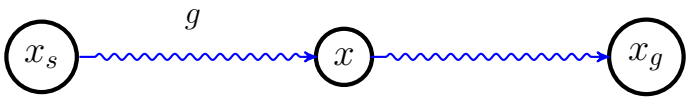


Figure 6.6: The figure shows a node x along one path from x_s to x_g. The cost to come g is the cost of the best path (discovered so far) between x_s and x. Such path is not necessarily a shortest path between x_s and x_g, because a better one could be discovered at a later time.

It is easy to show that if x lies along a shortest path between x_s and x_g, then the cost to come g must be the smallest among all paths from x_s to x. In fact, if there were a different path with cost $g' < g$, then a better path from x_s to x_g could be obtained by swapping the original path of cost g with the one with cost g', but this contradicts our assumption that the path was optimal. Similarly, one can reason about the length of the path between x and x_g. Dijkstra's algorithm works by iteratively lowering the cost to come for the vertices in the graph, until it can no longer be lowered (and hence it is optimal).

In Dijkstra's algorithm, the container $OPEN$ is a priority queue whose key is the cost to come g. Every time the cost to come of a node in the queue is lowered, the queue needs to be updated using the *rebalance* method. Algorithm 2 sketches the pseudocode for the algorithm.

In the initialization step (loop starting at line 1), the cost of all nodes is set to ∞ and the parents are initialized to *null* (lines 2 and 3). These values reflect that no path from x_s to any node has been discovered yet. The cost of x_s is set to 0 (line 4) before it is inserted in

```
   Data: G = (V, E), x_s ∈ V, x_g ∈ V, c : E → ℝ_{≥0}
   Result: Shortest path from x_s to x_g if it exists, or FAILURE
 1 foreach x ∈ V do
 2 |   x.g ← ∞;
 3 |   x.parent ← null;
 4 x_s.g ← 0;
 5 OPEN.initializeEmpty();
 6 OPEN.insert(x_s);
 7 CLOSED.initializeEmpty();
 8 while not OPEN.empty() do
 9 |   x ← OPEN.remove();
10 |   CLOSED.insert(x);
11 |   if x = x_g then
12 |   |   return ExtractPath(x_s, x_g);
13 |   foreach x' ∈ V such that (x, x') ∈ E do
14 |   |   if x'.g = ∞ then
15 |   |   |   x'.parent ← x ;
16 |   |   |   x'.g ← x.g + c(x, x');
17 |   |   |   OPEN.insert(x');
18 |   |   else if x.g + c(x, x') ≤ x'.g then
19 |   |   |   x'.g ← x.g + c(x, x');
20 |   |   |   x'.parent← x ;
21 |   |   |   OPEN.rebalance(x');
22 return FAILURE;
```

Algorithm 2: Dijkstra's algorithm

the queue (line 6) because, by definition, the cost of going from x_s to x_s is 0 (no action must be taken to go from x_s to x_s, so no cost is incurred). As for BFS and DFS, x_s is inserted into the $OPEN$ queue before the main loop starts, and $CLOSED$ is initialized to the empty container (line 7). In the main loop, the node x with the smallest cost to come g is removed from $OPEN$ (line 9) and moved into $CLOSED$. If the node is equal to x_g, the algorithm terminates and a path of minimum cost is returned (line 12). Note that this is correct, i.e., a better path from x_s to x_g cannot be found because all costs are non-negative, and since x_g is removed from the $OPEN$ list, it means it has the lowest cost among those in the queue (recall that it is prioritized by the cost to come g). No shortest path can be found later on because all nodes still in $OPEN$ have cost to come $g(x)$ no smaller than $g(x_g)$, and all those that could be included in $OPEN$ at a later point will have a cost to come no smaller than those in $OPEN$ right now. Therefore, it is impossible to find a lower cost for any vertex x once it is removed from $OPEN$. Otherwise, if x is not x_g, it is expanded (line 13). If a neighboring node x' has cost equal to ∞, then it is being discovered for the first time (line 14). In such a case, the parent is set to x (line 15) and the cost to come is set to $x.g + c(x, x')$ (line 16) because a path with this cost has just been discovered by appending the edge (x, x') to the best path between x_s and x. Moreover, the node is inserted in $OPEN$ (line 17). If

instead the cost of a node x' is different from ∞ but larger than $x.g + c(x, x')$ (line 18), then it means that a path had been previously found, but a better one has just been determined. In this case, the cost to come g of x' is updated (lowered) to $x.g + c(x, x')$ (line 19), and the parent is also modified and set to x (line 20) because the new shortest path from x_s to x' is obtained by appending (x, x') to the best path between x_s and x. In this case, x' was already in $OPEN$ because it was inserted when its cost was first lowered from ∞. Therefore, the rebalance operation is applied to adjust the position of the node in the queue since its key has been modified (line 21). As for BFS and DFS, if the algorithm exits the main loop, it means that no more vertices can be discovered and x_g has not been found among those processed already. Therefore, no path exists and the algorithm returns failure (line 22).

Computational Complexity. The clear difference between Dijkstra's algorithm and BFS/DFS is the different policy used to handle $OPEN$. The complexity of the operations insert and remove obviously depends on the underlying data structure used to implement the priority queue, and this also affects the time complexity of the rebalance operation. If $OPEN$ is implemented as a binary heap, then the overall complexity is $\mathcal{O}((|V|+|E|)\log|V|)$. An even more efficient algorithm can be obtained using a Fibonacci heap (see [13] for more details). In this latter case, the time complexity is $\mathcal{O}(|V|\log|V| + |E|)$.

Theorem 6.4. *If a path between x_s and x_g does not exist, Dijkstra's algorithm returns FAILURE. If a path between x_s and x_g exists, Dijkstra returns a path of minimum cost, as per the path cost defined in Eq.* (6.3).

Dijkstra's algorithm is another standard algorithm in graph theory, and the proof for Theorem 6.4 can also be found in most algorithm textbooks, e.g., [13]. Dijkstra's algorithm is often also described as a *label correcting algorithm* because it repeatedly updates (corrects) the g values (labels) associated with the vertices. The reader is referred to [29] or [7] for more details on label correcting algorithms.

Example 6.4. *Consider the weighted graph shown in figure 6.7 where each edge is now associated with a non-negative cost.*

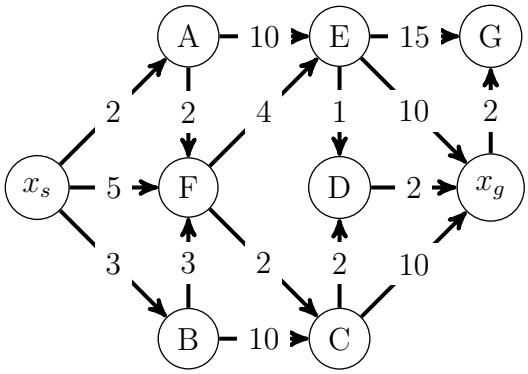


Figure 6.7: A simple weighted directed graph.

The following table is analogous to Table 6.1 and shows how the algorithm progresses. For each node, the parent and the g cost are displayed. In the OPEN priority queue, nodes

are shown with their associated priority key, sorted from the lowest key to the highest key. Note how, from step 3 to 4, the keys for both nodes C and E are lowered, and their respective positions in the queue are swapped. Moreover, observe that, differently from BFS and DFS, the algorithm terminates when the goal node x_g is extracted from OPEN (step 8), and not when it is first discovered (step 5), because when it is inserted in OPEN for the first time, its cost g is not necessarily the lowest possible.

Step	$OPEN$	x_s	A	B	C	D	E	F	G	x_g
0	$x_s/0$	$N/0$	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞	N/∞
1	$A/2, B/3, F/5$	$N/0$	$x_s/2$	$x_s/3$	N/∞	N/∞	N/∞	$x_s/5$	N/∞	N/∞
2	$B/3, F/4, E/12$	$N/0$	$x_s/2$	$x_s/3$	N/∞	N/∞	$A/12$	$A/4$	N/∞	N/∞
3	$F/4, E/12, C13$	$N/0$	$x_s/2$	$x_s/3$	$B/13$	N/∞	$A/12$	$A/4$	N/∞	N/∞
4	$C/6, E/8$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	N/∞	$F/8$	$A/4$	N/∞	N/∞
5	$D/8, E/8, x_g/18$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	N/∞	$C/18$
6	$E/8, x_g/10$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	N/∞	$D/10$
7	$x_g/10$	$N/0$	$x_s/2$	$x_s/3$	$F/6$	$C/8$	$F/8$	$A/4$	$E/23$	$D/10$

Figure 6.8 shows the resulting tree obtained after running Dijkstra's algorithm on the given graph. Each vertex is associated with the cost of the shortest path from x_s discovered by the algorithm before it terminates.

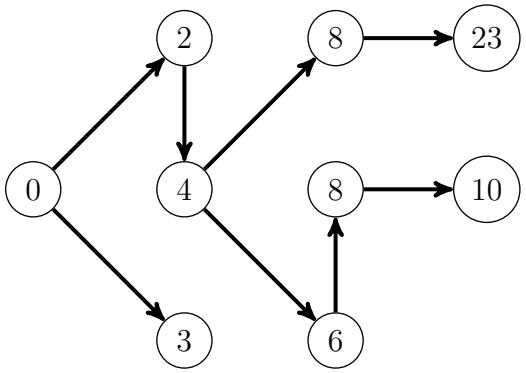


Figure 6.8: Tree produced by Dijkstra's algorithm.

Observe that in the final tree node G is associated with a cost 23 and that is not optimal because a path through x_g can reach G with cost 12. However, since the algorithm terminates when x_g is removed from OPEN, it is not guaranteed that the best path from x_s to every node in the graph has been found. This is the case for G and, in general, this may happen for the nodes still in OPEN when the algorithm terminates. Indeed, the common feature of nodes in OPEN is that their optimal cost has yet to be determined.

In the previous example, we have seen that when the algorithm terminates, it determines the cost of an optimal path between x_s and x_g, but an optimal path between x_s and nodes different from x_g is not necessarily found. To alter this behavior, one could modify Algorithm 2 to obtain the so-called *single source shortest path* that computes the shortest path from x_s to all vertices in the graph reachable from x_s. Algorithm 3 sketches this minor modification.

Note that in this case x_g is not part of the input, because the algorithm computes the shortest path between x_s and all vertices in G.

```
   Data: G = (V, E), x_s ∈ V, c : E → ℝ_≥0
   Result: Shortest from x_s to each vertex in V reachable from x_s
 1 foreach x ∈ V do
 2 |   x.parent ← null;
 3 |   x.g ← ∞;
 4 x_s.g ← 0;
 5 OPEN.initializeEmpty();
 6 CLOSED.initializeEmpty();
 7 OPEN.insert(x_s);
 8 while not OPEN.empty() do
 9 |   x ← OPEN.remove();
10 |   CLOSED.insert(x);
11 |   foreach x' ∈ V such that (x, x') ∈ E do
12 |   |   if x'.g = ∞ then
13 |   |   |   x'.g ← x.g + c(x, x');
14 |   |   |   x'.parent ← x ;
15 |   |   |   OPEN.insert(x');
16 |   |   else if x.g + c(x, x') ≤ x'.g then
17 |   |   |   x'.g ← x.g + c(x, x');
18 |   |   |   x'.parent← x ;
19 |   |   |   OPEN.rebalance(x');
```

Algorithm 3: Single source shortest path algorithm

When the algorithm terminates, one can easily verify if a vertex is reachable from x_s or not by checking the cost to come g attribute. If it is different from ∞, it means that an optimal path was determined, and by recursively following the *parent* attribute it is possible to determine a path of minimum cost. Figure 6.9 shows the result obtained running Algorithm 3 on the graph given in Figure 6.7. Compare this tree with the one in Figure 6.8.

6.3.5 A* algorithm

A* is one of the most influential algorithms in artificial intelligence and is a so-called *informed* search method. The term *informed* is used because A* assumes the availability of a heuristic function h estimating the *cost-to-go* from each vertex to x_g. Note that while in Dijkstra's algorithm g is the *exact* cost to come from the source x_s to a vertex, in A* h is an *estimate* of the cost to go from x to the goal vertex x_g. The intuition is that g and h combined give an estimate of the cost of the shortest path between x_s and x_g constrained to include x. The heuristic h "informs" the algorithm in the sense that it focuses the search in the most promising direction. This heuristic is not assumed to be available to BFS, DFS, or Dijkstra's algorithm, and these algorithms are therefore said to execute an *uninformed* search. The better the heuristic, the higher the increase in performance; that is, the fewer vertices

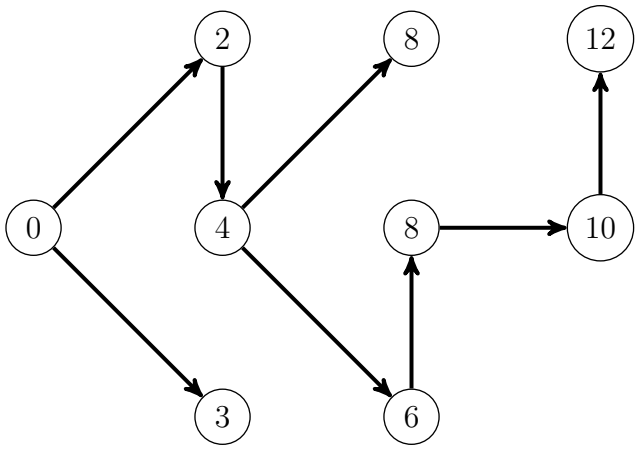


Figure 6.9: Tree produced by Single Source Shortest Path algorithm.

expanded before the optimal solution is found. To understand the role of the heuristic, consider Figure 6.10, showing a path from x_s to x_g passing through x. Let g be the exact cost to come from x_s to x along the path, and h_{exact} the exact cost to go from x to x_g. Therefore, the overall cost of this path is $g + h_{exact}$. If the path is a shortest path from x_s to x_g, then based on the observation we made while discussing Dijkstra's algorithm, the path from x_s to x must be a shortest path, and similarly the path from x to x_g must be a shortest path. Of course, since one of the very reasons to run a planning algorithm is to determine these paths, we neither know them nor their costs beforehand. Dijkstra's algorithm iteratively refines the cost to come g until it eventually converges to the optimal value, but it never considers the cost to go. In A* we assume the availability of an *estimate* for h_{exact}, and we indicate it as h. Therefore, if at a certain point we have determined the exact, optimal cost to come g, an estimate for the overall cost of the path is $g + h$. This is an estimate because h it includes h which is an estimate and not an exact cost.

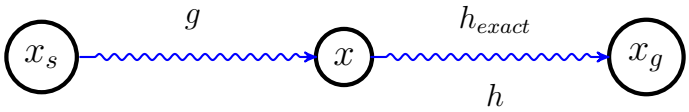


Figure 6.10: The figure shows a node x along one path from x_s to x_g. The cost to come g is the cost of the best path (discovered so far) between x_s and x. h_{exact} is the cost of the shortest path between x and x_g, whereas h is an estimate of h_{exact}.

To ensure the correctness of the algorithm, the heuristic h must be *admissible*, i.e., it must never overestimate the cost to go; that is, it must always satisfy $h \leq h_{exact}$. This concept is formalized by the following definition.

Definition 6.3. *(Admissible heuristic.) Let $G = (V, E)$ be a weighted graph where $c : E \to \mathbb{R}_{\geq 0}$ is the cost function. Let $x_g \in V$ be a goal vertex and for each vertex $v \in V$ let $c(p_{v,x_g})$ be the cost of a shortest path from v to x_g. An* admissible heuristic *is a function $h : V \to \mathbb{R}_{\geq 0}$ such that for each vertex v, $h(v) \leq c(p_{v,x_g})$.*

Note that based on the above definition, the function $h(v) = 0$ for each $v \in V$ is an admissible heuristic because $c(p_{v,x_g}) \geq 0$ for each vertex v. In fact, Dijkstra's algorithm is the special case of A* obtained when using this trivial (albeit correct) heuristic. Moreover, if h is an admissible heuristic, then $h(x_g) = 0$ because the cost to go from x_g to x_g is of course 0. Another desirable property of heuristic functions is consistency, although this concept is more restrictive and, in general, not necessary to ensure correctness.

Definition 6.4. *(Consistent heuristic.) Let $G = (V, E)$ be a weighted graph where $c : E \to \mathbb{R}_{\geq 0}$ is the cost function, and let $x_g \in V$ be a goal vertex. A function $h : V \to \mathbb{R}_{\geq 0}$ is a consistent heuristic if*

1. *for a pair of vertices $u, v \in V$ with $(v, u) \in E$, it holds that $h(v) \leq c(v, u) + h(u)$;*
2. *$h(x_g) = 0$.*

It is easy to see that a heuristic is consistent if it does not overestimate the cost of any edge $c(v, u)$. A consistent heuristic is also admissible, but not all admissible heuristics are consistent. Figure 6.11 and Table 6.3 show a simple case of a graph and a heuristic that is admissible but not consistent. Observe that the minimum cost to go from x_s to x_g is 7, and therefore $h(x_s) = 6$ is admissible. However, $c(x_s, A) < h(x_s) - h(A)$, thus making the heuristic inconsistent.

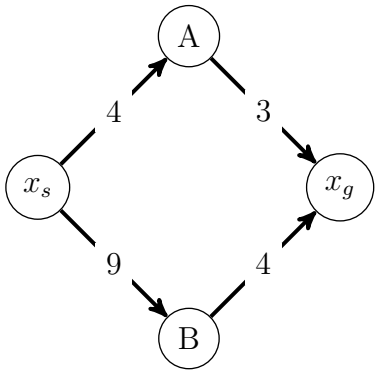


Figure 6.11: A weighted directed graph where every edge is associated with a non-negative cost.

Node	h
x_s	6
x_g	0
A	1
B	1

Table 6.3: Inconsistent estimate for the graph in Figure 6.11

If a heuristic function is consistent, the implementation of A* is simpler and the algorithm is more efficient, in the sense that it will expand fewer nodes. If instead the heuristic is admissible but not consistent, the algorithm is still correct, but its implementation is

slightly more complicated and it will expand more nodes, thus being less efficient. This is somewhat intuitive because a consistent heuristic is more informative than a heuristic that is just admissible, and it better directs A* search. In the limit, if $h(v) = c(p_{v,x_g})$, i.e., if the estimate coincides with the true cost to go, A* will expand only the nodes on the shortest path between x_s and x_g, thus minimizing the number of expanded nodes. However, this limit case is unrealistic, because if the estimate coincides with the true shortest cost, then it is pointless to solve the planning problem since the solution is already known. Nevertheless, a good heuristic is one that is as close as possible to $c(p_{v,x_g})$, and the more accurate the heuristic, the bigger the performance gap between A* and Dijkstra's algorithm.

If a consistent heuristic h is used, A* works like Dijkstra's algorithm, with the only difference that vertices in $OPEN$ are prioritized by the estimated cost $f(v) = g(v) + h(v)$ instead of just $g(v)$. Note that with $h(v) = 0$ we have an admissible heuristic and we indeed obtain Dijkstra's algorithm. Algorithm 4 sketches the pseudocode for A* when a consistent heuristic h is used. The reader should compare it with Algorithm 2.

Data: $G = (V, E)$, $x_s \in V$, $x_g \in V, c : E \to \mathbb{R}_{\geq 0}$
Result: Shortest path from x_s to x_g if it exists, or FAILURE
foreach $x \in V$ **do**
  $x.parent \leftarrow$ **null**;
  $x.g \leftarrow \infty$;
  $x.f \leftarrow \infty$;
$x_s.g \leftarrow 0$;
$x_s.f \leftarrow x_s.g + h(x_s)$;
$OPEN$.initializeEmpty();
$CLOSED$.initializeEmpty();
$OPEN$.insert(x_s);
while not $OPEN$.empty() **do**
  $x \leftarrow OPEN$.remove();
  $CLOSED$.insert(x);
  if $x = x_g$ **then**
    return ExtractPath(x_s, x_g);
  foreach $x' \in V$ *such that* $(x, x') \in E$ **do**
    if $x'.g = \infty$ **then**
      $x'.g \leftarrow x.g + c(x, x')$;
      $x'.f \leftarrow x'.g + h(x')$;
      $x'.parent \leftarrow x$;
      $OPEN$.insert(x');
    else if $x.g + c(x, x') \leq x'.g$ **then**
      $x'.g \leftarrow x.g + c(x, x')$;
      $x'.f \leftarrow x'.g + h(x')$;
      x'.parent$\leftarrow x$;
      $OPEN$.rebalance(x');
return FAILURE;

Algorithm 4: A* algorithm with consistent heuristic.

As in Algorithm 2, A* starts by initializing the attributes of all nodes (loop in line 2). In this case, three attributes are maintained for every node: *parent* and g as in Algorithm 2, and additionally f, which is the key used to prioritize the *OPEN* data structure. Before inserting x_s into *OPEN*, its g and f attributes are set to 0 and $h(x_s)$, respectively (lines 5 and 6). Then the main loop starts and proceeds similarly to Dijkstra's algorithm. The main differences are two. First, when a vertex is discovered for the first time (line 16), it is necessary to set not only the *parent* and g attributes, but also the f attribute (line 18). Second, if a better path to reach a vertex is found (line 21), it is necessary to update the cost to come g, the total cost estimate f, and the *parent*. Note that the condition governing this update (line 21) is based on the cost to come (g) only, and not on the total cost estimate f.

Example 6.5. *Figure 6.12 shows a weighted graph, whereas Table 6.4 displays the h value (estimated cost to go) for each of its vertices. The reader should verify that this estimate is consistent and therefore admissible. Consequently, we can use Algorithm 4.*

As we did previously for Dijkstra's algorithm, it is instructive to examine step by step how the algorithm works. The key feature here is to consider that f and g are updated as the algorithm progresses, and the OPEN queue is now prioritized by the value $f = g + h$. This is displayed in the following table. For space reasons, we omit the step number and only show the ordered elements in OPEN without the values of f used to prioritize them. For each node, we use the notation $P/g/f$, where P is the parent, g is the cost to come, and $f = g + h$.

OPEN	x_s	A	B	C	D	E	F	x_g
x_s	$N/0/5$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$
B, A, F	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$N/\infty/\infty$	$N/\infty/\infty$	$N/\infty/\infty$	$x_s/5/7$	$N/\infty/\infty$
F, A, C	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$B/8/9$	$N/\infty/\infty$	$N/\infty/\infty$	$B/3/5$	$N/\infty/\infty$
C, A, E	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$F/4/5$	$N/\infty/\infty$	$F/10/14$	$B/3/5$	$N/\infty/\infty$
x_g, A, D, E	$N/0/5$	$x_s/1/7$	$x_s/2/5$	$F/4/5$	$C/6/10$	$F/10/14$	$B/3/5$	$C/6/6$

When x_g is extracted from OPEN the computation terminates, and figure 6.13 shows the tree produced after A is run on the graph.*

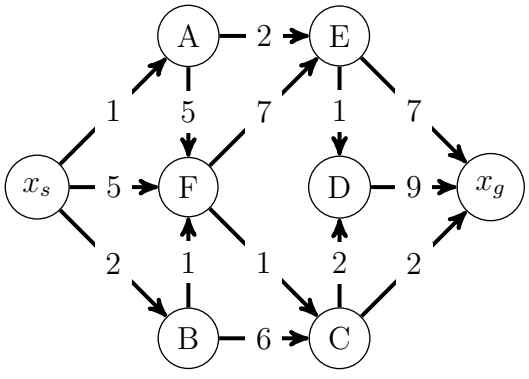


Figure 6.12: A weighted directed graph where every edge is associated with a non-negative cost.

Node	h
x_s	5
x_g	0
A	6
B	3
C	1
D	4
E	4
F	2

Table 6.4: Consistent estimate cost to go h for the graph in Figure 6.12

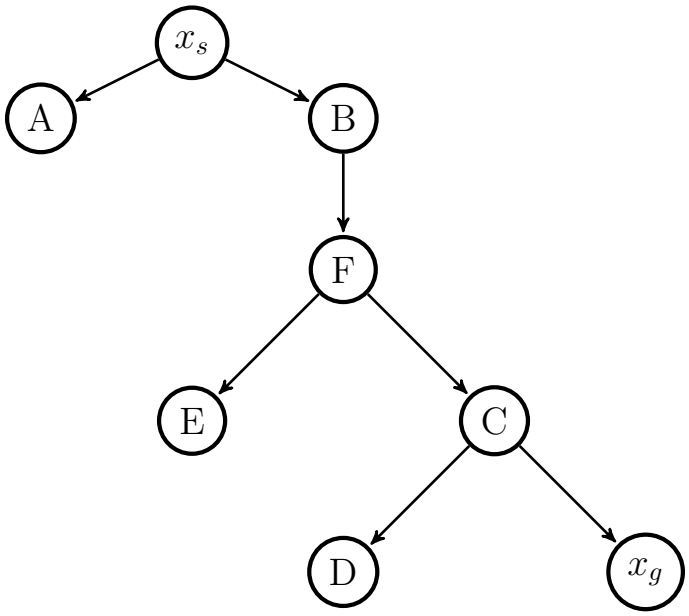


Figure 6.13: Tree produced by the A* algorithm.

If the heuristic function h is admissible but not consistent, nodes that have already been moved to the $CLOSED$ list may be rediscovered with a lower cost. In such cases, these nodes need to be moved back into the $OPEN$ list. Algorithm 5 shows the modifications required in the algorithm to handle this situation.

Theorem 6.5. *If a path between x_s and x_g does not exist, A* returns FAILURE. If a path between x_s and x_g exists and h is an admissible heuristic, A* returns a path of minimum cost, as per the path cost defined in Eq.* (6.3)*.*

What is the advantage of using a heuristic, and what constitutes a good heuristic? Thanks to the heuristic, A* focuses the expansion of nodes towards promising directions, and will in general find a solution expanding fewer nodes than Dijkstra's algorithm (and so it will be faster). Of course, in the ideal situation one would like to have a perfect heuristic, i.e., one such that $h(v) = c(p_{v,x_g})$ for each vertex. But in this case solving the planning problem would be useless, because the heuristic would already provide the optimal path.

```
   Data: G = (V, E), x_s ∈ V, x_g ∈ V, c : E → ℝ_{≥0}
   Result: Shortest path from x_s to x_g if it exists, or FAILURE
 1 foreach x ∈ V do
 2 |   x.parent ← null;
 3 |   x.g ← ∞;
 4 |   x.f ← ∞;
 5 x_s.g ← 0;
 6 x_s.f ← x_s.g + h(x_s);
 7 OPEN.initializeEmpty();
 8 CLOSED.initializeEmpty();
 9 OPEN.insert(x_s);
10 while not OPEN.empty() do
11 |   x ← OPEN.remove();
12 |   CLOSED.insert(x);
13 |   if x = x_g then
14 |   |   return ExtractPath(x_s, x_g) ;
15 |   foreach x' ∈ V such that (x, x') ∈ E do
16 |   |   if x'.g = ∞ then
17 |   |   |   x'.g ← x.g + c(x, x');
18 |   |   |   x'.f ← x'.g + h(x');
19 |   |   |   x'.parent ← x ;
20 |   |   |   OPEN.insert(x');
21 |   |   else if (x.g + c(x, x') ≤ x'.g) AND x' ∈ OPEN then
22 |   |   |   x'.g ← x.g + c(x, x');
23 |   |   |   x'.f ← x'.g + h(x');
24 |   |   |   x'.parent ← x ;
25 |   |   |   OPEN.rebalance(x');
26 |   |   else if (x.g + c(x, x') ≤ x'.g) AND x' ∈ CLOSED then
27 |   |   |   x'.g ← x.g + c(x, x');
28 |   |   |   x'.f ← x'.g + h(x');
29 |   |   |   x'.parent ← x ;
30 |   |   |   OPEN.insert(x');
31 |   |   |   CLOSED.remove(x');
32 return FAILURE;
```

Algorithm 5: A* algorithm with admissible but not consistent heuristic.

To determine good admissible heuristics, some domain knowledge is usually very useful. For example, in planning problems related to navigation tasks where the state of the robot is its position and the robot must reach a given target location, an immediate admissible heuristic is given by the length of the straight segment between the robot pose and the goal location.

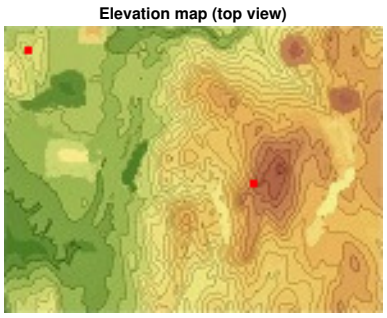
6.3.6 Examples

We now compare how BFS, Dijkstra, and A* solve the same planning problem. Figure 6.14a shows an elevation map retrieved from the internet. Green areas indicate areas of lower elevation, whereas brown areas are associated with locations at higher elevations. Figure 6.14b shows a mesh representation of the same area. The task is to move from the top-left location marked with a red dot to the location in the middle, also marked with a red dot. From each location, it is possible to take four actions (up/down/left/right) to move to a nearby location. Unfeasible actions are removed for locations near the borders or at the corners. Figure 6.14c shows the path determined by the BFS algorithm. Recall that in BFS all actions have the same costs, and the algorithm finds the path with the minimum number of transitions. Figure 6.14d shows instead the contents of the $CLOSED$ list once the algorithm terminates. Nodes in the $CLOSED$ list are marked as green pixels on the map. Note the uniform diagonal frontier generated by the algorithm. This confirms that BFS expands nodes according to the distance intended as the number of hops from the source.

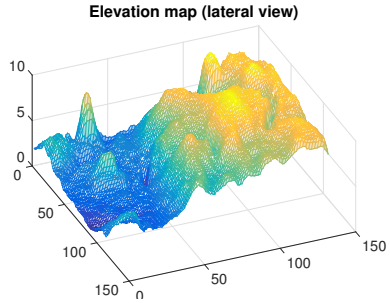
We next consider the weighted case and compare the paths produced by Dijkstra's and A* . The cost of an action is induced by the elevation map. To this end, let $e(x)$ be the elevation associated with state x, as per Figure 6.14b. For $(x, x') \in E$, if $e(x) \geq e(x')$, then $c(x, x') = 1$, i.e., moving downhill or to a cell at the same elevation has cost 1. If instead $e(x) \leq e(x')$, then $c(x, x') = 1 + K(e(x') - e(x))$, i.e., moving uphill costs 1 plus a term proportional to the difference in elevation. Hence it may make more sense to take a longer path (in terms of number of actions), rather than a steeper path. Figure 6.14e shows that path returned by Dijkstra's algorithm to determine a path of smallest cost according to the cost function just defined. The cost of the path is 168. Figure 6.14f shows instead the contents of the $CLOSED$ list after a solution is found. Note that in this case the frontier is not uniform because nodes are extracted from the $OPEN$ list according to their cost-to-come. Finally, Figure 6.14g shows an optimal path determined by A* and Figure 6.14h shows instead the contents of the $CLOSED$ list after A* terminates. First, compare Figure 6.14g with 6.14e. The two paths are different, but the associated cost is the same, i.e., 168. This is consistent with the former observation that both Dijkstra and A* solve the same problem, i.e., they determine a path of minimal cost. In this case, the path is different but the cost is the same. Next, compare Figure 6.14h with 6.14f. As for the BFS algorithm, nodes in the $CLOSED$ list are displayed as bright green pixels. The figures clearly show that A* takes advantage of the provided heuristic and better focuses the search, thus terminating after having expanded far fewer nodes. The more accurate the estimate h, the wider the gap.

6.4 Navigation Functions

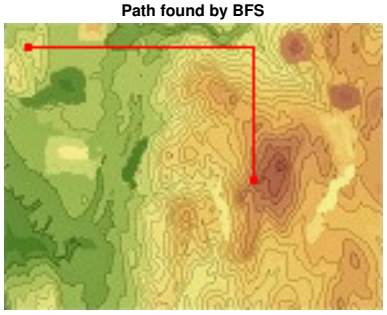
In section 6.3.4, we introduced the single-source shortest path algorithm as a minor modification of Dijkstra's algorithm. This algorithm is interesting because it does not compute just a plan, but rather a family of plans, i.e., a plan for each possible vertex in the graph. One can think of this algorithm as a form of pre-processing. After having run the algorithm over a given planning graph, we can answer any planning question of the type "Compute a plan



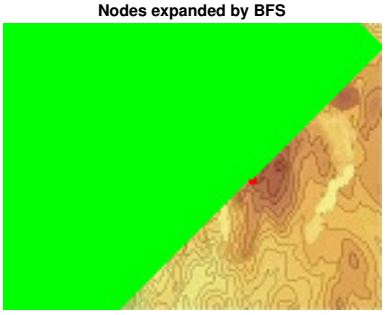
(a) Elevation map (top view)



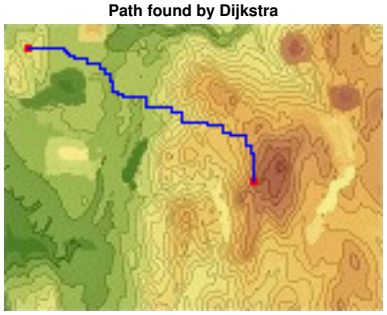
(b) Elevation map (lateral view)



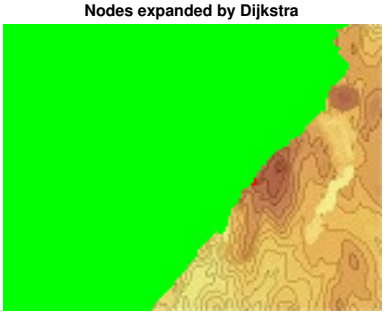
(c) Path determined by BFS



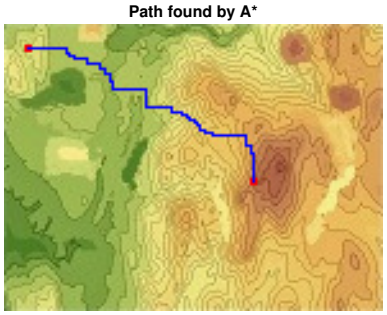
(d) Nodes expanded by BFS



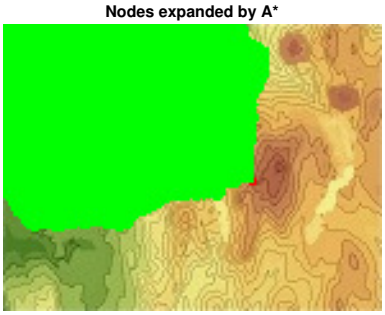
(e) Path determined by Dijkstra



(f) Nodes expanded by Dijkstra



(g) Path determined by A*



(h) Nodes expanded by Astar

Figure 6.14: Planning examples.

from x_s to v" for any v in the planning graph. Of course, if we change the starting vertex x_s, the tree returned by Dijkstra's algorithm no longer provides the right answer. Navigation functions are conceptually similar, but they swap the roles between source and goal vertex. A navigation function provides a family of plans to reach a goal vertex x_g starting from

any vertex in the graph. This concept is rather simple (almost trivial) for the deterministic case but is important to handle uncertainties, e.g., when the robot may veer of course while navigating to x_g. In such case, it is not necessary to plan a new path because the navigation function provides the correct action to execute irrespective of where the robot is. The above intuition can be formalized as follows. For a given planning graph $G = (V, E)$, a navigation function ψ assigns a non-negative value[4] to each vertex, i.e., $\psi : V \to \mathbb{R}_{\geq 0}$. Note that this definition so far is rather generic. A navigation function ψ defines a plan as follows. For vertex $x \in V$, let $e(x)$ be the set of edges (x, y) outgoing from x. Then

$$e' = \arg \min_{(x,y) \in e(x)} \{\psi(y)\}$$

In other words, among all edges outgoing from x, the navigation function implicitly defines a plan by selecting the edge (action) leading to the connected vertex y with the lowest value of ψ. Ties are arbitrarily resolved. Recalling that each edge is associated with an action, the navigation function therefore defines a plan.

Note that e' not only identifies an action but also a vertex y reachable from x. This vertex will be indicated as $e'(x)$. Unless we impose additional properties on the navigation function, the associated plan is, in general, not very useful. The following definition identifies the useful navigation functions.

Definition 6.5. *Let $G = (V, E)$ be a planning graph and let $x_g \in V$ be a goal vertex. A navigation function $\psi : V \to \mathbb{R}_{\geq 0}$ is* feasible *if it satisfies the following three conditions.*

1. *$\psi(x_g) = 0$.*
2. *If $x \in V$ is a vertex from which there is no path to x_g, then $\psi(x) = \infty$.*
3. *If x is a state from which x_g can be reached, then for $y = e'(x)$ we have $\psi(y) < \psi(x)$.*

Feasible navigation functions are also called *potential functions* because, by following the negative gradient, they can be used to reach the goal state x_g. Figure 6.15 shows a valid navigation function for a grid world. The goal vertex x_g is marked by the value 0, and black obstacle cells are assigned the value ∞ (not shown in the figure).

It is straightforward to verify that if at every cell we move to a neighboring cell with lower ψ value and we iterate this method, we eventually reach x_g. How can we compute a navigation function? For a case like the one displayed in the figure, and assuming that $(x, y) \in E \Rightarrow (y, x) \in E$, it is sufficient to run BFS from x_g and let the algorithm go as long as there are more elements in the $OPEN$ queue, much like we did for the single source shortest path.

Navigation functions can be used to define control policies, i.e., functions that assign to each state an action to be executed. Policies are commonly indicated with the letter π. If we start at a state x, the action to execute is $\pi(x)$. Once we execute this action, we move to a state y. From there we again follow the policy and execute action $\pi(y)$, and so on. As fornerly stated, this approach is not particularly meaningful if everything is deterministic, but will

[4]To avoid additional notation, with slight imprecision we assume $\infty \in \mathbb{R}_{\geq 0}$, i.e., the navigation function may assign ∞ to some vertices.

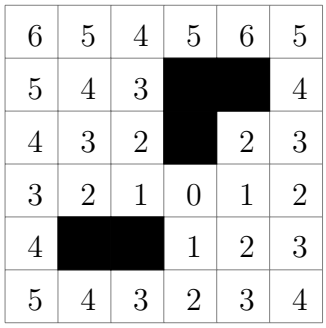


Figure 6.15: Feasible navigation function for a grid world. The goal vertex x_g is the cell with the value 0.

be important to implement planners capable of dealing with noisy state transition equations. Policies of this type are also called *feedback control policies* or *feedback plans* (recall section 6.1) because they are based on the assumption that the state is always known, i.e., that the state is observable.

Finally, note that the term navigation is also often used to indicate another process integrating perception and obstacle avoidance with plan execution, for example to avoid a person that may be approaching the robot. Navigation functions as described in this section are off-line processes that do not include any perception but rather rely on a provided map of the environment. Therefore not able to implement this type of avoidance. However, since they are simple to compute, they can be repeatedly computed online integrating both preexisting information and sensor data acquired on the fly, thus providing a system capable of handling dynamic obstacles and changes in the environment.

6.5 ROS Actions

Before discussing how planning is implemented in ROS through the navigation stack Nav2, it is necessary to dig a bit deeper into ROS actions, which were briefly introduced in Section 2.11. Actions are extensively used in Nav2 that will be discussed later. Actions implement *non-blocking, pre-emptable* function calls, i.e., control is returned to the client (caller) after the action is initiated, and its execution can be interrupted (pre-empted) by the client before it is completed. The functionality of an action is performed by an action server that runs on a separate thread from the client. Therefore, server and client run in parallel. Actions are appropriate when the task performed by the server may take a long time to complete. In such a case, if progress stalls, the client may decide to interrupt the execution of an action. This is, for example, the case when the robot is commanded to reach a certain location. While the planning phase itself may be quick, it may take a significant amount of time for the robot to get to the desired final location. In this case, a non-blocking action is called, so that the client can still perform other operations while the robot is moving towards its goal location. A fundamental difference between actions and services is that during the execution of an action, feedback information is passed back to the client so that progress can be monitored and additional steps undertaken if needed (e.g., terminating the action through pre-emption if no progress is reported or the objective has changed.) Moreover,

once an action terminates, a result is passed back to the client. With a service, instead, no feedback information is provided to the client during the exectution. Just the result is communicated at the end. The following terms define the components involved in an action.

Action Server : the node offering and implementing an action.

Action Client : the node sending a request to an action server.

Goal : the message sent by the client to the server when an action is initiated. The type and content of the goal message depends on the action being called.

Feedback : a message periodically sent back by the server to the client to report about progress. Multiple feedback messages of the same type are in general sent while the action is being executed.

Result : a unique message sent by the server back to the client once the action is completed. The message usually includes information about the outcome of the action (e.g, success or failure.)

From the client standpoint, it is not only possible to terminate an action before it is completed (pre-emption), but it is also possible to enter a busy-waiting state waiting for the action to complete. This option, however, should be used with caution, as it goes against the very reason actions are introduced. As in the case of messages and services, ROS comes with its own set of predefined actions, but it is possible to define new ones. In the following, we will only discuss how to use existing actions and will therefore focus on the client side. The reader is referred to the ROS official documentation for the steps necessary to introduce new actions and to implement an action server.

Figure 6.16 illustrates the typical interaction between an action client and an action server. Green sections on the vertical bars indicate that the associated node is performing some computation, whereas red segments indicate a state of busy waiting.

After a goal request is sent, the client continues its own computation, and messages sent back by the action server are asynchronously processed through callback functions, exactly as we do with a subscribed topic. In fact, it is necessary to call one of the `spin` functions to retrieve these messages, since they are passed through dedicated topics.

Actions in ROS are offered through the `rclcpp_action` package. From the client perspective, the main function is `create_client`, which creates a client object that can then interact with an action server. The function takes as a parameter a pointer to a node and the name of the action for which a client is being created (see listing 6.1 for an example). Once a client object has been created, the following member functions can be called to interact with the associated action server (refer to the official ROS 2 documentation for more details):

`wait_for_action_server` : waits until the action server is ready to receive a goal. The function can optionally accept a timeout parameter to limit the wait time. If no timeout is given, the function is blocking.

`action_server_is_ready` : returns true if the action server is ready to accept a goal request.

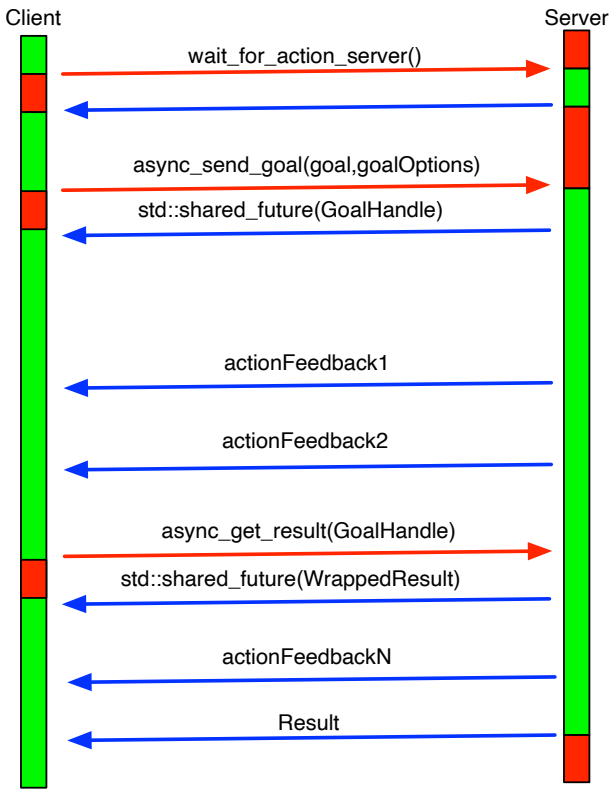


Figure 6.16: Interactions between action server and action client. Red sections on the bars lines indicate busy waiting, whereas green sections indicate that the node is active.

`async_send_goal` : sends a goal to the action server and registers callback functions to be called when the action completes, when the action becomes active, and when feedback messages are received from the action. The three callbacks are stored in an object of type `SendGoalOptions`. The function returns a shared *future* object to a `GoalHandle` (see below for a discussion about futures.)

`async_get_result` : gets the result for an active goal specified through a goal handle returned by `async_send_goal`. The function returns a shared future to a `WrappedResult` holding the result.

`async_cancel_goal` : pre-empts the action by canceling the current goal. The function returns a future that is set when the cancel request is honored.

Before we discuss more in depth these functions, a few more details are needed.

6.5.1 Futures

As formerly stated, action clients and servers run concurrently as separate threads and are implemented on top of the C++ standard libraries for concurrency support. Consequently, some of the results produced by the action server are returned as instances of the templatized class `std::future` or `std::shared_future`. A *future* is an object that is used to gain access

to a shared state and provide access to the result produced by an asynchronous operation. Specifically, in this context, a future is used to pass results between the server and the client, so that the client can verify if the server has terminated its operation and provided the result. In ROS, one can use the function `spin_until_future_complete` to spin until the result expected to be passed back through a future is ready (thus blocking the caller). We already saw an instance of this call in Listing 5.7. Once the future has been filled, the result can be obtained with the member function `get`. The reader is referred to the official C++ implementation for more details about futures, while in the following examples we will show the bare minimum to interact with ROS actions.

6.5.2 Goals, Goal Options, Goal Handles and Wrapped Results

In dealing with actions, there are some specific classes that are important. The function `async_send_goal` accepts two parameters. The first is a `Goal` specifying the parameters for the action being called. The second parameter is an instance of `SendGoalOptions` storing the pointers to three callback functions to be called while the action is executed. The three callback functions are called to process the response from the action server when an action is sent, the feedback messages sent during the action execution, and the result sent by the action server when the action terminates. In the following, we provide a minimal example showing how to interact with an action server. More specifically, we will call an action called `Spin` that is provided by the TurtleBot robot and that makes the robot turn in place. Note, however, that the action `Spin` has nothing to do with the function `spin` we saw earlier. The structure of the `Goal` parameter passed as the first parameter to `async_send_goal` can be ascertained using `ros2 interface show`. For example, the `Spin` action discussed in the example is an action of type `nav2_msgs::action::Spin`, and its interface is as follows:

```
float32 target_yaw
builtin_interfaces/Duration time_allowance
    int32 sec
    uint32 nanosec
---
#result definition

# Error codes
# Note: The expected priority order of the error should match the message order
uint16 NONE=0
uint16 UNKNOWN=700
uint16 TIMEOUT=701
uint16 TF_ERROR=702
uint16 COLLISION_AHEAD=703

builtin_interfaces/Duration total_elapsed_time
    int32 sec
    uint32 nanosec
uint16 error_code
```

```
string error_msg
---
#feedback definition
float32 angular_distance_traveled
```

From the output we see that the goal consists of a structure with two fields. `target_yaw` is a floating point value specifying how much the robot should turn (the value is given in radians). `time_allowance` is an instance of `Duration` from the package `builtin_interfaces` and specifies how much time is allowed for action completion. If not specified, in the current implementation of Nav2, it defaults to 10 seconds. For an action of type `T`, the `Goal` object is created with a call to `T::Goal()` (see example 6.1 for the syntax details.)

`async_send_goal` returns either the pointer to a future to `GoalHandle` (see Figure 6.16) or a `nullptr` if the server rejects the goal. At this point, the client can proceed in two ways. It can call `spin_until_future_complete` on the returned future to get the `GoalHandle` and then call `async_get_result` on the handle to get a future to a `WrappedResult` that will be completed when the action finishes. The `WrappedResult`, when ready, holds a result code (as per the values shown in the action interface), as well as a shared pointer to the result segment in the action definition. The result segment in the action definition is the middle one, so in the previous example it would be `total_elapsed_time` plus an integer `error_code` and a string `error_msg`. Alternatively, the client can simply call one of the spinning functions until eventually the callback function associated with the result is called. The callback function also receives a `WrappedResult`, so both methods end up getting the same values.

Listing 6.1 shows how a minimal client example can establish a connection to an action server, send a goal, and then process the messages sent back.

Listing 6.1: Action Client

```
1  #include <rclcpp/rclcpp.hpp>
2  #include <rclcpp_action/rclcpp_action.hpp>
3  #include <nav2_msgs/action/spin.hpp>
4  #include <action_msgs/msg/goal_status.hpp>
5
6  class ActionCaller : public rclcpp::Node {
7
8  public:
9    using GoalHandleSpin =
10     rclcpp_action::ClientGoalHandle<nav2_msgs::action::Spin>;
11   ActionCaller() : Node("actioncaller") {
12     spin_client = rclcpp_action::create_client<nav2_msgs::action::Spin>
13       (this,"spin");
14     rotating = false;
15   }
16
17   void RotateRobot(double target_yaw) {
18     using namespace std::placeholders;
19     spin_client->wait_for_action_server();
20     auto goal_message = nav2_msgs::action::Spin::Goal();
21     goal_message.target_yaw = target_yaw;
22
23     auto send_goal_options =
24       rclcpp_action::Client<nav2_msgs::action::Spin>::SendGoalOptions();
```

```
25      send_goal_options.goal_response_callback =
26        std::bind(&ActionCaller::response_callback, this, _1);
27      send_goal_options.feedback_callback =
28        std::bind(&ActionCaller::feedback_callback, this, _1, _2);
29      send_goal_options.result_callback =
30        std::bind(&ActionCaller::result_callback, this, _1);
31
32      auto send_goal_future =
33        spin_client->async_send_goal(goal_message,send_goal_options);
34      rclcpp::spin_until_future_complete(get_node_base_interface(),
35                                         send_goal_future);
36    }
37
38    bool RobotIsRotating() { return rotating; }
39
40  private:
41    rclcpp_action::Client<nav2_msgs::action::Spin>::SharedPtr spin_client;
42    bool rotating;
43    void response_callback(const GoalHandleSpin::SharedPtr & goal_handle)
44    {
45      if (!goal_handle) {
46        RCLCPP_ERROR(this->get_logger(), "Goal was rejected by action server");
47      } else {
48        RCLCPP_INFO(this->get_logger(), "Goal accepted by action server");
49        rotating = true;
50      }
51    }
52
53    void result_callback(const rclcpp_action::ClientGoalHandle
54                         <nav2_msgs::action::Spin>::WrappedResult & result)
55    {
56      if (int(result.code) == action_msgs::msg::GoalStatus::STATUS_SUCCEEDED)
57        RCLCPP_INFO(get_logger(),"Rotation completed with success");
58      rotating = false;
59    }
60
61    void feedback_callback(rclcpp_action::ClientGoalHandle
62                           <nav2_msgs::action::Spin>::SharedPtr,
63                           const std::shared_ptr
64                           <const nav2_msgs::action::Spin::Feedback> f)
65    {
66      RCLCPP_INFO(get_logger(),"Angle %f",f->angular_distance_traveled);
67    }
68  };
69
70  int main(int argc,char **argv) {
71
72    rclcpp::init(argc,argv); // initialize the ROS subsystem
73    ActionCaller node; // create node
74    node.RotateRobot(2.0); // turn ~ 114 degrees
75    while ( node.RobotIsRotating() )
76      rclcpp::spin_some(node.get_node_base_interface());
77    rclcpp::shutdown(); // shutdown ROS
78    return 0;
```

```
79 }
```

The class `ActionCaller` calls the action `spin` that is part of Nav2, which will be discussed next. The action will spin the robot in place to a given angle. To offer this functionality, the class features three callback methods (`response_callback`, `result_callback`, and `feedback_callback`) as well as a pointer to an object of type `rclpp_action::Client` called `spin_client`. Note that `rclpp_action::Client` uses templates to specify the type of service used by the client (`nav2_msgs::action::Spin`).

The client object is initialized in the constructor with the function `create_client`, whose first parameter is a pointer to a node and the second is a string with the name of the action (`spin`, in this case). The action is then called in the method `RotateRobot`. First, we wait for the action server to become available by calling `wait_for_action_server`. Then, a goal message to be sent to the server is created with the `Goal` method of `nav2_msgs::action::Spin` and is initialized with the assigned target yaw. In this case, since `time_allowance` is not specified, we use the default value of 10 seconds. The goal message is sent to the server together with an instance of `SendGoalOptions` that includes three pointers to the callback functions to be called when the action server acknowledges the receipt of the goal, when feedback is received, and when the result is received. The request to the server is then sent using the method `async_send_goal`, which takes as parameters the goal message and the goal options. This function activates the action and returns a future that the client can use to process the message returned by the action server in response to the goal request. This is done by calling `spin_until_future_completed`. This is a blocking function that returns a future to a `GoalHandle`. In this case, rather than waiting or spinning on the future, in the main function, after the `Spin` method has been called, we use `spin_some` to process incoming feedback and result messages. Eventually, when the `result_callback` function is called, the variable `rotating` is set to `false` and the cycle ends. Note that after the action is called, the caller and the action server proceed in parallel.

To test this simple program and see it in action open a shell and give the following commands

```
ros2 launch gazeboenvs tb4_simulation.launch.py
```

This will launch a Gazebo simulation with the TurtleBot robot. In a separate shell, run

```
ros2 run examples actioncall
```

This will make the robot turn in place in Gazebo. To see more articulated examples of interactions with ROS actions, see the package `navigation` in the MRTP GitHub.

6.6 The navigation stack Nav2

Because of its practical importance, ROS supports planning in various ways. However, as stated earlier in this chapter, an open-loop strategy, where a plan is executed without incorporating any feedback from sensors, is not a practical approach (see Figure 1.6). For this reason, the planning components available in ROS are tightly integrated into a more

complex system called Nav2 (*navigation stack*), which provides both planning and execution with sensor feedback, as well as replanning when needed. From a very high-level standpoint, Nav2 integrates all the components shown in Figure 1.3, i.e., perception, planning, and execution. Nav2 is a complex, highly configurable software system that implements a wide variety of planning and navigation algorithms. It is used by many companies to deploy successful commercial products and many of its core developers are from the corporate world.

A complete discussion of Nav2 is beyond the scope of these notes. The reader is referred to its official website[5] for a detailed overview of its many interconnected components. Although we provide an introduction to Nav2 here, some of its core functionalities rely on localization and mapping algorithms that have not yet been covered and will be discussed in later chapters. Nevertheless, this is an appropriate point to introduce the component, as it demonstrates how the material presented so far is essential for deploying robots in the physical world. In simplified terms, the navigation stack extracts a planning graph from a given map of the environment, computes a plan, and then executes it by sending appropriate commands to the robot's motors. Nav2's default configuration and primary use cases are geared toward 2D planning and navigation, making it well-suited to the types of robots considered in these notes. However, it is worth mentioning that it can also incorporate input from sensors such as depth cameras, which provide three-dimensional data. Additionally, its plugin-based architecture allows users to retune or extend components to handle more complex scenarios, like outdoor navigation in uneven terrain. Figure 6.17 sketches the high-level architecture of Nav2.

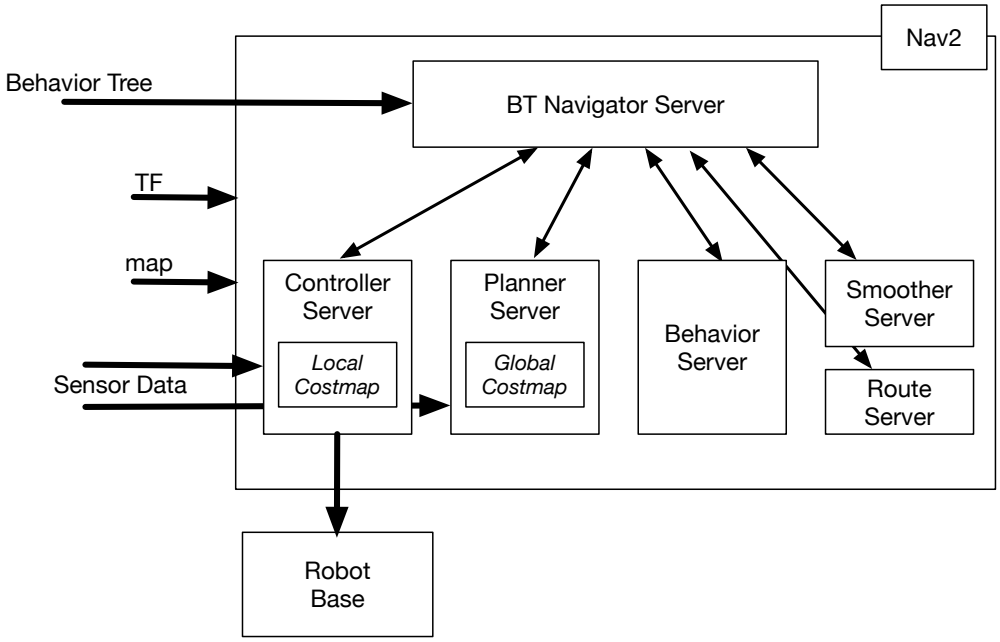


Figure 6.17: General overview of Nav2 (figure adapted from `https://docs.nav2.org/`.)

The larger box represents the overall Nav2 system. Its inputs can be identified on the left, and its output appears at the bottom. Inputs include:

[5] `https://docs.nav2.org/`

- a behavior tree specifying how to orchestrate the interactions among submodules;
- transformations from `tf2`, which describe the kinematic arrangement of objects in the environment (including the current pose of the robot), as well as the configuration of sensors mounted on the robot;
- a map of the environment in which the robot is operating (see Figure 6.1);
- sensor data from the robot's onboard systems, particularly sensors capable of measuring distances to obstacles.

The output of Nav2 is a stream of commands sent to the robot's actuators. One of the key strengths of Nav2 is its adaptable architecture, which can be configured for different robot bases. This allows developers to write high-level robot software that is largely agnostic to the underlying hardware configuration.

Inside Nav2, we find six modules, each implemented as a ROS server that provides actions or services. At the top is the BT (Behavior Tree) Navigator Server, which implements the highest level of planning and handles the execution of high-level commands, such as "Go to pose p" or "Navigate through the following waypoints: p_1, p_2, p_3." This server is responsible for orchestrating the functionalities provided by the other five servers to accomplish a given task. In doing so, it balances goal-directed behaviors (e.g., moving toward the goal) with so-called *recovery behaviors*, which are executed when the robot encounters unexpected situations or ceases to make progress. As shown in the figure, the BT Navigator Server receives as input a Behavior Tree, i.e., a set of rules that specify how to switch between different tasks. The BT Navigator Server will be discussed in more detail in Section 6.9. The other five servers offer specific functionalities that will be described shortly and are activated by the BT Navigator Server. Note that sensor data are provided to two of these servers: the Controller Server and the Planner Server. Finally, it is worth noting that only the Controller Server is responsible for interacting directly with the robot base by issuing `cmd_vel` commands to move the robot.

During the execution stage, nodes within Nav2 continuously poll the sensors to make necessary corrections; in other words, they implement a *closed-loop* system. Nav2 also requires that an estimate of the robot's pose be continuously available via `tf`, in order to determine which action to execute next. This estimate can come either directly from a sensor (e.g., GPS) or from a more sophisticated estimation algorithm (e.g., a particle filter or Kalman filter, both of which will be discussed in Chapter 8), possibly fusing data from multiple sources. When setting up the navigation stack, all of these components must be configured, and multiple nodes are typically launched when using it. The functionalities of the five additional servers are briefly described next. These servers are also referred to as *plugins* because they can be reconfigured using plugin modules.

Planner Server: The Planner Server is responsible for computing a global plan between the assigned start and goal poses. The computed plan is informed by the available map of the environment and can take into account the kinematic model of the robot. The output of the planner is an *open-loop* reference path. The default implementation of this server uses some of the planning algorithms discussed earlier in this chapter (see Section 6.7 for more details).

Controller Server: The Controller Server (also known as the *local planner*) follows the path computed by the Planner Server. It does this by issuing velocity commands to the underlying platform. In doing so, the controller implements a *closed-loop* strategy, i.e., it continuously monitors the onboard sensors to ensure the robot follows the assigned path while avoiding obstacles. The Controller Server is informed by the kinematic model of the robot and, importantly, by sensor data (see Section 6.8 for more details).

Smoother Server: The Smoother Server modifies a path produced by the Planner Server to account for additional criteria that the planner may not have considered. For example, the smoother may eliminate sharp turns (to smooth the path) or increase clearance from obstacles. This server may be queried optionally.

Behavior Server: This module implements maneuvers to handle failures or unforeseen events (e.g., when the robot gets stuck and cannot make progress). The behaviors implemented by this module are triggered by the BT Navigation Server when the robot fails to make progress toward its assigned goal. Currently, this server implements three maneuvers: spin, wait, and back up. The spin service we invoked in the example in Section 6.5 is, in fact, one of the maneuvers offered by the Behavior Server.

Route Server: The Route Server is a high-level planner that computes paths on a navigation graph. This differs from the Planner Server, which computes paths based on the environment map, although the navigation graph may have been extracted from the environment map.

Planning and executing a path in a cluttered environment (e.g., navigating between two locations in a shopping mall) is generally a complex task. It becomes even more challenging when the robot must account for dynamic events such as people approaching or changes in the environment (e.g., a moved obstacle). For this reason, a simple pipeline consisting of (1) planning, (2) smoothing, and (3) following the path using the controller is often not sufficiently robust. More sophisticated behaviors are necessary, for example, backing away from a suddenly appearing obstacle or replanning an entirely new path after encountering an unforeseen dead end. To ensure robustness and fault tolerance, the BT Navigator Server is introduced. Nav2 includes a predefined set of behavior trees, allowing beginning users to rely on these defaults without needing to design their own. In the above breakdown of functionalities, the Planner Server is also referred to as the *global planner*, while the Controller Server is referred to as the *local planner*. The distinction between these two will be discussed shortly.

6.6.1 Localization, Maps, and Costmaps

The availability of a map and the ability to localize the robot within that map are essential to most planning and navigation algorithms, and Nav2 is no exception. During startup, Nav2 loads a static environment map using a bitmap representation similar to Figure 6.1. This map is typically made available through a topic called `/map`, of type `nav_msgs::msg::OccupancyGrid`. A detailed discussion of map representations and mapping algorithms will be given in subsequent chapters, but for the time being, the reader may

refer back to the brief discussion on map representations provided at the end of Section 6.1. Poses passed to Nav2 are expressed with respect to the `map` frame defined in the map data (see Subsection 4.13.5). Additionally, as the robot moves, it must continuously estimate its own location relative to the `map` frame. This assumption is consistent with our earlier discussion on graph-based planning, where we assumed the robot always knows the vertex (state) where it is located (refer to the grid world examples discussed previously). In indoor environments, localization can be achieved by matching readings from onboard sensors to the provided map (localization will be discussed in Chapter 8). In outdoor environments, the robot's pose may instead be obtained from a GPS receiver or, more commonly, by fusing multiple data streams using a Kalman filter. For now, it is sufficient to note that a dedicated node launched when Nav2 starts provides this localization functionality.

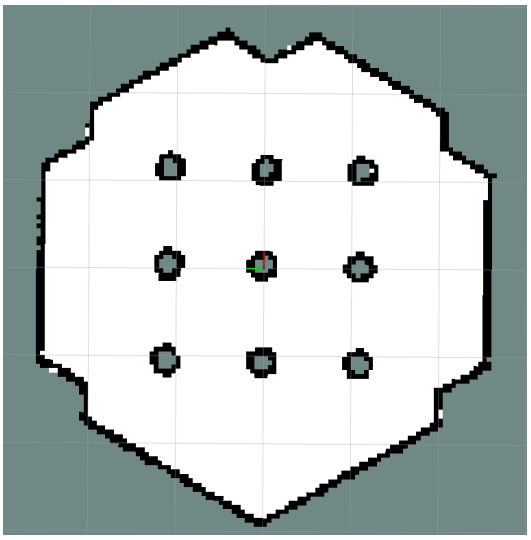
Nav2 defines and maintains two costmaps that influence the global and local planners described above (refer to Figure 6.18 for the following discussion). These costmaps are called the *global costmap* and the *local costmap*. As shown in Figure 6.17, sensor data influence both of these costmaps. Importantly, while both maps are affected by sensor input, they are commonly configured so that updates occur at different frequencies, with the local costmap being updated at a higher rate. The global costmap is used by the global planner to find a long-term, feasible path from the robot's current position to a distant goal. This is achieved using planning algorithms such as Dijkstra and A* , which operate on weighted graphs. The local costmap, in contrast, is used by the local planner (control server) to perform short-term, reactive collision avoidance while following the global path determined by the global planner. While the global costmap, by definition, is intended to be global and span the entire workspace, the local costmap is smaller and covers a rectangular area centered around the robot (e.g., it may cover a square area with a size of 3 meters). Both costmaps are obtained by superimposing different layers that combine various information sources. The specific layers included in each costmap can be configured and generally vary, though some configurations are more common than others. The global costmap typically overlays the static layer, the obstacle layer, and the inflation layer. The local costmap, on the other hand, typically includes the inflation layer and the obstacle (or voxel) layer. The static layer may or may not be included in the local costmap; when it is, it serves to seed the local costmap. The different layers are defined as follows:

Static Layer: Represents the static map of the environment retrieved from `/map` at startup.

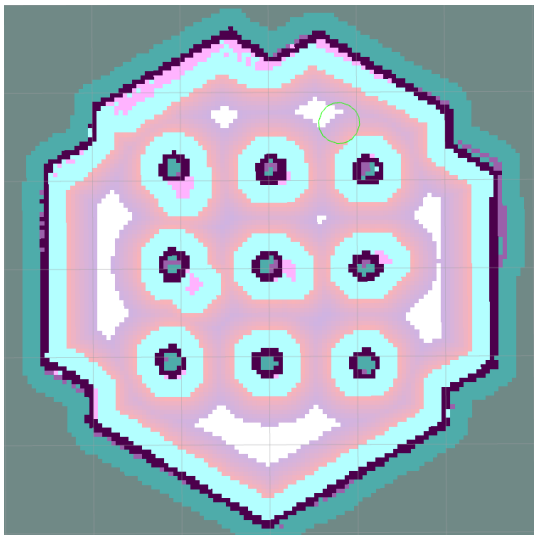
Obstacle Layer: Incorporates information about static and dynamic obstacles detected by the onboard sensors. This layer handles dynamic components that were not modeled in the static layer.

Voxel Layer: Similar to the obstacle layer, but relies on sensors providing three-dimensional data, such as depth cameras or three-dimensional range finders.

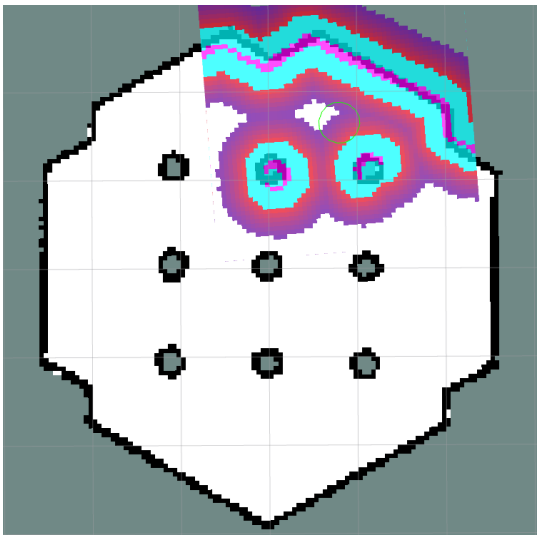
Inflation Layer: The inflation layer adds a safety margin by increasing the costs of cells surrounding the obstacles (thus inflating them). This encourages the planner to stay clear of obstacles when possible. The inflation layer retrieves its data from the other layers. The width of the inflation can be configured using a parameter called `inflation_radius`, which should be chosen based on the size of the robot.



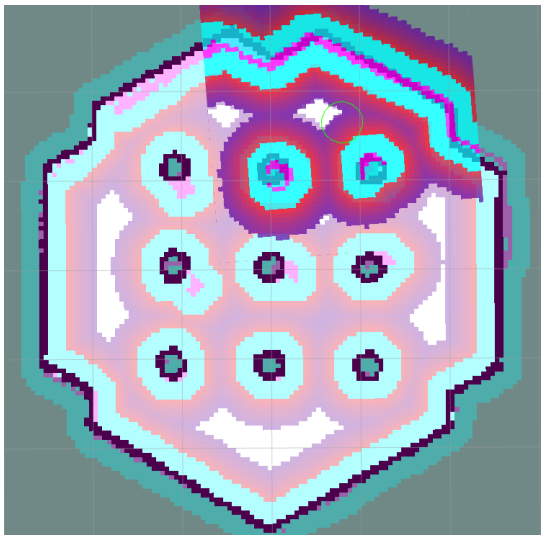
(a) Map of the environment (as provided through the `/map` topic.) This corresponds to the static layer in the global costmap



(b) Global costmap with the static layer and the inlation layer.



(c) Local costmap with the obstace layer and the inflation layer.



(d) Superimposition of the global and local costmaps.

Figure 6.18: Visualization of the different costmaps used by Nav2.

These costmaps are handled by the `nav2_costmap_2d` package, and they are all instances of `nav_msgs::msg::OccupancyGrid`. Accordingly, they are represented as bidimensional regular grids, where each cell contains a cost value between 0 and 255. Free space is represented by the value 0, while forbidden regions (also called a *lethal obstacle*) are represented by the value 254. The value 255 is used to represent cells with an unknown state and obstacles have the value 253. Intermediate values are also used, with higher values guiding

the planner to prefer paths that maintain a safe clearance from obstacles (recall how planning algorithms consider costs when computing an optimal path). The cost associated with each cell is used by both the planner and the controller servers. The global costmap can be retrieved from `/global_costmap/costmap`, while the local costmap can be retrieved from `/local_costmap/costmap`. Although usually not necessary, it is also possible to retrieve the individual layers (the topics and types can be determined using `ros2 topic`). The values computed by both the local and global maps are influenced by the shape of the specific robot used to initialize Nav2. By using both a global and a local map, Nav2 is able to produce plans that address both long-term objectives (e.g., reaching a faraway location) and short-term ones (e.g., avoiding a suddenly appearing obstacle or correcting the trajectory when the robot veers off course due to disturbances).

6.7 The Planner Server

The planner server can be configured to run different planning algorithms depending on the application (leveraing its architecture based on plugins). A complete list of available plugins is available on the Nav2 website. Here we only discuss the `NavfnPlanner` plugin, which is widely used. `NavfnPlanner` computes a path from the current pose to an assigned target pose. More precisely, it computes a path between a start and an assigned goal cell in a grid (with the start grid cell identified by the current robot pose). To accomplish this, the first step is converting the global costmap (i.e., a grid) into a weighted graph. Each cell in the costmap corresponds to a vertex in the graph, and edges between vertices are added only if the corresponding cells are adjacent. Edge costs are set based on the values stored in the costmap. More precisely, the cost of an edge from vertex v_i to vertex v_j is based on the value stored in the costmap grid cell corresponding to vertex v_j. The value is not simply copied but is transformed linearly so that each cost falls between two predetermined values called `COST_NEUTRAL` and `COST_OBSTACLE`, which by default are set to 50 and 253, respectively. `NavfnPlanner` computes a navigation function by setting the goal vertex potential value to 0 and then propagating potential values to neighboring cells using[6] either Dijkstra or A* (Dijkstra is the default). This is akin to the single-source shortest path algorithm (see Algorithm 3), where the queue OPEN may be sorted using either the cost-to-come g (Dijkstra) or the estimated cost $f = h + g$ (A*). Once the start cell is reached, a path from the starting cell to the goal cell is retrieved performing gradient descent the computed navigation function (recall the discussion in Section 6.4.) Figure 6.19 shows an example of a path produced by the `NavfnPlanner` to move the robot from the top right location to the bottom left location. Note how the path stays clear of obstacles. This is because the traversal cost for a cell, factored in by both Dijkstra and A* is higher for cells closer to obstacles.

[6]The cost-to-come uses a slightly more complex formula than simple addition, but the underlying principle is the same.

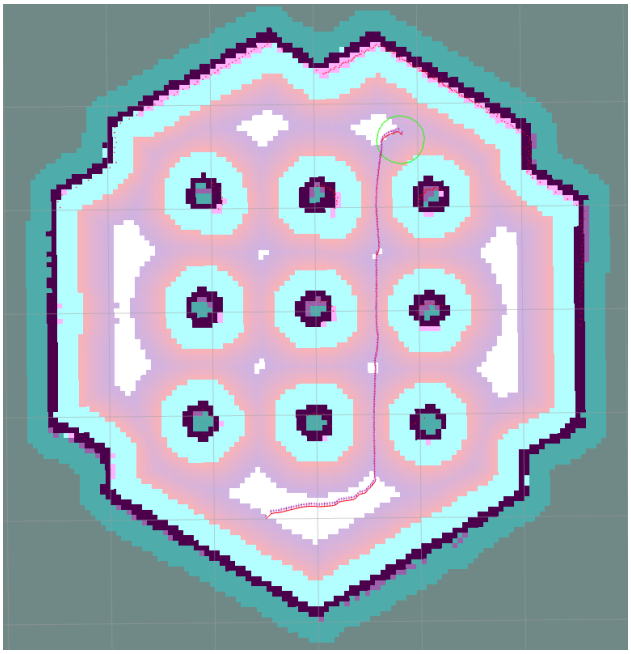


Figure 6.19: Global path (red line) produced by the NavfnPlanner. The start location is at the top, while the goal location is at the bottom. The green circle shows the shape of the robot.

6.8 The Controller Server

The controller server is implemented by the node `controller_server` through the action `follow_path`. The controller server implements the local planner, whose task is to follow the global path generated by the planner server. This path-following process integrates collision avoidance and reactive motion control by querying the local costmap, which is continuously updated using data from onboard sensors. This allows the controller server to avoid obstacles that may have been missing in the global costmap (e.g., humans moving around the environment) and were therefore unknown to the global planner server. It is the local planner that ultimately sends commands to move the robot. The controller server operates at a high frequency, and therefore each velocity command sent to the robot (i.e., messages of type `geometry_msgs::msg::TwistStamped`) is intended to be executed only for a short duration before being replaced by a newly computed one. The interaction between the planner server and the controller server is orchestrated by the BT Navigator server, described in the next subsection. The BT Navigator server first calls the planner server, providing a desired goal pose, and receives back a path, i.e., a message of type `nav::msg::Path`, whose structure is shown below:

```
# An array of poses that represents a Path for a robot to follow.

# Indicates the frame_id of the path.
std_msgs/Header header
  builtin_interfaces/Time stamp
```

```
    int32 sec
    uint32 nanosec
  string frame_id

# Array of poses to follow.
geometry_msgs/PoseStamped[] poses
  std_msgs/Header header
    builtin_interfaces/Time stamp
      int32 sec
      uint32 nanosec
    string frame_id
  Pose pose
    Point position
      float64 x
      float64 y
      float64 z
    Quaternion orientation
      float64 x 0
      float64 y 0
      float64 z 0
      float64 w 1
```

As we can see, a path consists of an sequence of poses expressed in the frame `frame_id`. The path is then passed to the controller server through a call to the `follow_path` action.

Like the planner server, the controller server can be configured through dynamic plugins to implement its reactive path-following function using a variety of algorithms. These are called *controller plugins*, and a complete list is available on the Nav2 website. Here we discuss the DWB controller, which is widely used in practice and is the default in the examples we present. DWB is a configurable implementation of a local planner based on an algorithm known in the literature as the Dynamic Window Approach (DWA). DWB is based on the following idea. First, a set of candidate local trajectories is generated. Different plugins can be used to generate trajectories using various algorithms (these algorithms are referred to as *trajectory generator plugins*.) Next, the set of local trajectories is scored based on how well they track the global trajectory and how safe they are. Then, the best one determines the velocity commands that will be executed. The algorithm responsible for scoring the local trajectories is called a *critic*. Interestingly, multiple critics can be used at the same time, and the overall score is obtained by summing the individual scores. This approach is illustrated in Figure 6.20. As of now, there are two algorithms for trajectory generation:

Standard Trajectory Generator: This algorithm samples the control space of the robot (e.g., rotational and translational velocities). For each sample, it applies the forward kinematic model of the robot for a fixed (and short) amount of time and determines what the resulting trajectory would be. For the forward simulation, it uses a set of equations similar to those presented in Section 4.9 (e.g., Eq. (4.29) or Eq. (4.30) and Eq. (4.31)).

Limited Acceleration Generator: This is similar to the previous one but samples a

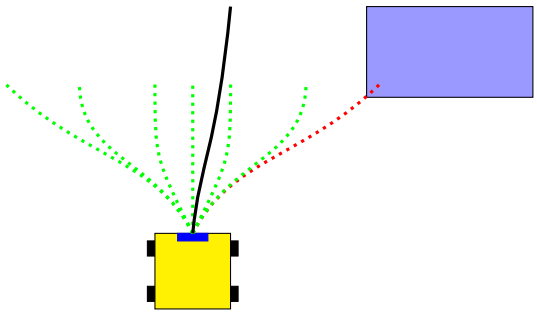


Figure 6.20: Conceptual illustration of how the DWB planner works. Given the global trajectory to follow (solid black line), a set of K tentative local trajectories is generated by a trajectory generation plugin ($K = 7$ in this example). All these trajectories are then evaluated by the critic, and the best one is executed. In this case, the red trajectory would receive a low score because it collides with an obstacle in the local costmap.

smaller space constrained by the acceleration limits of the robot, making it more efficient in terms of planning time.

According to the ROS documentation, the two methods perform similarly in terms of planning success; therefore, the second one is often preferred due to its greater efficiency. DWB comes with ten different critic plugins that can be used in combination, where each critic scores a trajectory and a weighted sum of the individual scores provides a comprehensive evaluation. Critic plugins consider aspects such as (this is a partial list): how well the local plan aligns with the global plan; how much forward progress the local plan achieves; the cost of the path relative to the local costmap; and so on. Figure 6.21 shows the interplay between the global and local plans.

6.9 The BT Navigator Server

Planning in a complex, possibly dynamic environment often requires repeated planning or performing maneuvers to recover from unforeseen circumstances. Therefore, a pipeline more complex than plan/smooth/execute is necessary. Nav2 implements these high level strategies through the BT Navigator Server which is the orchestrator and top-level decision-maker in the Nav2 stack. The BT Navigator Server is implemented by the `bt_navigator` node which provides two main actions: `navigate_to_pose` and `navigate_through_poses`. The BT Navigator server implements its high level functionalities through behavior trees (hence the name), a planning formalism to specify how different operations (e.g., plan, backup, replan, etc.) should be triggered depending on how the execution of the plan evolves. Nav2 comes with multiple behavior trees to implement the various high level functionalities offered and when Nav2 starts it is possible to specify which one should be passed to the BT Navigator server (see Figure 6.17). Additionally, it is also possble to specify new behavior trees in XML.

In the following we discuss the logic embedded in the behavior tree governing the motion to a desired pose in response to a call to the `navigate_to_pose` action. From a high level perspective, this system is composed of two subsystems. The first, called the *navigation subtree*, is in charge of making progress towards the goal pose. It is expected that most of

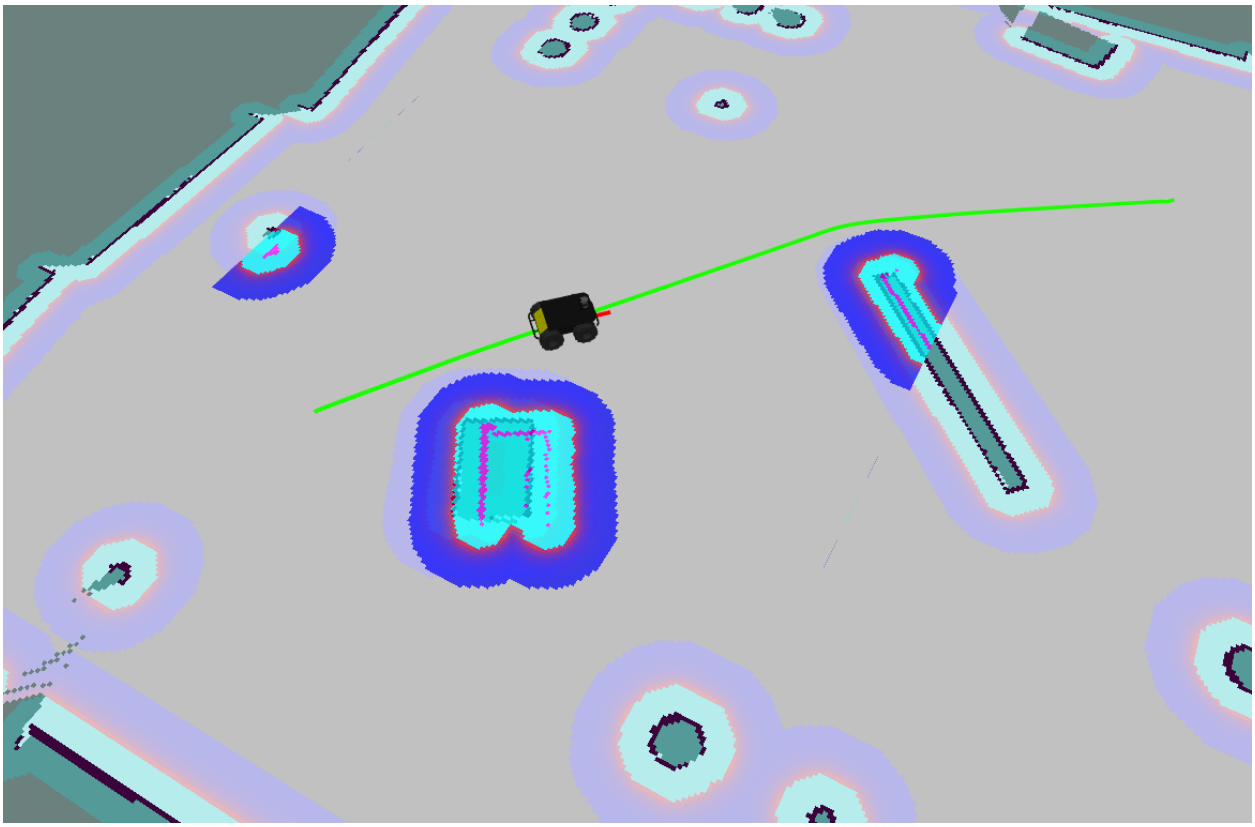


Figure 6.21: Global plan (green) computed by the planner server and local plan (red) computed by the controller server.

the time the robot is executing the tasks defined in the navigation subtree. The second, called the *recovery subtree*, is in charge of recovering from problematic situations that may arise while navigating to the goal pose (e.g., the robot cannot make progress.)

From a high level perspective, the logic is as follows. The robot starts executing the navigation subtree. When it gets stuck, it switches to the recovery subtree. After the recovery subtree terminates its execution, the execution goes back to the navigation subtree. The alternation between the two can continue, i.e., if the robot gets stuck again the recovery subtree is triggered again, although after a fixed (and configurable) number of switches without reaching the goal position the BT Navigator Server terminates the execution, signaling a failure in the attempt to reach the goal pose (in the default settings, the recovery subtree is triggered at most six times.)

The navigation subtree works as follows. The planner server computes a global path to the goal pose and passes the path to the controller server for execution. To make things more robust, the global path is not computed once, but rather recomputed at a fixed frequency (1 Hz by default) to ensure that changes in the environment modeled into the global costmap are taken into consideration. Every time a path is recomputed, it is passed to the controller server (recall that the controller server only tracks a segment of the global path.) If either the planner server or the controller server fails, then a *contextual recovery* is tried, i.e., the planner checks 1) if the goal has changed; or 2) if clearing the associated costmap results in

the server being able to resume the execution. Resuming the execution in this case means going back to the plan/control cycle. If the contextual recovery fails, the robot switches to the recovery subtree.

The goal of the recovery tree is to perform a set of actions that may restart the navigation tree. The recovery tree starts by checking if the goal has changed. If that is the case, then the recovery terminates and the control goes back to the navigation tree. If the goal has not changed, then the navigation tree executes four operations in a round-robin fashion. The four operations are 1) clear the costmaps; 2) spin; 3) wait; 4) backup. Note that the last three operations are actions offered by the behavior server. As soon as one of these actions is successful, control goes back to the navigation tree. However, if the navigation tree is still unable to resume, control comes back to the recovery tree but rather than restarting from the first operation (clear cost maps) it will start from the first one that was not tried in the previous round (e.g., if the previous execution of recovery succeeded through spin but navigation still fails, the next operation will be wait, and so on.)

6.10 Interacting with Nav2

Given the complexity of functionalities offered by Nav2, interacting with it may seem more complicated than it is. In fact, despite the fact that countless functionalities can be reconfigured, in most instances one can simply rely on the default settings and switch to fine tuning only when these do not work properly. In this section, we show how to call Nav2 actions both from the command line and from code. Before starting, it is necessary to make sure that Nav2 is up and running. In particular, it is important for the localization module to be properly initialized with the starting location of the robot. Here we rely on the Gazebo simulations provided with Jazzy and slightly adapted[7] to make the following examples work. To initialize the simulation environment, simply type:

```
ros2 launch gazeboenvs tb4_simulation.launch.py
```

This will open the same simulation environment we saw in Section 4.12 and will properly initialize Nav2. At this point, we can send a goal position to the BT Navigator server via the command line interface, for example:

```
ros2 action send_goal /navigate_to_pose nav2_msgs/action/NavigateToPose
"{pose: {header: {frame_id: map}, pose: {position: {x: 10.0, y: 4.0,
z: 0.0}, orientation: {x: 0.0, y: 0.0, z: 0.707, w: 0.707}}}}" --feedback
```

After having connected to the server, the action commences, printing the action feedback messages to the screen. At the end, the action result is printed to the screen as well (the following shows the last feedback message and the result):

[7] If you start the same simulations coming with the `nav2_bringup` package, the initial position of the robot is not initialized and you will have to do it manually, either from rviz2 or by calling a service. The one provided in the MRTP website comes with the initial position already set.

```
Feedback:
  current_pose:
    header:
      stamp:
        sec: 281
        nanosec: 232000000
      frame_id: map
    pose:
      position:
        x: 9.96245331540172
        y: 4.109822104188528
        z: 0.0
      orientation:
        x: 0.0
        y: 0.0
        z: 0.6133636328253297
        w: 0.7898006418883908
navigation_time:
  sec: 9
  nanosec: 231000000
estimated_time_remaining:
   sec: 3
   nanosec: 290356111
number_of_recoveries: 0
distance_remaining: 0.14307163655757904

Result:
   error_code: 0
error_msg: ''

Goal finished with status: SUCCEEDED
```

Note that even though the target 2D pose was $(x, y) = (10, 4)$, the robot stops at a slightly different position $(9.962, 4.109)$. The same is true for the orientation. This is because Nav2 is run with the default configuration that considers a pose reached with a certain error tolerance. This configurable parameter can be changed to make the action execution more precise.

The other alternative to interface with Nav2 is to write ROS nodes that call Nav2 services and actions. To simplify this task, Nav2 provides a so-called *Simple Commander API* whose goal is to offer "navigation as a library" to Python3 programmers. The `navigation` package in the MRTP GitHub implements a C++ porting of most[8] functionalities of the same library. Listing 6.2 shows how the library can be used to interact with the navigation library. The reader is referred to the MRTP website for a detailed description of the API. To test the

[8]The C++ porting was initially developed based on the Foxy version and replicates the API offered at that time. In Jazzy, a few more functionalities have been added and those at the moment have not been ported yet.

code first start the following Gazebo simulation:

```
ros2 launch nav2_bringup tb3_simulation.launch.py headless:=False
```

and then run the demo:

```
ros2 run navigation testpackage
```

This command starts both Gazebo and RViz. It is instructive to look at the trajectories and maps being plotted in Rviz while the robot goes through the the different steps.

Listing 6.2: Navigation Library Client

```
#include <rclcpp/rclcpp.hpp>
#include <navigation/navigation.hpp>
#include <iostream>
#include <memory>

/* Tests all functionalities in the navigation library */

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

  rclcpp::init(argc,argv);
  Navigator navigator(true); // create node with debug info only

  // First test initialization methods
  geometry_msgs::msg::Pose::SharedPtr init =
                              std::make_shared<geometry_msgs::msg::Pose>();
  init->position.x = -2;
  init->position.y = -0.5;
  init->orientation.w = 1;
  navigator.SetInitialPose(init); // test SetInitialPose
  navigator.WaitUntilNav2Active(); // test WaitUntilActive

  // Now start testing functionalities

  navigator.Spin(); // test Spin action
  while ( ! navigator.IsTaskComplete() ) {  // test IsTaskComplete
    auto feedback_ptr = navigator.GetFeedback(); // test GetFeedback
    auto ptr_spin = std::static_pointer_cast
                      <const nav2_msgs::action::Spin::Feedback>(feedback_ptr);
    std::cout << "Feedback: angular distance traveled: " <<
                  ptr_spin->angular_distance_traveled << std::endl;

  }
  auto result = navigator.GetResult(); // test GetResult
  if ( result == rclcpp_action::ResultCode::SUCCEEDED )
    std::cout << "Spin action succeeded" << std::endl;
  else
    std::cout << "Spin goal was not achieved" << std::endl;

  navigator.Spin(-1.57); // execute Spin action again to cancel it
  int i = 0;
```

```
41    // wait for 3 feedback messages and then cancel
42    while ( ( ! navigator.IsTaskComplete() ) && (i < 3) ) {
43      i++;
44    }
45    navigator.CancelTask(); // test CancelTask
46    result = navigator.GetResult();
47    if ( result == rclcpp_action::ResultCode::CANCELED )
48      std::cout <<"Spin action was canceled as intended" << std::endl;
49    else
50      std::cout <<"Cancel task did not return the expected result." << std::endl;
51
52
53    // test GoToPose
54    geometry_msgs::msg::Pose::SharedPtr
55                          goal_pos = std::make_shared<geometry_msgs::msg::Pose>();
56    goal_pos->position.x = 2;
57    goal_pos->position.y = 1;
58    goal_pos->orientation.w = 1;
59    // move to new pose
60    navigator.GoToPose(goal_pos);
61    while ( ! navigator.IsTaskComplete() ) {
62            auto feedback_ptr = navigator.GetFeedback();
63            auto ptr_gotopose = std::static_pointer_cast
64              <const nav2_msgs::action::NavigateToPose::Feedback>(feedback_ptr);
65            std::cout << "Distance remaining: "
66                      << ptr_gotopose->distance_remaining << std::endl;
67    }
68    result = navigator.GetResult();
69    if ( result == rclcpp_action::ResultCode::SUCCEEDED )
70      std::cout << "GoToPose action succeeded" << std::endl;
71    else
72      std::cout << "GoToPose goal was not achieved" << std::endl;
73
74    // test clearLocalCostMap
75    std::cout << "Clearing local costmap" << std::endl;
76    navigator.ClearLocalCostmap();
77
78    // test clearLocalCostMap
79    std::cout << "Clearing global costmap" << std::endl;
80    navigator.ClearGlobalCostmap();
81
82    // test clearAllCostMaps
83    std::cout << "Clearing all costmaps" << std::endl;
84    navigator.ClearAllCostmaps();
85
86
87    // test Backup
88    navigator.Backup(0.1,0.1); // backup 0.1m at 0.1 m/s
89    while ( ! navigator.IsTaskComplete() ) {
90
91    }
92    result = navigator.GetResult();
93    if ( result == rclcpp_action::ResultCode::SUCCEEDED )
94      std::cout << "Backup action succeeded" << std::endl;
```

```
95    else
96      std::cout << "Backup goal was not achieved" << std::endl;
97
98    // test GetGlobalCostmap
99    std::shared_ptr<nav2_msgs::msg::Costmap> global_costmap =
100                                              navigator.GetGlobalCostmap();
101   std::cout << "Global costmap has dimensions " <<
102                    global_costmap->metadata.size_x << "," <<
103                    global_costmap->metadata.size_y << std::endl;
104
105   // test GetLocalCostmap
106   std::shared_ptr<nav2_msgs::msg::Costmap> local_costmap =
107                                              navigator.GetLocalCostmap();
108   std::cout << "Global costmap has dimensions " <<
109                              local_costmap->metadata.size_x << "," <<
110                              local_costmap->metadata.size_y << std::endl;
111
112   // test GetPath
113   goal_pos = std::make_shared<geometry_msgs::msg::Pose>();
114   goal_pos->position.x = 2;
115   goal_pos->position.y = -1;
116   goal_pos->orientation.w = 1;
117   // move to new pose
118   auto path = navigator.GetPath(goal_pos);
119   while ( ! navigator.IsTaskComplete() ) {
120
121   }
122   result = navigator.GetResult();
123   if ( result == rclcpp_action::ResultCode::SUCCEEDED ) {
124     std::cout << "GetPath action succeeded" << std::endl;
125     std::cout << "Received a path with " << path->poses.size() <<
126                                       " intermediate poses" << std::endl;
127   }
128   else
129     std::cout << "GetPath goal was not achieved" << std::endl;
130
131   // test FollowPath (but only if a path was returned)
132   if ( result == rclcpp_action::ResultCode::SUCCEEDED ) {
133     navigator.FollowPath(path);
134     while ( ! navigator.IsTaskComplete() ) {
135
136     }
137     result = navigator.GetResult();
138     if ( result == rclcpp_action::ResultCode::SUCCEEDED )
139       std::cout << "FollowPath action succeeded" << std::endl;
140     else
141       std::cout << "FollowPath goal was not achieved" << std::endl;
142   }
143
144   // test FollowWaypoints
145   geometry_msgs::msg::PoseStamped p1,p2,p3;
146   p1.pose.position.x = 2;
147   p1.pose.position.y = 1;
148   p2.pose.position.x = -2;
```

```
149   p2.pose.position.y = 1;
150   p3.pose.position.x = -2;
151   p3.pose.position.y = -1;
152   std::vector<geometry_msgs::msg::PoseStamped> pointList;
153   pointList.push_back(p1);
154   pointList.push_back(p2);
155   pointList.push_back(p3);
156   navigator.FollowWaypoints(pointList);
157   while ( ! navigator.IsTaskComplete() ) {
158
159   }
160   result = navigator.GetResult();
161   if ( result == rclcpp_action::ResultCode::SUCCEEDED )
162     std::cout << "FollowWaypoints action succeeded" << std::endl;
163   else
164     std::cout << "FollowWaypoints goal was not achieved" << std::endl;
165
166   // test ChangeMap -- should fail
167   navigator.ChangeMap("bogusmap.png");
168   result = navigator.GetResult();
169   if ( result == rclcpp_action::ResultCode::SUCCEEDED )
170     std::cout << "ChangeMap action succeeded" << std::endl;
171   else
172     std::cout << "ChangeMap goal was not achieved" << std::endl;
173
174   rclcpp::shutdown(); // shutdown ROS
175   return 0;
176 }
```

The library is accessed by creating an instance of the class `Navigator` and then calling its various methods. An important point to consider is that, differently from the previous example, the simulation file provided with the `nav2_bringup` package does not set the initial position of the robot. Therefore, before starting to interact with Nav2, it is necessary to set the robot's initial pose by calling the method `SetInitialPose`. If the pose is instead already set, this is not necessary. Then, a call to `WaitUntilNav2Active` stops the program, if needed, until the various components of Nav2 are up and running. From that point onwards, it is possible to call the various methods as shown in the example. Calls to methods that invoke actions (such as `Spin`, `GoToPose`, etc.) are non-blocking. These can be canceled while being executed by calling the method `CancelTask`, and it is also possible to retrieve feedback messages while they are executed with the method `GetFeedback`. Action results can be accessed using the method `GetResult`. On the contrary, calls to methods that invoke services (such as `ClearLocalCostMap` or `ChangeMap`) are blocking, but are nevertheless quick to execute.

Further Reading

LaValle's book [29] provides a thorough introduction to planning algorithms with a particular emphasis on robotics problems. The classic AI book by Russel and Norvig also covers search planning algorithms [47]. Another book presenting planning algorithms within a robotics

framework is [11]. Finally, [13] is the definitive guide about algorithms, including the graph search methods presented in this chapter. The reader is referred to it for proofs about complexity, correctness, etc. The Nav2 system is extensively discussed in its website, while a more concise introduction can be found in [37]. Additional articles, such as [32, 35, 36], provide details about the various algorithms implemented inside Nav2's components. The Dynamic Window Approach was originally proposed in [23].

Chapter 7

Perception

7.1 Introduction

So far we have not explicitly considered how the robot could acquire information about its surrounding environment or about itself. This has not been necessary for two reasons. First, it was assumed that the robot already had a model of its world (e.g., the planning graph), and it was moreover hypothesized that the outcome of actions was fully predictable. Under these conditions one can develop algorithms that do not rely on perception, or in other words, one can rely on an *open loop* approach. As formerly stated, these hypotheses are either unrealistic (the effect of actions is never fully predictable) or impractical (a robot is often tasked with operating in an environment for which a full model is not available or changes over time). Consequently, the open loop approach is doomed to fail in practice. Indeed, as we discussed in section 6.6, realistic systems to implement navigation in the real world integrate perception while plans are executed.

Consequently, robots are equipped with numerous sensors to extract information about their surrounding environment or about themselves. In Chapter 1 we have introduced the formalism to consider perception in robotics. In particular, we have introduced the state observation equation (Eq. (1.2)) to model this process. The following is the discrete, time-invariant version of the relationship (Eq. (1.8)):

$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{u}_t). \tag{7.1}$$

As anticipated in Chapter 1, in many instances it is possible to assume that the sensor reading is independent of the input given to the robot, and therefore the above relationship can often be simplified as follows (time-invariant version of Eq. (1.9)):

$$\mathbf{z}_t = h(\mathbf{x}_t). \tag{7.2}$$

These equations outline that the sensor reading $\mathbf{z}_t$ at time t is a function of the state at time t (i.e., $\mathbf{x}_t$) and possibly of the input, though most often this is not the case, as in Eq. (7.2). This notation tacitly makes two assumptions. The first is that if the robot is equipped with multiple sensors, and they are all queried at time t, then their collective readings can be *stacked together* in a single vector $\mathbf{z}_t$ with an adequate number of components. If this is not the case, e.g., different sensors are queried at different times, or if one prefers to keep

the various readings separate, then it is necessary to introduce multiple state observation functions $h_1, h_2, \ldots$, to separately consider every sensor. The other assumption is that the function h implicitly models the environment. Indeed, the sensor reading is not only a function of the robot state $\mathbf{x}_t$ (and possibly of the input $\mathbf{u}_t$), but of the environment, too. For example, consider the simple case of a robot equipped with a sensor measuring the distance from the closest obstacle in front of the robot. This can be easily implemented with a laser range finder. The distance returned by the sensor is obviously a function of both where the robot is in the environment and of the environment itself. However, for a static environment, this dependency can be included in h itself and therefore not explicitly included among the arguments of the function h. In some cases considered later on, this dependency may be explicitly noted. This is, for example, the case when the environment changes over time, or when it is convenient to explicitly outline this dependency because of the task we are solving (e.g., building a map of the environment.)

A realistic sensor, however, is not well modeled by Eq. (7.1) nor Eq. (7.2), because they do not account for the unavoidable noise affecting the perception process. More realistic relationships would be

$$\mathbf{z}_t = h(\mathbf{x}_t, \mathbf{u}_t, \psi_t) \qquad \mathbf{z}_t = h(\mathbf{x}_t, \psi_t)$$

where ψ_t is a disturbance at time t. The impact of noise and how to deal with it will be considered in Chapter 8. In this chapter, instead, we focus on how sensors can be queried in ROS. All sensors are subject to measurement errors. The simplest way to think of these errors is through an additive noise component, i.e.,

$$h(\mathbf{x}_t, \mathbf{u}_t, \psi_t) = h_c(\mathbf{x}_t, \mathbf{u}_t) + \psi_t$$

or

$$h(\mathbf{x}_t, \psi_t) = h_c(\mathbf{x}_t) + \psi_t$$

where h_c is the error-free sensor reading ("correct" sensor reading) and ψ_t is the error. These two components cannot be separated, but one may in general have some statistical description for the error ψ_t. In other cases, the error may affect the correct sensor reading in more subtle ways. Note, however, that *the correct sensor reading* h_c is mostly useful for conceptual purposes, but is not really accessible in reality. This model will be heavily used in Chapter 8, where estimation algorithms are discussed.

7.1.1 Dead Reckoning

The equations described above are not strictly applicable for one class of sensors very popular in mobile robotics that go under the name of *dead reckoning*. These sensors produce an estimate of the state (e.g., pose) by integrating the state derivative (e.g., velocity) over time.

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\nu) d\nu. \tag{7.3}$$

As per Eq. (1.1), the state derivative $\dot{\mathbf{x}}$ at time t depends on the input $\mathbf{u}$, and therefore strictly speaking, the value returned at a given time t is a function of the initial value at

time 0 and the sequence of inputs given up to time t (and not just of the state and input at time t as in Eq. (7.1).) The odometry sensor we present in section 7.4.6 is a prime example of this type of sensor. Sensors based on dead reckoning share the following features. First, they require an initial condition, i.e., $\mathbf{x}(0)$ in Eq. (7.3) (oftentimes this is set to a default value, e.g., a vector of 0s). Second, their error tends to grow over time, as is typical of processes relying on integration. Finally, note that Eq. (7.3) describes a numerical method to obtain a state estimate. Therefore, an odometry sensor must incorporate this computation internally. However, such computation in some cases is relatively striaghtforward as Eq. (7.3) is implemented as a discrete time approximation.

7.2 Sensors

Sensor technology continuously evolves. Consequently, any survey aiming at describing the technical details of the state of the art is doomed to quickly become obsolete. In these notes, we abstract away from the underlying technology and focus instead on the type of information returned by sensors, how it can be used to develop more robust robot control systems, and how it is handled in ROS.

Sensors can be characterized in different ways. A first distinction is between *proprioceptive* and *exteroceptive* sensors. Proprioceptive sensors return information about the robot. For example, they may return its velocity, its orientation in space, the battery level, and so on. Exteroceptive sensors return information about the outside of the robot, i.e., about the environment. These include, for example, cameras, range finders (i.e., sensors measuring the distances to obstacles), etc. Most of the time, robots use both types of sensors. The following lists are in no way meant to be exhaustive. Our emphasis is on the type of information they provide, without considering the underlying physical processes generating the information collected.

7.2.1 Proprioceptive sensors

Encoder: an encoder provides information about the angular position and/or velocity of a motor. Typically, there is one encoder per motor (or per wheel). By relating the angular velocity of a motor with the models presented in Section 4.9, it is possible to infer, for example, the velocity of the robot. The value returned by this sensor is typically a vector with two numeric values (e.g., angular position and angular velocity) per motor. Encoder values are often used to implement dead reckoning.

Accelerometer: as the name suggests, this sensor measures the acceleration along one or more axes. An accelerometer returns a vector of values, i.e., the acceleration components along a set of predefined axes (typically one or three).

Gyroscope: a gyroscope measures angular velocity or angular orientation relative to a predefined frame.

Inertial Measurement Unit: under this category we include a family of sensors integrating accelerometers and gyroscopes to return both linear acceleration as well as angular

velocities and/or orientation. These sensors are also sometimes referred to as *inertial navigation units* or using other names. The terminology is not standard. The term *inertial navigation system* is instead used to indicate the system including both the sensor and the computational unit processing the data returned by the sensor. The specific values provided by the sensor depend on the underlying technology, and typically include acceleration along three axes, orientation (e.g., as a quaternion), and angular velocities.

Battery sensor: this sensor provides information about the battery system powering the robot. Typically, it provides a vector of numeric values, like the current voltage, the current charge, etc.

7.2.2 Exteroceptive sensors

Sonar: a sonar returns the distance to the closest obstacle in a given direction (see Figure 7.1). Sonars typically have a relatively short maximum range (e.g., three meters), and a relatively wide opening, so they can be effectively used for safety purposes (obstacle detection and avoidance), but are not ideal for localizing features in the environment or building maps. Robots often mount various sonars, and the sensor system returns a vector with one value per sonar. Sonars are rather noisy and imprecise but extremely inexpensive.

Laser rangefinder: a laser rangefinder (also called LiDAR for Light Detection and Ranging) measures distances along a set of directions. Those most commonly found in robotics are so-called *planar rangefinders*, meaning that they return distances along a plane (see Figure 7.1). Typically, they have an angular field of view of 180 or 270 degrees, and a spacing of 1 or half a degree between two adjacent readings. The sensor returns a vector with one value per reading. As for the sonar, a laser rangefinder has a maximum range, and when a returned value equals the maximum range, it typically means that the closest obstacle in that direction is farther than the maximum range. Laser rangefinders are much more accurate than sonars, but also more expensive.

3D rangefinder: this type of sensor can be thought of as the three-dimensional version of a laser rangefinder, i.e., its readings are not arranged on a plane, but rather span a solid angle. The value returned is typically represented by a point cloud, i.e., a set of point coordinates referred to a reference frame placed on the sensor. Three-dimensional LiDAR sensors are extensively used in autonomous vehicles. Technology for these sensors is continuously improving, and there exist very different approaches to generate a point cloud. While the information produced by these sensor is very rich, the main drawback is their cost and the fact that they produce a high volume of information that may require quite some computational power for timely processing.

GPS: a GPS sensor returns the latitude and longitude of the robot (and typically also the altitude) by measuring the signals received from a set of satellites. Although the GPS returns the pose of the robot, it is considered an exteroceptive sensor because it determines this information by measuring quantities external to the robot.

Camera (2D): robots often use one or more cameras. These return a matrix of values encoding the information about every pixel (e.g., RGB values, etc.).

Depth Camera: a depth camera (also called RGB-D camera) produces measurements for the depth (distance) of the pixels in the image, thus producing a 3D point cloud (see Figure 7.2). Fueled by sustained user demand for a variety of applications, technological advances in this domain are at an all-time high.

Contact sensor: contact sensors are used to determine if the robot is touching something. In mobile robots, they are often placed around the robot. Each sensor returns a binary value (contact/no contact). These sensors are also called *bumper sensors*.

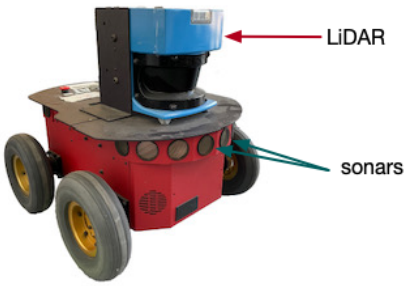


Figure 7.1: A Pioneer P3AT robot with a range finder and sonars.

Figure 7.2: An Intel RealSense depth camera.

7.3 Sensors in ROS

ROS provides support for sensors in two ways. First, it provides the package `sensor_msgs` defining various messages useful to exchange information returned by sensors. In addition, it provides nodes to interact with various hardware devices. In essence, these nodes implement *drivers*, i.e., they exchange information with the underlying hardware often using proprietary or device-specific protocols, and they provide data to the rest of the application by publishing suitable messages on assigned topics. From a design point of view, the node subscribing to these topics is independent and decoupled from the pipeline upstream, i.e., it can ignore the process that led to the messages appearing on the topic it subscribes to. This is consistent with ROS overall architecture based on loosely coupled nodes and makes code much more portable and easier to maintain. Table 7.1 shows the associations between messages defined

Message Type	Sensor
`BatteryState`	Status of battery (e.g., voltage and other info)
`Image`	Camera image
`Imu`	Inertial Measurement Unit
`JointState`	Status of set of joints (position, velocity, torque)
`LaserScan`	Planar range finder (e.g., laser)
`NavSatFix`	GPS
`PointCloud`	Collection of 3D points returned by a 3D rangefinder (deprecated)
`PointCloud2`	Collection of 3D points returned by a 3D scanner (2nd version)
`Range`	Single range reading (e.g., sonar)

Table 7.1: Association between messages in the `sensor_msgs` package and sensors.

in the `sensor_msgs` package and sensors (only a subset of messages is displayed). A notable sensor missing in the table is the 2D camera. The package `sensor_msgs` includes messages of type `sensor_msg::msg::Image` and `sensor_msg::msg::ImageCompressed` to transmit and receive images and compressed images (essentially, a matrix of data). However, given that computer vision is a discipline on its own, in the following we will not investigate this type of sensor and the associated ROS infrastructure.

7.4 Sensor messages of common use

In this section we discuss some of the messages most commonly used in ROS to exchange and process sensor data.

7.4.1 Laser Scan

Messages of type `sensor_msgs::LaserScan` are used to transmit data produced by a planar laser range finder, such as the Sick LMS 200 mounted on the robot shown in Figure 7.1. A laser range finder is a sensor that returns an array of readings with distances along a set of predefined directions lying on a plane. Each message contains the data from a single scan and has the following structure:

```
std_msgs/Header header # timestamp in the header is the acquisition time of
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id
      # the first ray in the scan.
      #
      # in frame frame_id, angles are measured around
      # the positive Z axis (counterclockwise, if Z is up)
      # with zero angle being forward along the x axis
```

```
float32 angle_min            # start angle of the scan [rad]
float32 angle_max            # end angle of the scan [rad]
float32 angle_increment      # angular distance between measurements [rad]

float32 time_increment  # time between measurements [seconds] - if your scanner
                        # is moving, this will be used in interpolating position
                        # of 3d points
float32 scan_time            # time between scans [seconds]

float32 range_min            # minimum range value [m]
float32 range_max            # maximum range value [m]

float32[] ranges # range data [m]
                 # (Note: values < range_min or > range_max should be discarded)
float32[] intensities   # intensity data [device-specific units].  If your
                        # device does not provide intensities, please leave
                        # the array empty.
```

The message includes the familiar `std_msgs::msg::Header` field, which contains information about when the scan was taken and the frame with respect to which the results are reported. Typically, the field `frame_id` in `header` is set to `base_laser`, and the relationship between `base_laser` and `base_link` can be determined through the `tf_static` topic. In the coordinate frame specified by `frame_id`, angles are measured around the positive z-axis, with zero angle pointing forward along the x-axis. Note that ranges smaller than `range_min` or larger than `range_max` are considered spurious and should be discarded. The three fields `angle_min`, `angle_max`, and `angle_increment` are expressed in radians and define the geometry of the sensor (see Figure 7.3 for the meaning of these parameters). The fields `range_min` and `range_max` define the valid interval for range measurements; readings outside these bounds should be discarded. The field `ranges` is an array containing the range readings. While the number of elements can be explicitly retrieved using the `size` attribute, it can also be inferred from `angle_min`, `angle_max`, and `angle_increment`. The field `intensities` contains an *intensity* value for each range reading; its interpretation is sensor-specific, and it will not be used in the following. Similarly, the fields `time_increment` and `scan_time` provide timing information which, in simple applications, can be ignored.[1] Figure 7.4 shows a rendering in RViz of the data returned by a range scanner with a 360 degrees field of view. The pose of the robot is indicated by the frame on the lower right, and the robot is operating in the environment shown in figure 6.18.

Listing 3.15 shows the code of a node reading from a rangefinder from the topic `scan`, extracting the closest value, and returning it on a different topic called `closest`. This is shown in listing 7.1 (as for example 3.15. To test this code you need to run

```
ros2 launch gazeboenvs tb4_simulation.launch.py use_rviz:=True
```

to generate the data. To run the node run

[1] What are these times used for? If the robot moves while performing a scan, the sensor itself moves as well. These fields can be used to compensate for the sensor motion during the scan.

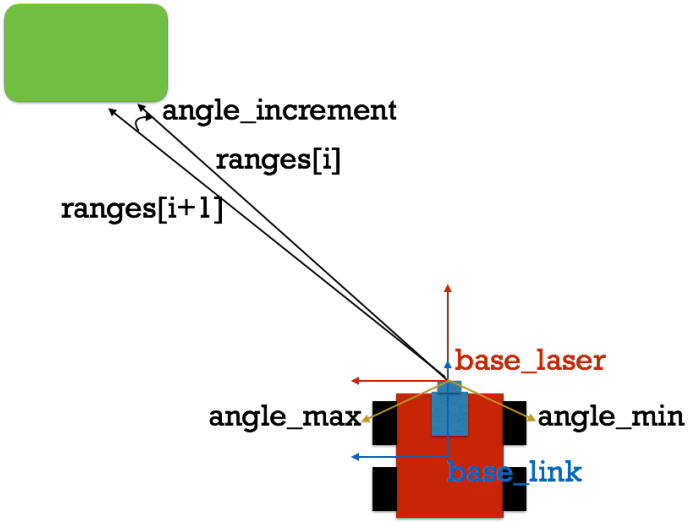


Figure 7.3: Interpretation of the parameters included in a `sensor_msgs::msg::LaserScan` message.

```
ros2 run examples pubsubstl
```

and to visualize the output run

```
ros2 topic echo closest
```

Listing 7.1: Publisher/subscriber node

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/laser_scan.hpp> // to receive laser scans
#include <std_msgs/msg/float32.hpp> // to send floating point numbers
#include <algorithm> // for stl min algorithm

class FindClosest : public rclcpp::Node {
public:
  FindClosest():Node("pubsubstl") {
    pubf = this->create_publisher<std_msgs::msg::Float32>("closest",1000);
    sub = this->create_subscription<sensor_msgs::msg::LaserScan>
          ("scan",10,std::bind(&FindClosest::processScan,
                               this,std::placeholders::_1));
  }

private:
  void processScan(const sensor_msgs::msg::LaserScan::SharedPtr msg) {
    std::vector<float>::const_iterator minval =
      min(msg->ranges.begin(),msg->ranges.end());
    std_msgs::msg::Float32 msg_to_send;
    msg_to_send.data = *minval;
    pubf->publish(msg_to_send); // publish result
  }
  rclcpp::Publisher<std_msgs::msg::Float32>::SharedPtr pubf;
```

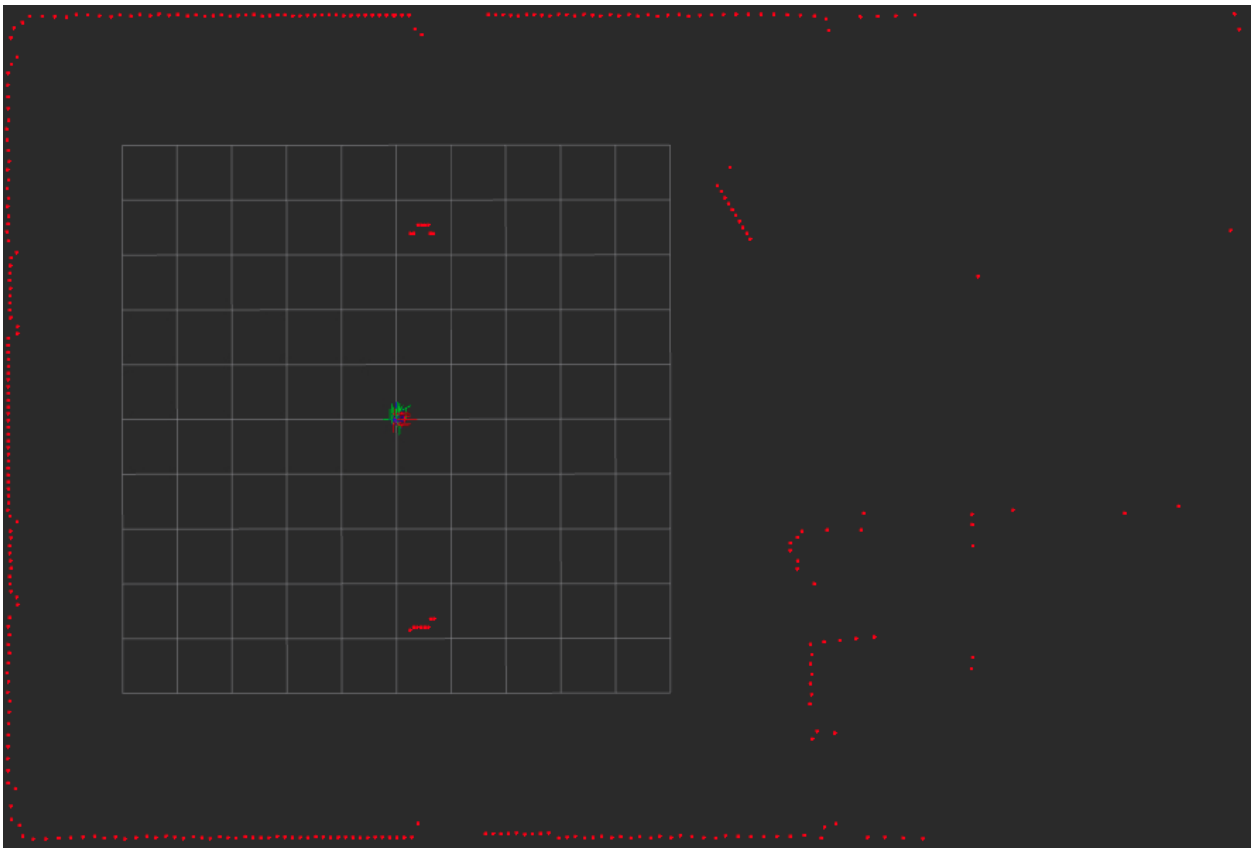


Figure 7.4: RViz visualization of data returned by a range scanner with a 360 degree field of view. Red dots indicate the readings returned by the LiDAR. The robot is located in the middle where the frames are displayed. This fgure is best interpreted looking at the Gazebo simulation, too.

```
24    rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr sub;
25  };
26
27  int main(int argc,char ** argv) {
28
29    rclcpp::init(argc,argv);
30    rclcpp::spin(std::make_shared<FindClosest>());  // create and spin
31    rclcpp::shutdown();
32    return 0;
33  }
```

7.4.2 Single Range

Messages of type `sensor_msgs::Range` are used to acquire data from single-reading range sensors, such as ultrasound or infrared. These sensors also return distance measurements and operate on the principle of emitting a signal and detecting its reflection (if any). Robots often mount several of these sensors, like the P3AT shown in Figure 7.1, which has sixteen sonars (eight on the front and eight on the back). Each sensor, however, returns its own

reading. The structure of the message is as follows:

```
std_msgs/Header header # timestamp in the header is the time the ranger
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id
                        # returned the distance reading

# Radiation type enums
# If you want a value added to this list, send an email to the ros-users list
uint8 ULTRASOUND=0
uint8 INFRARED=1

uint8 radiation_type    # the type of radiation used by the sensor
                        # (sound, IR, etc) [enum]

float32 field_of_view   # the size of the arc that the distance reading is
                        # valid for [rad]
                        # the object causing the range reading may have
                        # been anywhere within -field_of_view/2 and
                        # field_of_view/2 at the measured range.
                        # 0 angle corresponds to the x-axis of the sensor.

float32 min_range       # minimum range value [m]
float32 max_range       # maximum range value [m]
                        # Fixed distance rangers require min_range==max_range

float32 range           # range data [m]
                        # (Note: values < range_min or > range_max should be
                        # discarded)
                        # Fixed distance rangers only output -Inf or +Inf.
                        # -Inf represents a detection within fixed distance.
                        # (Detection too close to the sensor to quantify)
                        # +Inf represents no detection within the fixed distance
                        # (Object out of range)

float32 variance        # variance of the range sensor
                        # 0 is interpreted as variance unknown
```

Besides the usual `header` field, there is the `radiation_type`, which indicates the type of signal used to determine the reading—either ultrasound (value 0) or infrared (value 1). The message accommodates both range sensors that return distances within a range, as well as fixed-distance rangers that determine whether an obstacle is closer than a fixed threshold (hence the name).

For range sensors returning distances, similar to the `LaserScan` message, there are two fields, `min_range` and `max_range`, which specify the range outside of which readings should

be discarded. Additionally, `range` indicates the distance returned, and `variance` provides the associated variance.

For fixed-distance rangers, only two readings are returned: `-Inf` if the obstacle is closer than the specified range, and `+Inf` if it is farther.

In both cases, `field_of_view` specifies the sensor's opening angle in radians (see Figure 7.5). In essence, the distance reading corresponds to an obstacle that reflected the signal and is located somewhere within the field of view.

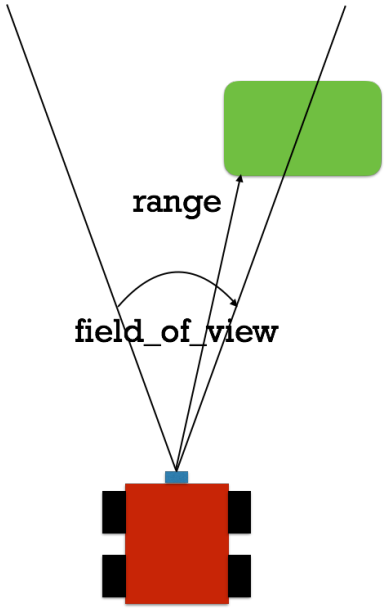


Figure 7.5: Meaning of the parameters included in a `sensor_msgs::Range` message.

7.4.3 Inertial Measurement Unit

Messages of type `sensor_msgs::msg::Imu` are used to exchange information produced by inertial measurement units (IMUs). Since there is considerable variability among these devices, the message is structured to include all possible estimates. However, when working with a specific device, some quantities may not be computed; in such cases, the corresponding field includes a flag indicating the lack of data. IMUs combine multiple accelerometers and gyroscopes to determine linear accelerations and angular velocities. Moreover, through integration, orientation can also be determined. The structure of the message is as follows:

```
std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id

geometry_msgs/Quaternion orientation
  float64 x 0
```

```
  float64 y 0
  float64 z 0
  float64 w 1
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
  float64 x
  float64 y
  float64 z
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
  float64 x
  float64 y
  float64 z
float64[9] linear_acceleration_covariance # Row major x, y z
```

In addition to the `header`, the message includes the orientation (represented as a quaternion), the angular velocity, and the linear acceleration. Each of these fields is accompanied by an associated 3×3 covariance matrix. If a specific device does not return one of the fields—or a component within a field (e.g., it does not return the linear acceleration along the **z** axis)—the first element in the corresponding covariance matrix is set to -1 to indicate that the data is missing. For example, if the IMU does not provide the linear acceleration, then `linear_acceleration_covariance[0]` is set to -1. Therefore, before using any of the fields, it is always necessary to first check the covariance matrix to determine whether the data is actually provided by the sensor. Note that the covariance matrices are filled in row-major order, and accelerations are expressed in m/s^2.

7.4.4 GPS

Robots operating outdoors often rely on GPS modules to determine their location through the *Global Positioning System* or *Global Navigation Satellite System* (these terms can be considered synonyms in the following). The message `sensor_msgs::msg::NavSatFix` is used to transmit this type of data, and its structure is as shown by the command `ros2 interface show`.

```
std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id

# Satellite fix status information.
NavSatStatus status
#
  int8 STATUS_UNKNOWN = -2        #
```

```
  int8 STATUS_NO_FIX =  -1        #
  int8 STATUS_FIX =      0        #
  int8 STATUS_SBAS_FIX = 1        #
  int8 STATUS_GBAS_FIX = 2        #
  int8 status -2 #
  uint16 SERVICE_UNKNOWN = 0  #
  uint16 SERVICE_GPS =     1
  uint16 SERVICE_GLONASS = 2
  uint16 SERVICE_COMPASS = 4      #
  uint16 SERVICE_GALILEO = 8
  uint16 service

# Latitude [degrees]. Positive is north of equator; negative is south.
float64 latitude

# Longitude [degrees]. Positive is east of prime meridian; negative is west.
float64 longitude

# Altitude [m]. Positive is above the WGS 84 ellipsoid
# (quiet NaN if no altitude is available).
float64 altitude

# Position covariance [m^2] defined relative to a tangential plane
# through the reported position. The components are East, North, and
# Up (ENU), in row-major order.
#
# Beware: this coordinate system exhibits singularities at the poles.
float64[9] position_covariance

# If the covariance of the fix is known, fill it in completely. If the
# GPS receiver provides the variance of each measurement, put them
# along the diagonal. If only Dilution of Precision is available,
# estimate an approximate covariance from that.

uint8 COVARIANCE_TYPE_UNKNOWN = 0
uint8 COVARIANCE_TYPE_APPROXIMATED = 1
uint8 COVARIANCE_TYPE_DIAGONAL_KNOWN = 2
uint8 COVARIANCE_TYPE_KNOWN = 3

uint8 position_covariance_type
```

As usual, the `header` defines the time when the reading was taken. `frame_id` is the position of the antenna receiver, and not of the vehicle (assuming this is `base_link`.) The meaning of the other fields is self-explanatory. An important field is `status`, which indicates whether the fix is valid and also specifies which satellite system was used. The `status` field should always be checked before using the fix in a message of type `NavSatFix`.

A technology that can greatly enhance GPS accuracy is Real-Time Kinematic GPS, usu-

ally referred to as RTK GPS. RTK GPS increases localization accuracy by using a *base station* placed at a known fixed location and broadcasting suitable correction signals. With this setup, a robot or a vehicle (referred to as *rovers* in this context) can increase the accuracy of their localization by integrating not only the signals it receives from the satellites, but also the corrections it receives from the base station. RTK GPS can provide position with centimeter-level precision and is increasingly used in outdoor robotic applications. Importantly, from the ROS standpoint, RTK GPS and GPS use the same interface, i.e., they both provide their data through `sensor_msgs::msg::NavSatFix` messages.

7.4.5 Point Clouds

ROS provides two different messages to represent point clouds produced by sensors depth cameras, 3D LiDARs, and similar devices. The older is `sensor_msgs::msg::PointCloud`, whose structure is as follows:

```
# Time of sensor data acquisition, coordinate frame ID.
std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id

# Array of 3d points. Each Point32 should be interpreted as a 3d point
# in the frame given in the header.
geometry_msgs/Point32[] points
  #
  #
  float32 x
  float32 y
  float32 z

# Each channel should have the same number of elements as points array,
# and the data in each channel should correspond 1:1 with each point.
# Channel names in common practice are listed in ChannelFloat32.msg.
ChannelFloat32[] channels
#
  string name
  float32[] values
```

After the header with the timestamp and the frame name, there is an array of three-dimensional points with x, y, z coordinates, referenced to the frame `frame_id` specified in the `Header`. The `channels` array contains as many elements as `points`, and carries additional data for each point, such as distance, RGB color, and so on. The type of information is specified in the `name` field, and the corresponding data is stored in the `values` field. Allowable values for the `name` field can be found in the specification of the `ChannelFloat32`

message.

Messages of type `sensor_msgs::msg::PointCloud` are deprecated. Users are advised to use the newer `sensor_msgs::msg::PointCloud2`, which has the following structure.

```
# Time of sensor data acquisition, and the coordinate frame ID
# (for 3d points).
std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields
    uint8 INT8    = 1
    uint8 UINT8   = 2
    uint8 INT16   = 3
    uint8 UINT16  = 4
    uint8 INT32   = 5
    uint8 UINT32  = 6
    uint8 FLOAT32 = 7
    uint8 FLOAT64 = 8
    string name      #
    uint32 offset    #
    uint8  datatype  #
    uint32 count     #

bool    is_bigendian # Is this data bigendian?
uint32  point_step   # Length of a point in bytes
uint32  row_step     # Length of a row in bytes
uint8[] data         # Actual point data, size is (row_step*height)

bool is_dense        # True if there are no invalid points
```

The main difference is that this type of messages allows to represent n-dimensional data. Another difference with `sensor_msgs::msg::PointCloud` is that different data types can be used, as evidenced by the constants defined in the `fields` field. Messages of this type are useful to interface ROS with the Point Cloud Library (PCL), an external, independent library for 2D/3D point processing. Support for the integration between ROS and PCL is offered by the `perception_pcl` stack.

7.4.6 Odometry

Most mobile robots can use information about their own motion to estimate where they are. In the simplest possible scenario, it is possible to project, or integrate forward, the commands given to the motors to estimate the robot's position (for example, using equations like (4.30) for a differential drive platform). This approach was introduced in Section 7.1.1 and is known as *odometry*. More generally, odometry estimates both pose and velocity and integrates data from multiple sources (e.g., IMUs, encoders, etc.). Odometry data is exchanged through messages of type `nav_msgs::msg::Odometry`. The structure is as follows:

```
# Includes the frame id of the pose parent.
std_msgs/Header header
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id

# Frame id the pose points to. The twist is in this coordinate frame.
string child_frame_id

# Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/PoseWithCovariance pose
  Pose pose
    Point position
      float64 x
      float64 y
      float64 z
    Quaternion orientation
      float64 x 0
      float64 y 0
      float64 z 0
      float64 w 1
  float64[36] covariance

# Estimated linear and angular velocity relative to child_frame_id.
geometry_msgs/TwistWithCovariance twist
  Twist twist
    Vector3  linear
      float64 x
      float64 y
      float64 z
    Vector3  angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

The `frame_id` field in the header refers to the `odom` frame in ROS. The interpretation of the remaining sections is straightforward. The only notable aspect is that both `position` and `twist` include a 6×6 covariance matrix to represent uncertainty. Indeed, the main consideration regarding the pose provided by odometry is that it drifts over time due to the accumulation of errors, as is typical of methods based on integration. Therefore, relying exclusively on this information to determine the robot's pose is prone to failure. However, odometry information (pose and velocities) is often a key component of sensor fusion algorithms that combine data from multiple sources. This will be discussed in subsequent chapters.

7.4.7 Images

For completeness, we provide the structure of the messages `sensor_msgs::msg::Image` and `sensor_msgs::msg::CompressedImage`, even though they are not listed in Table 7.1. The structure of `sensor_msgs::msg::Image` is as follows, where we see that, unsurprisingly, an image is represented as a matrix of pixels.

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image

std_msgs/Header header # Header timestamp should be acquisition time of image
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id
# Header frame_id should be optical frame of camera
# origin of frame should be optical center of cameara
# +x should point to the right in the image
# +y should point down in the image
# +z should point into to plane of the image
# If the frame_id here and the frame_id of the CameraInfo
# message associated with the image conflict
# the behavior is undefined

uint32 height                # image height, that is, number of rows
uint32 width                 # image width, that is, number of columns

# The legal values for encoding are in file
# include/sensor_msgs/image_encodings.hpp
# If you want to standardize a new string format, join
# ros-users@lists.ros.org and send an email proposing a new encoding.

string encoding       # Encoding of pixels -- channel meaning, ordering, size
# taken from the list of strings in include/sensor_msgs/image_encodings.hpp

uint8 is_bigendian    # is this data bigendian?
```

```
uint32 step           # Full row length in bytes
uint8[] data          # actual matrix data, size is (step * rows)
```

After the `header`, we find the dimensions of the image (`height` and `width`). The `encoding` string specifies the schema used to encode the image and can be any value defined in the file `include/sensor_msgs/image_encodings.h`. Examples of defined encodings include `rgb8` for images in which each pixel is in RGB format with 8 bits per channel, `rgba16` for an RGB-alpha encoding with 16 bits per channel, and so on. The field `step` defines the length of a row in bytes, and `data` contains the image data. Note that `data` is an array of unsigned integers containing `step` $\times$ `height` elements. Finally, the field `is_bigendian` specifies how to interpret values stored in more than one byte.

In real-world applications, one typically relies on additional packages to acquire, transmit, and process images. These include drivers for specific devices, such as `usb_cam` to interface with V4L USB cameras or `cv_camera` to capture images using OpenCV, and `cv_bridge` to conver them into OpenCV.image format. Image processing in ROS applications typically relies on external libraries such as OpenCV.

Finally, the structure of messages of type `sensor_msgs::msg::CompressedImage` is the following.

```
# This message contains a compressed image.

std_msgs/Header header # Header timestamp should be acquisition time of image
  builtin_interfaces/Time stamp
    int32 sec
    uint32 nanosec
  string frame_id
# Header frame_id should be optical frame of camera
# origin of frame should be optical center of cameara
# +x should point to the right in the image
# +y should point down in the image
# +z should point into to plane of the image

string format               # Specifies the format of the data


uint8[] data                # Compressed image buffer
```

The interpretation should by now be straightforward. The field `format` specifies how the image is compressed. The reader is referred to the full message specification for a list of available formats.

Further reading

Various robotics textbooks like [50] and [27] include chapters describing the physical principles supporting various sensing technologies. A rich, albeit meanwhile dated, survey on sensor technologies is given in [9]. For details about GPS sensors and algorithms, see [38].

Chapter 8

Estimation and Filtering

8.1 Introduction

In this chapter, we introduce the mathematical foundations of the estimation problem, with particular attention to applications from the robotics domain. Some of the algorithms we will present are readily available in ROS. However, for the sake of explanation, this chapter focuses on theory only, while details related to ROS implementations will be given in Chapter 9.

Informally speaking, *estimation* is the process of determining a quantity from data obtained through measurements that are incomplete and/or inaccurate. The data used to answer an estimation question must therefore carry direct or indirect information about the quantity being estimated. Determining the height of a building by measuring the length of its shadow is an example of an estimation problem. *Filtering* is a specific type of estimation problem in which the quantity being estimated is the state of a dynamical system, and thus changes over time. In the example introduced above, the length of the shadow (i.e., the measurement) changes during the day, but the height of the building does not, so it is not a filtering problem. A typical example of a filtering problem in robotics is determining the pose of a robot that moves in its environment. In filtering problems, the state is estimated by considering uncertain measurements, as well as the uncertain dynamics governing the state evolution over a given sequence of inputs. The following diagram is a slight variation of what we saw in Chapter 1. Note that, in this case, the sensor reading at time t is just a function of the state at time t, i.e., $\mathbf{x}_t$.

$$\begin{array}{ccccccccc} \mathbf{x_0} & \xrightarrow{\mathbf{u}_1} & \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1) & \xrightarrow{\mathbf{u}_2} & \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2) & \xrightarrow{\mathbf{u}_3} & \mathbf{x}_3 = f(\mathbf{x}_2, \mathbf{u}_3) & \longrightarrow & \dots \\ & & \downarrow & & \downarrow & & \downarrow & & \\ & & \mathbf{z}_1 = h(\mathbf{x}_1) & & \mathbf{z}_2 = h(\mathbf{x}_2) & & \mathbf{z}_3 = h(\mathbf{x}_3) & & \end{array}$$

The objective of the filtering problem is to estimate $\mathbf{x}_t$ given the sequence of inputs $\mathbf{u}_1, \dots, \mathbf{u}_t$ and the sequence of sensor measurements $\mathbf{z}_1, \dots, \mathbf{z}_t$. A fundamental complication in this formulation is that both the state transition equation and the sensor measurements are noisy. Accordingly, the following diagram is the one we should consider when thinking about realistic estimation and filtering problems.

$$\mathbf{x_0} \xrightarrow{\mathbf{u}_1} \mathbf{x}_1 = f(\mathbf{x}_0, \mathbf{u}_1, \zeta_1) \xrightarrow{\mathbf{u}_2} \mathbf{x}_2 = f(\mathbf{x}_1, \mathbf{u}_2, \zeta_1) \xrightarrow{\mathbf{u}_3} \mathbf{x}_3 = f(\mathbf{x}_2, \mathbf{u}_3, \zeta_3) \longrightarrow \dots$$
$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$
$$\mathbf{z}_1 = h(\mathbf{x}_1, \psi_1) \qquad \mathbf{z}_2 = h(\mathbf{x}_2, \psi_2) \qquad \mathbf{z}_3 = h(\mathbf{x}_3, \psi_3)$$

In this case, the disturbances ζ_i and ψ_i are not accessible to the estimation algorithm, and therefore, even if one assumes f and h are deterministic, it would still be impossible to exactly predict the next state or the sensor reading, even when $\mathbf{x}$ and $\mathbf{u}$ are known. An alternative way to model these noise sources is to assume that f and h are probability distributions rather than deterministic functions. That is, $\mathbf{x}_i$ is treated as a sample drawn from a distribution f that depends on $\mathbf{x}_{i-1}$ and $\mathbf{u}_i$, and $\mathbf{z}_i$ is a sample drawn from a distribution h that depends on $\mathbf{x}_i$.

Estimation and filtering algorithms are central to mobile robotics applications, and they are used to solve fundamental problems such as localization and mapping. This is illustrated in the following motivating example.

Example 8.1. *A differential drive robot moving on the plane is equipped with a sensor that returns the distance to a landmark whose position ${}^A\mathbf{p}_l$, expressed in the world frame A, is known (see Figure 8.1). The motion of the robot is uncertain, and the sensor returns noisy readings.*

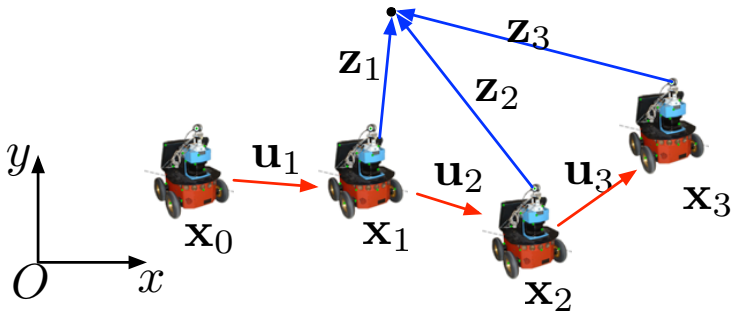


Figure 8.1: Localization as a filtering problem.

The initial pose of the robot, ${}^A\mathbf{p}_0^r$, expressed in frame A, is also known. Three commands, $\mathbf{u}_1$, $\mathbf{u}_2$, and $\mathbf{u}_3$, are given to the robot in sequence, and after each command the sensor is queried, returning the values $\mathbf{z}_1$, $\mathbf{z}_2$, and $\mathbf{z}_3$. That is, $\mathbf{z}_1$, $\mathbf{z}_2$, and $\mathbf{z}_3$ are the measured distances from ${}^A\mathbf{p}_l$ when the robot is at positions $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$, respectively. A filtering algorithm can then be used to estimate ${}^A\mathbf{p}_1^r$, ${}^A\mathbf{p}_2^r$, and ${}^A\mathbf{p}_3^r$, i.e., the pose of the robot as it evolves over time. The estimate is based on the sequence of inputs given to the robot and the sensor readings collected. As is common in the estimation literature, the quantity being estimated is denoted as $\mathbf{x}_i$, with the index i indicating its dependence on time.

The previous example sketches an instance of the *localization* problem, a fundamental challenge in mobile robotics. Note that, because of the uncertainty in pose evolution, the pose for $t > 0$ becomes a random quantity even if the initial pose is known with no uncertainty. Therefore, even if one knows $\mathbf{x}_0$ exactly, after the robot executes the first input $\mathbf{u}_1$ (assumed

to be known without uncertainty), the next state $\mathbf{x}_1$ is still a random quantity due to noise. In this case, the state is a three-dimensional random vector representing the pose x, y, ϑ. Estimating $\mathbf{x}_t$ through filtering, therefore, means computing a probability density function[1] for $\mathbf{x}_t$. This simple example also introduces another important concept: *recursive filtering*. In most practical scenarios, one does not collect all sensor readings and all inputs up to time t and then estimate $\mathbf{x}_t$. Instead, sensor readings and inputs are processed incrementally as they are received. Starting from $\mathbf{x}_0$, $\mathbf{u}_1$, and $\mathbf{z}_1$, the filtering algorithm computes an estimate for $\mathbf{x}_1$. After $\mathbf{u}_2$ and $\mathbf{z}_2$ are received, an estimate for $\mathbf{x}_2$ is computed, and so on. However, for efficiency reasons, this is not done by reprocessing from scratch all data received up to that point. Instead, the estimate for $\mathbf{x}_2$ is computed using $\mathbf{u}_2$, $\mathbf{z}_2$, and the estimate for $\mathbf{x}_1$ computed at the previous time step. This approach is called recursive filtering. The advantage is that the time complexity to compute the new estimate is independent of the length of the history.

Remark 8.1. *In computer science, recursive algorithms are often considered conceptually elegant but inefficient (e.g., the recursive algorithm to compute Fibonacci numbers or the factorial). However, in estimation theory, the opposite is true:* recursive *estimation algorithms are widely used precisely because they are very efficient.*

Two problems related to filtering are *smoothing* and *prediction*. In *smoothing*, the estimate for $\mathbf{x}_t$ is computed considering measurements and inputs that occurred after t, i.e., $\mathbf{u}_{t+1}, \ldots, \mathbf{u}_{t+k}$ and $\mathbf{z}_{t+1}, \ldots, \mathbf{z}_{t+k}$ for some $k > 1$. In most robotics applications, we are interested in solutions to the filtering problem because it is solved online; however, in some cases smoothing is applicable. For example, one may try to recover *a posteriori* the trajectory followed by a robot after the mission terminates and all inputs and sensor readings are available. Another relevant problem in some applications is *prediction*, where we estimate $\mathbf{x}_t$ starting from $\mathbf{u}_1, \ldots, \mathbf{u}_{t-k}$ and $\mathbf{z}_1, \ldots, \mathbf{z}_{t-k}$ for some $k > 1$. In the following, for the sake of brevity, we focus exclusively on filtering and refer the reader to the references at the end of the chapter for smoothing and prediction algorithms.

The filtering algorithms we consider in this chapter belong to the class of Bayesian filters and build upon the theory of Bayesian inference. In the next section, we provide a very brief introduction to the mathematical foundations of this problem, and the reader is referred to the references at the end of the chapter for more details.

8.2 Math Preliminaries

In this section we provide a concise introduction to the general estimation problem. Albeit elegant, this formulation finds few practical applications because it relies on the ability to compute various integrals in closed form. Therefore, in the following sections we study special instances of this general framework that are computationally more efficient and can be integrated into tightly timed robot control software. The topics discussed next assume basic knowledge of probability theory. The reader is referred to Appendix A for a quick review of the relevant material.

[1] In the following, for simplicity, we consider density functions, but similar concepts apply when the result of the filtering process is a probability mass distribution.

There exist two approaches to estimation. In the *classical* approach, the quantity being estimated is a constant (scalar or vector) that is unknown. In the *Bayesian* approach, the quantity being estimated is assumed to be a random variable from a set of possible values, say Θ. Accordingly, a prior over Θ is assumed, and the objective is to produce a posterior (recall the terminology used when introducing Bayes' rule). In the following, we exclusively deal with *Bayesian* estimation techniques[2]. Accordingly, a prior about the quantity being estimated will always be assumed to be available.

Let $\mathbf{x}_t$ be the state of a dynamic system we want to estimate, where the subscript t indicates the time step. As per our Bayesian standpoint, let $\mathbf{x}_0$ be the prior about the state (i.e., the knowledge we have at time 0 before the filtering process starts), let $\mathbf{u}_1, \ldots, \mathbf{u}_t$ be the inputs applied up to time t, and $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_t$ be the sensor readings collected up to time t. It is customary to indicate estimated quantities using a *hat* symbol, and the estimate for the state at time t is then indicated as $\hat{\mathbf{x}}_t$. So, $\mathbf{x}$ is the unknown state we want to estimate, and $\hat{\mathbf{x}}$ is its estimate as produced by an estimation algorithm. The estimation problem can be then be cast as follows

$$\hat{\mathbf{x}}_t = g(\mathbf{x}_0, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_t, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_t) \tag{8.1}$$

where function g is called *estimator*, and the algorithm computing g is called *estimation algorithm*. Recall that in a Bayesian framework $\mathbf{x}_t$ is a random variable and its estimate is a probability density function.

Therefore, the estimate $\hat{\mathbf{x}}_t$ provides the following posterior probability

$$p(\mathbf{x}_t = x \mid \mathbf{x}_0, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_t, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_t)$$

for each possible value x that the random variable $\mathbf{x}_t$ may assume. The above expression is often written in shorter form as

$$p(x_t \mid \mathbf{x}_0, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_t, \mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_t).$$

In some textbooks, the posterior distribution is also called *belief*. Evidently, there exist infinite estimators, and we will look for an estimator that is *good* (or ideally even *optimal*) according to a chosen criterion. An ideal criterion would be

$$\mathbb{E}_{\mathbf{x}_t, \hat{\mathbf{x}}_t}[||\hat{\mathbf{x}}_t - \mathbf{x}_t||^2] \tag{8.2}$$

but we do not know $\mathbf{x}_t$, so this is not viable in practice. However, we know the noisy measurement $\boldsymbol{z}_t$ and through the function h we can predict what the measurement would be if our estimation were correct, i.e., $\hat{\mathbf{z}} = h(\hat{\mathbf{x}})$. Hence the following criterion could instead be used:

$$\mathbb{E}_{\mathbf{z}_t, \hat{\mathbf{z}}_t}[||\hat{\mathbf{z}}_t - \mathbf{z}_t||^2] \tag{8.3}$$

where the term $\boldsymbol{z} - \hat{\boldsymbol{z}}$ is called *measurement residual*.

[2]There is a long-standing debate about the relative merits and demerits of either approach. This is beyond the scope of these notes

Eq. (8.1) reveals the importance of recursive estimation techniques. As more and more data is collected over time, the estimation algorithm would depend on more and more inputs, and this could slow it down if they have to be all taken into account at every step. For this reason, the estimation algorithms we will consider in this chapter are *recursive estimators*, i.e., they have the following form:

$$\hat{\mathbf{x}}_t = g(\hat{\mathbf{x}}_{t-1}, \mathbf{u}_t, \mathbf{z}_t). \tag{8.4}$$

The estimate at time t is just a function of the estimate at time $t-1$ (i.e., $\hat{\mathbf{x}}_{t-1}$) and the last input $\mathbf{u}_t$ and observation $\mathbf{z}_t$. The estimate at time $t-1$ in turn depends on $\hat{\mathbf{x}}_{t-2}$, $\mathbf{u}_{t-1}$ and $\mathbf{z}_{t-1}$. These relationships can be unfolded all the way to $\hat{\mathbf{x}}_1$ that can be computed from $\mathbf{x}_0, \mathbf{u}_1$ and $\mathbf{z}_1$. It can therefore be seen how the prior $\mathbf{x}_0$ is essential to bootstrap the Bayesian estimation process.

In the beginning of this section we noted that in the Bayesian setting the state $\mathbf{x}_t$ is a random variable. Therefore the estimation algorithm g will return a description of the random variable $\hat{\mathbf{x}}_t$, e.g., its PDF. If we make the assumption that $\hat{\mathbf{x}}$ is a random variable following a known canonical distribution (e.g., a multivariate Gaussian), the estimation algorithm will determine the parameters defining the distribution (e.g., mean and covariance matrix). This will be the case for the Kalman filter algorithm presented later on. This class of problems is called *parametric estimation*, as the problem is to determine the parameters of a PDF modeling the quantity being estimated. However, it is also possible to assume that the random variable being estimated has a PDF that is not described in closed form by a finite set of parameters, but can rather be represented or approximated using some other description. In such case the estimation algorithm will return a representation for this non-parametric description. This is for example the case of the particle filter. Both cases will be considered in the remainder of this chapter.

In Bayesian estimation, besides the availability of the prior $\mathbf{x}_0$ we assume also the availability of the following two models.

1. State Transition Model:

$$p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) \tag{8.5}$$

   This is the probability distribution for the next state $\mathbf{x}_t$ conditioned on the current state $\mathbf{x}_{t-1}$ and the current input $\mathbf{u}_t$. This model captures the uncertainty the robot experiences as it moves around.

2. Sensor Model:

$$p(\mathbf{z}_t|\mathbf{x}_t) \tag{8.6}$$

   This is the probability distribution for the sensor reading at time t given the current state. This model captures the uncertainty in the perceptual process, and is influenced by both the state of the robot (explicitly) and the environment in which the robot operates (implicitly). Concrete examples will be given later on.

Note that while in principle it may be complex to determine these two probability distributions, in the following we will simply assume they are available and they can be computed in constant time when needed (for a fixed choice of their arguments).

8.3 Discrete Estimation Algorithms

As robots operate in the physical world, many of the quantities of interest are naturally modeled as real numbers, e.g., position, orientation, etc. In some instances it will be possible to formulate estimation algorithms operating on continuous variables (e.g., the Kalman Filter.) However, in many other circumstances, approaches based on discretization will instead be embraced. While these pose some practical challenges, they are nevertheless useful to study because they rely on the same concepts that will also be utilized to implement widely used algorithms.

Utilizing a discretized approach means that even though the underlying quantity may be continuous, its domain is subdivided into a finite partition, and an approximate discretized representation is used. Stated differently, the quantity being estimated is a discrete random variable. For example, when considering the pose of a robot moving inside a room, we may subdivide the environment into a finite number of equally sized square cells and approximate the (x, y) position with the centers of the cells. Similarly, we can discretize the orientation ϑ as well, thus getting a three-dimensional grid with two dimensions for the position discretized position and one dimension for the discretized orientation. Using this discretized approach, the pose of the robot is therefore a discrete random variable defined by a PMF, and an estimation algorithm will therefore determine a PMF over the same alphabet.

In the following we introduce some notation that will make the description of estimation algorithms shorter. Let us assume that X is a discrete random variable that can assume n different values, i.e., its alphabet is $x_1, \ldots, x_n$. The random variable X is described by a PMF over $x_1, \ldots, x_n$, i.e.,

$$P(X = x_1) \qquad P(X = x_2) \qquad \ldots \qquad P(X = x_n)$$

Since this is a PMF, these values must all be non-negative and add up to 1, i.e., $\sum_{i=1}^{n} P(X = x_i) = 1$. In the following, for brevity we will sometimes write $P(x_1)$ as a shorthand for $P(X = x_1)$.

Example 8.2. *Assume that a robot is moving in an empty rectangular room and we are interested in estimating where it is (location only, without considering the orientation). To this end, we can subdivide the room into n equally sized rectangles that will be labeled $x_1, \ldots, x_n$. Let X be the random variable for the location of the robot (e.g., the projection on the xy plane of its center of mass.) The estimation algorithm will determine a PMF over $x_1, \ldots, x_n$ that should be interpreted as follows: $P(X = x_1)$ is the estimated probability that the robot is inside the rectangle labeled x_1, $P(X = x_2)$ is the estimated probability that the robot is inside the rectangle labeled x_2, and so on.*

All algorithms we will consider in the following are based on some variation of Bayes rule (see A.4):

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{8.7}$$

Before embarking in a full study of estimation algorithms using Bayes rule, let us state a fact that will have great practical importance. Assume X is a discrete random variable and assume we want to estimate its PMF over its finite alphabet $x_1, \ldots, x_n$. To this end,

we can query a sensor that returns a random variable Z carrying some information about X. This would implment a form of indirect estimation and is the prevailing approach. Let z be the value returned from the sensor. Let us furthermore assume that we know $P(z|x_i)$ for all x_is and all possible values of z. Assume moreover, that we have a prior for each of the x_i values, i.e., we know $P(X = x_i)$. Under these conditions, an estimation algorithm could therefore update this prior integrating the sensor reading, thus producing a posterior PMF using Bayes rule. We can start applying Bayes rule (8.7) to each of the values x_i, thus obtaining:

$$P(X = x_1|Z = z) = \frac{P(Z = z|X = x_1)P(X = x_1)}{P(Z = z)}$$

$$P(X = x_2|Z = z) = \frac{P(Z = z|X = x_2)P(X = x_2)}{P(Z = z)}$$

$$\dots$$

$$P(X = x_n|Z = z) = \frac{P(Z = z|X = x_n)P(X = x_n)}{P(Z = z)}$$

In the above expressions the numerators are all known because of the assumptions we made, but we did not assume to know $P(Z = z)$. However, this is unimportant because we can indirectly retrieve this value and complete our calculations. The reason is that $P(Z = z)$ appears in the denominator of all expressions, and moreover, since we are dealing with a PMF, the sum of all these probabilities on the right sides must add up to one. Therefore:

$$\begin{aligned}\sum_{i=1}^{n} P(X = x_i|Z = z) &= \sum_{i=1}^{n} \frac{P(Z = z|X = x_i)P(X = x_i)}{P(Z = z)} \\ &= \frac{1}{P(Z = z)} \sum_{i=1}^{n} P(Z = z|X = x_i)P(X = x_i) = 1\end{aligned}$$

Hence:

$$P(Z = z) = \sum_{i=1}^{n} P(Z = z|X = x_i)P(X = x_i)$$

and this can be calculated since all elements on the right-hand side are known. This result is not surprising since this expression is given by the total probability theorem (see theorem A.2.) Consequently, when applying Bayes rule to compute posterior PMFs we usually do not even indicate the denominator $\frac{1}{P(Z=z)}$ because it can be computed by normalization. In fact, the above expressions are usually written as

$$P(x_i|z) = \eta P(z|x_i)P(x_i)$$

where we used the shortened notation $P(x_i)$ for $P(X = x_i)$ and wrote η for $\frac{1}{P(z)}$.

Remark 8.2. *To apply Bayes rule one needs $P(B) > 0$. In our former discussion we have not explicitly assumed that $P(z) > 0$ because by stating that the sensor returned the reading z it follows that $P(Z = z)$ can not be zero.*

Example 8.3. *A robot is equipped with a camera and an image processing algorithm (classifier) to determine whether there is a given object in an image captured by the camera (say a human). Let X be the binary random variable for the event* "there is a human in the picture"*, i.e. $X = 1$ indicates that the event is true. The classifier algorithm returns a binary random variable Z to indicate whether it has detected a human in the image or not. The classification algorithm is not perfect, i.e., it may incur in missed detections (fails to detect a human in the picture when there is one), or false positives (indicates that there is a human in the picture when there is none). The performance of the sensor is the following:*

$$P[Z = 1|X = 0] = 0.2 \qquad P[Z = 0|X = 0] = 0.8$$

$$P[Z = 1|X = 1] = 0.7 \qquad P[Z = 0|X = 1] = 0.3.$$

Note that the performance of the sensor is asymmetric, i.e., its false positive probability (0.2) is different from its missed detection probability (0.3). Assuming that the prior of the event $X = 1$ is $P[X = 1] = 0.2$ and that the sensor returns $Z = 1$, determine the posterior for the random variable X conditioned on the sensor reading.

This question can be answered by a straightforward application of Bayes rule. First note that $P[X = 0] = 1 - P[X = 1] = 0.8$. Next, we compute the posterior for both the events $X = 1$ and $X = 0$ applying Bayes rule A.4:

$$P[X = 1|Z = 1] = \frac{P[Z = 1|X = 1]P[X = 1]}{P[Z = 1]} = \eta P[Z = 1|X = 1]P[X = 1]$$

$$P[X = 0|Z = 1] = \frac{P[Z = 1|X = 0]P[X = 0]}{P[Z = 1]} = \eta P[Z = 1|X = 0]P[X = 0]$$

Plugging in the values we get $P[X = 1|Z = 1] = \eta 0.7 \cdot 0.2$ and $P[X = 0|Z = 1] = \eta 0.2 \cdot 0.8$. To determine the normalizer η we exploit our previous observation that the posteriors must add up to one, i.e., $P[X = 1|Z = 1]+P[X = 0|Z = 1] = 1$. Therefore $\eta 0.2 \cdot 0.8+\eta 0.7 \cdot 0.2 = 1$, i.e.,

$$\eta = \frac{1}{0.2 \cdot 0.8 + 0.7 \cdot 0.2} = \frac{1}{0.3}$$

Plugging η into the previous formulas we therefore obtain the posteriors

$$P[X = 1|Z = 1] = \eta P[Z = 1|X = 1]P[X = 1] = \frac{1}{0.3} 0.7 \cdot 0.2 = \frac{0.14}{0.3} \approx 0.46\ldots$$

$$P[X = 0|Z = 1] = \eta P[Z = 1|X = 0]P[X = 0] = \frac{1}{0.3} 0.2 \cdot 0.8 = \frac{0.16}{0.3} \approx 0.53\ldots.$$

Before concluding, note that we derived η through normalization, but this is equivalent to compute $P[Z = 1]$ using the total probability theorem (see A.2).

A term pervasively used in robotics estimation is *belief*. Belief is the posterior probability conditioned on inputs and sensor readings (compare with Eq. (8.1)). Therefore, we could say that the estimation algorithms we will present later aim at computing a belief over a certain state space.

Before embarking on the full discussion of the Bayes filter to compute posterior distributions conditioned on both inputs and observations, let us first consider two simpler cases. First, consider the case where we want to compute the posterior distribution of a random variable X based on two sensor readings (rather than one). We indicate with Z_1 the random variable for the first sensor reading and Z_2 the random variable for the second sensor reading. Let the two sensor readings be z_1 and z_2, and as in the previous discussion, assume a prior distribution for X is given, and we moreover know $P(z_1|x_i)$ and $P(z_2|x_i)$ for each x_i. Finally, let us assume that the two variables Z_1 and Z_2 are independent from each other. More importantly, from now onwards we assume that the probability of making an observation at time t depends exclusively on the state at time t. The posterior for X can be computed using Eq. (**??**), i.e., Bayes rule with background knowledge:

$$P(x_i|z_1, z_2) = \frac{P(z_1|x_i, z_2)P(x_i|z_2)}{P(z_1|z_2)} = \frac{P(z_1|x_i)P(x_i|z_2)}{P(z_1)}.$$

The final expression was simplified because we assumed Z_1 and Z_2 are independent, and therefore $P(z_1|z_2) = P(z_1)$ and $P(z_1|x, z_2) = P(z_1|x)$. This expression can be further expanded using Bayes rule again to compute $P(x_i|z_2)$, i.e.,

$$P(x_i|z_1, z_2) = \frac{P(z_1|x_i)}{P(z_1)}\frac{P(x_i|z_2)}{1} = \frac{P(z_1|x_i)}{P(z_1)}\frac{P(z_2|x_i)P(x_i)}{P(z_2)} = \frac{P(z_1|x_i)}{P(z_1)}\frac{P(z_2|x_i)}{P(z_2)}P(x_i),$$

where in the last expression we evidenced $P(x_i)$ to show that the posterior integrating the observations can be seen as a scaling of the prior $P(x_i)$.

The reader should note that we have initially assumed that z_2 was the variable providing background knowledge in the first iteration of Bayes rule. This was an arbitrary decision and we could have started with z_1 as well. However, as it is immediate to verify, the final result would not change. This is consistent with the intuition that if one collects multiple independent measurements under the same conditions[3], the final result of the estimation process should not depend on the order in which these measurements are processed. It is at this point trivial to show that this observation extends to an arbitrary number of sensor readings, i.e.,

$$\begin{aligned} P(x_i|z_1, z_2, \ldots, z_n) &= \frac{P(z_1|x_i)}{P(z_1)}\frac{P(z_2|x_i)}{P(z_2)}\cdots\frac{P(z_n|x_i)}{P(z_n)}P(x_i) \\ &= \left(\prod_{j=1}^{n}\frac{P(z_j|x_i)}{P(z_j)}\right)P(x_i) = \eta\left(\prod_{j=1}^{n}P(z_j|x_i)\right)P(x_i) \end{aligned} \tag{8.8}$$

In this last expression we have again evidenced the η factor that can be determined through normalization without knowing the various $P(z_i)$. $P(z|x)$ is the aforementioned *sensor model*, and it is the probabilistic version of the state observation equation we introduced in Eq. (1.9), i.e., $\mathbf{z}_t = h(\mathbf{x}_t)$. The following example illustrates these concepts in a very simple setting.

[3]Technically speaking, this means that the observations are independent identically distributed random variables.

Example 8.4. *Assume a robot is located in an environment that has been discretized into 5 regions (cells). In each region there is a unique detectable landmark and the robot is equipped with a sensor to detect and identify the landmark located in its region. The sensor may return $l_1, l_2, \ldots, l_5$, to indicate which landmark it has detected (see Figure 8.2.) For example l_1 means the red landmark has been detected, l_2 means the blue landmark has been detected, and so on. Each landmark is detectable only when the robot is located in the corresponding cell.*

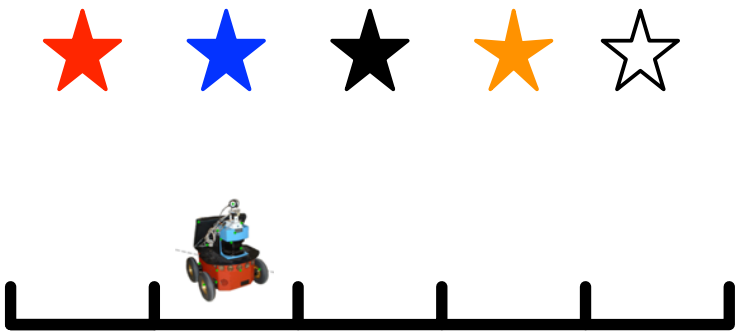


Figure 8.2: Each of the five possible locations is marked with a distinctive landmark (colored star). By querying an appropriate sensor capable of detecting the landmark in the region, the robot can estimate its location, i.e., a PMF for the discrete random variable representing the area in which it is located.

By querying the sensor n times, the objective is to estimate in which region the robot is located, i.e., we want to estimate $P(x_i|z_1, \ldots, z_n)$. To make things simpler, it is assumed that the robot does not move between the sensor readings. In an ideal world, if the robot had a perfect sensor not subject to any error, with a single sensor reading it could determine with certainty its location. However, as in the real world, we assume that the sensor is not perfect, and therefore multiple readings will be made to estimate the location of the robot. To answer this question using the previous formulas, we need to also specify the prior over $x_1, \ldots, x_5$ and $P(z = l_j|x_i)$ for all possible l_j and x_i. Let us assume that the prior is $P(x_1) = 0.6$ and $P(x_2) = P(x_3) = P(x_4) = P(x_5) = 0.1$. Note that the prior is not uniform but may rather model domain-specific a priori knowledge about where the robot could be. For the sensor model, let us assume the following: $P(l_i|x_i) = 0.6$ and $P(l_j|x_i) = 0.1$ for $j \neq i$. That is to say that if the robot is in region x_i, the sensor returns l_i with probability 0.6 (correct reading) and any of the other landmarks with equal probability (wrong reading). Note that, obviously, for a given x_i we have $\sum_{j=1}^{5} P(z = l_j|x_i) = 1$. Assume the robot queries the sensor $n = 5$ times obtaining l_3, l_1, l_3, l_3, l_3. What is the posterior distribution for each of the x_i? To answer this question we need to apply Eq. (8.8) to each of the x_is:

$$P(x_i|l_3, l_1, l_3, l_3, l_3) = \eta P(l_3|x_i)^4 P(l_1|x_i) P(x_i).$$

Substituting, the following posterior is obtained (note that numbers are rounded):

$$P(x_1|l_3, l_1, l_3, l_3, l_3) = 0.027$$
$$P(x_2|l_3, l_1, l_3, l_3, l_3) = 0.007$$
$$P(x_3|l_3, l_1, l_3, l_3, l_3) = 0.9708$$
$$P(x_4|l_3, l_1, l_3, l_3, l_3) = 0.007$$
$$P(x_5|l_3, l_1, l_3, l_3, l_3) = 0.007$$

Figure 8.3 shows how the posterior varies as the sensor readings are integrated. The top left histogram shows the prior, while each successive diagram shows the posterior after each of the 5 sensor readings are integrated. The last diagram shows the final results.

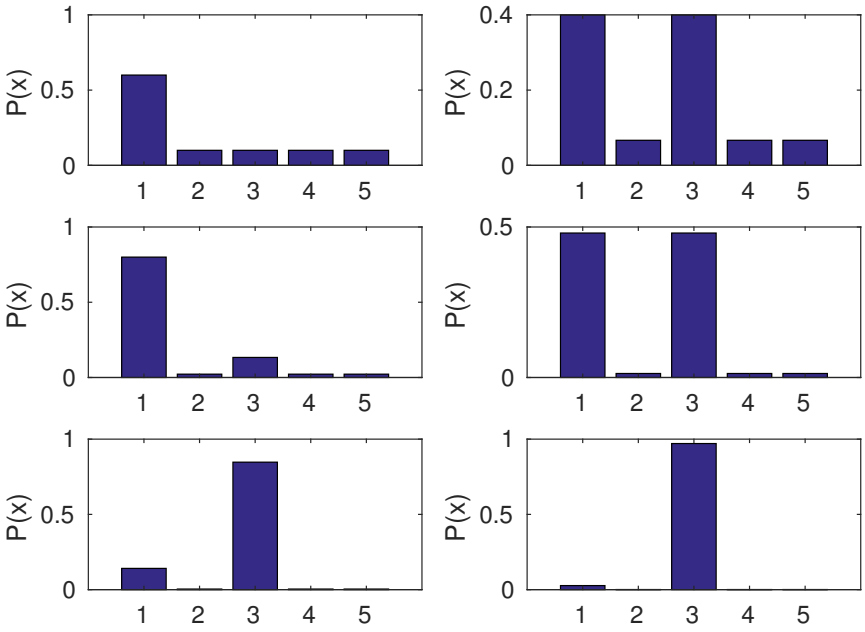


Figure 8.3: Prior (top left) and posterior (remaining diagrams) after the various sensor readings are sequentially considered.

Note that the diagrams show the posterior assuming that the sensor readings are integrated in the given order, i.e., first l_3, then l_1, then l_3, and so on. However, consistently with our previous discussion, the final result does not depend on the order, as per Eq. (8.8).

Similarly, we can consider the effect of using the probabilistic motion model introduced in Eq. (8.5). As formerly stated, this model gives the probability that the next state is $\mathbf{x}_t$ given that the current state is $\mathbf{x}_{t-1}$ and the applied input $\mathbf{u}_t$. With the motion model, we can determine the posterior for the state conditioned on a given input. The key is in using Eq. (8.5) in the total probability theorem, where the y_is are a partition of the state space (i.e., y_is are the elements of the alphabet of the random variable X being estimated.)

$$P(x|u) = \sum_{i=1}^{n} P(x|y_i, u)P(y_i). \tag{8.9}$$

In this case, too, we must assume the availability of the prior PMF $P(y)$.

Example 8.5. *Consider a robot moving along a single dimension, e.g., along a corridor with finite length. Let us assume the set of possible poses along the corridor has been discretized into a finite set of $n = 30$ equally sized segments (cells). A prior non-uniform distribution about the initial pose of the robot is available and displayed in figure 8.4 (that is the $P(y_i)$ in Eq.* (8.9)*).*

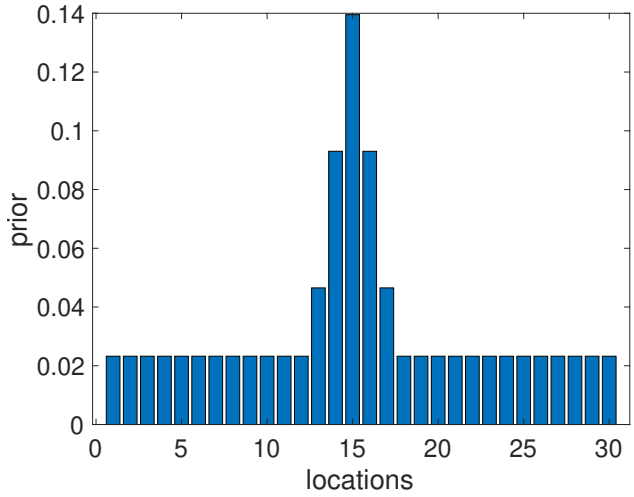


Figure 8.4: Prior distribution for the robot pose along the corridor.

The robot then executes one action u aimed at moving one step to the right. When the robot is at either end of the corridor (rightmost or leftmost cell) it cannot move past the boundary. The motion model in Eq. (8.9) *is as follows. For the first 28 cells (i.e., all except the last two at the very right) the model is*

$$P(x'|x, u) = \begin{cases} 0.1 & \text{if } x' = x \\ 0.7 & \text{if } x' = x + 1 \\ 0.2 & \text{if } x' = x + 2 \end{cases}$$

This means the robot moves one cell to the right with probability 0.7, moves two cells to the right (overshoots) with probability 0.2, and remains in the same cell (skids in place) with probability 0.1. For cell number 29, the robot either moves one cell to the right with probability 0.9 or remains there with probability 0.1 (in this case the robot cannot overshoot because of the corridor dead end.) Finally, if the robot is in cell number 30 (rightmost cell), it remains there with probability 1 because it cannot move to the right at all. The motion model, combined with the prior given in figure 8.4 and Eq. (8.9) *allows us to compute a posterior for each pose. The result is shown in figure 8.5.*

Some interesting observations can be made. First, as expected, the peak of the distribution shifted to the right, consistently with the motion executed. The distribution of the peak,

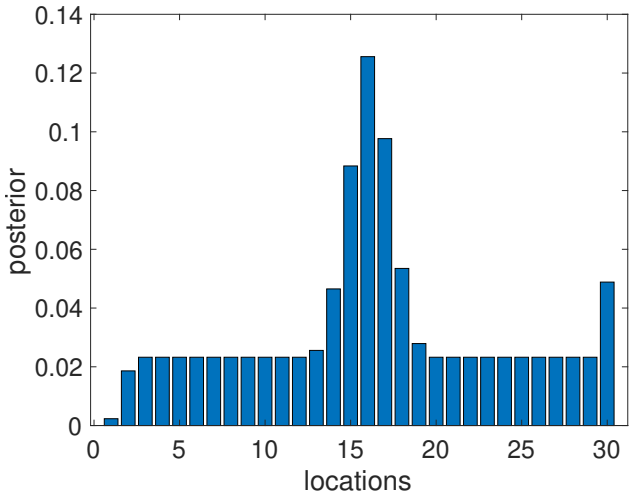


Figure 8.5: Posterior distribution for the robot pose along the corridor.

moreover, is more spread and the peak is lower. This is consistent with our former statement that executing actions will in general increase uncertainty. At the left, we observe that the probability of the first cell is significantly lower. This is because if the robot were there, there would be only a 0.1 probability of remaining there (so the updated value is 0.1 times the prior.) At the other end, we notice a new peak. This is consistent with the fact that the robot cannot go past the last position. So the probability increases because the robot will end up in the last cell as a result of three possible events: 1) starting from cell 30 and remaining there; 2) starting from cell 29 and moving one cell to the right (expected move); 3) starting from cell 28 and moving two cells to the right (overshooting).

As a final comment, one could observe that if the robot were to repeatedly execute the action u (move one cell to the right), the peak at the end would keep growing until eventually reaching a value close to 1, while all other values would tend to 0. This actually makes sense because if the corridor has a finite length and the robot keeps attempting to move to the right, it will with very high probability eventually reach the dead end to the right. However, the probability will not be 1 (why?). The reader is encouraged to analytically verify why this is the case.

As for the previous case, we can consider the estimate based on two inputs rather than one, i.e., $P(x|u_1, u_2)$. In this case, assume that the two inputs are given in sequence, i.e., u_1 is applied first, and u_2 is applied next. Intuitively, if one thinks about composing maneuvers with a vehicle, inverting the order of maneuvers leads to different final results, and therefore the order matters.

Example 8.6. *Consider a differential drive robot and two inputs* $\mathbf{u}_1$ *and* $\mathbf{u}_2$*. For the time being, let us assume that the robot is not affected by any noise.* $\mathbf{u}_1$ *has the rotational velocity equal to* $\frac{\pi}{2}$ *rad/s and the translational velocity equal to* 0*, whereas* $\mathbf{u}_2$ *has the translational velocity equal to* 1 *m/s and the rotational velocity equal to 0. Both inputs are executed for 1 s. If we execute first* $\mathbf{u}_1$ *and then* $\mathbf{u}_2$*, the robot first rotates in place and then translates. If we swap the order, the robot first translates and then rotates in place. At the end, the robot will have the same orientation, but its* (x, y) *location will be different. Therefore, the final*

pose depends on the order in which the inputs are executed.

This intuition can be easily analytically confirmed, and it is of course true also when disturbances affect the robot.

Remark 8.3. *The previous derivations rely on the availability of the sensor model $p(z|x)$ and of the motion model $P(x|y,u)$. Where do we get them from? Various approaches are possible. For sensors, error characterizations are often provided by the manufacturer. Alternatively, both for sensor and motion models, it is possible to derive them from data. Following this approach, one collects multiple sensor readings or executes certain inputs multiple times and then derives a model fitting the data. The model could be "black-box," a computational module that maps inputs to outputs with certain guarantees that it fits the data it was trained on. The method to perform this mapping is unrelated to the underlying physical processes that were used to collect the data, but simply models the correspondences. Alternatively, the model could be grounded in the physical phenomena explaining the sensor process or the actuation mechanism. Irrespective of that, in the following we simply assume that both the sensor and motion models are given and are part of the input to the problem.*

8.4 Recursive Discrete Bayes Filter

Building upon the material presented in the previous section, we can now introduce the discrete Bayes filter. The discrete Bayes filter offers an attractive framework for efficiently estimating non-parametric distributions. However, it should be noted that other estimation methods presented later on, such as the Kalman filter, are special cases of the general Bayes filter method.

Starting from Eq. (8.1), we want to determine a posterior (i.e., the *belief*) for the state at time t conditioned on all sensor readings and inputs given up to that time, as well as knowledge about the initial state x_0. This can be written as follows:

$$P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_t).$$

Before starting our derivation, we reiterate a couple of assumptions made in the previous sections. Sensor readings depend on the current state only, but are independent of each other, the inputs, and the previous states. In addition, the motion model is Markovian; that is, the PMF for the next state depends exclusively on the current state and the current input. Let us call $\hat{x}_t$ the estimate at time t, conditioned on all available data up to that time, i.e.,

$$\hat{x}_t = P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_t). \tag{8.10}$$

At this point, we can rework this expression using the insights derived in the previous section.

$$\hat{x}_t = P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_t) \tag{8.11}$$

$$= \frac{P(z_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1}, x_t)P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1})}{P(z_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1})} \tag{8.12}$$

$$= \frac{P(z_t|x_t)P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1})}{P(z_t)} \tag{8.13}$$

$$= \frac{P(z_t|x_t)\sum_{x_{t-1}} P(x_t|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1}, x_{t-1})P(x_{t-1}|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1})}{P(z_t)} \tag{8.14}$$

$$= \frac{P(z_t|x_t)\sum_{x_{t-1}} P(x_t|x_{t-1}, u_t)P(x_{t-1}|x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1})}{P(z_t)} \tag{8.15}$$

$$= \frac{P(z_t|x_t)\sum_{x_{t-1}} P(x_t|x_{t-1}, u_t)P(x_{t-1}|x_0, u_1, \ldots, u_{t-1}, z_1, \ldots, z_{t-1})}{P(z_t)} \tag{8.16}$$

$$= \frac{P(z_t|x_t)\sum_{x_{t-1}} P(x_t|x_{t-1}, u_t)\hat{x}_{t-1}}{P(z_t)} \tag{8.17}$$

$$= \eta P(z_t|x_t)\sum_{x_{t-1}} P(x_t|x_{t-1}, u_t)\hat{x}_{t-1} \tag{8.18}$$

The above chain of equalities can be derived as follows. Eq. (8.12) follows from Eq. (8.11) by applying Eq. (**??**) where $x_0, u_1, \ldots, u_t, z_1, \ldots, z_{t-1}$ are the background knowledge variables. Eq. (8.13) is obtained exploiting the assumption that the sensor reading depends only on the current state x_t and is independent from previous sensor readings, states and inputs. Next, Eq. (8.14) is obtained from Eq. (8.13) applying the total probability theorem (Eq. (A.2)) where the partition Ω is the set of states at time $t-1$. We next exploit the hypothesis that the motion model is Markovian and the state at time t is exclusively a function of the state at $t-1$ and of the last input u_t. This leads to Eq. (8.15). The next equations follows observing that x_{t-1} does not depend on u_t, i.e., the input applied once[4] we are in x_{t-1}. Therefore u_t can be dropped from the set of conditioning variables and we get Eq. (8.16). Next observe that the last term in Eq. (8.16) is nothing but $\hat{x}_{t-1}$ as per Eq. (8.10). This leads to Eq. (8.17) and Eq. (8.18) follows writing η for the normalization constant. The last expression can be seen as the product of two terms, i.e., $\sum_{x_{t-1}} P(x_t|x_{t-1}, u_t)\hat{x}_{t-1}$ and $\eta P(z_t|x_t)$. The first, is the prediction step, and it increases uncertainty, whereas the second is the correction step decreasing uncertainty by querying the sensor.

The estimation technique we just presented is also called *histogram filter* because it assigns a posterior to each of the discrete values for X. Hence, these values can be visualized as a histogram, especially when considering unidimentional situations like those described in Examples 8.4 and 8.5. The implementation of the algorithm is rather straightforward, assuming that the motion model and sensor model are given. Algorithm 6 sketches the pseudocode, assuming that X is a discrete random variable whose alphabet has n elements, i.e., $x_1, x_2, \ldots, x_n$.

[4] u_t will influece x_t but does not influence x_{t-1} because it is executed *after* the system has reached state x_{t-1}.

Data: $\hat{x}_{t-1}$, z_t, u_t
Result: Estimate $\hat{x}_t$
1 **for** $i \leftarrow 1$ **to** n **do**
2 $\quad x'_i \leftarrow \sum_{x_{t-1}} P(x'_i|x_{t-1}, u_t)\hat{x}_{t-1}$;
3 **for** $i \leftarrow 1$ **to** n **do**
4 $\quad x''_i \leftarrow P(z_t|x_i)x'_i$;
5 $\frac{1}{\eta} \leftarrow \sum_{i=1}^{n} x''_i$;
6 **for** $i \leftarrow 1$ **to** n **do**
7 $\quad \hat{x}_i \leftarrow \eta x''_i$;

Algorithm 6: Discrete Bayes Filter

The computational complexity of the algorithm clearly depends on the number of elements in the alphabet n. Assuming that $P(x_t|x_{t-1}, u_t)$ and $P(z_t|x'_i)$ can be computed in $\mathcal{O}(1)$, the complexity of the algorithm is $\mathcal{O}(n^2)$. The quadratic term stems from the prediction step (for loop in line 1). Note however that this result relies on the assumption that the sensor and actuation models can be compute in constant time. One critical aspect is the dependency on n. If the discrete Bayes filter is used to solve the localization algorithm, one would split the state space in cells (normally equally sized). For example, one could partition the state space for the pose x, y, ϑ in a regurlar grid. From an accuracy stand point one would like to have the cells as small as possible, because each point inside the same grid cell is indistinguishable. However, this means increasing n, and thus slowing the algorithm. There exist various computational expedients one may apply to counter this problem, but these come at the price of a more complicated implementation. For this, and various other reasons, other algorithms have been developed and are often used in practice.

8.5 Particle Filters

Particle filters are used to solve non-linear, non-Gaussian estimation problems[5], and represent the posterior through a set of *particles*. In the following, the term *particle* is synonymous with *sample* drawn from a certain distribution. The intuition is the following. Imagine drawing a set of independent samples from a random variable with a fixed, unknown PDF. Such samples will be denser in areas where the PDF is higher and more sparse where the PDF is lower. Therefore, from the set of samples, one can get an idea about the unknown PDF they are drawn from. As the number of samples grows, one can get more and more confident about the shape of the underlying PDF.

For example, in Figure 8.6 we show the histogram for 1000 samples drawn from a Gaussian distribution with mean $\mu_x = 2$ and $\sigma = 5$. Even without knowing the underlying parameters, by looking at the samples one can start to guess these values. For example, already with just 1000 samples it appears that the mean is somewhere around 2.

Unsurprisingly, as the number of samples grows, the histogram of the sample distribution

[5]When distributions are Gaussian, the Kalman Filter or the Extended Kalman Filter can be used in the linear and non-linear case, respectively.

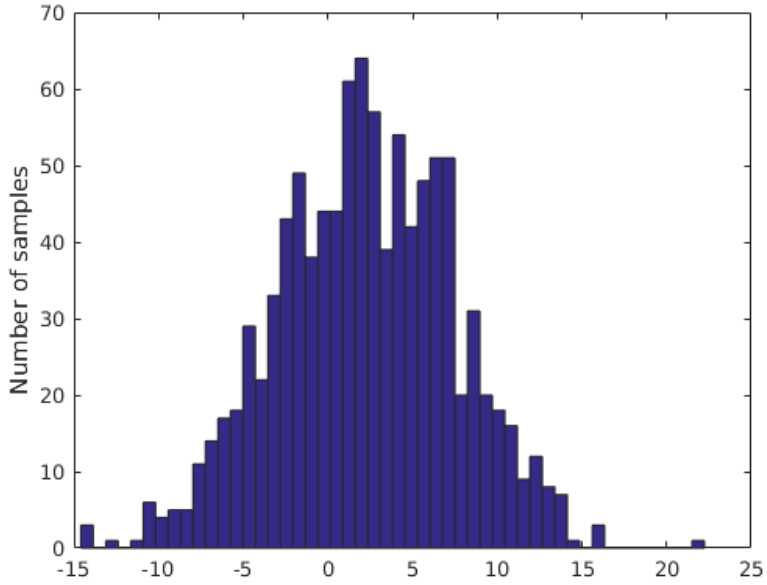


Figure 8.6: Histogram of the distribution of 1000 samples drawn from a $\mathcal{N}(2, 5)$.

resembles more and more the underlying PDF. This is evident in Figure 8.7, where we show the distribution of 10,000 samples drawn from the same underlying PDF.

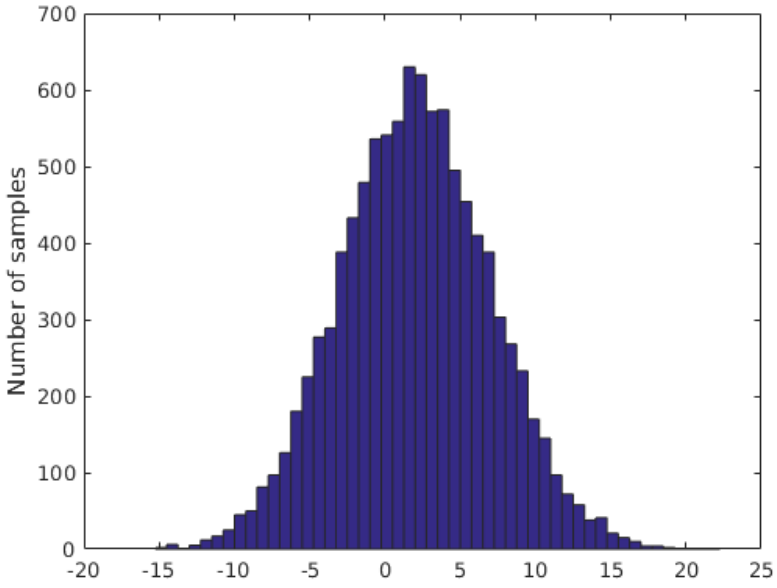


Figure 8.7: Histogram of the distribution of 10000 samples drawn from a $\mathcal{N}(2, 5)$.

While estimating the parameters of a Gaussian from a set of samples may be instructive, the power of this approach lies in estimating more complex posteriors, in particular non-

parametric distributions. This is shown in Figure 8.8, where we again display the histograms for 1000 and 10,000 samples drawn from the same underlying PDF (in this case, a bimodal distribution.)

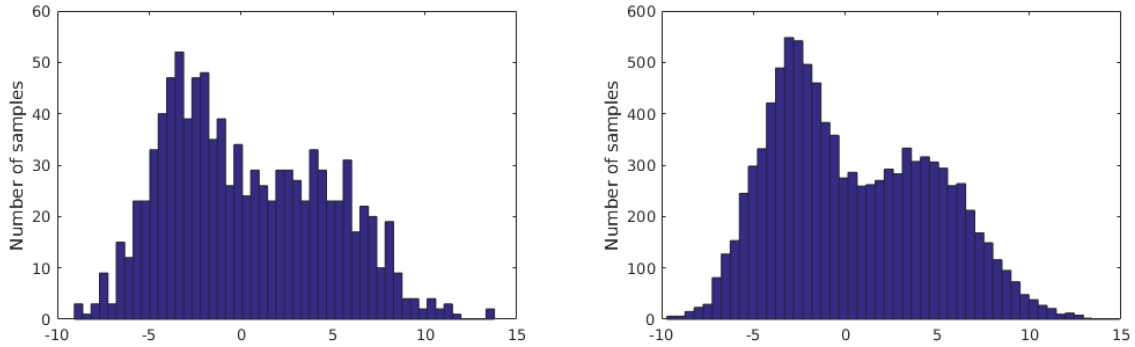


Figure 8.8: Histogram of the distribution of 1000 samples (left) and 10000 samples (right) from the same bimodal distribution.

The theory of estimation with particle filters has broad applicability and a general discussion is beyond the scope of these notes. In the following, we restrict our attention to using particle filters to estimate the pose of the robot. Indeed, particle filters became popular in robotics because they provide an efficient, general solution to this problem. Building upon our former observations, the idea behind estimation with particle filters is to estimate a posterior through a set of samples drawn from it. Of course, this is easier said than done, because the very reason to run an estimation algorithm is to derive the posterior, so one cannot directly draw samples from it.

This problem can be somehow circumvented by assuming the capability of sampling from other, easier distributions. In particular, earlier on we assumed the availability of a motion model, i.e., $P(x'|x,u)$, that is, the ability to determine the probability that the next state is x' assuming that the current state is x and the applied input is u. This setup is appropriate in the case of the discrete Bayes filter. In general, in the continuous case, we would consider $f(x'|x,u)$, where f is a PDF, rather than the probability P of an element. Recall that for fixed x and u, f is a PDF, i.e., it is non-negative and it must integrate to 1. To implement the particle filter algorithm, we assume the ability to generate samples from a random variable with PDF f for each possible choice of x and u. The trick is that while we cannot draw samples from the posterior probability we want to estimate, drawing samples from f is simpler. The reason is that f captures the underlying motion ability of the robot, so we can determine it in an analytic way. From now onwards, we assume that such a PDF is given and that samples can be generated.

The other element we need is the sensor model, that is $g(h|x)$ where g is a density function. In fact, in this case, too, we move from the discrete domain to the continuous domain, so PMFs are substituted by PDFs. For a fixed x, g is a PDF, and therefore it is non-negative and integrates to 1. Building upon these elements, the recursive particle filter estimation algorithm starts from a set of particles representing the estimate $\hat{x}_{t-1}$ and generates a new set of particles representing the estimate $\hat{x}_t$. To do so, as for the case of the discrete Bayes filter, it also needs the latest input u_t and the latest sensor reading z_t. Let

$\mathcal{P}_{t-1}$ be the set of particles representing the estimate $\hat{x}_{t-1}$. Let us assume it consists of N particles and let $\hat{x}^i_{t-1}$ be the i-th particle in $\mathcal{P}_{t-1}$. The algorithm works in two stages.

In the first stage, for each particle in $\mathcal{P}_{t-1}$ a new (intermediate) particle is generated and assigned a weight. The ith new particle, called x^i_t, is created by sampling from the PDF $f(\hat{x}^i_{t-1}, u_t)$, and its weight is set to $g(z_t|x^i_t)$. Let w_i be the weight assigned to the new particle, and let $\mathcal{P}'_t$ be the set of the newly generated intermediate particles.

In the second stage, $\mathcal{P}_t$ is computed by sampling particles from $\mathcal{P}'_t$. Each sample is independently drawn from $\mathcal{P}'_t$ with repetitions and with probability proportional to the weight of the particle. This step is called *importance sampling*. Algorithm 7 sketches this approach.

Data: $\mathcal{P}_{t-1}$, z_t, u_t
Result: New set of particles $\mathcal{P}_t$
1 **for** $i \leftarrow 1$ **to** N **do**
2 | $x^i_t \sim f(\hat{x}^i_{t-1}, u_t)$;
3 | $w_i \leftarrow g(z_t|x^i_t)$;
4 **for** $i \leftarrow 1$ **to** N **do**
5 | $\hat{x}^i_t \sim$ sample from $\mathcal{P}'_t$ with probability proportional to w_i;

Algorithm 7: Particle filter for localization

The first loop starting at line 1 generates the new set of intermediate particles, whereas the second loop at line 4 performs the importance sampling step. The informal intuition is the following. The algorithm uses each existing particle $\hat{x}^i_{t-1}$ jointly with the latest input u_t to sample a new candidate particle x^i_t. This represents a *prediction* about how the robot could have moved if $\hat{x}^i_{t-1}$ was the true pose. Note that since x^i_t is sampled from a PDF, such a sample already models the uncertainty associated with the motion. Moreover, even if there are multiple identical particles (say $\hat{x}^i_{t-1} = \hat{x}^j_{t-1}$), the prediction step will, with high probability, produce predictions $x^i_t \neq x^j_t$ due to the stochastic sampling. The next step assigns to each prediction a weight using the sensor model PDF g. Informally speaking, predictions that "explain well" the sensor reading z_t will receive a high weight through g, while those that poorly match the sensor reading will receive a low weight. Observe that the assigned weight depends on both the input and the sensor reading, because g is conditioned on the predicted particle and the predicted particle takes the input u_t as a parameter for the sampling f. This way, when the importance sampling step occurs, particles that are in good agreement with both u_t and z_t have a higher chance of being selected. Moreover, even if a particle is selected more than once during the resampling step, this is not a problem, and is in fact desirable (think about the examples discussed in the previous figures.)

Algorithm 7 gives a general idea of the particle filter estimation process but omits many important details. The first, of course, concerns the number of particles to be used to estimate the posterior. A large number of particles is desirable because it, in principle, allows a smaller approximation error, but at the same time, at each iteration, all particles must be reprocessed to generate the new set, so from this standpoint, a smaller number of particles is preferable. Determining the right number of particles for a given localization problem is in general not simple and requires some experience, although there exist more

sophisticated versions of the algorithm that will adjust the size of the particle set on the fly. Another relevant problem is how to initialize $\mathcal{P}_0$, i.e., the initial set of particles. This is one of the appealing aspects of estimation with particle filters. If one has a good estimate of the initial pose of the robot, i.e., a prior distribution, $\mathcal{P}_0$ shall be initialized by drawing samples from that distribution. If, on the contrary, a prior is not available, one can initialize $\mathcal{P}_0$ using a uniform distribution over the set of possible states. Note that this lack of knowledge in fact corresponds to assuming a uniform prior over the set of possible poses. In this case, the filter is used to solve the so-called *global* localization problem. One final problem is how to translate the posterior represented by the particles into a usable representation. For example, in a localization problem, one is interested in (x, y, ϑ), possibly with an associated uncertainty margin, rather than a set of particles.

We next display a couple of examples of solutions to the localization problem using the particle filter. Both are derived using ROS, and the reader is referred to section 9.4 for details about the node implementing this algorithm. Both cases are executed in the map displayed in figure 8.9.

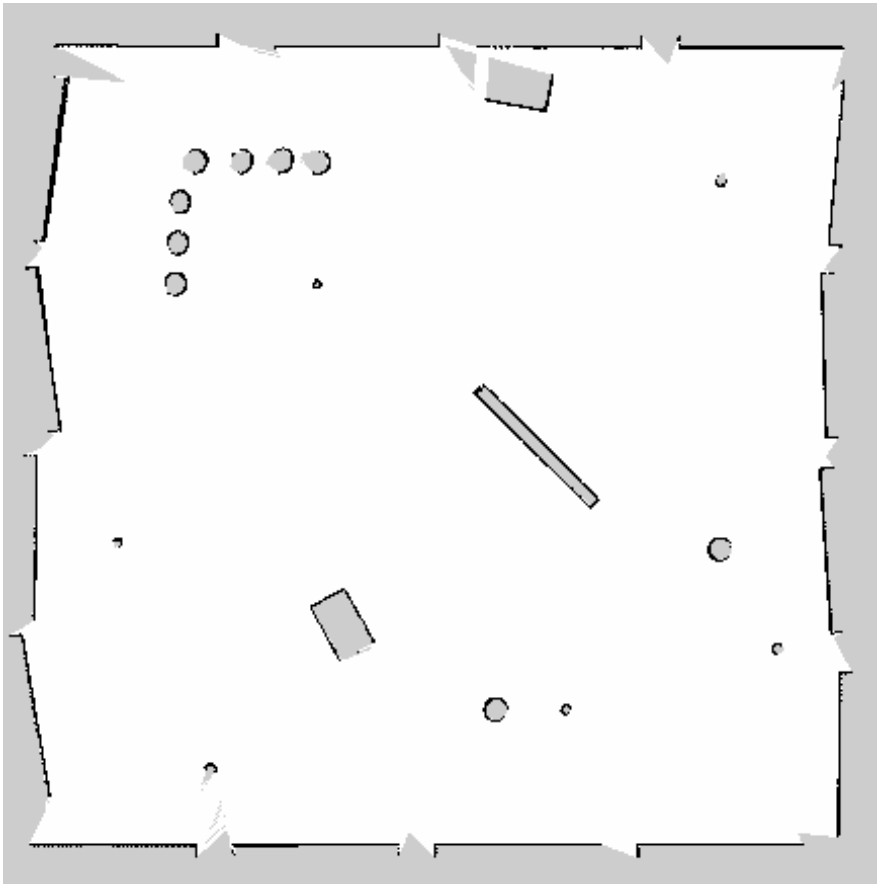


Figure 8.9: Map used for the localization example with the particle filter.

Figure 8.10 shows the evolution of the particles for the case where the prior is initially set to a Gaussian centered in the middle of the map. Note how the particle set first shrinks, but also occasionally spreads. The number N at the top is the number of particles, and it

varies through the run because the node implementing the algorithm features an advanced technique to adjust the size on the fly.

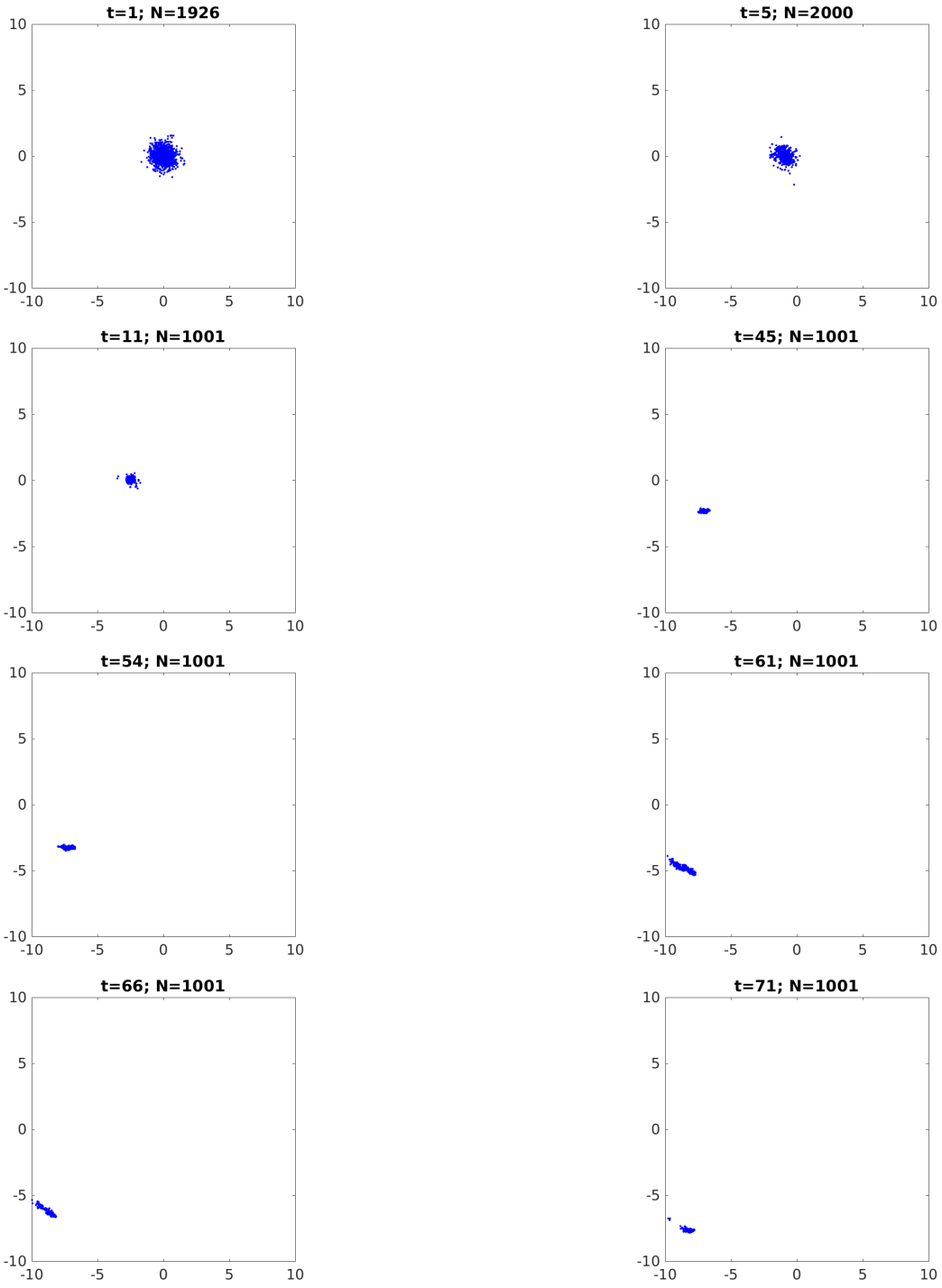


Figure 8.10: Particle Filters Example.

Figure 8.11 instead shows a run where the particle filter is initialized using a uniform distribution for the particles. This is useful when there is no prior information about the

initial location of the robot. Hence, a uniform prior is used to represent the lack of prior knowledge about the location of the robot. This is well shown in the snapshot at time $t = 1$. As time progresses, the particles concentrate in few regions, as shown for time $t = 7$ and $t = 20$. By the time $t = 33$ the filter is essentially tracking two hypotheses, as shown by the two clusters of particles. This ambiguity continues for a while (next three snapshots), but by the time $t = 70$ (last snapshot) all particles have converged in the vicinity of single location.

Remark 8.4. *In Section 8.2, we pointed out that estimation is often cast as an optimization problem where the goal is to minimize a suitably defined error measure (e.g., the root mean square error of the measurement residual, as per Eq.* (8.3)*). One should note that, while particle filters perform very well in practice and are often the method of choice in many applications, it is not straightforward to quantify the estimation error.*

8.6 Probabilistic Motion Models

The discrete Bayes filter presented in Section 8.4 and the particle filter presented in Section 8.5 rely on the availability of probabilistic models for the motion model. In particular, for the discrete Bayes filter, it is necessary to compute $P(x_{t+1}|x_t, u_t)$ (see Algorithm 6), whereas for the particle filter, it is necessary to sample from the distribution $f(x_t, u_t)$ (see Algorithm 7). In this section, we briefly discuss how these models can be formulated. The reader is referred to Chapter 5 in [53] for a detailed discussion of these topics and the theoretical justification of these relationships. The formulas we present in this section are in fact taken from [53] and are specialized for the case where we deal with a mobile robot whose configuration is (x, y, ϑ) and the input is given as translational and rotational velocity, i.e., $u_t = (v_t, v_r)$ (see Section 4.9).

We start by providing the formulas to compute $P(x_{t+1}|x_t, u_t)$ that is needed for the discrete Bayes filter. To this end, we assume the availability of a function $f_G(x, \sigma^2)$ that provides the PDF in x of a Gaussian distribution with zero mean and variance σ^2. The idea is the following: given two poses x_t and x_{t+1}, compute the nominal input u' that would drive the robot from x_t to x_{t+1} under the assumption that there is no noise and the motion is carried out in time Δt (a fixed parameter). The probability of executing u' instead of u_t is then related to $P(x_{t+1}|x_t, u_t)$. However, there is an additional source of error to be considered—namely, the fact that the final orientation of the robot in x_{t+1} is affected by an orientation error assumed to be independent from the input errors[6]. Algorithm 8 builds upon these ideas and provides $P(x_{t+1}|x_t, u_t)$.

The algorithm computes the nominal input u' that would drive the robot from x_t to x_{t+1} in a noise-free scenario. This is done by the `InverseModel` function. In addition, a final orientation error $\delta\vartheta$ is computed as well. Finally, the probability is returned by multiplying together the densities of three Gaussians. Note that the variance of these distributions depends on a six-dimensional vector $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_6)$ that models the accuracy (or lack thereof) of the robot being used. From a practical standpoint, these parameters could be

[6] See [53] for a detailed discussion about why this is a necessary assumption.

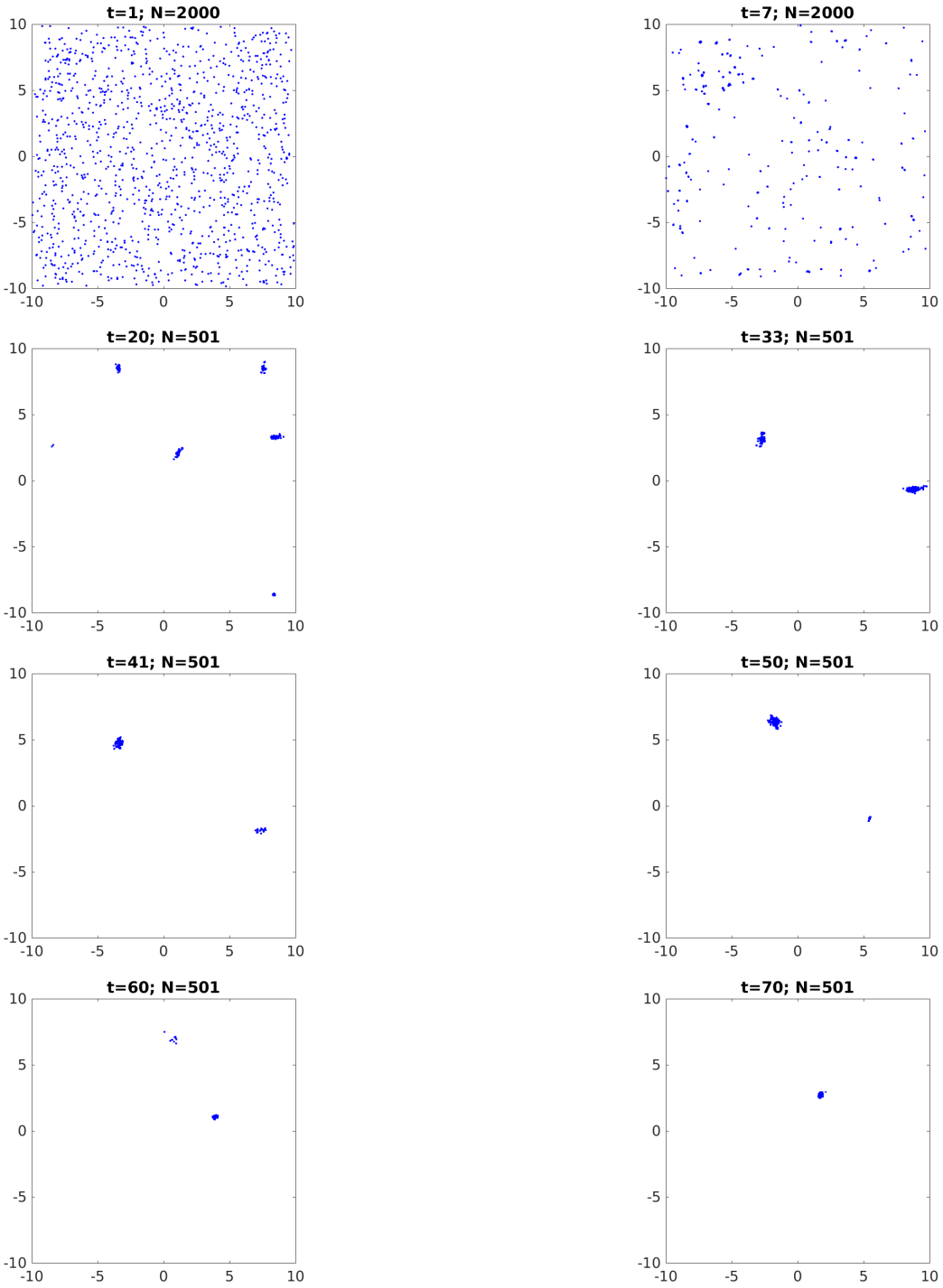


Figure 8.11: Particle Filters Example.

estimated through a (time-consuming) experimental approach, where many trajectories are collected and the parameters are then estimated.

Following a similar reasoning, given x_t and u_t, a new pose can be obtained by sampling from the PDF $f(x_t, u_t)$, as needed by the particle filter. The idea is to alter u_t with appro-

Data: $x_{t+1} = (x', y', \vartheta')$, $x_t = (x, y, \vartheta)$, $u_t = (v_t, v_r)$
Result: $P(x_{t+1}|x_t, u_t)$
$u' = (v_t', v_r') \leftarrow \text{InverseModel}(x_{t+1}, x_t)$;
$\delta\vartheta \leftarrow \frac{\vartheta' - \vartheta}{\Delta t} - v_r'$;
return $f_G(v_t - v_t', \alpha_1 v_t^2 + \alpha_2 v_r^2) \cdot f_G(v_r - v_r', \alpha_3 v_t^2 + \alpha_4 v_r^2) \cdot f_G(\delta\vartheta, \alpha_5 v_t^2 + \alpha_6 v_r^2)$;

Algorithm 8: Probabilistic Motion Model

priate noise to obtain a new input u', and to use this new input to determine the new pose with a deterministic motion model (see Eq. (4.30)). As with the probabilistic motion model, an additional orientation error is added at the end. Algorithm 9 illustrates this idea. The algorithm relies on the availability of a function $s(\sigma^2)$ that returns a sample from $\mathcal{N}(0, \sigma^2)$. The function `ForwardModel` implements Eq. (4.30), and, like Algorithm 8, the algorithm relies on the vector of parameters $\boldsymbol{\alpha}$.

Data: $x_t = (x, y, \vartheta)$, $u_t = (v_t, v_r)$
Result: $x_{t+1} \sim f(x_t, u_t)$
$v_t' \leftarrow v_t + s(\alpha_1 v_t^2 + \alpha_2 v_r^2)$;
$v_r' \leftarrow v_r + s(\alpha_3 v_t^2 + \alpha_4 v_r^2)$;
$\delta\vartheta \leftarrow s(\alpha_5 v_t^2 + \alpha_6 v_r^2)$;
$(x', y', \vartheta') \leftarrow \text{ForwardModel}(x, y, \vartheta, v_t', v_r')$;
$\vartheta' \leftarrow \vartheta' + \delta\vartheta\Delta t$;
return (x', t', ϑ');

Algorithm 9: Sample generation

8.7 Kalman Filter

We now switch to a different parametric estimation technique based on the assumption that the posterior is represented as a (multivariate) Gaussian distribution. Consequently, the estimation algorithm aims at determining the mean vector and covariance matrix of the posterior distribution. Before embarking on the full discussion, it is useful to recall some important facts about Gaussians and their transformations.

Let $\mathbf{x}$ be a Gaussian random vector with mean $\boldsymbol{\mu}_x$ and covariance $\boldsymbol{\Sigma}_x$. Recall that mean and covariance fully define the PDF of a Gaussian distribution. Let $\mathbf{y}$ be a random vector obtained from $\mathbf{x}$ through a linear transformation, i.e.,

$$\mathbf{y} = \mathbf{A}\mathbf{x}.$$

Then $\mathbf{y}$ is also a Gaussian random vector with mean $\boldsymbol{\mu}_y = \mathbf{A}\boldsymbol{\mu}_x$ and covariance $\boldsymbol{\Sigma}_y = \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T$. More generally, if $\mathbf{y}$ is obtained from $\mathbf{x}$ through an affine transformation

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b} \tag{8.19}$$

where $\mathbf{b}$ is a constant vector, then $\mathbf{y}$ is still a Gaussian random vector with mean $\boldsymbol{\mu}_y = \mathbf{A}\boldsymbol{\mu}_x + \mathbf{b}$ and covariance $\boldsymbol{\Sigma}_y = \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T$.

Next, consider the case where $\mathbf{x}$ is the sum of two independent Gaussian vectors $\mathbf{y}$ and $\mathbf{z}$,

$$\mathbf{x} = \mathbf{y} + \mathbf{z}.$$

Then $\mathbf{x}$ is also Gaussian, with mean $\boldsymbol{\mu}_x = \boldsymbol{\mu}_y + \boldsymbol{\mu}_z$ and covariance $\boldsymbol{\Sigma}_x = \boldsymbol{\Sigma}_y + \boldsymbol{\Sigma}_z$. If $\mathbf{x}$ is a random vector (not necessarily Gaussian) and $\mathbf{y}$ is obtained from $\mathbf{x}$ through the affine transformation in Eq. (8.19), then the above relationships still hold, i.e., $\boldsymbol{\mu}_y = \mathbf{A}\boldsymbol{\mu}_x + \mathbf{b}$ and $\boldsymbol{\Sigma}_y = \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T$. These equalities can be easily shown by applying the definitions of expectation and covariance, and by noting that integration is a linear operator. However, in this latter case where $\mathbf{x}$ is a random vector (not necessarily Gaussian) , $\boldsymbol{\mu}_y$ and $\boldsymbol{\Sigma}_y$ do not necessarily define the PDF of $\mathbf{y}$ because $\mathbf{y}$ is not Gaussian.

Example 8.7. *Let $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x)$ and $\mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_y, \boldsymbol{\Sigma}_y)$ be two independent random vectors. Let $\mathbf{b}$ be a constant vector, and define $\mathbf{z} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} + \mathbf{b}$ where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{b}$ are matrices and a vector with suitable dimensions. In this case, $\mathbf{z}$ is also a Gaussian vector with mean $\boldsymbol{\mu}_z = \mathbf{A}\boldsymbol{\mu}_x + \mathbf{B}\boldsymbol{\mu}_y + \mathbf{b}$ and covariance $\boldsymbol{\Sigma}_z = \mathbf{A}\boldsymbol{\Sigma}_x\mathbf{A}^T + \mathbf{B}\boldsymbol{\Sigma}_y\mathbf{B}^T$. These results can be easily derived using the relationships described above.*

8.7.1 Linear Case

We start by considering a dynamical system whose state $\mathbf{x}$ evolves in discrete time according to the following linear model (see also Eq. (1.10) and Eq. (1.11)):

$$\mathbf{x}_t = \mathbf{A}_t\mathbf{x}_{t-1} + \mathbf{B}_t\mathbf{u}_t + \mathbf{v}_t \tag{8.20}$$

where $\mathbf{v}_t$ is a Gaussian vector with zero mean and covariance matrix $\mathbf{R}_t$. Moreover, we assume that $\mathbf{v}_i$ is independent of $\mathbf{v}_j$ for each $i \neq j$. The input $\mathbf{u}_t$ is assumed to be known and free of uncertainty. For greater generality, the matrices $\mathbf{A}_t$ and $\mathbf{B}_t$ are considered time-dependent, as indicated by the index t. An observation of the state is obtained through a linear transformation:

$$\mathbf{z}_t = \mathbf{H}_t\mathbf{x}_t + \mathbf{w}_t \tag{8.21}$$

where $\mathbf{w}_t$ is a Gaussian vector with zero mean and covariance matrix $\mathbf{Q}_t$. Similarly, we assume that $\mathbf{w}_i$ is independent of $\mathbf{w}_j$ for each $i \neq j$. The matrix $\mathbf{H}_t$ is also assumed to be time-dependent.

Because of the noise present in both Eq. (8.20) and Eq. (8.21), the state $\mathbf{x}$ and observation $\mathbf{z}$ become random vectors even if all matrices and the input are deterministic. In this scenario, the objective of the estimation process is to determine a statistical description of these random vectors, and in particular of $\mathbf{x}$. The Kalman Filter (KF) approach assumes that $\mathbf{x}$ is normally distributed, and therefore aims to derive its mean vector and covariance matrix as these two parameters fully define its PDF.

Remark 8.5. *We have assumed that $\mathbf{A}_t$, $\mathbf{B}_t$, and $\mathbf{H}_t$ may vary over time, but they are known for every time step t. In many practical instances, these matrices are constant. However, for greater generality, the framework is presented using a time-varying formulation.*

Remark 8.6. *We have assumed that the input $\mathbf{u}_t$ is known, as is $\mathbf{B}_t$. In some cases, one may assume that the input is also a Gaussian random vector. The subsequent derivations would not change significantly (see a later remark).*

Consistent with the recursive estimation approach, our objective is to provide an estimate for the state at time t, starting from its estimate at time $t-1$, the control $\mathbf{u}_t$ at time t, the sensor reading $\mathbf{z}_t$ at time t, and the covariance matrices $\mathbf{R}_t$ and $\mathbf{Q}_t$. In particular, since our estimate aims to determine the mean and covariance of $\mathbf{x}_t$, we will assume knowledge of the previously estimated mean and covariance at time $t-1$. Therefore, the inputs to the estimation problem are:

- $\boldsymbol{\mu}_{t-1}$: mean of the estimate for $\mathbf{x}$ at time $t-1$;
- $\boldsymbol{\Sigma}_{t-1}$: covariance of the estimate for $\mathbf{x}$ at time $t-1$;
- $\mathbf{u}_t$: input at time t;
- $\mathbf{z}_t$: observation at time t;
- $\mathbf{R}_t$: covariance matrix for the system evolution noise at time t;
- $\mathbf{Q}_t$: covariance matrix for the observation noise at time t.

The outputs of the estimation problem are:

- $\boldsymbol{\mu}_t$: mean of the estimate for $\mathbf{x}$ at time t;
- $\boldsymbol{\Sigma}_t$: covariance of the estimate for $\mathbf{x}$ at time t.

The Kalman Filter provides a recursive, optimal method to solve the estimation problem based on the two steps we have already encountered: *prediction* and *correction* (also called *update*). Before discussing the details of the algorithm, it is useful to once again consider the following sequence:

$$\mathbf{x}_0 \xrightarrow{\mathbf{u}_1} \mathbf{x}_1 \xrightarrow{\mathbf{z}_1} \mathbf{x}_1 \xrightarrow{\mathbf{u}_2} \mathbf{x}_2 \xrightarrow{\mathbf{z}_2} \mathbf{x}_2 \ldots$$

In Algorithm 10, we provide the equations that implement the prediction and correction steps.

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \mathbf{u}_t, \mathbf{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
/* Prediction */
1 $\bar{\boldsymbol{\mu}}_t = \mathbf{A}_t \boldsymbol{\mu}_{t-1} + \mathbf{B}_t \mathbf{u}_t$;
2 $\bar{\boldsymbol{\Sigma}}_t = \mathbf{A}_t \boldsymbol{\Sigma}_{t-1} \mathbf{A}_t^T + \mathbf{R}_t$;
/* Update */
3 $\mathbf{K}_t = \bar{\boldsymbol{\Sigma}}_t \mathbf{H}_t^T (\mathbf{H}_t \bar{\Sigma}_t \mathbf{H}_t^T + \mathbf{Q}_t)^{-1}$;
4 $\boldsymbol{\mu}_t = \bar{\boldsymbol{\mu}}_t + \mathbf{K}_t (\mathbf{z}_t - \mathbf{H}_t \bar{\boldsymbol{\mu}}_t)$;
5 $\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t \mathbf{H}_t) \bar{\boldsymbol{\Sigma}}_t$;
6 **return** $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$;

Algorithm 10: Linear Kalman Filter

The *prediction* step produces an estimate for $\mathbf{x}_t$ (i.e., its mean and covariance) after accounting for the effects of the input $\mathbf{u}_t$, but before incorporating the measurement $\mathbf{z}_t$. The

mean and covariance obtained after the prediction step are denoted as $\bar{\boldsymbol{\mu}}_t$ and $\bar{\boldsymbol{\Sigma}}_t$, where the bar indicates that these values have been computed without considering the most recent observation. The *correction* step refines the estimate for $\mathbf{x}_t$ by incorporating the latest sensor reading $\mathbf{z}_t$. This step updates the predicted values $\bar{\boldsymbol{\mu}}_t$ and $\bar{\boldsymbol{\Sigma}}_t$, and yields the final estimate at time t, denoted by $\boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_t$. Note that to *bootstrap* the method, we need an initial estimate for the mean and covariance; that is, we require $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$.

The KF method is particularly interesting when the initial estimate is Gaussian distributed, i.e., $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$ represent the mean and covariance of a Gaussian random vector. In this case, thanks to the properties outlined earlier, the dynamic state $\mathbf{x}_t$ remains Gaussian because it results from the sum of an affine transformation of Gaussian variables, $\mathbf{A}\mathbf{x}_{t-1}+\mathbf{B}\mathbf{u}_t$, and another Gaussian variable, $\mathbf{v}_t$. If $\mathbf{x}_0$ is a random vector with mean $\boldsymbol{\mu}_0$ and covariance $\boldsymbol{\Sigma}_0$ but is not Gaussian, the method is still valid and the relationships still hold. The KF algorithm continues to provide the correct mean and covariance; however, these quantities no longer fully characterize the probability distribution of $\mathbf{x}_t$ because it is not Gaussian.

Remark 8.7. *We previously stated that the KF method provides an* optimal *estimate. This optimality is defined with respect to a specific objective function that minimizes the estimation error, as given in Eq.* (8.2)*.*

Remark 8.8. *If the vector $\mathbf{u}_t$ is also a Gaussian random vector with mean μ_{u_t} and covariance matrix Σ_{u_t}, then the first equation in the prediction step becomes $\bar{\mu}_t = \mathbf{A}_t\mu_{t-1} + \mathbf{B}_t\mu_{u_t}$, and the second equation becomes $\bar{\Sigma}_t = \mathbf{A}_t\Sigma_{t-1}\mathbf{A}_t^T + \mathbf{B}_t\Sigma_{u_t}\mathbf{B}_t^T + \mathbf{R}_t$. The correction step remains unchanged.*

Remark 8.9. *In some cases, inputs and observations may not alternate in a one-to-one fashion. For example, the system might receive an observation every three inputs, or it may receive multiple observations in a row. In such cases, the step corresponding to the missing data is simply skipped, i.e., one would perform multiple predictions in sequence or multiple corrections in sequence.*

Remark 8.10. *The term $(\mathbf{z}_t - \mathbf{H}_t\bar{\mu}_t)$ is called* innovation*.*

Before continuing with the nonlinear case, it is instructive to consider a very simple unidimensional example.

Example 8.8. *Let us consider a unidimensional case governed by the following simple equations:*

$$x_t = x_{t-1} + u_t$$
$$z_t = x_t$$

Let v_t and w_t have constant variance, as specified below. The first input u_1 is 2, and the first sensor reading z_1 is 6. Finally, let $x_0 \sim \mathcal{N}(1, 0.5)$. Figure 8.12 shows the result of the estimation process for x_1 for different values of R and Q, as indicated in the titles of the subcharts.

In all figures, the red distribution represents the prior (constant in all examples), the green distribution represents the prediction, the blue distribution represents the measurement, and

the black distribution represents the posterior. Note that as the ratio between R and Q varies, the shape of the posterior (black distribution) changes accordingly. In the first three cases, the posterior has lower variance than the prior, but in the last case this is no longer true, as evidenced by the lower peak. Moreover, observe how the mean of the posterior may be closer to the mean of the measurement or the prediction; this also depends on the ratio between R and Q.

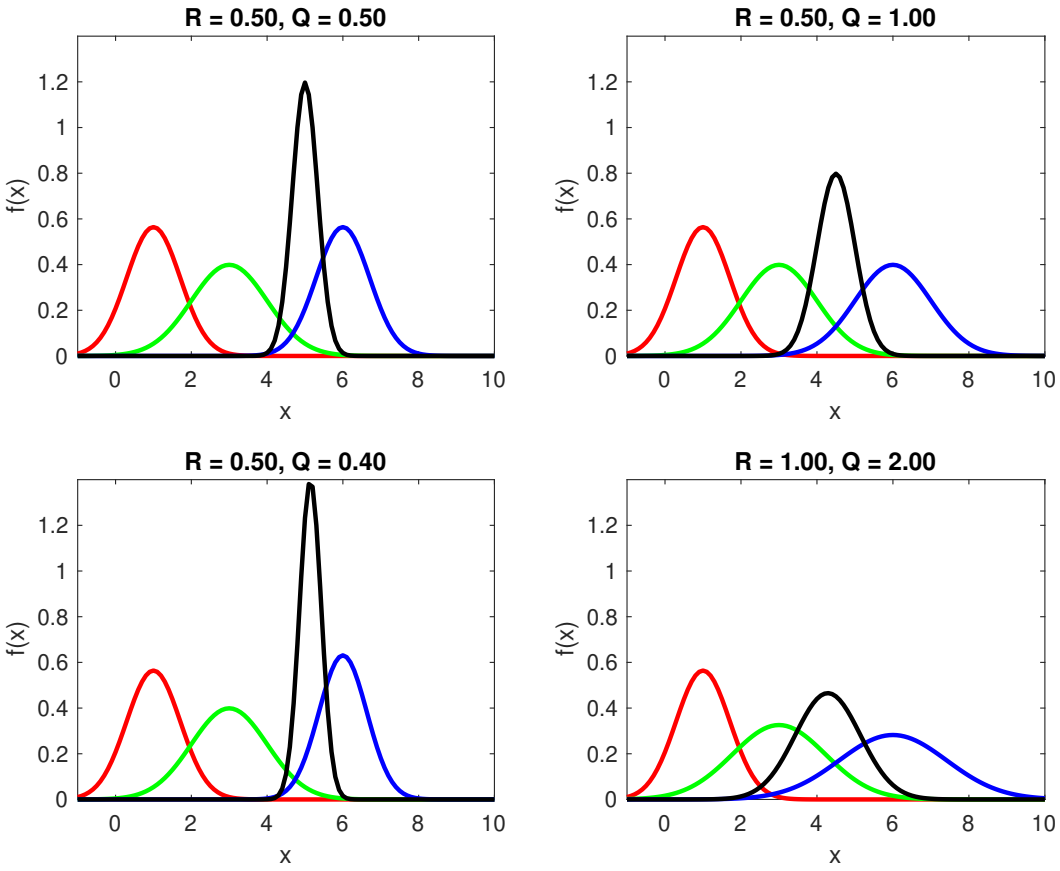


Figure 8.12: Prior (red), prediction (green), measurement (blue), posterior (black) for different values of Q and R.

Example 8.9. *We now consider a small variation of the previous example, in which the robot is equipped with two sensors that measure the same unidimensional state. This scenario is modeled as follows:*

$$x_t = x_{t-1} + u_t$$
$$z_t = \mathbf{H}x_t$$

where the observation matrix $\mathbf{H}$ *and the covariance matrix of the observation noise are given by*

$$\mathbf{H} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \qquad \mathbf{Q} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

Observe the structure of $\mathbf{H}$*. In the ideal, noise free scenario, the first sensor reading returns the state* x *(because the first element in* $\mathbf{H}$ *is 1), while the second sensor returns* $2x$ *(because the second element in* $\mathbf{H}$ *is 2). As in the previous example, let the initial state be distributed as* $x_0 \sim \mathcal{N}(1, 0.5)$*, let the control input be* $u_1 = 1$*, and assume the first sensor reading is* $\mathbf{z}_1 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$.

The prediction step follows the same state transition model and yields a predicted mean and variance:

$$\bar{\mu}_1 = 2, \qquad \bar{\sigma}_1^2 = 1.$$

However, the correction step is different due to the multiple, possibly conflicting, sensor measurements. In this case, the first sensor reading suggests that the prediction underestimates the state (the prediction is 2, but the sensor indicates 3), while the second sensor reading suggests it overestimates it (the prediction is still 2, but the sensor indicates 1.5 because it returns the value multiplied by 2). The Kalman gain resolves this conflict by weighting each observation based on its reliability:

$$\mathbf{K}_1 = \bar{\sigma}_1^2 \mathbf{H}^T \left(\mathbf{H} \bar{\sigma}_1^2 \mathbf{H}^T + \mathbf{Q} \right)^{-1} = [0.5263 \quad 0.2105].$$

Using this Kalman gain, the posterior mean and variance are:

$$\mu_1 = \bar{\mu}_1 + \mathbf{K}_1 (\mathbf{z}_1 - \mathbf{H} \bar{\mu}_1) = 2.3158,$$

$$\sigma_1 = (1 - \mathbf{K}_1 \mathbf{H}) \bar{\sigma}_1^2 = 0.0526.$$

This example highlights how the Kalman filter effectively combines multiple sensor readings with different noise characteristics to produce an optimal estimate of the system state.

8.7.2 Nonlinear Case

In many practical cases, the state evolves according to a nonlinear relationship and the observation is a nonlinear function of the state. In this case, Eqs. (8.20) and (8.21) are replaced by the following nonlinear relationships:

$$\mathbf{x}_t = f_t(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{v}_t \tag{8.22}$$

$$\mathbf{z}_t = h_t(\mathbf{x}_t) + \mathbf{w}_t \tag{8.23}$$

The same assumptions hold regarding the noise terms $\mathbf{v}_t$ and $\mathbf{w}_t$, i.e., 0 mean and Gaussian distributed. Additionally, as in the linear case, we consider time-dependent state evolution function f_t and observation function h_t. The Extended Kalman Filter (EKF) provides an estimate of $\mathbf{x}_t$ by linearizing the nonlinear functions f_t and h_t. However, the estimate is no longer optimal, as the linearization introduces approximation error. The EKF uses the following Jacobian matrices:

$$\mathbf{A}_t = \frac{\partial f_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}}, \qquad \mathbf{B}_t = \frac{\partial f_t(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}}, \qquad \mathbf{H}_t = \frac{\partial h_t(\mathbf{x})}{\partial \mathbf{x}}. \tag{8.24}$$

Data: $\boldsymbol{\mu}_{t-1}, \boldsymbol{\Sigma}_{t-1}, \mathbf{u}_t, \mathbf{z}_t$
Result: $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$
/* Prediction */
1 $\bar{\mu}_t = f_t(\mu_{t-1}, \mathbf{u}_t)$;
2 $\bar{\Sigma}_t = \mathbf{A}_t \Sigma_{t-1} \mathbf{A}_t^T + \mathbf{R}_t$;
/* Update */
3 $\mathbf{K}_t = \bar{\Sigma}_t \mathbf{H}_t^T (\mathbf{H}_t \bar{\Sigma}_t \mathbf{H}_t^T + \mathbf{Q}_t)^{-1}$;
4 $\mu_t = \bar{\mu}_t + \mathbf{K}_t(\mathbf{z}_t - h_t(\bar{\mu}_t))$;
5 $\Sigma_t = (\mathbf{I} - \mathbf{K}_t \mathbf{H}_t)\bar{\Sigma}_t$;
6 **return** $\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t$;

Algorithm 11: Extended Kalman Filter

These matrices are evaluated at the latest state estimate μ_{t-1}. Based on these definitions, the EKF algorithm shown in Algorithm 11 performs the prediction and correction steps as follows (compare with Algorithm 10).

In the first equation of the prediction step, we use the nonlinear state evolution function f_t, and in the second equation of the correction step, we use the nonlinear observation function h_t. Note also that in the previous formulation, we have not used the matrix $\mathbf{B}_t$ because we again assumed that the input is known with no uncertainty. However, if this was not the case, we would use the same formulation given in the previous remark where we considered $\mathbf{u}_t$ as Gaussian.

8.7.3 Numerical Example

We provide a fully worked-out example of two estimation steps for the EKF. The state transition equations and observation equations are inspired by a differential robot moving on the plane. Its state is $\mathbf{x} = [x\ y\ \theta]^T$ and consists of the pose (x, y) and heading (θ). The robot has two inputs, i.e., $\mathbf{u} = [l\ r]^T$, where l is its translational speed and r is its rotational speed. The robot can either move forward or rotate in place, but it cannot move along an arc. This means that at any given time t, either $l_t \neq 0$ and $r_t = 0$, or $l_t = 0$ and $r_t \neq 0$, but it cannot be that both $l_t \neq 0$ and $r_t \neq 0$ simultaneously. Under these assumptions, the state transition equation has the following structure, where it was tacitly assumed that $\Delta t = 1$:

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t) + \mathbf{v}_t = \begin{bmatrix} x_{t-1} + l_t \cos(\theta_{t-1}) \\ y_{t-1} + l_t \sin(\theta_{t-1}) \\ \theta_{t-1} + r_t \end{bmatrix} + \mathbf{v}_t.$$

$\mathbf{v}_t$ is a zero mean Gaussian vector with covariance matrix $\mathbf{R}$

$$\mathbf{R} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.2 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$$

The robot is equipped with two sensors. The first one gives the distance from a landmark at a know location $(x_L, y_L) = (2, 2)$, and the second provides the heading deviation

(difference) from a reference direction $\theta_R = 0.1$. These two sensors can be modeled as follows:

$$\mathbf{z}_t = h_t(\mathbf{x}_t) + \mathbf{w}_t = \begin{bmatrix} \sqrt{(x_L - x_t)^2 + (y_L - y_t)^2} \\ \theta_t - \theta_R \end{bmatrix} + \mathbf{w}_t$$

where $\mathbf{v}_t$ is a zero mean Gaussian vector with covariance matrix (assumed constant)

$$\mathbf{Q} = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.05 \end{bmatrix}$$

The initial state is Gaussian distributed with mean $\boldsymbol{\mu}_0 = [0\ 0\ 0]^T$ and covariance

$$\boldsymbol{\Sigma}_0 = \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}$$

Assume that the first two inputs are $\mathbf{u}_1 = [0.5\ 0]^T$ and $\mathbf{u}_2 = [0\ 0.2]^T$, and the first two sensor readings are $\mathbf{z}_1 = [2.4\ 0]^T$ and $\mathbf{z}_2 = [2.1\ 0.2]^T$. Based on this information, compute $\boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}_2$.

To answer the question, we use the EKF, starting by computing the Jacobian matrices $\mathbf{A}$ and $\mathbf{H}$.

$$\mathbf{A} = \frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial(x_{t-1} + l_t \cos(\theta_{t-1}))}{\partial x_{t-1}} & \frac{\partial(x_{t-1} + l_t \cos(\theta_{t-1}))}{\partial y_{t-1}} & \frac{\partial(x_{t-1} + l_t \cos(\theta_{t-1}))}{\partial \theta_{t-1}} \\ \frac{\partial(y_{t-1} + l_t \sin(\theta_{t-1}))}{\partial x_{t-1}} & \frac{\partial(y_{t-1} + l_t \sin(\theta_{t-1}))}{\partial y_{t-1}} & \frac{\partial(y_{t-1} + l_t \sin(\theta_{t-1}))}{\partial \theta_{t-1}} \\ \frac{\partial(\theta_{t-1} + r_t)}{\partial x_{t-1}} & \frac{\partial(\theta_{t-1} + r_t)}{\partial y_{t-1}} & \frac{\partial(\theta_{t-1} + r_t)}{\partial \theta_{t-1}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -l_t \sin(\theta_{t-1}) \\ 0 & 1 & l_t \cos(\theta_{t-1}) \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H} = \frac{\partial h}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial(\sqrt{(x_L - x_t)^2 + (y_L - y_t)^2})}{\partial x_t} & \frac{\partial(\sqrt{(x_L - x_t)^2 + (y_L - y_t)^2})}{\partial y_t} & \frac{\partial(\sqrt{(x_L - x_t)^2 + (y_L - y_t)^2})}{\partial \theta_t} \\ \frac{\partial(\theta_t - \theta_R)}{\partial x_t} & \frac{\partial(\theta_t - \theta_R)}{\partial y_t} & \frac{\partial(\theta_t - \theta_R)}{\partial \theta_t} \end{bmatrix}$$
$$= \begin{bmatrix} \frac{x_t - x_L}{\sqrt{(x_L - x_t)^2 + (y_L - y_t)^2}} & \frac{y_t - y_L}{\sqrt{(x_L - x_t)^2 + (y_L - y_t)^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that both $\mathbf{A}$ and $\mathbf{H}$ are evaluated at the most recent estimates. For $\mathbf{A}$, this will be the estimate at the previous step (note the indices $t-1$), while for $\mathbf{H}$ it is the result of the prediction step (note the indices t).

From these matrices and the provided data, we can compute the desired estimation. We start with the prediction at time 1 (where μ_{x0} is the x component of $\boldsymbol{\mu}_0$, and so on):

$$\bar{\boldsymbol{\mu}}_1 = f(\boldsymbol{\mu}_0, \mathbf{u}_1) = \begin{bmatrix} \mu_{x0} + l_1 \cos(\mu_{\theta 0}) \\ \mu_{y0} + l_1 \sin(\mu_{\theta 0}) \\ \mu_{\theta 0} + r_1 \end{bmatrix} = \begin{bmatrix} 0 + 0.5 \cos(0) \\ 0 + 0.5 \sin(0) \\ 0 + 0 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix}$$

To compute $\bar{\boldsymbol{\Sigma}}_1$ we use the matrix $\mathbf{A}$ evaluated at $\boldsymbol{\mu}_0$ (most recent estimate)

$$\bar{\boldsymbol{\Sigma}}_1 = \mathbf{A}\boldsymbol{\Sigma}_0\mathbf{A}^T + \mathbf{R} = \begin{bmatrix} 1 & 0 & -l_1\sin(\mu_{\theta 0}) \\ 0 & 1 & l_1\cos(\mu_{\theta 0}) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.3 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.2 \end{bmatrix} \begin{bmatrix} 1 & 0 & -l_1\sin(\mu_{\theta 0}) \\ 0 & 1 & l_1\cos(\mu_{\theta 0}) \\ 0 & 0 & 1 \end{bmatrix}^T + \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.2 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.35 & 0.1 \\ 0 & 0.1 & 0.3 \end{bmatrix}$$

Next, we can perform the correction step. We start with the Kalman Gain

$$\mathbf{K}_1 = \bar{\boldsymbol{\Sigma}}_1\mathbf{H}_1^T(\mathbf{H}_1\bar{\boldsymbol{\Sigma}}_1\mathbf{H}_1^T + \mathbf{Q})^{-1}$$

for which we need the matrix $\mathbf{H}$ evaluated at $\bar{\boldsymbol{\mu}}_1$ (most recent estimate)

$$\mathbf{H}_1 = \begin{bmatrix} \frac{\bar{\mu}_{x1} - x_L}{\sqrt{(x_L - \bar{\mu}_{x1})^2 + (y_L - \bar{\mu}_{y1})^2}} & \frac{\bar{\mu}_{y1} - y_L}{\sqrt{(x_L - \bar{\mu}_{x1})^2 + (y_L - \bar{\mu}_{y1})^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.6 & -0.8 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Plugging this matrix into the previous expression we get

$$\mathbf{K}_1 = \begin{bmatrix} -0.6176 & -0.1412 \\ -0.5294 & 0.1647 \\ -0.0235 & 0.8518 \end{bmatrix}$$

from which we can complete the correction step and compute μ_1

$$\begin{aligned} \boldsymbol{\mu}_1 =& \bar{\boldsymbol{\mu}}_1 + \mathbf{K}_1(\mathbf{z}_1 - h(\bar{\boldsymbol{\mu}}_1)) = \\ =& \begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.6176 & -0.1412 \\ -0.5294 & 0.1647 \\ -0.0235 & 0.8518 \end{bmatrix} \left(\begin{bmatrix} 2.4 \\ 0 \end{bmatrix} - h\left(\begin{bmatrix} 0.5 \\ 0 \\ 0 \end{bmatrix} \right) \right) = \begin{bmatrix} 0.5476 \\ 0.0694 \\ 0.0875 \end{bmatrix} \end{aligned}$$

and then $\boldsymbol{\Sigma}_1$

$$\begin{aligned} \boldsymbol{\Sigma}_1 &= (\mathbf{I} - \mathbf{K}_1\mathbf{H}_1)\bar{\boldsymbol{\Sigma}}_1 \\ &= \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} -0.6176 & -0.1412 \\ -0.5294 & 0.1647 \\ -0.0235 & 0.8518 \end{bmatrix} \begin{bmatrix} -0.6 & -0.8 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.35 & 0.1 \\ 0 & 0.1 & 0.3 \end{bmatrix} = \\ &= \begin{bmatrix} 0.3147 & -0.1588 & -0.0071 \\ -0.1588 & 0.1853 & 0.0082 \\ -0.0071 & 0.0082 & 0.0426 \end{bmatrix} \end{aligned}$$

To determine $\boldsymbol{\mu}_2$ and $\boldsymbol{\Sigma}_2$ we perform exactly the same computations, but starting our most recent estimates, i.e., $\boldsymbol{\mu}_1$ and $\boldsymbol{\Sigma}_1$ instead $\boldsymbol{\mu}_0$ and $\boldsymbol{\Sigma}_0$.

$$\bar{\boldsymbol{\mu}}_2 = f(\boldsymbol{\mu}_1, \mathbf{u}_2) = \begin{bmatrix} \mu_{x1} + l_2\cos(\mu_{\theta 1}) \\ \mu_{y1} + l_2\sin(\mu_{\theta 1}) \\ \mu_{\theta 1} + r_2 \end{bmatrix} = \begin{bmatrix} 0.5476 \\ 0.0694 \\ 0.2875 \end{bmatrix}$$

$$\bar{\boldsymbol{\Sigma}}_2 = \mathbf{A}\boldsymbol{\Sigma}_1\mathbf{A}^T + \mathbf{R} = \begin{bmatrix} 1 & 0 & -l_2\sin(\mu_{\theta 1}) \\ 0 & 1 & l_2\cos(\mu_{\theta 1}) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.3147 & -0.1588 & -0.0071 \\ -0.1588 & 0.1853 & 0.0082 \\ -0.0071 & 0.0082 & 0.0426 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -l_2\sin(\mu_{\theta 1}) \\ 0 & 1 & l_2\cos(\mu_{\theta 1}) \\ 0 & 0 & 1 \end{bmatrix}^T + \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.2 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} = \begin{bmatrix} 0.5147 & -0.1588 & -0.0071 \\ -0.1588 & 0.3853 & 0.0082 \\ -0.0071 & 0.0082 & 0.1426 \end{bmatrix}$$

$$\mathbf{H}_2 = \begin{bmatrix} \frac{\bar{\mu}_{x2} - x_L}{\sqrt{(x_L - \bar{\mu}_{x2})^2 + (y_L - \bar{\mu}_{y2})^2}} & \frac{\bar{\mu}_{y2} - y_L}{\sqrt{(x_L - \bar{\mu}_{x2})^2 + (y_L - \bar{\mu}_{y2})^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -0.6012 & -0.7991 & 0 \\ 0 & 0 & 1.0000 \end{bmatrix}$$

$$\mathbf{K}_2 = \bar{\boldsymbol{\Sigma}}_2\mathbf{H}_2^T(\mathbf{H}_2\bar{\boldsymbol{\Sigma}}_2\mathbf{H}_2^T + \mathbf{Q})^{-1} = \begin{bmatrix} -0.4812 & -0.0425 \\ -0.5596 & 0.0360 \\ -0.0016 & 0.7404 \end{bmatrix}$$

From the Kalman gain $\mathbf{K}_2$ and the matrix $\mathbf{H}_2$ we can then finish the correction step:

$$\begin{aligned} \boldsymbol{\mu}_2 =& \bar{\boldsymbol{\mu}}_2 + \mathbf{K}_2(\mathbf{z}_2 - h(\bar{\boldsymbol{\mu}}_2)) = \\ =& \begin{bmatrix} 0.5476 \\ 0.0694 \\ 0.2875 \end{bmatrix} + \begin{bmatrix} -0.4812 & -0.0425 \\ -0.5596 & 0.0360 \\ -0.0016 & 0.7404 \end{bmatrix} \left(\begin{bmatrix} 2.1 \\ 0.2 \end{bmatrix} - h\left(\begin{bmatrix} 0.5476 \\ 0.0694 \\ 0.2875 \end{bmatrix} \right) \right) = \begin{bmatrix} 0.6991 \\ 0.2466 \\ 0.2973 \end{bmatrix} \end{aligned}$$

and then $\boldsymbol{\Sigma}_2$

$$\begin{aligned} \boldsymbol{\Sigma}_2 &= (\mathbf{I} - \mathbf{K}_2\mathbf{H}_2)\bar{\boldsymbol{\Sigma}}_2 \\ &= \left(\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} -0.4812 & -0.0425 \\ -0.5596 & 0.0360 \\ -0.0016 & 0.7404 \end{bmatrix} \begin{bmatrix} -0.4424 & -0.8968 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \cdot \\ &\quad \begin{bmatrix} 0.5147 & -0.1588 & -0.0071 \\ -0.1588 & 0.3853 & 0.0082 \\ -0.0071 & 0.0082 & 0.1426 \end{bmatrix} = \begin{bmatrix} 0.4266 & -0.2607 & -0.0021 \\ -0.2607 & 0.2661 & 0.0018 \\ -0.0021 & 0.0018 & 0.0370 \end{bmatrix} \end{aligned}$$

Note how after one iteration the covariance matrix $\boldsymbol{\Sigma}_2$ is no longer diagonal, and after two iterations all its off-diagonal elements are different from 0, indicating that the components are no longer uncorrelated.

8.8 Mapping as an Estimation Problem

So far in this chapter, estimation algorithms have been mostly used to solve problems related to localization, i.e., estimating the pose of the robot in a *known map*. At the beginning of

the chapter, we clarified that when we use the sensor model $p(z|x)$ in localization algorithms, we implicitly assume that the map of the environment is known. For instance, in the last example, we assumed the location of the landmark was known, i.e., we assumed knowledge of the environment where the robot operates. However, one can also swap the perspective and use estimation algorithms to determine the map of the environment under the assumption that the pose of the robot is known. This problem, too, can be cast as an estimation question and is known as *mapping*, which will be further expanded in Chapter 9.

We should clarify upfront that in many practical scenarios, the assumption that the pose of the robot is known does not hold. Nevertheless, this problem is interesting for two reasons. First, in some specific cases, one can ensure that the pose of the robot is known (e.g., by using some external infrastructure, such as RTK GPS), and then use the mapping algorithm to build a representation of the environment. Second, the mapping problem is also an estimation problem of independent interest. As the name suggests, mapping algorithms are estimation algorithms aimed at building a spatial representation (i.e., a map) of the environment. Different spatial models have been proposed and used by the robotics community over the years. In this section, we only consider occupancy grids, while more models will be introduced in Chapter 9. In the following, we present an algorithm to build occupancy grid maps starting from sequences of poses and sensor readings. An occupancy grid is a spatial representation where the space is divided into a regular grid of equally sized cells (see Figure 6.1 for an example). Accordingly, the map m can be seen as a collection of n cells. To each grid cell, we associate a binary random variable indicating whether the cell is occupied or not. The value 1 indicates that the cell is occupied, and let m_i be the binary random variable associated with the ith cell in the map. The mapping problem can then be seen as the problem of estimating the joint posterior distribution for the n binary random variables associated with the n cells in the map. Before getting into the analytic derivation, it is instructive to consider a simple idealized case to gain some intuition. Consider the situation sketched in Figure 8.13, where a robot at a known position x_t is equipped with a laser range finder. To simplify the discussion, let us consider just one of the readings returned by the sensor, and let us call it z_t. The grid cells in the map can be divided into three groups.

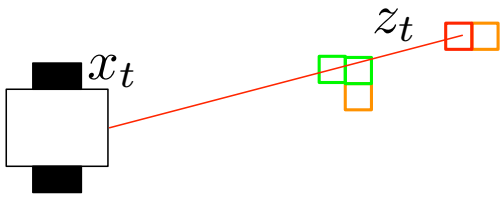


Figure 8.13: Mapping principle

For some cells, like those depicted in orange in the figure, the sensor reading z_t carries no information. For these cells, the probability $p(m_i = 1)$ should therefore not be updated based on this sensor reading. For other cells, like those shown in green, the sensor information indicates that $p(m_i = 1)$ should be lowered. In this case, the probability should be decreased because the laser beam traverses the cells; however, in general, this probability update needs to be tailored to the specific sensor being used. Finally, for some cells, like the one in red, $p(m_i = 1)$ should be increased because the sensor reading indicates that this cell is occupied. As with the cells whose probability is lowered, the nature of the sensor clarifies

why the probability should be increased (in this case, because the beam is reflected by the cell, indicating the presence of an obstacle). For a range finder, the above reasoning should be repeated for all readings returned by the sensor.

Moreover, note that by knowing the pose of the robot, the geometry of the sensor, and the values it returns, it is possible to partition all grid cells in the map into one of the three above groups. An outstanding question in the above discussion is by how much the probability should be either increased or lowered when an update is necessary. This question is related to the specific sensor being used. Additionally, due to the fact that sensors are affected by errors, no conclusion should be drawn from a single sensor reading; that is, a single reading indicating that a cell is occupied should not be used to infer that $p(m_i = 1) = 1$ (or $p(m_i = 1) = 0$ if the sensor indicates the cell is free).

We next turn these intuitions into a recursive Bayesian estimation algorithm. The following discussion closely follows the presentation in [53] (chapter 4 and chapter 9) and [11] (chapter 9), and the reader is referred to these excellent references for more details. As is immediately evident, considering the map as a joint collection of n grid cells scales poorly with the size of the map, because with n grid cells one can have 2^n possible maps, and it would be computationally too demanding to compute the posterior of each of those maps. Therefore, a standard assumption to simplify the problem is that all grid cells are independent of each other, and we consequently develop an algorithm to estimate the posterior of a single grid cell. This algorithm will then be applied to all grid cells in the map.

Taking a Bayesian approach, our objective is to estimate the following posterior:

$$p(m_i = 1 | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_n)$$

where m_i is the ith grid cell in map m. Note that in this case we do not consider the inputs given to the robot because we assume the pose is known, so the inputs carry no additional information. As in our previous estimation problems, we start by applying Bayes' rule and simplifying the following expressions by making suitable independence assumptions. For brevity, we write $p(m_i)$ for $p(m_i = 1)$:

$$p(m_i | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_n) = \tag{8.25}$$

$$= \frac{p(z_n | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1}, m_i) p(m_i | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})}{p(z_n | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})} \tag{8.26}$$

$$= \frac{p(z_n | x_n, m_i) p(m_i | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})}{p(z_n | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})} \tag{8.27}$$

$$= \frac{p(m_i | x_n, z_n) p(z_n | x_n) p(m_i | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})}{p(m_i) p(z_n | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})} \tag{8.28}$$

$$= \frac{p(m_i | x_n, z_n) p(z_n | x_n) p(m_i | x_1, x_2, \ldots, x_{n-1}, z_1, z_2, \ldots, z_{n-1})}{p(m_i) p(z_n | x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})} \tag{8.29}$$

The rationale of the derivation is as follows. Eq. (8.26) is obtained from Eq. (8.25) by applying Bayes' rule with background knowledge. Next, Eq. (8.27) is obtained from Eq. (8.26) by noting that the sensor reading at time n (z_n) depends only on the grid cell m_i

and the pose at time n (x_n), but not on the previous sensor readings or previous poses. We then apply Bayes' rule again to the term $p(z_n|x_n, m_i)$, i.e., $p(z_n|x_n, m_i) = \frac{p(m_i|x_n,z_n)p(z_n|x_n)}{p(m_i|x_n)}$. Further observing[7] that $p(m_i|x_n) = p(m_i)$, we obtain Eq. (8.28). Finally, to derive Eq. (8.29), we observe that x_n does not carry any information about m_i if we do not know z_n, so we drop it. Without additional assumptions, this expression cannot be further simplified, and it still includes some terms we do not know. To circumvent these issues, we go through the same derivation to estimate $p(m_i = 0)$. Of course, $p(m_i = 0) = 1 - p(m_i = 1)$, but the derivation is still convenient to simplify the unknown terms in Eq. (8.29). In the following, for brevity, we write $p(\neg m_i)$ for $p(m_i = 0)$. Following exactly the same derivation we discussed above, we get:

$$p(\neg m_i|x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_n) = \tag{8.30}$$

$$= \frac{p(\neg m_i|x_n, z_n)p(z_n|x_n)p(\neg m_i|x_1, x_2, \ldots, x_{n-1}, z_1, z_2, \ldots, z_{n-1})}{p(\neg m_i)p(z_n|x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})} \tag{8.31}$$

At this point we take the ratio between the two estimates, i.e.,

$$\frac{p(m_i|x_1, x_2 \ldots, x_n, z_1, z_2, \ldots, z_n)}{p(\neg m_i|x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_n)}$$

and after substituting Eq. (8.29) and Eq. (8.30) we notice that the terms $p(z_n|x_n)$ and $p(z_n|x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_{n-1})$ cancel out. This leads to the following expression:

$$\frac{p(m_i|x_1, \ldots, x_n, z_1, \ldots, z_n)}{p(\neg m_i|x_1, \ldots, x_n, z_1, \ldots, z_n)} = \frac{p(m_i|x_n, z_n)p(\neg m_i)p(m_i|x_1, \ldots, x_{n-1}, z_1, \ldots, z_{n-1})}{p(m_i)p(\neg m_i|x_n, z_n)p(\neg m_i|x_1, \ldots, x_{n-1}, z_1, \ldots, z_{n-1})} \tag{8.32}$$

At this point, recall that $p(m_i) = 1 - p(\neg m_i)$, and we further introduce the following quantity, defined for all events x whose probability is neither 0 nor 1:

$$Odds(x) = \frac{p(x)}{1 - p(x)}. \tag{8.33}$$

Using this definition, we can rewrite Eq. (8.32) as follows:

$$Odds(p(m_i|x_1, \ldots, x_n, z_1, \ldots, z_n)) = \frac{Odds(p(m_i|x_n, z_n)) \cdot Odds(p(m_i|x_1, \ldots, x_{n-1}, z_1, \ldots, z_{n-1}))}{Odds(m_i)}. \tag{8.34}$$

For computational reasons that will soon become evident, rather than directly working with the *Odds*, it is more practical to work with its logarithm, exploiting the fact that the log function is invertible. The last equation can therefore be rewritten as follows:

[7] One could speculate that knowing the robot pose is x_n tells us something about whether cell m_i is occupied or not. In particular, if m_i is within the robot's footprint at state x_n, the cell must be free. However, it is convenient to make this independence assumption, and it still provides an acceptable approximation.

$$\begin{aligned}
&\log Odds(p(m_i|x_1,\ldots,x_n,z_1,\ldots,z_n)) = \\
&\quad = \log\left(\frac{Odds(p(m_i|x_n,z_n))\cdot Odds(p(m_i|x_1,\ldots,x_{n-1},z_1,\ldots,z_{n-1}))}{Odds(m_i)}\right) \\
&\quad = \log Odds(p(m_i|x_n,z_n)) + \log Odds(p(m_i|x_1,\ldots,x_{n-1},z_1,\ldots,z_{n-1})) - \log Odds(m_i)
\end{aligned} \tag{8.35}$$

This last expression shows once again the recursive nature of the estimation algorithm. The $\log Odds$ obtained after integrating the last pose x_t and the last sensor reading z_t is computed by adding a term considering these quantities ($\log Odds(p(m_i|x_n,z_n))$) to the $\log Odds$ of the previous estimate ($\log Odds(p(m_i|x_1,\ldots,x_{n-1},z_1,\ldots,z_{n-1}))$). Note moreover that the estimate also depends on the $\log Odds$ of the prior, i.e., $\log Odds(m_i)$, and that this term will be 0 if the prior is $p(m_i) = 0.5$, i.e., if the prior information assigns equal probability that m_i is free or occupied.

Once the $\log Odds$ have been computed, $p(m_i)$ can be obtained by inverting Eq. (8.33). Putting everything together, then, the algorithm updates every cell in the map considering all the sensor readings received. As a consequence of the above derivation, it can be observed that $p(m_i)$ should never be exactly 0 or 1, because in either case the $\log Odds$ are undefined. Indeed, if $p(m_i) = 1$ or $p(m_i) = 0$, then there is no point in applying an estimation algorithm altogether, since there is no uncertainty. The previous derivation leads to an algorithm that can be used to update the posterior of a single cell in the occupancy grid map. In particular, it updates the estimate $\log Odds$ of the binary variable associated with the cell m_i, starting from the previous estimate, and the latest pose x_n and sensor reading z_n. Algorithm 12 sketches the solution. In the algorithm, l_n stores the estimate of $\log Odds$ at stage n for the whole map being built, and l_n^i is the ith component, i.e., the $\log Odds$ for m_i at time t.

Data: $x_n = (x, y, \vartheta)$, z_n, l_{n-1}, m_i
Result: l_n
1 **if** m_i *in perceptual range of* x_n, z_n **then**
2 $\quad l_n^i \leftarrow \log Odds(m_i|x_n,z_n) + l_{n-1}^i - \log Odds(m_i)$;
3 **else**
4 $\quad l_n^i \leftarrow l_{n-1}^i$;
5 **return** l_n;

Algorithm 12: Mapping with known poses

The algorithm first determines if the latest sensor reading z_n combined with the latest pose x_n can be used to update the posterior for m_i (line 1). If that is not the case, then the $\log Odds$ for m_i is not updated, and the previous value is simply copied (line 4). If instead the posterior needs to be updated, in line 2 the value is updated by applying Eq. (8.35). The update takes the form of the typical recursive formulation, where the new estimate is obtained by updating the previous estimate.

Further Reading

Estimation has a rich history. Classic textbooks on the topic include [3, 7, 52]. A more recent textbook on the topic is [48]. For a discussion more focused on estimation problems in robotics, the reader is referred to [11, 27, 53], and in particular to [53], as it is the leading reference for this topic. For smoothing, see [1]. The implementation of `ekf_localization` in ROS is discussed in [40].

Chapter 9

Localization and Mapping

9.1 Introduction

The filtering and estimation techniques introduced in Chapter 8 can be used to solve two important problems in robotics: localization and mapping. Basic examples illustrating the role of estimation algorithms in these problems were presented earlier when introducing the fundamentals. In this chapter, we delve deeper into these two essential problems and explain how they are addressed in ROS through a set of readily available nodes that can be easily integrated into new applications. While these nodes can be used as "black boxes," understanding the fundamental algorithms discussed in the previous chapter is crucial to appreciating the strengths and limitations of these nodes and configuring them effectively.

The following quotes from a 1991 paper [15] are often cited to emphasize the practical importance of localization in mobile robotics applications.

> "We believe that position estimation is a primary problem that must be solved for autonomous vehicles working in structured environments. [...]
>
> Using sensory information to locate the robot in its environment is the most fundamental problem to providing a mobile robot with autonomous capabilities."

Although these quotes are more than 30 years old, they remain valid and highly relevant to this day when developing software for mobile robots.

Localization is the problem of using sensory data to determine the position of a robot inside a given map. This corresponds to determining the pose of the robot (e.g., x, y, ϑ, assuming a mobile differential drive robot) with reference to an assigned frame. To this end, it may be useful to review the standard frames introduced in Section 4.13.5. Conversely, *mapping* is the problem of building a map of the environment assuming that the location of the robot is known. Hence, the two problems are intertwined and often studied together. However, it is necessary to clarify that *the* most important problem from a practical standpoint is the so-called *SLAM* problem, i.e., *simultaneous localization and mapping*, where the robot has to build a map and at the same time localize itself in the map being built. SLAM algorithms are substantially more complex than localization and mapping alone, and will not be covered here. However, we will discuss some ready-to-use ROS nodes that implement SLAM algorithms.

There exist numerous variations of the mapping and localization problems. With regard to localization, three different problems are commonly considered.

Tracking is the problem maintaining (tracking) a posterior about the robot pose given knowledge about its initial pose.

Global Localization is the problem of determining the robot pose given no prior information about the robot pose. After the robot has been successfully localized, then one could switch to tracking, although as we will see, some algorithms can seamlessly switch from one mode the other.

Kidnapped Robot Problem is the problem of determining the robot location given a wrong initial estimate about its pose. This problem is important because it addresses the ability of recovering from a complete localization failure. This variant is different from global localization because in the former the robot starts with a prior modeling lack of knowledge, whereas in the latter the robot starts with a wrong estimate about its pose.[1]

With regard to mapping, in addition to the occupancy grids introduced in previous chapters, several alternative models have been proposed and studied in the literature. The continued use of different spatial representations highlights that no single model is universally superior; rather, the choice of model is typically driven by the specific requirements of the application. The following list outlines some of the most commonly used models.

Occupancy grids: An occupancy grid divides the environment into a regular grid and assigns to each grid cell the probability that the cell is occupied—hence the name. The maps shown in Figures 6.1 and 8.9 are examples of occupancy grid maps built by a mobile robot. Occupancy grid maps are appealing for at least two reasons. First, they are very easy for humans to interpret because they resemble blueprints we are familiar with. Second, they are very practical for tasks like robot navigation, as they encode occupied areas that the robot should avoid to prevent collisions, as outlined when we discussed Nav2.

Feature based maps: A feature-based map includes the location of landmarks that the robot can detect. The location of each landmark is often represented probabilistically, e.g., using a Gaussian distribution. The type of landmarks depends on the sensor(s) used. For example, in an outdoor scenario, one could map light poles, trees, or similar features.

Appearance based maps: An appearance-based map consists of a graph where each vertex is associated with an image, and edges are added between similar vertices (i.e., images). To determine whether a vertex should be added, a similarity metric is used. Appearance-based maps can be used to perform tasks like visual navigation, though this topic will not be discussed further. An example is shown in Figure 9.1.

[1]The word *kidnapped* is used do describe this problem because this is what would happen if a correctly localized robot is lifted (kidnapped) and moved to a different location. In this case, once the robot is "freed," its most recent pose estimate would be wrong, but the robot would not be aware of it.

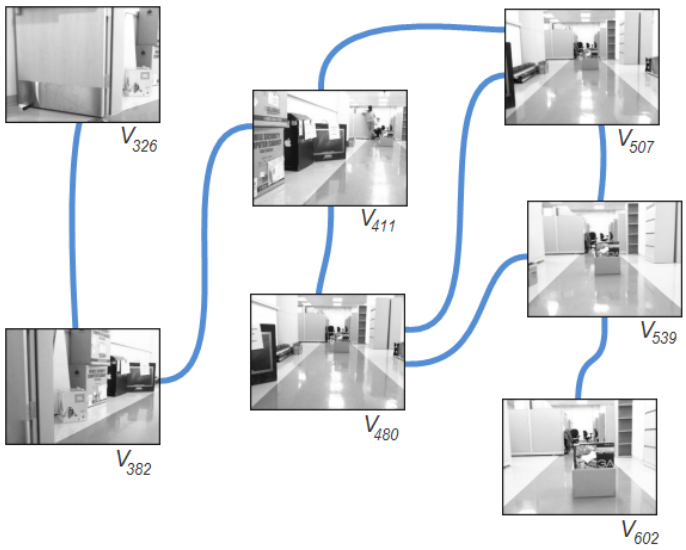


Figure 9.1: An example of an appearance-based map, where images are associated with vertices and edges are added between vertices whose images are sufficiently similar according to a given similarity metric. Image taken from [19].

Topological maps: A topological map consists of a graph where vertices are associated with *places*, and edges between two vertices are added when it is possible to move between the associated places.

Occupancy grid maps and feature based maps can also be classified as *metric maps* because they include metric information, such as distances between landmarks and other measurements.

9.2 Localization

Before delving into how localization algorithms are implemented in ROS, we begin with a deeper discussion of how the EKF can be used to solve the localization problem in a feature based map.

The EKF is most often used to solve the tracking problem, i.e., to maintain a Gaussian posterior given a Gaussian prior and a stream of inputs and sensor data. When using a standard EKF, it is important to remember that a basic implementation relies on a uni-modal posterior (the Gaussian distribution), and therefore it is not suitable for tracking multiple hypotheses. During the prediction step, the algorithm integrates the most recent inputs, while during the correction step, the provided map is used to compute the innovation and apply corrections. The architecture shown in Figure 9.2 (adapted from [30], page 60) illustrates how the EKF is used to localize a robot navigating in a feature based map with landmarks placed at known locations.

The *prediction* and *correction* steps have already been discussed in Section 8.7 and are exactly the same. However, in a feature-based map including multiple landmarks placed at known locations, there is potentially an additional aspect to consider, i.e., *data association* or

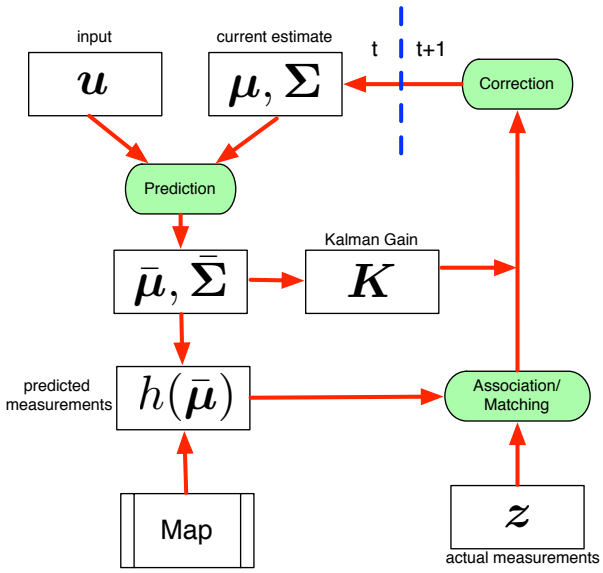


Figure 9.2: EKF localization cycle

matching. After the state has been predicted, by using the map and the sensor model h, it is possible to predict what the measurements would be if the state of the robot were equal to the predicted mean, i.e., $h(\boldsymbol{\mu})$. This is necessary to compute the innovation term. For example, assuming that the sensor could measure the distance between the robot and all the features in the map (say k features), the sensor would return a vector with k values. However, in a practical application the actual measurements could include only a subset of these k values, for example because some of the features are not detected due to distance or occlusions. Alternatively, the sensor could return the distances to all k features, but the order in which they are returned may not be known. This could be, for example, the case when the features are indistinguishable. The association step is needed to match the predicted measurements (or a suitable subset) to the actual measurements, so that the innovation $\mathbf{z} - h(\boldsymbol{\mu})$ can be properly computed. Association is, in general, a difficult problem to solve. For this reason, in many cases the environment is pre-conditioned so that all features are uniquely identifiable, and the association problem is therefore bypassed or simplified. However, in many circumstances it is not possible to modify the environment to ease this problem, and one has to deal with the data association challenge.

9.2.1 Pose tracking in a feature map with EKF

We show a concrete example where the EKF is used to solve the tracking problem. In this case, the robot moves according to the same motion model used in the example provided in Section 8.7.3, and the robot starts from the pose $(0, 0, 0)$. However, in this case, the robot is equipped with a sensor that returns the distances from three landmarks placed at known locations. The landmarks are distinguishable and always detectable, and therefore the data association problem does not arise. The sensor model is:

$$\mathbf{z}_t = h_t(\mathbf{x}_t) + \mathbf{w}_t = \begin{bmatrix} \sqrt{(x_L^1 - x_t)^2 + (y_L^1 - y_t)^2} \\ \sqrt{(x_L^2 - x_t)^2 + (y_L^2 - y_t)^2} \\ \sqrt{(x_L^3 - x_t)^2 + (y_L^3 - y_t)^2} \end{bmatrix} + \mathbf{w}_t$$

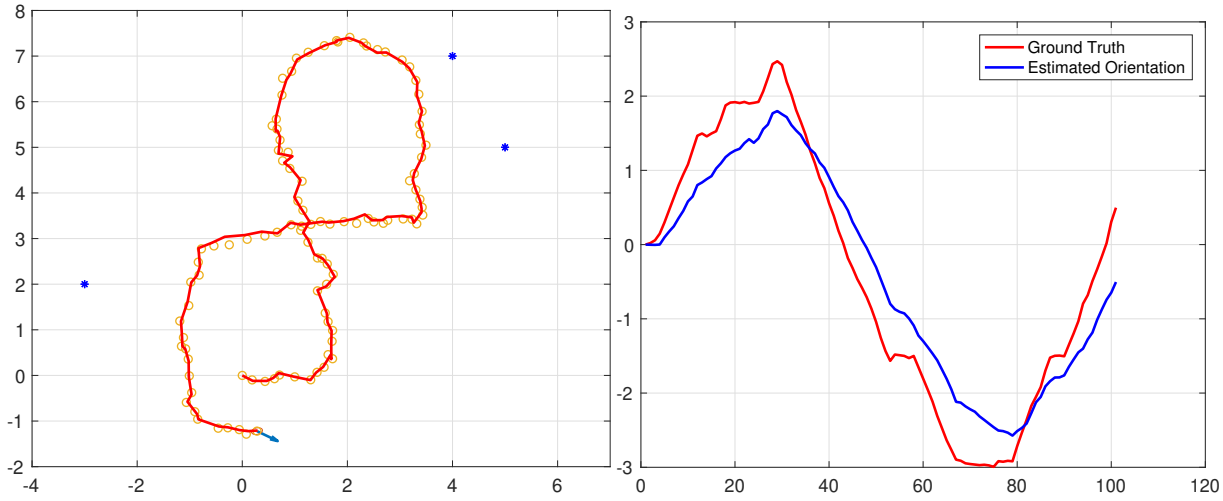
where (x_L^i, y_L^i) is the known location of the ith landmark. Since the landmarks have been assumed to be distinguishable, the ith entry in $\mathbf{z}_t$ is always the distance from (x_L^i, y_L^i). $\mathbf{v}_t$ is a zero mean Gaussian vector with constant covariance matrix

$$\mathbf{Q} = \begin{bmatrix} 0.001 & 0 & 0 \\ 0 & 0.001 & 0 \\ 0 & 0 & 0.001 \end{bmatrix}.$$

In this case the robot is not equipped with any sensor returning direct information about its heading, as ϑ does not appear in the expression for the sensor function h. The initial pose of the robot is Gaussian distributed with mean $\mu_0 = [0\ 0\ 0]$ and covariance matrix:

$$\mathbf{\Sigma}_0 = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 0.001 \end{bmatrix}.$$

We next illustrate the results of the estimation process for two different cases. In the first setup, inputs and measurements regularly alternate at every cycle, i.e., every prediction step is followed by a correction step. Figure 9.3 shows the results. On the left, we see the real path followed by the robot (red path), the estimated pose (yellow dots), and the locations of the three landmarks (blue stars). Throughout the trajectory, the estimated position remains fairly close to the ground truth. In the right figure, we plot the ground truth orientation as well as the estimated orientation. Note the *lag* between the actual trajectory and the estimated trajectory.



(a) Position Error (b) Orientation Error.

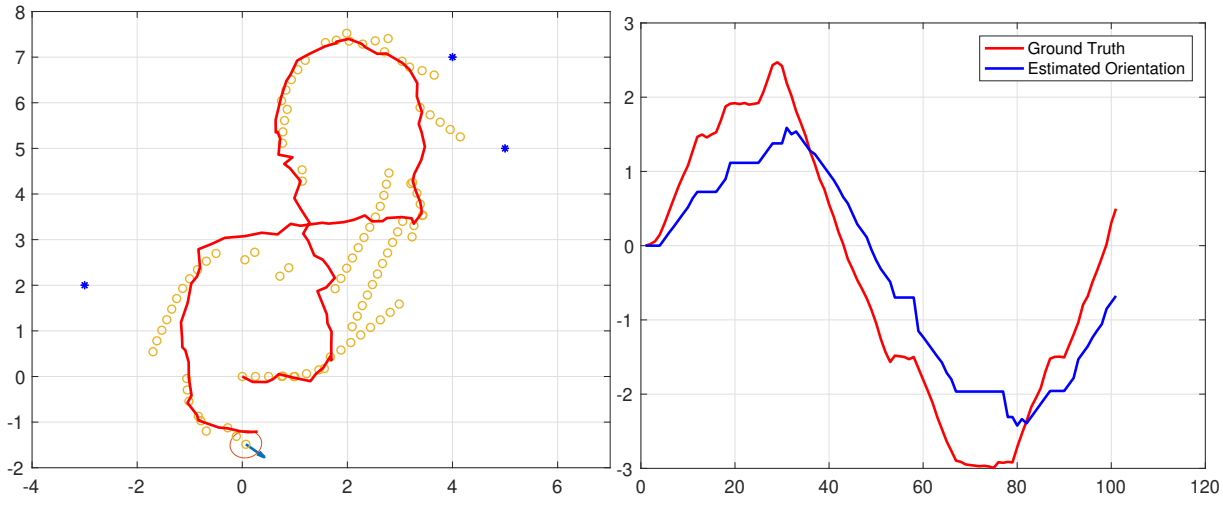
Figure 9.3: Position and orientation error for the case of alternating prediction and correction. In both charts the red line shows the ground truth.

It is interesting to examine the covariance matrix $\mathbf{\Sigma}$ at the end of the estimation process:

$$\mathbf{\Sigma} = \begin{bmatrix} 0.000979 & -0.000161 & 0.000119 \\ -0.000161 & 0.000502 & 0.000102 \\ 0.000119 & 0.000102 & 0.013212 \end{bmatrix}$$

The first important observation is that even though the initial covariance matrix $\mathbf{\Sigma}_0$ was diagonal, i.e., the covariances between all components (off-diagonal values) were 0, at the end of the estimation process this is no longer the case. That is, although the initial components of the state were independent Gaussian variables, in the end there is non-zero covariance between the state components. Hence, it is possible to (indirectly) estimate the orientation, even if there is no sensor directly measuring it. The other interesting observation pertains to the elements on the diagonal of $\mathbf{\Sigma}$, i.e., the variances of the individual components of the state being estimated. The variances of the first two components (x and y) are significantly smaller than the variance for the third component (ϑ), indicating greater uncertainty for the latter.

In the second case, shown in Figure 9.4, observations are instead received sporadically, i.e., the EKF does not regularly alternate prediction and correction, but rather performs multiple predictions in a row before applying a correction when a sensor reading is received. This can be clearly seen by the mismatch between the red curve and the yellow dots in Figure 9.4. In fact, the *linear sequences* of yellow dots are obtained in between corrections (predictions only), while the jumps occur when a sensor reading is received and the correction step is applied. In Figure 9.4, note the segments where the estimated orientation does not change (horizontal segments in the blue line). These correspond to cases where the robot is moving straight in between receiving sensor readings. In that situation, the EKF is performing prediction only, and according to the motion model, no change in orientation is made, resulting in constant orientation estimates.



(a) Position Error (b) Orientation Error.

Figure 9.4: Position and orientation error for the case with sporadic sensor readings. In both charts the red line shows the ground truth.

9.3 Extended Kalman Filter in ROS

The package `robot_localization` provides a ready-to-use implementation of the EKF[2] in a node called `ekf_node`. `ekf_node` implements a generic state estimation algorithm that can integrate an arbitrary number of sensors and inputs and can be applied to robots with different motion models. The node solves the estimation problem for robots moving in three-dimensional space and is therefore equally applicable to mobile wheeled robots and drones alike. The estimation node tracks the pose of a 15-dimensional vector, including position and orientation, their time derivatives (velocities), and the second derivatives (accelerations) of the position. Indicating with r, p, w the roll, pitch, and yaw orientation angles, the tracked state vector is $\mathbf{x} = [x, y, z, r, p, w, \dot{x}, \dot{y}, \dot{z}, \dot{r}, \dot{p}, \dot{w}, \ddot{x}, \ddot{y}, \ddot{z}]^T$. The EKF implementation in `ekf_node` follows the nonlinear estimation formulation presented in Section 8.7.2, i.e., it considers nonlinear models for both motion and sensing. The input vector $\mathbf{u}$ is assumed to be a six dimensional vector including linear and angular accelerations, i.e.,

$$\mathbf{u}_t = \begin{bmatrix} \ddot{x}_{t-1} \\ \ddot{y}_{t-1} \\ \ddot{z}_{t-1} \\ \ddot{r}_{t-1} \\ \ddot{p}_{t-1} \\ \ddot{w}_{t-1} \end{bmatrix}$$

while the state transition equation $\mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_t)$ is

$$f(\mathbf{x}_{t-1}) = \begin{bmatrix} f_x(\mathbf{x}_{t-1}, \mathbf{u}_t) \\ f_y(\mathbf{x}_{t-1}, \mathbf{u}_t) \\ f_z(\mathbf{x}_{t-1}, \mathbf{u}_t) \\ r_{t-1} + (\dot{r}_{t-1} + \sin r_{t-1} \tan p_{t-1} \dot{p}_{t-1} + \cos r_{t-1} \tan p_{t-1} \dot{w}_{t-1})\Delta t \\ p_{t-1} + (\cos r_{t-1} \dot{p}_{t-1} - \sin r_{t-1} \dot{w}_{t-1})\Delta t \\ w_{t-1} + (\sin r_{t-1} \frac{1}{\cos p_{t-1}} \dot{p}_{t-1} + \cos r_{t-1} \frac{1}{\cos p_{t-1}} \dot{w}_{t-1})\Delta t \\ \dot{x}_{t-1} + \ddot{x}_{t-1}\Delta t \\ \dot{y}_{t-1} + \ddot{y}_{t-1}\Delta t \\ \dot{z}_{t-1} + \ddot{z}_{t-1}\Delta t \\ \dot{r}_{t-1} + \ddot{r}_{t-1}\Delta t \\ \dot{p}_{t-1} + \ddot{p}_{t-1}\Delta t \\ \dot{w}_{t-1} + \ddot{w}_{t-1}\Delta t \\ \ddot{x}_{t-1} \\ \ddot{y}_{t-1} \\ \ddot{z}_{t-1} \end{bmatrix}$$

[2]To be precise, the package also provides another node implementing a filter called the *unscented Kalman filter*. However, we will not discuss it here because we have not covered its theoretical foundations.

and

$$\begin{aligned} f_x(\mathbf{x}_{t-1}, \mathbf{u}_t) = x_{t-1} &+ \cos y_{t-1} \cos p_{t-1} \dot{x}_{t-1} \Delta t + \\ &(\cos y_{t-1} \sin p_{t-1} \sin r_{t-1} - \sin y_{t-1} \cos r_{t-1}) \dot{y}_{t-1} \Delta t + \\ &(\cos y_{t-1} \sin p_{t-1} \cos r_{t-1} + \sin y_{t-1} \sin r_{t-1}) \dot{z}_{t-1} \Delta t + \\ &\frac{1}{2} \left(\dot{x}_{t-1} \ddot{x}_{t-1} + \dot{y}_{t-1} \ddot{y}_{t-1} + \dot{z}_{t-1} \ddot{z}_{t-1} \right) \Delta t \end{aligned}$$

$$\begin{aligned} f_y(\mathbf{x}_{t-1}, \mathbf{u}_t) = y_{t-1} &+ \sin y_{t-1} \cos p_{t-1} \dot{x}_{t-1} \Delta t + \\ &(\sin y_{t-1} \sin p_{t-1} \sin r_{t-1} + \cos y_{t-1} \cos r_{t-1}) \dot{y}_{t-1} \Delta t + \\ &(\sin y_{t-1} \sin p_{t-1} \cos r_{t-1} - \cos y_{t-1} \sin r_{t-1}) \dot{z}_{t-1} \Delta t + \\ &\frac{1}{2} \left(\dot{x}_{t-1} \ddot{x}_{t-1} + \dot{y}_{t-1} \ddot{y}_{t-1} + \dot{z}_{t-1} \ddot{z}_{t-1} \right) \Delta t \end{aligned}$$

$$\begin{aligned} f_z(\mathbf{x}_{t-1}, \mathbf{u}_t) = z_{t-1} &- \sin p_{t-1} \dot{x}_{t-1} \Delta t + \\ &\cos p_{t-1} \sin r_{t-1} \dot{y}_{t-1} \Delta t + \\ &\cos p_{t-1} \cos r_{t-1} \dot{z}_{t-1} \Delta t + \\ &\frac{1}{2} \left(\dot{x}_{t-1} \ddot{x}_{t-1} + \dot{y}_{t-1} \ddot{y}_{t-1} + \dot{z}_{t-1} \ddot{z}_{t-1} \right) \Delta t \end{aligned}$$

The reader may be taken aback by the complex expressions appearing in the definition of $f(\mathbf{x}_{t-1}, \mathbf{u}_t)$. Of course, it is not necessary to remember these relationships to use the node, but the formulas show that the EKF implementation allows the prediction step to work regardless of the motion model governing the robot. The prediction step considers how the pose, orientation, and velocity of a rigid body moving in three dimensions evolve over time, under the assumption that the body is subject to external forces and torques generating linear and angular accelerations, as per our assumption of $\mathbf{u}$. Derivations for these formulas can be found in textbooks focusing on robot mechanics, such as [31,49]. The Jacobian matrix of the nonlinear sensing function (see Eq.(8.24)) is assumed to contain only zeros and ones. This means that the sensors should directly return the state variables being estimated. For a sensor returning all 15 variables, $\mathbf{H}$ would be the identity matrix. Sensors returning only a subset of the 15 variables being tracked (say m) can be easily handled by zeroing out the components associated with the variables not measured, thus yielding an $m \times 15$ matrix. For example, to integrate inputs from a sensor such as odometry providing x, y, w (i.e., the 2D pose), the matrix $\mathbf{H}$ would be

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

This structure ensures that only the three components returned by the sensor will be updated. Likewise, if one has an IMU returning $\dot{x}, \dot{y}, \dot{z}$, the metrix would be

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Fortunately, the programmer does not have to deal with updating these matrices, as this can be easily configured through a text configuration file described later on. Similarly, the same configuration file can be used to set the values of the covariance matrices for disturbances, i.e., $\mathbf{Q}$ and $\mathbf{R}$. Note that while the general EKF formulation presented in Chapter 8 allows for time-varying matrices, the ROS implementation instead assumes these are fixed, and therefore we have dropped the subscript t since there is no time dependency.

The standard way to run the EKF node in ROS is to put all parameters in a YAML configuration file and then pass the file to the node, e.g.,

```
ros2 run robot_localization ekf_node --ros-args --params-file ekfnode.yaml
```

Because the EKF node can be configured in a very complex way, the configuration file can quickly become quite large. The most practical approach is to start with the template file `ekf.yaml` provided as part of the `robot_localization` package and adjust it for the specific configuration being used. The reader is referred to this file and to the online documentation for a detailed explanation of the many parameters. A key aspect to consider when setting up the filter is that it can receive inputs from an arbitrary number of devices providing data through topics of type:

- `geometry_msgs::PoseWithCovarianceStamped`
- `geometry_msgs::TwistWithCovarianceStamped`
- `nav_msgs::Odometry`
- `sensor_msgs::Imu`.

At this point, it should be clear how the data provided by these sensor streams fit into the equations shown at the beginning of this section. The output from the filter is published to a topic of type `nav_msgs::Odometry` and typically called `odom/filtered`, though note that this may be part of a namespace and therefore appear with a different fully qualified name. To see an instance of `ekf_node` in action, you can run the following command:

```
ros2 launch gazeboenvs husky_orchard.launch.py
```

With `ros2 node info`, we see that the `ekf_node` processes data from the odometry sensor and one IMU. The estimate is published to the topic `/a201_0000/platform/odom/filtered` as the launch file defines its own namespace. Echoing the output topic of the EKF, one can observe that, consistent with EKF theory, the node outputs both the mean and the covariance of the estimate, and these are updated as the robot is moved around with the GUI appearing on the right side of the Gazebo window. Importantly, the filter provides the estimate for the pose of the `base_link` frame relative to the `odom`. This is consistent with the conventions mentioned at the end of Chapter 4, when it was stated that poses expressed in the `odom` should evolve with continuity. `ekf_node` also provides various services. Among these, `SetPose` is used to set the tracked state during the estimation process, `toggle` is turn the filter on or off, and `reset` is to reset it (i.e., forget all past history and restart). `ros2 interface show` can be used to see how to interact with these services.

Remark 9.1. *The* ***ekf_node*** *implementation does not allow direct replication of the pose tracking approach presented in Section 9.2.1 because it cannot integrate sensor readings that return distances from landmarks located at known positions. To make this information available to* ***ekf_node****, one should convert it into a suitable message, e.g., by turning variations in measured distances from landmarks into estimates of motion encoded in a message of type* ***nav_msgs::Odometry****. Approaches known as* visual odometry *perform this conversion by estimating a robot's movement through the analysis of sequences of images from one or more cameras.*

9.4 Particle Filters in ROS

The `nav2_amcl` package provides a node called `amcl` that implements an algorithm called Adaptive Monte Carlo Localization (AMCL from now onwards). As discussed in the previous chapter, a careful implementation is needed to turn the basic particle filter idea into a robust localization algorithm. AMCL specializes the basic particle filter described in Section 8.5 with advanced features, such as the ability to adjust, on the fly, the size of the particle set (the technical details are beyond the scope of these notes). This allows starting with a large set of particles initially, for example when the robot's pose is subject to significant uncertainty, and then adaptively refining (typically reducing) this number as the task progresses and the estimate becomes more accurate. `amcl` works only for a robot moving in a planar environment, i.e., it estimates position and yaw orientation (x, y, ϑ). As of now, the only sensor used for localization is the laser scanner. To see the `amcl` particle filter in action, we can run the following Gazebo simulation

```
ros2 launch gazeboenvs tb4_simulation.launch.py use_rviz:=True
```

and in RViz it is possible to enable the visualization of particles[3], as shown in Figure 9.5.

`amcl` solves two types of localization problems. The first is *pose tracking*, i.e., given an initial estimate for the pose, it integrates sensor readings and given commands to propagate a posterior and maintain (*track*) the pose estimate. When solving the tracking problem, the initial set of particles is drawn from a Gaussian distribution with a given mean and covariance. The second problem it solves is the *global localization* problem, i.e., given no prior estimate of the robot pose, it attempts to estimate it. In this case, the initial set of particles is generated and spread across the environment using a uniform distribution.

Its behavior can be configured through a large number of parameters that can be set either in a YAML configuration file or through the parameter server.

9.4.1 Subscribed topics

To track the pose of the robot, `amcl` subscribes to `/scan` to receive sensor readings from the laser range finder. It also subscribes to `/map` to obtain the map of the environment and to `/tf` to receive the necessary transformations, e.g., the transformation between the range finder and the `base_link`. The fourth topic it subscribes to is `/initialpose`, where

[3]Note that until the robot is not moved the particles are not displayed.

Figure 9.5: Particle filter visualization in RViz. The green dots (which are actually arrows) show possible positions and orientations. As expected, particles are more dense near the actual location of the robot, indicating that pose tracking is working.

messages of type `geometry_msgs::PoseWithCovarianceStamped` are sent. As the name suggests, this topic can be used to initialize the initial set of particles by sampling from a Gaussian distribution with the given mean (pose) and covariance. If no node is publishing to this topic, `amcl` attempts to retrieve the initial pose from the parameter server, and if this also fails, the initial pose is set to $(0, 0, 0)$ with a diagonal covariance matrix containing fixed values. Particles are then generated by drawing samples from a Gaussian distribution centered on the initial pose and using the provided covariance matrix. This initialization step is intended to solve the aforementioned tracking problem, i.e., to create an initial set of particles drawn from a distribution that models preliminary knowledge about the robot's pose.

9.4.2 Published topics

`amcl` publishes the estimated robot pose to the topic `/amcl_pose`. Messages published to this topic are of type `geometry_msgs/PoseWithCovarianceStamped`, as each pose is associated with a covariance matrix and a timestamp. Importantly, the pose returned is referred to the frame `map`, indicating that the pose estimation may be discontinuous. To return a single pose from the current set of particles, `amcl_pose` divides the particles into clusters and assigns a weight to each cluster. The returned pose is then the mean of the particles in the cluster with the highest weight, along with its covariance. In addition, `amcl` also publishes the entire set of particles to the `/particlecloud` topic as a message of type `geometry_msgs/PoseArray`. The topic `/initialpose`, as the name suggests, can be used to initialize the particle filter with a message of type `geometry_msgs/PoseWithCovarianceStamped`.

9.4.3 Implemented services

`amcl` implements numerous services to asynchronously interact with the filter. The service `/reinitialize_global_localization` is used to reset the set of particles and distribute

them uniformly over free space. This service should be called to solve the global localization problem. Listing 9.1 shows how this service can be called.

Listing 9.1: Resetting `amcl` for global localization

```
#include <rclcpp/rclcpp.hpp>
#include <std_srvs/srv/empty.hpp>
#include <chrono>

using namespace std::chrono_literals;

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

    rclcpp::init(argc,argv);
    rclcpp::Node::SharedPtr nodeh;
    nodeh = rclcpp::Node::make_shared("resetparticles"); // create node

    rclcpp::Client<std_srvs::srv::Empty>::SharedPtr client =
    nodeh->create_client<std_srvs::srv::Empty>
                          ("/reinitialize_global_localization");
    // wait...
    while (! client->wait_for_service())
      RCLCPP_INFO(nodeh->get_logger(),"Waiting for service to be available");

    auto request = std::make_shared<std_srvs::srv::Empty::Request>();
    auto response = client->async_send_request(request);
    if (rclcpp::spin_until_future_complete(nodeh, response) ==
        rclcpp::FutureReturnCode::SUCCESS) {
      RCLCPP_INFO(nodeh->get_logger(), "Particles reset");
    }
    else // Error:
      RCLCPP_ERROR(nodeh->get_logger(),"Error while resetting particles");

    rclcpp::shutdown();
    return 0;
}
```

If you run this node with the previous simulation environment up and running and visualize the particles with RViz, you will see that after running the node, the particles are spread uniformly over the free space in the map. The service `/request_nomotion_update` should be called to force the node to perform an update even when no motion commands are being sent. The reason is that, to ensure stability, `amcl` suspends its updates when the robot is stationary. By invoking this service, it is possible to force the robot to perform a single update. It is therefore necessary to call this service repeatedly if continuous updates are desired while the robot is not moving. Both of these services receive requests of type `std_srvs::srv::Empty` because they do not require any input parameters and return no results.

9.4.4 Parameters

`amcl` relies on numerous parameters to define its functioning. These can be either set in the configuration file, or through a YAML file, similar to the EKF node. Two important

parameters are the minimum number of particles `min_particles` and maximum number of particles `max_particles`. If not explicitly set, they default to 500 and 2000, respectively. When the filter is initialized, it initially creates `max_particles` particles drawn from a Gaussian distribution, as per the above discussion, but they can be resampled from a uniform distribution over free space, using the `/reinitialize_global_localization` service.

9.5 SLAM in ROS

In Chapter 8, we showed how both the mapping and the localization problem can be cast as estimation problems. However, we also mentioned that, from a practical standpoint, a much more important problem is the so-called SLAM, where the robot at the same time must build a map and localize itself within it. SLAM is a much harder problem than localization, and we will not discuss the algorithmic details. The good news is that ROS provides a ready-to-use implementation of a 2D SLAM algorithm through the `slam_toolbox` package. To see the SLAM algorithm in action, you can run the following command:

```
ros2 launch gazeboenvs tb4_simulation.launch.py slam:=True use_rviz:=True
```

In this case, RViz starts with an empty map. However, if you move the robot using the RViz interface, you will see that a map begins to be incrementally built. Figure 9.6 shows an example.

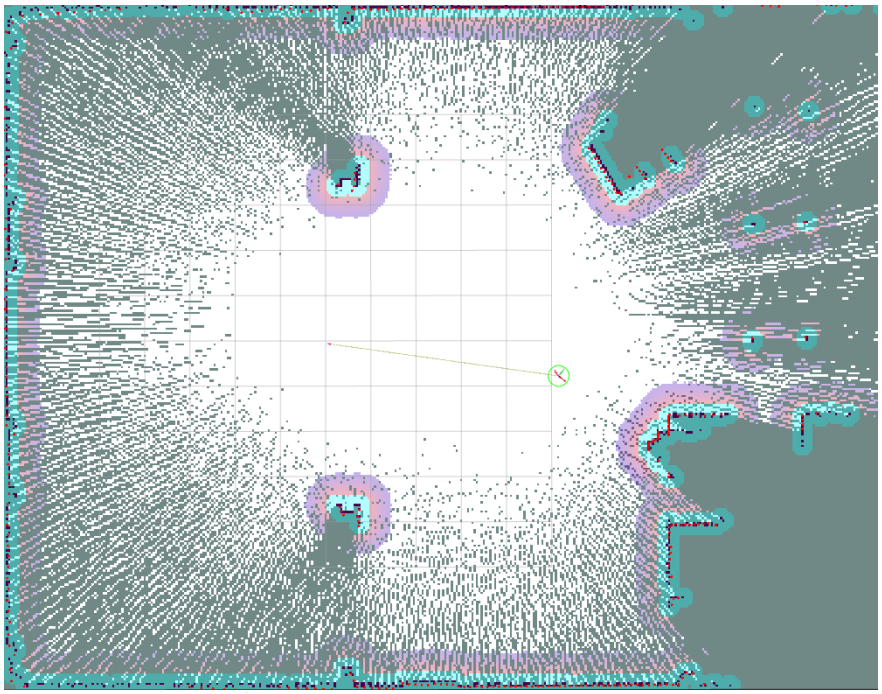


Figure 9.6: Partially built map obtained by the SLAM algorithm.

The `slam_toolbox` node subscribes to `/scan` to receive data from the range finder and to `/tf` to obtain the location of the sensor on the robot. The node publishes both

the map (of type `nav2_msgs::OccupancyGrid`) to the topic `/map` and the pose (of type `geometry_msgs::PoseWithCovarianceStamped`) to the topic `/pose`. It also exposes numerous services for tasks such as saving the map, resetting the algorithm, and more. A complete list can be obtained with `ros2 node info /slam_toolbox`.

Remark 9.2. *The examples we have seen illustrate how the algorithms work in practice and how it is possible to interact with them, e.g., retrieving the pose produced by* ***`ekf_node`*** *or the map from* ***`slam_toolbox`****. These examples relied on having Nav2 properly configured for the robot being used, either in simulation or in the real world. Configuring Nav2 from scratch for a new robot can be a time-consuming task, although the Nav2 website provides detailed tutorials on how to do it. On the positive side, given the widespread adoption of ROS, most commercial platforms now come with Nav2 already configured, allowing one to skip the setup process entirely.*

Further Reading

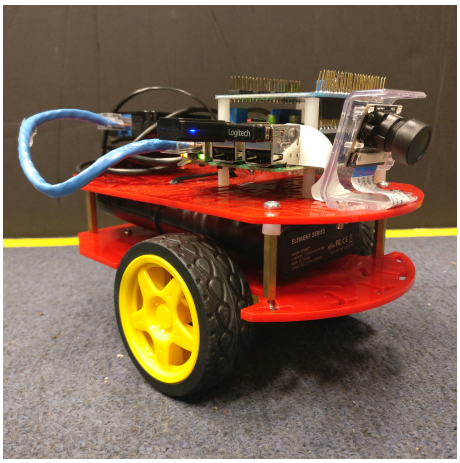
An earlier implementation of EKF in ROS is discussed in [39] and it provides informative examples showing how different sensors perform individually and through fusion. The SLAM toolbox is described in [34].

Literature in localization, mapping and SLAM is vast and has greatly evolved through the years, albeit one can say that in numerous settings the SLAM problem can be considered solved. A classic reference explaining the fundamentals is [53], although much progress has been made since the book appeared. Another classic introductory tutorial is [5, 18].

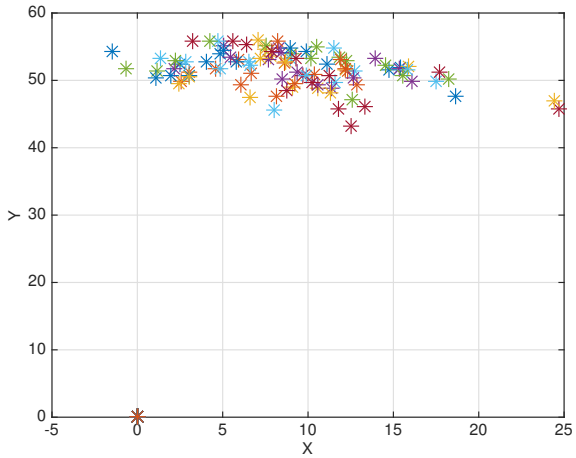
Appendix A

Probability

Robots operate in the physical world and are subject to countless sources of uncertainty. Some uncertain outcomes are due to the laws of physics and unavoidable mechanical limitations. Figure A.1 illustrates this problem. The DuckieBot (shown in Figure A.1a) executed the maneuver *move forward 50 centimeters* 100 times, always starting from the position $(0, 0)$. Figure A.1b shows the distribution of the positions at the end of the maneuver.



(a) The DukieBot



(b) Endpoints obtained repeating the same maneuver 100 times.

Figure A.1: The DukieBot and its trajectories. The robot always starts at $(0, 0)$ pointing upwards. The plotted dots are the position of the mid point between the two wheels.

In this case, the distribution is clearly skewed to the right because one of the two motors was not working properly. While the figure depicts a case that may appear extreme, such problems are far from unusual. Similar behaviors are, for example, observed when the wheels are inflated to different pressures (and thus have different radii), or when the surface is irregular, etc. This uncertainty cannot be avoided, although it can be limited with careful (or more expensive) design. However, it can never be completely eliminated. Other events that can add uncertainty are *external*, e.g., one of the motors may unexpectedly stop working, causing the robot to fail to move altogether. Similar phenomena are experienced when

considering robot sensors. For example, if we repeatedly query a GPS receiver to determine our latitude and longitude, even without moving, the returned location typically fluctuates, and occasionally one can see "jumps" of a few meters.

Consequently, to develop truly robust robot systems capable of successfully completing their assigned missions despite these uncertainties, it is necessary to develop algorithms that explicitly account for these sources of unpredictability. The theory of probability provides the right framework to formulate and solve these problems. In this chapter, we provide a short recap of the main concepts necessary to develop the contents presented in this book. The reader is referred to the references at the end of the chapter for more details.

A.1 Sets and Algebras

We assume the reader is familiar with the concepts of set, union, complement, etc.

Definition A.1 (Algebra)**.** *Given a set Ω, an* algebra *on Ω is a collection $\mathcal{A}$ of subsets of Ω satisfying the following conditions:*

1. $\varnothing \in \mathcal{A}$*: the empty set is in* $\mathcal{A}$*.*
2. $A \in \mathcal{A} \Rightarrow \bar{A} \in \mathcal{A}$*:* $\mathcal{A}$ *is closed under complement.*
3. $A, B \in \mathcal{A} \Rightarrow A \cup B \in \mathcal{A}$*:* $\mathcal{A}$ *is closed under finite union.*

From the above properties, we can show that an algebra is also closed under intersection.

Theorem A.1. *Let $\mathcal{A}$ be an algebra on Ω and let $A, B \in \mathcal{A}$. Then, $A \cap B \in \mathcal{A}$.*

Proof. The theorem can be easily proven using one of De Morgan's laws, i.e., $\overline{A \cap B} = \bar{A} \cup \bar{B}$. We start by observing that

$$A \cap B = \overline{\overline{A \cap B}} = \overline{\overline{A} \cup \overline{B}}$$

where the last equality follows from De Morgan's law. Next, since we assumed $A, B \in \mathcal{A}$, then $\overline{A}, \overline{B} \in \mathcal{A}$ by property 2, and $\overline{A} \cup \overline{B} \in \mathcal{A}$ by property 3. Finally, the complement of this union is also in $\mathcal{A}$ by property 2, and therefore we have shown that $A \cap B \in \mathcal{A}$. □

Example A.1. *Given a set Ω, the collection of subsets $\mathcal{A} = \{\varnothing, \Omega\}$ is the simplest algebra on Ω. It is straightforward to verify that it satisfies all three conditions stated above.*

Example A.2. *Let $\Omega = \{1, 2, 3, 4, 5, 6\}$. The following collection of subsets of Ω is an algebra on Ω:*

$$\mathcal{A} = \{\varnothing, \Omega, \{1, 3, 5\}, \{2, 4, 6\}\}.$$

Example A.3. *Let Ω be a set with a finite number of elements. Then $\mathcal{A} = 2^{\Omega}$ is an algebra on Ω (recall that 2^{Ω} is the power set of Ω).*

Definition A.2 (Partition)**.** *Let Ω be a set. A collection of sets $A_1, \ldots, A_n$ is a* partition *of Ω if the following properties hold:*

1. $A_i \subseteq \Omega$ *for* $1 \leq i \leq n$*;*
2. $A_i \cap A_j = \varnothing$ *for all* $i \neq j$*;*
3. $A_1 \cup A_2 \cup \cdots \cup A_n = \Omega$*.*

A.2 Probability Space

Definition A.3 (Probability Space)**.** *A* probability space[1] *is given by* $(\Omega, \mathcal{A}, P)$ *where:*

- Ω *is a set (usually called* sample set*; its elements are referred to as* elementary outcomes*);*
- $\mathcal{A}$ *is an algebra on* Ω *(usually called* event space*);*
- $P : \mathcal{A} \to \mathbb{R}$ *is a function (called* probability distribution *on* Ω*) subject to the following constraints:*
  1. $P(A) \geq 0$ *for each* $A \in \mathcal{A}$*;*
  2. $P(\Omega) = 1$*;*
  3. *if* $A, B \in \mathcal{A}$ *and* $A \cap B = \varnothing$*, then* $P(A \cup B) = P(A) + P(B)$*.*

$P(A)$ is called *the probability of the event* A. The above properties also imply that if $A \subseteq B$, then $P(A) \leq P(B)$. An *experiment* is defined by the sample set Ω, a set of relevant events $\mathcal{A}$, and the probability distribution P. In the following and in the rest of the book we often write $P(A, B)$ as a shorthand for $P(A \cap B)$ where A and B are two events over the same probability space.

Example A.4. *Let us define the probability space associated with the experiment of tossing an unfair coin. To this end, we need to define the sample set* Ω*, its algebra* $\mathcal{A}$ *and the probability distribution* P*.*

1. $\Omega = \{H, T\}$ *(heads, tails)*
2. $\mathcal{A} = \{\varnothing, \Omega, \{H\}, \{T\}\}$
3. $P(\Omega) = 1$, $P(\{H\}) = 0.25$, $P(\{T\}) = 0.75$, $P(\varnothing) = 0$

It is immediate to verify that $(\Omega, \mathcal{A}, P)$ *satisfy the properties specified in the definition of probability space.*

Remark A.1. *In the previous example, we described the experiment of tossing an* unfair *coin. Normally, when tossing a fair coin, we assume that the probability of obtaining heads is equal to the probability of obtaining tails, i.e., 0.5. In our case, however, we assumed a higher probability of getting heads. From the perspective of defining a probability space, both scenarios are valid, as long as the probability distribution* P *satisfies the properties outlined in the definition of a probability space.*

Fully defining the probability space by enumerating all events and their probabilities can be a tedious task when $\mathcal{A}$ contains many elements. This is for example the case when $\mathcal{A} = 2^{\Omega}$ and Ω has more than a handful of elements. In this case one can define P only for the elements in Ω (elementary outcomes) subject to the following constraints:

[1] To be precise this definition is not correct, because we should impose additional properties on $\mathcal{A}$ (we should require it is a so-called σ-algebra).

1. $P(\omega) \geq 0$ for each $\omega \in \Omega$;
2. $\sum_{\omega_i \in \Omega} P(\omega_i) = 1$.

It is easy to verify that starting from these two properties one can build a probability distribution on Ω satisfying the definition of probability space.

Example A.5. *Let us define the probability space associated with the experiment of rolling a fair die. According to the latest observation we define P only for the elements in Ω.*

1. $\Omega = \{1, 2, 3, 4, 5, 6\}$
2. $\mathcal{A} = 2^{\Omega}$
3. $P(\{1\}) = P(\{2\}) = P(\{3\}) = P(\{4\}) = P(\{5\}) = P(\{6\}) = \frac{1}{6}$

These definitions allow to answer questions regarding any event, i.e., any element in $\mathcal{A}$. For example, we can determine the probability of rolling the die and getting an odd face. The associated event is $A_{odd} = \{1, 3, 5\}$. Note that by virtue of our definition of $\mathcal{A}$ we have $A_{odd} \in \mathcal{A}$. Its probability is immediately determined as

$$P(A_{odd}) = P(\{1, 3, 5\}) = P(\{1\}) + P(\{3\}) + P(\{5\}) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}.$$

A.3 Basic Probability Facts

From now on, unless otherwise specified, when talking about multiple events we will implicitly assume they belong to the same probability space.

Definition A.4 (Conditional Probability)**.** *Let A, B be two events with $P(B) > 0$. We define the* conditional probability *of A given B as*

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{A.1}$$

Because we assume $P(B) > 0$, this quantity is well defined. Moreover, we can equivalently write

$$P(A \cap B) = P(A|B)P(B).$$

Figure A.2 provides the intuition behind this definition.

The probability of an event A can be seen as the ratio between its measure and the measure of the sample space Ω. Conditioning on an event B amounts to defining a new, restricted sample space whose measure is $P(B)$. In this new space, the event A "shrinks" to the part compatible with B, namely $A \cap B$. The ratio between the measure of $A \cap B$ and the measure of B is then the probability of the remaining part of A after conditioning, which is exactly the formula in Eq. (A.1).

Eq. (A.1) can be extended to cases with more than one conditioning event. If B and C are two events such that $P(B \cap C) > 0$ (compatible events), then we define

$$P(A|B, C) = \frac{P(A \cap B \cap C)}{P(B \cap C)}.$$

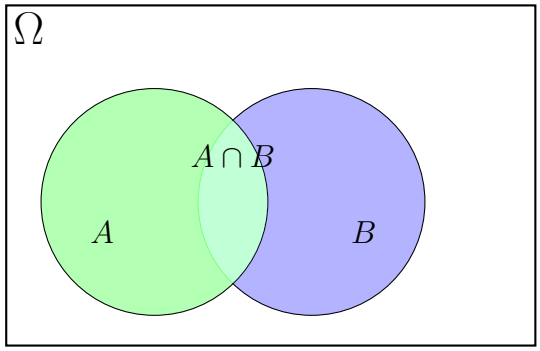


Figure A.2: Interpretation of the definition of conditional probability.

Following the same reasoning, it is possible to define the conditional probability for any finite number of conditioning events.

Example A.6. *Consider again the probability space introduced in Example A.5 for the experiment* rolling a fair die. *Let us consider the events* "the face is 4", "the face is 1", *and* "the face is even", *denoted respectively by*

$$A = \{4\}, \quad B = \{1\}, \quad C = \{2, 4, 6\}.$$

From our definition of P, we have $P(A) = P(B) = \frac{1}{6}$, while $P(C) = \frac{1}{2}$.

Applying Eq. (A.1), *we get the probability that the face is 4 given that it is even:*

$$P(A|C) = \frac{P(A \cap C)}{P(C)} = \frac{P(\{4\})}{\frac{1}{2}} = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3}.$$

Similarly, the probability that the face is 1 given that it is even is:

$$P(B|C) = \frac{P(B \cap C)}{P(C)} = \frac{P(\varnothing)}{\frac{1}{2}} = \frac{0}{\frac{1}{2}} = 0.$$

Two theorems that we will use repeatedly later are the total probability theorem and Bayes' theorem.

Theorem A.2 (Total probability theorem)**.** *Let B be an event and $A_1, \ldots, A_n$ a partition of Ω. Then we have*

$$P(B) = P(B|A_1)P(A_1) + P(B|A_2)P(A_2) + \cdots + P(B|A_n)P(A_n). \tag{A.2}$$

Proof. The proof follows from elementary set properties:

$$\begin{aligned} P(B) &= P(B \cap \Omega) = P(B \cap (A_1 \cup A_2 \cup \cdots \cup A_n)) \\ &= P((B \cap A_1) \cup (B \cap A_2) \cup \cdots \cup (B \cap A_n)) \\ &= P(B \cap A_1) + P(B \cap A_2) + \cdots + P(B \cap A_n) \\ &= P(B|A_1)P(A_1) + P(B|A_2)P(A_2) + \cdots + P(B|A_n)P(A_n). \end{aligned}$$

□

Note that using Eq. (A.1) we can equivalently rewrite Eq. (A.2) as follows:

$$P(B) = P(B \cap A_1) + P(B \cap A_2) + \cdots + P(B \cap A_n). \tag{A.3}$$

Theorem A.3 (Bayes' theorem)**.** *Assume $P(B|A)$ and $P(A|B)$ are well defined (i.e., $P(A) > 0$ and $P(B) > 0$). Then we have*

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{A.4}$$

Proof. We just need to apply the definitions and elementary algebra:

$$P(B|A) = \frac{P(A \cap B)}{P(A)} \Rightarrow P(A \cap B) = P(B|A)P(A).$$

Substituting into the definition of $P(A|B)$ we obtain the desired result:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}.$$

□

In estimation algorithms, the terms appearing in Eq. (A.4) are usually given specific names: $P(A|B)$ is called *posterior*, $P(A)$ is called *prior*, $P(B|A)$ is called *likelihood* and $P(B)$ is called *evidence*. Bayes' theorem can also be rewritten using the total probability theorem to express $P(B)$, i.e.,

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} = \frac{P(B|A)P(A)}{P(B|A_1)P(A_1) + P(B|A_2)P(A_2) + \cdots + P(B|A_n)P(A_n)}.$$

Bayes' theorem can also be formulated in the so-called *Bayes' theorem with background knowledge*, i.e., conditioning all terms in Eq. (A.4) on a third event (say C):

$$P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)} \tag{A.5}$$

Note that the role of the conditioning variables in Eq. (A.5) can be swapped, i.e., from a formal standpoint one can also write

$$P(A|B, C) = \frac{P(C|A, B)P(A|B)}{P(C|B)}.$$

The same reasoning can be applied when there are n conditioning events $B_1, B_2, \ldots, B_n$, i.e., one can write

$$P(A|B_1, B_2, \ldots, B_n) = \frac{P(B_1|A, B_2, \ldots, B_n)P(A|B_2, \ldots, B_n)}{P(B_1|B_2, \ldots, B_n)}.$$

Note that the choice of events to keep as background knowledge is arbitrary, i.e., one could equally rewrite the same equation by swapping B_1 with any of $B_2, \ldots, B_n$. All these relationships will be extremely useful when dealing with Bayes filters (Chapter 8).

Definition A.5 (Independent events)**.** *Two events A and B are said to be* independent *if* $P(A \cap B) = P(A)P(B)$.

An immediate consequence of this definition is that if A is independent from B, then $P(A|B) = P(A)$, as can be immediately verified from the definition of $P(A|B)$ in Eq. (A.1). Note that the concept of independent events is purely mathematical. When the probability space is set up to model a physical experiment, we may be able to determine from domain knowledge that two events are independent. However, in many robotics algorithms, certain events will be assumed independent even if this is not apparent from the physical setup. In those cases, independence is assumed to simplify expressions and derive more efficient algorithms, and experience shows that this approximation is most often negligible.

Definition A.6 (Conditional Independence)**.** *Two events A and B are conditionally independent given an event C if*

$$P(A \cap B|C) = P(A|C)P(B|C).$$

It is important to note that independence and conditional independence are not equivalent. That is, two independent events A and B may no longer be independent when a third event C is given. Conversely, two events conditionally independent given C may no longer be independent when C is not the conditioning event.

A.4 Random Variables

Let $(\Omega, \mathcal{A}, P)$ be a probability space and let $X : \Omega \to \mathbb{R}$ be a function from the sample set into the real numbers. For $k \in \mathbb{R}$ we define the set

$$\{X \leq k\} = \{\omega \in \Omega \ : \ X(\omega) \leq k\}.$$

By definition, for each $k \in \mathbb{R}$ it follows that $\{X \leq k\} \subseteq \Omega$, i.e., it is a set of elementary outcomes.

Definition A.7 (Random Variable)**.** *Let $(\Omega, \mathcal{A}, P)$ be a probability space. A function $X : \Omega \to \mathbb{R}$ is a* random variable *over $(\Omega, \mathcal{A}, P)$ if it satisfies the following condition:*

- *for each $k \in \mathbb{R}$, the set $\{X \leq k\}$ is an event over Ω, i.e., an element of $\mathcal{A}$.*

Example A.7. *Consider again the probability space $(\Omega, \mathcal{A}, P)$ defined in Example A.5. Let $X : \Omega \to \mathbb{R}$ be the function defined as follows: $X(1) = 4$, $X(2) = -1$, $X(3) = 6$, $X(4) = 0$, $X(5) = 10$, $X(6) = 4$. In this case, by inspection we can construct $\{X \leq k\}$ for any $k \in \mathbb{R}$. For example, for $k = 5$*

$$\{X \leq k\} = \{1, 2, 4, 6\}.$$

More generally, it is immediate to verify that X is a random variable, because for each $k \in \mathbb{R}$ the set $\{X \leq k\}$ is an event in $\mathcal{A}$. This is trivially true because we defined $\mathcal{A} = 2^{\Omega}$, i.e., $\mathcal{A}$ includes all possible subsets of Ω.

Starting from the previous definition, we can introduce the *cumulative function* of a random variable.

Definition A.8 (Cumulative Distribution Function)**.** *Given a random variable X over $(\Omega, \mathcal{A}, P)$, the cumulative distribution function $F_x : \mathbb{R} \to [0, 1]$ of X is defined as*

$$F_x(a) = P(\{X \leq a\}).$$

Since we assumed that X is a random variable, we have $\{X \leq a\} \in \mathcal{A}$, and therefore $P(\{X \leq a\})$ is well defined because P is defined for each event in $\mathcal{A}$. Moreover, if $a < b$ then $\{X \leq a\} \subseteq \{X \leq b\}$, and therefore $F_x(a) \leq F_x(b)$, i.e., the cumulative distribution function is a non-decreasing function. Similarly, we can show that

$$\lim_{a \to -\infty} F_x(a) = 0 \qquad \lim_{a \to +\infty} F_x(a) = 1.$$

Example A.8. *Let us compute the cumulative distribution function for the random variable defined in Ex. A.7. The function is piecewise constant and defined as follows:*

$$F_x(a) = \begin{cases} 0 & \text{for } a < -1 \\ \frac{1}{6} & \text{for } -1 \leq a < 0 \\ \frac{1}{3} & \text{for } 0 \leq a < 4 \\ \frac{2}{3} & \text{for } 4 \leq a < 6 \\ \frac{5}{6} & \text{for } 6 \leq a < 10 \\ 1 & \text{for } a \geq 10 \end{cases}$$

To see why the function has this expression, we observe that $\{X \leq a\}$ changes only when a assumes the critical values $-1, 0, 4, 6,$ and 10. Applying the definitions: for $a < -1$, one has $\{X \leq a\} = \varnothing$ and therefore $F_x(a) = P(\varnothing) = 0$. For $-1 \leq a < 0$, one has $\{X \leq a\} = \{2\}$, so $F_x(a) = P(\{2\}) = \frac{1}{6}$. Following the same reasoning, for $0 \leq a < 4$, $\{X \leq a\} = \{2, 4\}$, so $F_x(a) = P(\{2, 4\}) = \frac{1}{3}$, and so on. Finally, for $a \geq 10$, $\{X \leq a\} = \Omega$ and then $F_x(a) = P(\Omega) = 1$.

Definition A.9 (Continuous Random Variable)**.** *A random variable X is* continuous *if F_x is a continuous function.*

Definition A.10 (Discrete random variable)**.** *A random variable X is* discrete *if F_x is piecewise constant, i.e., it has a countable number of discontinuities and it is constant in between discontinuities.*

The random variable we considered in Ex. A.7 and Ex. A.8 is therefore discrete. Figure A.3 shows the cumulative function for a discrete (left) and continuous (right) random variable.

Definition A.11 (Alphabet of a Discrete Random Variable)**.** *Let X be a discrete random variable. The set of discontinuity points of its cumulative function is called* alphabet*.*

Note that the alphabet can be either finite or infinite.

Example A.9. *The random variable defined in Example A.7 is discrete and its alphabet consists of 5 elements, i.e., $\{-1, 0, 4, 6, 10\}$.*

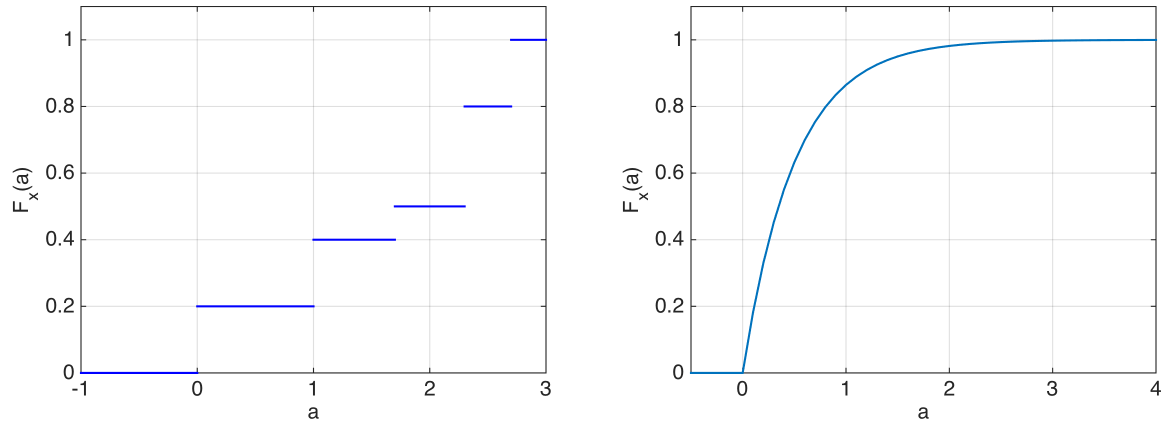


Figure A.3: Left: Cumulative function of a discrete variable. Right: Cumulative function of a continuous variable.

Definition A.12 (Probability Density Function)**.** *Let X be a continuous random variable. Then its* probability density function *(PDF from now onwards) is*[2]

$$f_x(a) = \frac{dF_x(a)}{da}.$$

Since $f_x(a)$ is the derivative of F_x and F_x is a non decreasing function, it follows that $f_x(a) \geq 0$ for each $a \in \mathbb{R}$. It is easy to show that $\int_{-\infty}^{+\infty} f_x(a)da = 1$ and moreover

$$P(a \leq X(\omega) \leq b) = \int_a^b f_x(a)da.$$

Note: in the following (for brevity) we will often write f instead of f_x.

Definition A.13 (Probability Mass Function)**.** *Let X be a discrete random variable and let $N = \{a_1, a_2, \dots\}$ be its alphabet. The* probability mass function *(PMF from now onwards) is a function $p : N \to [0, 1]$ defined as*

$$p_x(a_i) = F_x(a_i) - \lim_{a \to a_i^-} F_x(a).$$

If X is a discrete random variable, then for $a \in \mathbb{R}$ we will often write

$$p_x(X = a) = p_x(a) = p_a = P(\{\omega \in \Omega\} \mid X(\omega) = a).$$

From the basic probability properties if follows that $p_x(a) \geq 0$ for each $a \in N$. Using the above definitions and the properties of probability spaces, it is immediate to show the *normalization constraint*:

$$\sum_{a \in N} p_x(a) = 1.$$

We will often use the following random variables.

[2]We are implicitly defining the derivative where it exists. Once again, in the interest of space, we simplify the exposition by omitting some technical details.

Gaussian Random Variable

A random variable X is *Gaussian* or *Normal* if its PDF has the form

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Note that the PDF depends on two parameters, μ and σ^2. Parameter μ is the expectation of the random variable (see following) and σ^2 is its variance (its positive square root σ is its standard deviation). Often, we write $X \sim \mathcal{N}(\mu, \sigma^2)$ to indicate that X is a Normal random variable with expectation μ and variance σ^2. Figure A.4 shows the PDF and the cumulative density function for the standardized Gaussian variable $X \sim \mathcal{N}(0, 1)$.

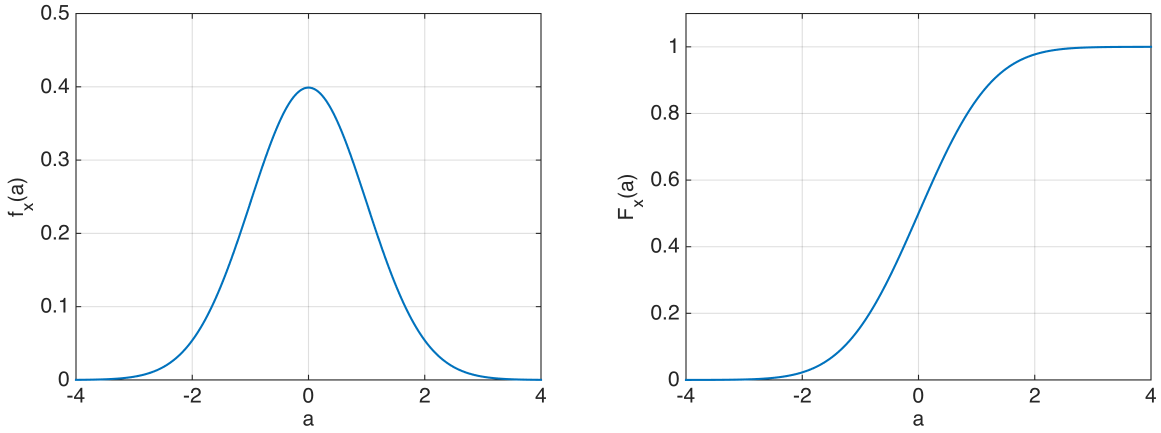


Figure A.4: Left: PDF of the standard random variable $\mathcal{N}(0, 1)$. Right: Cumulative density function of the standard random variable $\mathcal{N}(0, 1)$

Gaussian random variables are extensively used for multiple reasons. First, due to a result known as *central limit theorem*, the sum of multiple independent identically distributed random variables tends to be Gaussian distributed when the number of variables diverges. This is, for example, the case when multiple independent sensor readings are collected by a robot to gather information about its environment. Moreover, in numerous situations (e.g., the Kalman filter described in Chapter 8), analytic derivations can be simplified when the underlying distributions are assumed to be Gaussian. Finally, for a given variance value, Gaussian distributions provide the largest amount of uncertainty. Therefore, in uncertain scenarios with scarce prior information, choosing a Gaussian distribution is a good option to model lack of prior knowledge.

Uniform Random Variable

A random variable X is *uniform* if its PDF has the form

$$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases}.$$

We write $X \sim \mathcal{U}(a, b)$ to indicate that X is a uniform random variable whose density is non zero in the interval (a, b). Figure A.5 shows the PDF of a uniform random variable with $a = 1$ and $b = 3$.

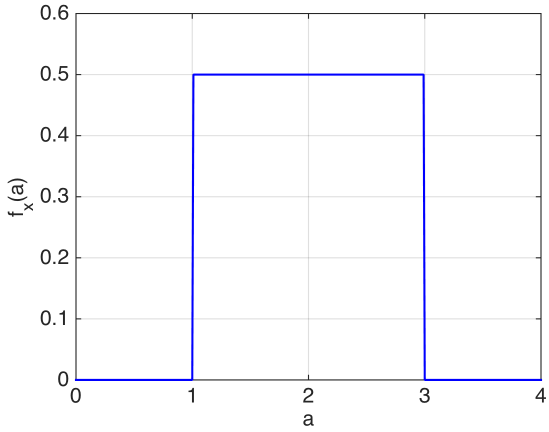


Figure A.5: PDF of a uniform random variable with $a = 1$ and $b = 3$.

Exponential Random Variable

A random variable X is *exponential* if its PDF has the form

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ \lambda e^{-\lambda x} & \text{if } x \geq 0 \end{cases}.$$

We write $X \sim \mathcal{E}(\lambda)$ to indicate that X is an exponential random with parameter λ. Figure A.6 shows the PDF of an exponential random variable for different values of λ.

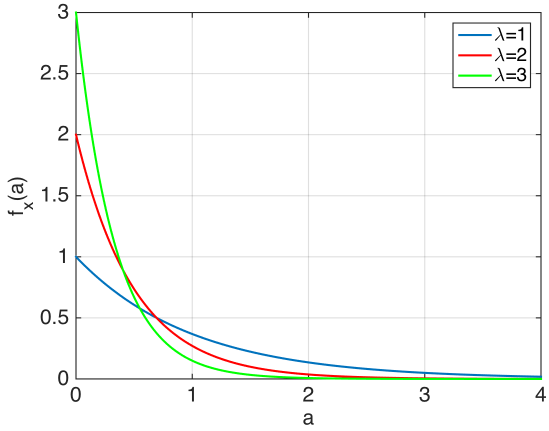


Figure A.6: PDF of an exponential random variable for different values of the λ parameter.

From the previous definitions it should be immediately clear that if X is a random variable and g is a real-valued function, then $Y = g(X(\omega))$ is also a random variable. This is true

both when X is discrete or continuous. The PMF or PDF of X is in general different from Y's (unless g is the identity function).

Example A.10. *Let $(\Omega, \mathcal{A}, P)$ be a probability space and X be a random variable over Ω. The real-valued function $g : \mathbb{R} \to \mathbb{R}$ defined as $g(x) = b$ defines a new random variable $Y(\omega) = g(X(\omega))$ for each $b \in \mathbb{R}$.*

It is immediate to show that Y is a discrete random variable with $N = \{b\}$. In fact, for each $k < b$ by definition we have

$$\{Y \leq k\} = \{\omega \in \Omega \ : \ Y(\omega) \leq k\} = \varnothing.$$

Vice versa, for each $k \geq b$ we have

$$\{Y \leq k\} = \{\omega \in \Omega \ : \ Y(\omega) \leq k\} = \Omega.$$

From these two equations it follows that $F_y(a) = 0$ for each $a < b$ and $F_y(a) = 1$ for each $a \geq b$. Consequently, $N = \{b\}$, and $p_Y(b) = 1$.

The above example shows that any constant real number can be seen as a random variable. Therefore, in the following we can compute quantities like expectation and variance also for constants.

A.5 Expectation of a Random Variable

Definition A.14 (Expectation)**.** *Let X be a random variable. If the following integral exists, then the* expectation *of X is defined as*

$$\mathbb{E}[X] = \int_{-\infty}^{+\infty} x f(x)\, dx.$$

If X is a discrete random variable, then the expression simplifies to

$$\mathbb{E}[X] = \sum_{\omega \in \Omega} X(\omega) P(\{\omega\}) = \sum_{x \in N} x\, p(x).$$

Example A.11. *From the definition it is immediate to verify that the expectation of a constant is equal to the constant itself, i.e., $\mathbb{E}[b] = b$. To see this, recall Example A.10, showing that a constant can be seen as a discrete random variable:*

$$\mathbb{E}[X] = \sum_{x \in N} x\, p(x) = \sum_{x \in \{b\}} x\, p(x) = b\, p(b) = b \cdot 1 = b.$$

For a random variable X, we will usually write μ_X for its expectation, i.e., $\mu_X = \mathbb{E}[X]$. Since the expectation of a random variable is defined through an integral, and the integral is a linear operator, it follows that expectation is also a linear operator. In particular, if we consider a new random variable $Y = aX + b$, we immediately obtain:

$$\mathbb{E}[Y] = \mathbb{E}[aX + b] = a\mathbb{E}[X] + \mathbb{E}[b] = a\mu_X + b.$$

If instead we consider a generic transformation $Y = g(X)$, it is possible to show that

$$\mathbb{E}[Y] = \int_{-\infty}^{+\infty} g(x) f(x)\, dx$$

for the continuous case, and

$$\mathbb{E}[Y] = \sum_{x \in N} g(x)\, p(x)$$

for the discrete case. This result is also known as the *fundamental theorem of expectation*.

With some calculations, it is possible to show that:

- If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $\mathbb{E}[X] = \mu$.
- If $X \sim \mathcal{U}(a, b)$, then $\mathbb{E}[X] = \frac{a+b}{2}$.
- If $X \sim \mathcal{E}(\lambda)$, then $\mathbb{E}[X] = \frac{1}{\lambda}$.

A.6 Variance of a Random Variable

Definition A.15 (Variance)**.** *Let X be a random variable with expectation μ_X. If the following integral exists, then the* variance *of X is defined as*

$$\mathrm{VAR}[X] = \int_{-\infty}^{+\infty} (x - \mu_X)^2 f(x)\, dx.$$

Similarly to what we saw for the expectation, for a discrete random variable the expression simplifies to

$$\mathrm{VAR}[X] = \sum_{\omega \in \Omega} (X(\omega) - \mu_X)^2 P(\{\omega\}) = \sum_{x \in N} (x - \mu_X)^2 p(x).$$

Example A.12. *From the definition it is immediate to verify that the variance of a constant is 0, i.e.,* $\mathrm{VAR}[b] = 0$.

The variance of a random variable X is also denoted as σ_X^2, whereas its positive square root σ_X is called the *standard deviation*.

Theorem A.4.

$$\mathrm{VAR}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

Proof. Just apply the definition and linearity of expectation:

$$\begin{aligned}
\mathrm{VAR}[X] &= \int_{-\infty}^{+\infty} (x - \mu_X)^2 f(x)\, dx = \int_{-\infty}^{+\infty} (x^2 - 2x\mu_X + \mu_X^2) f(x)\, dx \\
&= \int_{-\infty}^{+\infty} x^2 f(x)\, dx - 2\mu_X \int_{-\infty}^{+\infty} x f(x)\, dx + \mu_X^2 \int_{-\infty}^{+\infty} f(x)\, dx \\
&= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2.
\end{aligned}$$

□

As we have done for the expectation, it is useful to consider the variance of a variable $Y = aX + b$.

Theorem A.5. *Let X be a random variable and $Y = aX + b$ be a random variable obtained from X through an affine transformation. Then*

$$\mathrm{VAR}[Y] = a^2\mathrm{VAR}[X].$$

Proof.

$$\begin{aligned}\mathrm{VAR}[Y] &= \mathrm{VAR}[aX + b] = \mathrm{VAR}[aX] + \mathrm{VAR}[b] = \mathrm{VAR}[aX] \\ &= \int_{-\infty}^{+\infty} (ax - a\mu_X)^2 f(x)\,dx = a^2 \int_{-\infty}^{+\infty} (x - \mu_X)^2 f(x)\,dx = a^2\mathrm{VAR}[X].\end{aligned}$$

□

With some calculations, it is possible to show that:

- If $X \sim \mathcal{N}(\mu, \sigma^2)$, then $\mathrm{VAR}[X] = \sigma^2$.
- If $X \sim \mathcal{U}(a, b)$, then $\mathrm{VAR}[X] = \frac{(b-a)^2}{12}$.
- If $X \sim \mathcal{E}(\lambda)$, then $\mathrm{VAR}[X] = \frac{1}{\lambda^2}$.

A.7 Multiple Random Variables

Given a probability space $(\Omega, \mathcal{A}, P)$, it is possible to define more than one random variable over it. Assume X and Y are two random variables over $(\Omega, \mathcal{A}, P)$. Then we can consider the sets $\{X \leq k_1\}$ and $\{Y \leq k_2\}$ for $k_1, k_2 \in \mathbb{R}$. Since both X and Y are random variables, these sets are elements of the event space $\mathcal{A}$. The joint cumulative function is defined as

$$F_{xy}(k_1, k_2) = P(\{X \leq k_1\} \cap \{Y \leq k_2\}),$$

where the probability of the intersection is well defined because the intersection of two events is itself an event and hence part of $\mathcal{A}$.

From this definition, it is possible to define the joint PDF (if both X and Y are continuous) or the joint PMF (if both X and Y are discrete):

$$f_{xy}(k_1, k_2) = \frac{\partial^2 F_{xy}(k_1, k_2)}{\partial k_1 \partial k_2}.$$

From these definitions, it is immediate to define the concept of independent random variables by analogy with independent events.

Definition A.16 (Marginal Density Distribution)**.** *Let X and Y be two continuous random variables with joint PDF f_{xy}. The marginal PDF of X is defined as*

$$f_x(a) = \int_{\mathbb{R}} f_{xy}(a, b)\,db.$$

Similarly, the marginal PDF of Y is

$$f_y(b) = \int_{\mathbb{R}} f_{xy}(a, b)\,da.$$

Definition A.17 (Marginal Probability Mass Function)**.** *Let X and Y be two discrete random variables over the probability space $(\Omega, \mathcal{A}, P)$ with joint PMF p_{xy}. Let N_X be the alphabet of X and N_Y be the alphabet of Y. For each $a \in N_X$, the* marginal probability mass function *of X is defined as*

$$p_x(X = a) = \sum_{b \in N_Y} p_{xy}(X = a, Y = b).$$

Similarly, for each $b \in N_Y$, the marginal PMF of Y is

$$p_x(Y = b) = \sum_{a \in N_X} p_{xy}(X = a, Y = b).$$

The reader should notice the similarity between the formulas for marginalization and Eq. (A.3).

Definition A.18 (Independent Random Variables)**.** *Let X and Y be two random variables defined over the same probability space $(\Omega, \mathcal{A}, P)$. We say that X and Y are independent if for each $k_1 \in \mathbb{R}$ and $k_2 \in \mathbb{R}$, the events $\{X \leq k_1\}$ and $\{Y \leq k_2\}$ are independent.*

From this definition it follows that if X and Y are independent, then

$$F_{xy}(a_1, a_2) = F_x(a_1)F_y(a_2).$$

This additionally implies that if X and Y are continuous independent random variables, then

$$f_{xy}(a_1, a_2) = f_x(a_1)f_y(a_2),$$

and if X and Y are discrete random variables, then

$$p_{xy}(X = a, Y = b) = p_x(X = a)p_y(Y = b).$$

From these relationships, it is easy to show that if two random variables X and Y are independent, then

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y].$$

Definition A.19 (Covariance)**.** *Let X and Y be two random variables defined over the same probability space and let μ_X and μ_Y be their expectations. Their* covariance *is defined as*

$$\mathrm{COV}[X, Y] = \sigma_{xy} = \mathbb{E}[(X - \mu_X)(Y - \mu_Y)].$$

Note that by definition $\mathrm{COV}[X, Y] = \mathrm{COV}[Y, X]$. If X and Y are two random variables and $\mathrm{COV}[X, Y] = 0$, we say that X and Y are *uncorrelated*.

Theorem A.6. *Let X and Y be two independent random variables. Then* $\mathrm{COV}[X, Y] = 0$.

Proof. The theorem can be proved by applying the definition and recalling that the expectation of the product of two independent random variables is the product of their expectations:

$$\begin{aligned}
\mathrm{COV}[X, Y] &= \mathbb{E}[(X - \mu_X)(Y - \mu_Y)] = \mathbb{E}[XY - \mu_X Y - \mu_Y X + \mu_X \mu_Y] \\
&= \mathbb{E}[XY] - \mu_X \mathbb{E}[Y] - \mu_Y \mathbb{E}[X] + \mu_X \mu_Y \\
&= \mathbb{E}[X]\mathbb{E}[Y] - \mu_X \mu_Y - \mu_Y \mu_X + \mu_X \mu_Y \\
&= 0.
\end{aligned}$$

□

If $\text{COV}[X, Y] \neq 0$, its sign indicates whether the variables tend to vary in the same direction or not. Specifically, if $\text{COV}[X, Y] > 0$, then in expectation $X - \mu_X$ and $Y - \mu_Y$ have the same sign (either both positive or both negative), meaning the two variables tend to jointly increase or decrease. If $\text{COV}[X, Y] < 0$, they tend to move in opposite directions, i.e., when one increases the other decreases, and vice versa.

Remark A.2. *The previous theorem states that two independent random variables have 0 covariance. The converse, however, is not generally true: it is possible for two variables to have 0 covariance but not be independent.*

Definition A.20 (Correlation)**.** *Let X and Y be two random variables defined over the same probability space. Their* correlation *is defined as*

$$r_{xy} = \mathbb{E}[XY].$$

From the definition of covariance and correlation, it follows that

$$\text{COV}[X, Y] = r_{xy} - \mu_X \mu_Y.$$

Definition A.21 (Correlation coefficient)**.** *Let X and Y be two random variables defined over the same probability space. Their* correlation coefficient *is defined as*

$$\rho[X, Y] = \rho_{XY} = \frac{\text{COV}[X, Y]}{\sqrt{\text{VAR}[X]\text{VAR}[Y]}}.$$

It is easy to show that $-1 \leq \rho[X, Y] \leq 1$, and obviously $\rho[X, Y] = 0$ if X and Y are independent. The correlation coefficient measures *how strong* the linear relationship between the two variables is.

Theorem A.7. *$\rho[X, Y] = 1$ if and only if there exists a positive constant ζ such that $X - \mu_X = \zeta(Y - \mu_Y)$.*

If $\rho[X, Y] = -1$, the theorem still holds but for a negative constant. The expectation of the sum of two random variables is always

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] = \mu_X + \mu_Y.$$

The variance of the sum of two random variables can be written in this form only if X and Y are independent:

$$\text{VAR}[X + Y] = \text{VAR}[X] + \text{VAR}[Y] = \sigma_X^2 + \sigma_Y^2.$$

Definition A.22 (Jointly Gaussian Variables)**.** *Two Gaussian random variables X and Y are* jointly Gaussian distributed *if their joint PDF is*

$$f_{xy}(a_1, a_2) = \frac{1}{2\pi\sigma_X\sigma_Y\sqrt{1-\rho^2}} e^{\left\{-\frac{1}{2(1-\rho^2)}\left[\frac{(a_1-\mu_X)^2}{\sigma_X^2} - \frac{2\rho(a_1-\mu_X)(a_2-\mu_Y)}{\sigma_X\sigma_Y} + \frac{(a_2-\mu_Y)^2}{\sigma_Y^2}\right]\right\}}.$$

This PDF depends on five parameters: μ_X, μ_Y, σ_X^2, σ_Y^2, and ρ. The marginal distributions of X and Y are Gaussian with means μ_X and μ_Y and variances σ_X^2 and σ_Y^2, respectively. The parameter ρ is the correlation between X and Y.

A.8 Random Vectors

Let $(\Omega, \mathcal{A}, P)$ be a probability space and $\bar{X}$ a function $\bar{X} : \Omega \to \mathbb{R}^n$. Similarly to what we did for random variables, for $\mathbf{k} = [k_1\ k_2\ \dots\ k_n]^T \in \mathbb{R}^n$ we define the set

$$\{\bar{X} \leq \mathbf{k}\} = \{\omega \in \Omega : X_1(\omega) \leq k_1 \wedge X_2(\omega) \leq k_2 \wedge \dots \wedge X_n(\omega) \leq k_n\}$$

where $X_i(\omega)$ is the ith component of $\bar{X}(\omega)$.

Definition A.23 (Random Vector)**.** *Let $(\Omega, \mathcal{A}, P)$ be a probability space. An n-dimensional* random vector *is a function $\bar{X} : \Omega \to \mathbb{R}^n$ such that for each $\mathbf{k} \in \mathbb{R}^n$ the set $\{\bar{X} \leq \mathbf{k}\}$ is an element of $\mathcal{A}$.*

It is possible to show that in an n-dimensional random vector all its components are random variables, i.e.,

$$\bar{X} = [X_1\ X_2 \dots X_n]$$

where each of the X_i is random variables over $(\Omega, \mathcal{A}, P)$. Based on this definition, the cumulative function is defined similarly to the bidimensional case

$$F_{\bar{X}}(a_1, a_2, \dots, a_n) = P(\{X_1 \leq a_1, X_2 \leq a_2, \dots, X_n \leq a_n\})$$

As for the case of unidimensional variables, it is possible to define continuous and discrete random vectors. A random vector is continuous if $F_{\bar{X}}$ is continuous, and in this case the joint PDF can be defined as well

$$f_{\bar{X}}(x_1, x_2, \dots, x_n) = \frac{\partial^n F_{\bar{X}}(x_1, x_2, \dots, x_n)}{\partial x_1 \partial x_2 \dots \partial x_n}.$$

A random vector is instead discrete if its cumulative function is piecewise constant. As for the unidimensional case, the alphabet is the set of points in which the cumulative function is discontinuous. It is easy to show that a random vector is discrete if and only if its components are discrete random variables.

A.8.1 Expectation and Covariance of Random Vectors

Once the PDF or PMF has been defined for a random vector $\bar{X}$, it is straightforward to define its expectation. If $\bar{X}$ is an n-dimensional continuous random vector, then (provided the integral exists)

$$\mathbb{E}[\bar{X}] = \boldsymbol{\mu}_{\bar{X}} = \int_{\mathbb{R}^n} \mathbf{y}\, f_{\bar{X}}(\mathbf{y})\, d\mathbf{y},$$

and similarly, if $\bar{X}$ is a discrete random vector,

$$\mathbb{E}[\bar{X}] = \boldsymbol{\mu}_{\bar{X}}) = \sum_{\mathbf{y} \in N} \mathbf{y}\, p_{\bar{X}}(\mathbf{y}),$$

where N is the alphabet of $\bar{X}$ and $p_{\bar{X}}$ is its PMF. These formulas are simply the multidimensional generalizations of the corresponding unidimensional definitions.

The situation is slightly more complex for the variance.

Definition A.24 (Covariance matrix)**.** *Let $\bar{X}$ be an n-dimensional random vector with expectation $\boldsymbol{\mu}_X$. Its* covariance matrix *is the $n \times n$ matrix*

$$\mathrm{COV}[\bar{X}] = \boldsymbol{\Sigma}_{\bar{X}} = \mathbb{E}\big[(\bar{X} - \boldsymbol{\mu}_{\bar{X}})(\bar{X} - \boldsymbol{\mu}_{\bar{X}})^T\big]. \tag{A.6}$$

Note that for a random vector $\bar{X}$ we may write both $\mathrm{COV}[\bar{X}]$ and $\boldsymbol{\Sigma}_{\bar{X}}$ for its covariance matrix. The covariance matrix has certain structural properties. In particular, it is *symmetric*:

$$\boldsymbol{\Sigma}_{\bar{X}} = \boldsymbol{\Sigma}_{\bar{X}}^T,$$

and *positive semidefinite*, meaning that for every vector $\mathbf{x} \in \mathbb{R}^n$, the following inequality holds:

$$\mathbf{x}^T \boldsymbol{\Sigma}_{\bar{X}} \mathbf{x} \geq 0.$$

Since $\boldsymbol{\Sigma}_{\bar{X}}$ is positive semidefinite, we know from linear algebra that all its eigenvalues are nonnegative. By expanding Eq. (A.6) and performing some algebra, it is possible to see that the covariance matrix has the following structure:

$$\begin{aligned}
\boldsymbol{\Sigma}_{\bar{X}} &= \begin{bmatrix} \sigma_{X_1}^2 & \sigma_{X_1X_2} & \dots & \sigma_{X_1X_n} \\ \sigma_{X_2X_1} & \sigma_{X_2}^2 & \dots & \sigma_{X_2X_n} \\ \dots & \dots & \dots & \dots \\ \sigma_{X_nX_1} & \sigma_{X_nX_2} & \dots & \sigma_{X_n}^2 \end{bmatrix} \\
&= \begin{bmatrix} \sigma_{X_1}^2 & \rho_{X_1X_2}\sigma_{X_1}\sigma_{X_2} & \dots & \rho_{X_1X_n}\sigma_{X_1}\sigma_{X_n} \\ \rho_{X_1X_2}\sigma_{X_1}\sigma_{X_2} & \sigma_{X_2}^2 & \dots & \rho_{X_2X_n}\sigma_{X_2}\sigma_{X_n} \\ \dots & \dots & \dots & \dots \\ \rho_{X_nX_1}\sigma_{X_n}\sigma_{X_1} & \rho_{X_nX_2}\sigma_{X_n}\sigma_{X_2} & \dots & \sigma_{X_n}^2 \end{bmatrix}.
\end{aligned}$$

That is, the elements on the main diagonal of the covariance matrix are the variances of the components of the random vector $\bar{X}$, whereas the off-diagonal elements are the covariances between the components. More precisely, the generic element at position (i, j) with $i \neq j$ is $\mathrm{COV}[X_i, X_j]$. A special case occurs when the covariance matrix is diagonal; in this case, all covariances are zero, and the components of the vector are mutually uncorrelated. As with unidimensional random variables, it is useful to determine the expectation and covariance of a random vector obtained through an affine transformation of another random vector, e.g.,

$$\bar{Y} = \mathbf{A}\bar{X} + \mathbf{b}.$$

Note that $\bar{X}$ and $\bar{Y}$ do not need to have the same dimension, so $\mathbf{A}$ is not necessarily a square matrix. With this in mind, the expectation of $\bar{Y}$ is

$$\boldsymbol{\mu}_{\bar{y}} = \mathbb{E}[\bar{Y}] = \mathbb{E}[\mathbf{A}\bar{X} + \mathbf{b}] = \mathbb{E}[\mathbf{A}\bar{X}] + \mathbf{b} = \mathbf{A}\boldsymbol{\mu}_{\bar{x}} + \mathbf{b}.$$

The covariance matrix of $\bar{y}$ is given by

$$\mathrm{COV}[\bar{Y}] = \mathrm{COV}[\mathbf{A}\bar{X} + \mathbf{b}] = \mathrm{COV}[\mathbf{A}\bar{X}] = \mathbf{A}\boldsymbol{\Sigma}_{\bar{X}}\mathbf{A}^T.$$

A.9 Properties of Gaussian Distributions

Gaussian distributions play an important role in various robot algorithms and deserve special treatment. Note, however, that they are mostly used for computational advantages, while many physical phenomena are not Gaussian. A unidimensional Gaussian distribution is characterized by its expectation and variance, and we write $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$.

Definition A.25 (Gaussian Vector)**.** *A random vector is said to be a* Gaussian vector *if there exist a vector* $\boldsymbol{\mu}_{\bar{X}}$ *and a nonsingular matrix* $\boldsymbol{\Sigma}_{\bar{X}}$ *such that its PDF can be written as*

$$f_{\bar{X}}(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}_{\bar{X}}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})^T \boldsymbol{\Sigma}_{\bar{X}}^{-1}(\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})\right), \tag{A.7}$$

where $\mathbf{x} \in \mathbb{R}^n$ *and* $|\boldsymbol{\Sigma}_{\bar{X}}|$ *is the determinant of the covariance matrix.*

A Gaussian vector is also called a *multivariate Gaussian*, and we write $\bar{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\bar{X}}, \boldsymbol{\Sigma}_{\bar{X}})$. Note that for a multivariate Gaussian distribution the covariance matrix $\boldsymbol{\Sigma}_{\bar{X}}$ must be positive definite[3]. Consequently, all its eigenvalues are positive. Figure A.7 shows the PDF of a bidimensional Gaussian.

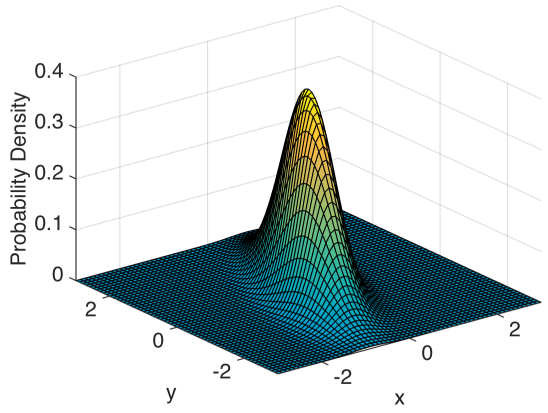


Figure A.7: PDF of a bidimensional Gaussian

The definition of a Gaussian vector allows us to extend the concept of n jointly Gaussian variables (generalizing the case of two variables, as discussed in Section A.7).

Definition A.26. *Let* $X_1, X_2, \ldots, X_n$ *be* n *random variables. We say that they are jointly Gaussian distributed if their joint distribution can be written as in Eq.* (A.7) *for a suitable vector* $\boldsymbol{\mu}_{\bar{X}}$ *and a suitable non-singular matrix* $\boldsymbol{\Sigma}_{\bar{X}}$*.*

In many practical scenarios, for a given value δ, it is useful to determine the smallest region $\mathcal{D}$ such that

$$\int_{\mathcal{D}} f_{\bar{X}}(\mathbf{x})\, d\mathbf{x} \geq \delta.$$

[3]The covariance matrix is in general positive semidefinite for a generic distribution. However, for the Gaussian case its inverse must exist for the PDF to be defined, and therefore the matrix must be positive definite.

The set $\mathcal{D}$ defines a *confidence region*, i.e., it specifies a bound on the probability that a realization of $\bar{X}$ lies outside $\mathcal{D}$. From the symmetry of the PDF, it follows immediately that $\mathcal{D}$ must be centered at $\boldsymbol{\mu}_{\bar{X}}$ and symmetric.

For the special case of a unidimensional random variable $X \sim \mathcal{N}(\mu, \sigma^2)$ the following relationships can be established through numerical integration:

$$\int_{\mu-\sigma}^{\mu+\sigma} f_x(\zeta)\, d\zeta \approx 0.68, \qquad \int_{\mu-2\sigma}^{\mu+2\sigma} f_x(\zeta)\, d\zeta \approx 0.95, \qquad \int_{\mu-3\sigma}^{\mu+3\sigma} f_x(\zeta)\, d\zeta \approx 0.997.$$

Hence, $\mathcal{D}$ is an interval centered at μ, and (unsurprisingly) its length increases with δ. While one can in principle compute such intervals for any δ, the three values above are the most widely used. In fact, the interval $(\mu - 3\sigma, \mu + 3\sigma)$ is perhaps the most common when working with Gaussian distributions, since the probability that a realization lies within this interval is approximately 0.997. Figure A.8 illustrates these intervals.

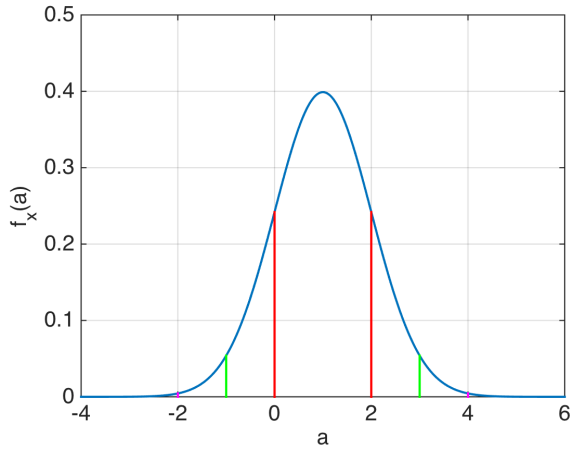


Figure A.8: Confidence intervals for the Gaussian $\mathcal{N}(1, 1)$. Red lines show the $(\mu-\sigma, \mu+\sigma)$ interval with area 0.68. Green lines show the $(\mu-2\sigma, \mu+2\sigma)$ interval with area 0.95. Purple lines show the $(\mu - 3\sigma, \mu + 3\sigma)$ interval with area 0.997.

The situation is more complex and interesting when considering an n-dimensional Gaussian random vector $\bar{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\bar{X}}, \boldsymbol{\Sigma}_{\bar{X}})$. To determine $\mathcal{D}$ for a given δ, we start with the simpler problem of identifying the set of points where the PDF is equal to a given value $k > 0$. This can be immediately obtained by plugging k into Eq. (A.7):

$$\frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}_{\bar{X}}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})^T \boldsymbol{\Sigma}_{\bar{X}}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})\right) = k.$$

Rearranging this expression, one can write

$$(\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})^T \boldsymbol{\Sigma}_{\bar{X}}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{\bar{X}}) = -2 \ln\left(\sqrt{(2\pi)^n |\boldsymbol{\Sigma}_{\bar{X}}|}\, k\right). \tag{A.8}$$

The term $(\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})^T \boldsymbol{\Sigma}_{\bar{X}}^{-1} (\mathbf{x} - \boldsymbol{\mu}_{\bar{X}})$ is called the *Mahalanobis distance* between $\mathbf{x}$ and $\boldsymbol{\mu}_{\bar{X}}$ and has important applications in robotics, in particular for data association problems.

Expanding the left-hand side of Eq. (A.8) and recalling basic geometry, one can observe that Eq. (A.8) describes the contour of an ellipse. The ellipse is centered at $\boldsymbol{\mu}_{\bar{X}}$ and has its axes oriented along the eigenvectors of $\boldsymbol{\Sigma}_{\bar{X}}$. Denoting by $(\lambda_i, \mathbf{e}_i)$ the i-th eigenvalue–eigenvector pair of the covariance matrix, the axes have length $\sqrt{k^*\lambda_i}$, where $k^* = -2\ln\left(\sqrt{(2\pi)^n|\boldsymbol{\Sigma}_{\bar{X}}|}\,k\right)$ is the right-hand side of Eq. (A.8).

Example A.13. *Consider a 2-dimensional Gaussian vector (also known as a bivariate Gaussian variable)* $\bar{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\bar{X}}, \boldsymbol{\Sigma}_{\bar{X}})$ *with* $\boldsymbol{\mu}_{\bar{X}} = [2 \quad 0.5]^T$ *and*

$$\boldsymbol{\Sigma}_{\bar{X}} = \begin{bmatrix} 2 & 2.5 \\ 2.5 & 4.25 \end{bmatrix}.$$

We want to determine and plot the locus of points for which the PDF is equal to $k = 0.1$. *We start by determining the eigenvalues and eigenvectors of* $\boldsymbol{\Sigma}_{\bar{X}}$. *They are* $\lambda_1 \approxeq 0.3835$, $\lambda_2 \approxeq 5.8665$, *and the corresponding eigenvectors are* $\mathbf{e}_1 = [-0.8398 \quad 0.5430]^T$ *and* $\mathbf{e}_2 = [0.5430 \quad 0.8398]^T$. *The center of the ellipse is given by* $\boldsymbol{\mu}_{\bar{X}}$ *and its axes are determined by the eigenvalues. To determine the contour, we need*

$$k^* = -2\ln(\sqrt{(2\pi)^n|\boldsymbol{\Sigma}_{\bar{X}}|}k) \approxeq 0.1185.$$

Hence the first axis has length $\sqrt{k^*\lambda_1} \approxeq 0.2132$ *and the second has length* $\sqrt{k^*\lambda_2} \approxeq 0.8337$. *With this information it is then possible to plot the ellipse (see Figure A.9).*

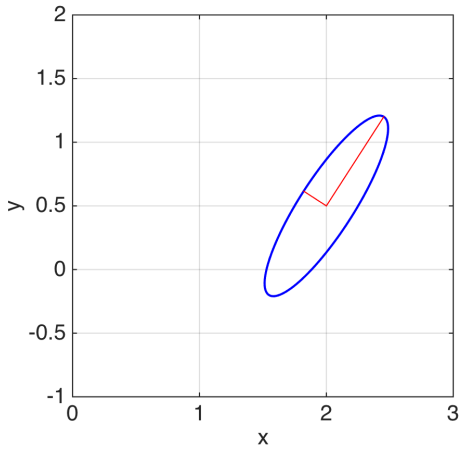


Figure A.9: Locus of points for which the PDF is equal to $k = 0.1$. This is also known as the density contour for 0.1 (or isocontour curve).

The reader can easily verify that as k grows, e.g., $k = 0.2$, the value of k^* becomes negative and the length of the axes is undefined because $\sqrt{k^*\lambda_i}$ is no longer a real number (recall that all eigenvalues of $\boldsymbol{\Sigma}_{\bar{X}}$ are strictly positive). This is correct, because for the chosen expectation and covariance matrix the PDF is always smaller than 0.2, and therefore the problem of determining the density contour for this value is ill posed. When the covariance matrix $\boldsymbol{\Sigma}_{\bar{X}}$ is diagonal, its eigenvectors are the n vectors of the standard basis for $\mathbb{R}^n$ and

therefore the axes of the ellipse are aligned with the axes of $\mathbb{R}^n$. Conversely, when the covariance matrix is not diagonal, as in the example above, the ellipse is not aligned with the axes. Having established that the level sets of the PDF are ellipses, we return to the problem of determining $\mathcal{D}$ for a given δ. At this point it should be evident that for a given δ we need a function defining the length of the semi-axes of the ellipse. It is possible to show that the length of the ith axis is $\sqrt{\chi^2_n(1-\delta)\lambda_i}$, where $\chi^2_n(1-\delta)$ is the $(1-\delta)$ percentile of a χ^2 distribution with n degrees of freedom and can be found in appropriate numerical tables.

Example A.14. *For the bivariate Gaussian distribution in Example A.13, we determine the region $\mathcal{D}$ associated with $\delta = 0.975$. From the previous discussion it follows that it is an ellipsoid centered on $\boldsymbol{\mu}_{\bar{X}} = [2 \quad 0.5]^T$ with axes in the directions identified by the eigenvectors $\mathbf{e}_1 = [-0.8398 \quad 0.5430]^T$ and $\mathbf{e}_2 = [0.5430 \quad 0.8398]^T$. The length of the semi-axes is determined by $\chi^2_2(1-\delta) = \chi^2_2(0.025) = 7.378$ (this value can be found in the χ^2 tables). Hence the first semi-axis has length $\sqrt{\chi^2_2(1-\delta)\lambda_1} = \sqrt{7.378 \cdot 0.3835}$, while the second has length $\sqrt{\chi^2_2(1-\delta)\lambda_2} = \sqrt{7.378 \cdot 5.8665}$.*

Another interesting property related to the ellipse associated with a Gaussian distribution is the following. The volume of an n-dimensional ellipse can be written as $K(n)a_1a_2\dots a_n$, where $K(n)$ is a function of the number of dimensions n and $a_1 \dots a_n$ are the lengths of the semi-axes. For example, for an ellipse in $\mathbb{R}^2$ the volume (area) is $\pi a_1 a_2$, in $\mathbb{R}^3$ the volume is $\frac{4}{3}\pi a_1 a_2 a_3$, and so on. From the previous examples it follows that the lengths of these semi-axes are proportional to the eigenvalues of the covariance matrix $\boldsymbol{\Sigma}_{\bar{X}}$. From linear algebra we also know that the trace of a square matrix (the sum of the elements on the main diagonal) is equal to the sum of its eigenvalues. Therefore the elements on the main diagonal of the covariance matrix of a Gaussian vector can be used to estimate the area of the ellipsoid even if the covariance matrix is not diagonal. In fact, the trace of a covariance matrix is also called *generalized variance* because it is a measure of variability of the associated random vector.

Random Gaussian variables and random Gaussian vectors are invariant to affine transformations. That is, if $X \sim \mathcal{N}(\mu_X, \sigma^2_X)$ and $Y = aX + b$, then $Y \sim \mathcal{N}(a\mu_X + b, a^2\sigma^2_X)$. Similarly, if $\bar{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\bar{X}}, \boldsymbol{\Sigma}_{\bar{X}})$ and $\bar{Y} = \mathbf{A}\bar{X} + \mathbf{b}$, then $\bar{Y} \sim \mathcal{N}(\mathbf{A}\boldsymbol{\mu}_{\bar{X}} + \mathbf{b}, \mathbf{A}\boldsymbol{\Sigma}_{\bar{X}}\mathbf{A}^T)$.

Remark A.3. *Let $\bar{X} \sim \mathcal{N}(\boldsymbol{\mu}_{\bar{x}}, \boldsymbol{\Sigma}_{\bar{x}})$ be an n-dimensional Gaussian vector with diagonal covariance matrix. This means that the covariance between all its components is 0. In the general case, covariance equal to 0 does not mean independence. However, for the special case of Gaussian random vectors, it is possible to show that this is true, i.e., the vector is equivalent to n independent Gaussian variables whose means are given by the components of $\boldsymbol{\mu}_{\bar{X}}$ and whose variances are given by the diagonal elements of $\boldsymbol{\Sigma}_{\bar{X}}$.*

A.10 Stochastic Processes

Stochastic processes are models used to describe stochastic phenomena evolving over time (or, more generally, a family of random variables or random vectors). Their formal definition is as follows.

Definition A.27 (Stochastic Process)**.** *Given a probability space $S = (\Omega, \mathcal{A}, P)$, let $\mathcal{T} \subset \mathbb{R}$ be an infinite set. A function $\boldsymbol{X} : \mathcal{T} \times \Omega \to \mathbb{R}$ is a* stochastic process *if for each $t' \in \mathcal{T}$ the function $\boldsymbol{X}(t', \omega)$ is a random variable.*

The set $\mathcal{T}$ is commonly referred to as *time*, although it could represent anything. Normally, one chooses either $\mathcal{T} = \mathbb{N}$ or $\mathcal{T} = \mathbb{R}$. In our case, we will most often consider $\mathcal{T} = \mathbb{N}$ to model the discrete-time nature of the algorithms we study. These types of processes are called *discrete-time stochastic processes*. If instead $\mathcal{T} = \mathbb{R}$, we have a *continuous-time stochastic process*.

Stochastic processes can model different phenomena. For example, $\boldsymbol{X}(t, \omega)$ can be interpreted as a family of random variables indexed by the elements in $\mathcal{T}$. When $\mathcal{T} = \mathbb{N}$, we will accordingly write $X_0, X_1, X_2, \ldots$ to indicate these variables. Alternatively, we could write $X(t)$ when $\mathcal{T} = \mathbb{R}$. Another perspective is to fix a specific $\omega' \in \Omega$ and then consider the function of time $\boldsymbol{X}(t, \omega')$. For a fixed ω', $\boldsymbol{X}(t, \omega')$ is deterministic. Hence, $\boldsymbol{X}(t, \omega)$ can also be seen as a map from Ω into the space of deterministic functions.

Once a given $n \in \mathcal{T}$ is fixed, X_n is a random variable, so we can compute its expectation, variance, and other statistical properties.[4] We denote the expectation of X_n by $\mu_X(n)$. As n varies over $\mathcal{T}$, the sequence $\mu_X(0), \mu_X(1), \mu_X(2), \ldots$ defines a function $\mu_X : \mathcal{T} \to \mathbb{R}$. For simplicity, we will write μ_n instead of $\mu_X(n)$.

For two elements of $\mathcal{T}$, say m and n, we can determine the correlation and covariance between the associated variables:

$$r(m, n) = \mathbb{E}[X_m X_n],$$

$$\mathrm{COV}(m, n) = \mathbb{E}[(X_m - \mu_m)(X_n - \mu_n)] = r(m, n) - \mu_m \mu_n.$$

From these definitions, we can introduce the concept of Gaussian processes.

Definition A.28 (Gaussian Process)**.** *A stochastic process $\boldsymbol{X}(t, \omega) : \mathcal{T} \times \Omega \to \mathbb{R}$ is called a* Gaussian process *if for each $n \in \mathbb{N}$ and each choice of $t_i \in \mathcal{T}$ $(1 \leq i \leq n)$, the n variables $\boldsymbol{X}(t_1, \omega), \boldsymbol{X}(t_2, \omega), \ldots, \boldsymbol{X}(t_n, \omega)$ are jointly Gaussian.*

Definition A.29 (White Noise)**.** *A stochastic process $\boldsymbol{X}(t, \omega) : \mathcal{T} \times \Omega \to \mathbb{R}$ is* white noise *if*

$$m \neq n \Rightarrow \mathrm{COV}[m, n] = 0.$$

Note that while some authors require a white-noise process to have zero mean, the definition we provide is more general and does not impose this requirement. If the process has zero mean, an equivalent definition is that $r(m, n) = 0$ for each $m \neq n$.

Further Reading

Most of the material presented in this chapter can be found in any probability theory textbook, like [8]. Some more advanced topics like stochastic processes are found in textbooks like [44].

[4] From now onwards, unless otherwise noted, we will consider the case $\mathcal{T} = \mathbb{N}$, i.e., we focus on discrete-time stochastic processes.

Index

Bibliography

[1] B.D.O. Anderson and J.B. Moore. *Optimal filtering*. Dover Publications, Inc., 1979.

[2] R. Arkin. *An introduction to behavior-based robotics*. MIT Press, 1998.

[3] K.J. Aström. *Introduction to stochastic control theory*. Dover Publications, 2006.

[4] K.J. Aström and R.M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2021.

[5] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *IEEE Robotics & Automation Magazine*, 13(3):108–117, 2006.

[6] G.A. Bekey. *Autonomous Robots*. MIT Press, 2005.

[7] D. P. Bertsekas. *Dynamic Programming & Optimal Control*, volume 1 and 2. Athena Scientific, 2005.

[8] D.P. Bertsekas and J.N. Tsitsiklis. *Introduction to Probability*. Athena, 2nd edition, 2008.

[9] J. Borenstein, H.R. Everett, and L. Feng. *Navigating mobile robotS: systems and techniques*. AK Peters, 1996.

[10] A. Cavalcanti, B. Dongol, R. Hierons, J. Timmins, and J. Woodcock, editors. *Software Engineering for Robotics*. Springer, 2021.

[11] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki, and S. Thrun. *Principles of robot motion*. MIT Press, 2005.

[12] P. Corke. *Robotics, vision and Control*. Springer, 2013.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[14] N. Correll, B. Hayes, C. Heckman, and A. Roncone. *Introduction to Autonomous Robots: Mechanisms, Sensors, Actuators, and Algorithms*. MIT Press, 2022.

[15] I. J. Cox. Blanche-an experiment in guidance and navigation of an autonomous robot vehicle. *IEEE Transactions on Robotics and Automation*, 7(2):193–204, 1991.

[16] J.J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson, 3rd edition, 2004.

[17] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, 2nd edition, 2010.

[18] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 13(2):99–110, 2006.

[19] G. Erinc and S. Carpin. Image-based mapping and navigation with heterogenous robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5807–5814, 2009.

[20] C. Fairchild and T.L. Harman. *ROS robotics by example*. Packt publishing, 2016.

[21] E. Fernandez, A. Mahtani, L. Sanchez Crespo, and A. Martinez. *Learning ROS for robotics programming*. Packt publishing, 2nd edition, 2015.

[22] T. Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, 2013.

[23] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics Automation Magazine*, 4(1):23–33, 1997.

[24] C. Le Goues, S. Elbaum, D. Anthony, Z.B. Celik, M. Castillo-Effen, N. Correll, P. Jamshidi, M. Quigley, T. Tabor, and Q. Zhu. Software engineering for robotics: Future research directions; report from the 2023 workshop on software engineering for robotics, 2024.

[25] L. Joseph. *Mastering ROS for robotics programming*. Packt publishing, 2015.

[26] L. Joseph and J. Cacace. *Mastering ROS 2 for Robotics Programming: Design, build, simulate, and prototype complex robots using the Robot Operating System 2+*. Packt Publishing, 2025.

[27] A. Kelly. *Mobile Robotics: Mathematics, Models, and Methods*. Cambridge University Press, 2013.

[28] D. Kortenkamp and R. Simmons. Robotic systems architectures and programming. In *Handbook of Robotics*, chapter 8, pages 187–206. Springer, 2008.

[29] S.M. LaValle. *Planning algorithms*. Cambridge academic press, 2006.

[30] J.J. Leonard and H.F. Durrant-Whyte. *Directed Sonar Sensing for Mobile Robot Navigation*. Boston : Kluwer Academic Publishers, 1992.

[31] K. Lynch and F.C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge Univrsity Press, first edition, 2017.

[32] S. Macenski, M. Booker, and J. Wallace. Open-source, cost-aware kinematically feasible planning for mobile and surface robotics. *Arxiv*, 2401.13078, 2024.

[33] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[34] S. Macenski and I. Jambrecic. Slam toolbox: Slam for the dynamic world. *Journal of Open Source Software*, 6(61):2783, 2021.

[35] S. Macenski, T. Moore, D.V Lu, A. Merzlyakov, and M. Ferguson. From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2. *Robotics and Autonomous Systems*, 168:104493, 2023.

[36] S. Macenski, S. Singh, F. Martin, and J. Gines. Regulated pure pursuit for robot path tracking. *Autonomous Robots*, 47:685–694, 2023.

[37] Steven Macenski, Francisco Martin, Ruffin White, and Jonatan Ginés Clavero. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[38] P. Misra and P. Enge. *Global Positioning System: Signals, Measurements, and Performance*. Ganga-Jamuna Press, 2010.

[39] T. Moore and D. Stouch. A generalized extended kalman filter implementation for the robot operating system. In *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.

[40] T. Moore and D. Stouch. A generalized extended Kalman filter implementation for the robot operating system. In *Proceedings of the 13th International Conference Intelligent Autonomous Systems*, pages 335–348. Springer International Publishing, 2016.

[41] R.R. Murphy. *An introduction to AI robotics*. MIT Press, 2000.

[42] W.S. Newman. *A Systematic Approach to Learning Robot Programming with ROS*. CRC Press, 2018.

[43] J. O'Kane. *A Gentle Introduction to ROS*. CreateSpace Independent Publishing Platform, 2013.

[44] A. Papoulis and S.U. Pillai. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill, 4th edition, 2002.

[45] E. Renard. *ROS 2 from Scratch: Get started with ROS 2 and create robotics applications with Python and C++*. Packt Publishing, first edition, 2024.

[46] F. Martín Rico. *A Concise Introduction to Robot programming with ROS2*. CRC Press, first edition, 2022.

[47] S. Russel and P. Norvig. *Artificial Intelligence: A modern approach*. Pearson, 3rd edition, 2009.

[48] S. Särkkä. *Bayesian filtering and smoothing*. Cambridge University Press, 2013.

[49] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics – Modelling, Planning and Control*. Springer, 2009.

[50] R. Siegward, I.R. Nourbaksh, and D. Scaramuzza. *Introduction to Autonomous Mobile Robots*. MIT Press, 2nd edition, 2011.

[51] M.W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot modeling and control*. Wiley, 2005.

[52] R.F. Stengel. *Optimal control and estimation*. Dover, 1994.

[53] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2006.

[54] S.H. Żak. *Systems and Control*. Oxford University Press, first edition, 2002.