

Topics on Robotics & Vision

D. T. McGuiness, PhD

**M.Sc
Robotics
Lecture Book**

2025.SS



Contents

1 Mobile Robot Localisation	3
1.1 Introduction	3
1.2 The problems of Noise and Aliasing	5
1.2.1 Sensor Noise	5
1.2.2 Sensor Aliasing	6
1.2.3 Effector Noise	7
1.2.4 An Error Model for Odometric Position Estimation	9
1.3 Localisation v. Hard-Coded Navigation	11
1.4 Representing Belief	14
1.4.1 Single Hypothesis Belief	14
1.4.2 Multiple Hypothesis Belief	16
1.5 Representing Maps	18
1.5.1 Continuous Representation	19
1.5.2 Decomposition Methods	21
1.5.3 Current Challenges	24
1.6 Probabilistic Map-Based Localisation	27
1.6.1 Introduction	27
1.6.2 Markov Localisation	29
1.6.3 Kalman Filter Localisation	35
1.7 Other Examples of Localisation Methods	40
1.7.1 Landmark-based Navigation	40
1.7.2 Globally Unique Localisation	41
1.7.3 Positioning Beacon systems	43
1.7.4 Route-Based Localisation	44
1.8 Building Maps	45
1.8.1 Stochastic Map Technique	46
1.8.2 Other Mapping Techniques	48
2 Welcome to Linux	53
2.1 Learning the Linux Command Line	53
2.1.1 A Short History on Computer Interfaces	54
2.1.2 Linux is a Nutshell	56
2.1.3 Linux Distributions	57
2.2 Installation	59
2.3 Docker	60
2.3.1 Dockerfile	61

2.3.2	Running the Container	66
3	Command Line Fundamentals	69
3.1	Introduction	69
3.2	The Structure of Commands	72
3.2.1	Some Rules Regarding the Syntax	73
3.3	Helpful Keyboard Shortcuts for the Terminal	75
3.4	When you need help with Commands	77
3.5	Additional Information	81
3.5.1	Use Tab completion on the Shell	81
3.5.2	The info command	81
3.5.3	The whatis command	82
4	Working with Files and Folders	85
4.1	Introduction	85
4.2	A Detailed Look in ls Command	90
4.3	Creating and Removing Folders	92
4.4	Move, Copy and Delete Files and Folders	94
4.5	Role to Users and sudo	96
4.6	File Permissions	98
4.7	Hard and Symbolic Links	101
4.7.1	Symbolic Links	101
4.8	The Linux File System	104
4.9	Common Command-Line Tools and Tasks	107
4.9.1	The UNIX Philosophy	107
4.9.2	Connecting Commands with Pipes	109
4.9.3	Viewing Text Files with cat, head, tail, and less	109
4.10	Advanced Topics	111
4.10.1	Find Linux Distribution and Kernel Information	111
4.10.2	Find System Hardware and Disk Information	112
5	Installation	115
5.1	ROS 2 Humble Hawksbill	115
5.1.1	Introduction	115
5.1.2	Setting up the Local	115
5.1.3	Setting Up the Source Files	116
5.1.4	Install ROS 2 Packages	117
5.1.5	Setting Up the Environment	118
5.2	Auto-Install Script	119
6	ROS Concepts	123
6.1	Introduction	123
6.2	Publisher and Subscriber Architecture	124
6.3	Nodes - The Building Blocks	125

6.4	The Discovery Process	126
6.5	Communication Between Nodes	127
6.5.1	Description	127
6.5.2	Messages	127
6.6	Topics	133
6.6.1	Publisher - Subscriber Architecture	133
6.6.2	Anonymity	133
6.6.3	Strongly-Typed	134
6.7	Services	135
6.7.1	Service Server	135
6.8	Actions	137
6.8.1	Action Server	137
6.8.2	Action Client	138
6.9	Parameters	139
6.9.1	A Detailed Look	139
6.9.2	Parameter Interaction	140
6.10	Working with Command Line	142
6.11	Launch File	144
7	Command Line Tools	145
7.1	Setting the Environment	145
7.2	Turtles and Graphs	148
7.3	A Deeper Look into Nodes	153
7.4	Working with Topics	156
7.5	Working with Services	161
7.6	Working with Parameters	165
7.7	A Practical Look into Actions	169
7.8	Launching Nodes	173
8	Client Libraries	175
8.1	Getting Started with Colcon	175
8.2	Creating a Workspace	179
8.3	Creating a Package	183
8.4	Writing a Simple Publisher & Subscriber	184
8.4.1	Writing the Publisher Node	184
8.4.2	Writing the Subscriber Node	188
8.4.3	Building and Running	190
8.5	Writing a Simple Service and Client	191
8.5.1	Writing the Service Node	192
8.5.2	Writing the Client Node	194
8.6	Creating Custom msg and srv Files	197
8.6.1	Creating Custom Definitions	198
8.6.2	Testing the Newly Built Interfaces	200
8.7	Using Parameters in a Class	205

8.8	Managing Dependencies	211
8.8.1	Explaining Rosdep	211
8.8.2	Explaining Pacakge Manifesto	211
8.9	Creating an Action	213
8.10	Writing an Action Server and Client	215
8.11	Writing a Launch File	216
9	Transform Library	221
9.1	A Gentle Introduction	221
9.2	Wrting a Static Broadcaster	225
9.3	Wrting a Listener	230
9.4	Adding a Frame	235
9.5	Wrting a Broadcaster	242

List of Figures

1.1	Navigation is one if not the most demanding and complicated task in Autonomous Mobile Robotics (AMR). However a successful implementation will result in a versatile AMR which can find its way in unknown environments such as exploring other planets [UoW2023].	3
1.2	General schematic for mobile robot localisation.	4
1.3	A sample environment.	11
1.4	An Architecture for Behavior-based Navigation	12
1.5	An Architecture for Map-based (or model-based) Navigation	12
1.6	On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown [xkcd2025].	13
1.7	The three (3) examples of single hypotheses of position using different map representation. a) real map with walls, doors and furniture b) line-based map -> around 100 lines with two parameters c) occupancy grid based map -> around 3000 grid cells sizing 50x50 cm d) topological map using line features (Z/S-lines) and doors -> around 50 features and 18 nodes	15
1.8	The presented robot-centric mapping framework enables mobile robots to create consistent elevation maps of the terrain. Mapping does not necessarily need to be done only in 2D as robots which will be used in outdoor environment would need the height of the map as well [fankhauser2018probabilistic].	18
1.9	A continuous representation using polygons as environmental obstacles.	19
1.10	Example of a continuous-valued line representation of EPFL. left: real map right: representation with a set of infinite lines.	20
1.11	The schematic for the Kalman filter localisation	38
1.12	An illustration showing the object-level landmarks in blue-boxes. (a,b) shows two different indoor scenarios. The blue boxes represent the 3D object detection of object-level landmarks. The red dots indicate the nodes of the topological map. The yellow lines indicate the edges of the topological map. The green curve is the feasible navigation trajectory generated based on the proposed method [wang2022object].	41
1.13	43
1.14	2005 DARPA Grand Challenge winner Stanley performed SLAM as part of its autonomous driving system [Kivaan2007].	45
1.15	General schematic for concurrent localization and map building.	46
1.16	A naive, local mapping strategy with small local error leads to global maps that have a significant error, as demonstrated by this real-world run on the left. By applying topological correction, the grid map on the right is extracted [bruce2000fast].	49

1.17 Stanford Racing and Victor Tango together at an intersection in the DARPA Urban Challenge Finals.	52
2.1 Hughes telegraph, an early (1855) teleprinter built by Siemens and Halske. The centrifugal governor to achieve synchronicity with the other end can be seen [Manske2009].	54
2.2 Nokia Bell Labs Murray Hill, NJ (Original)	54
2.3 Bourne shell interaction on Version 7 Unix (Original).	55
2.4 The kernel mapping of the Linux operating system.	56
2.5 The docker logo	60
3.1 A graphical interface from the late 1980s, which features a TUI window for a man page, a shaped window (oclock) as well as several iconified windows. In the lower right we can see a terminal emulator running a Unix shell, in which the user can type commands as if they were sitting at a terminal. - <i>From Wikipedia</i>	72
4.1 Beware of the sudo ghost.	96
4.2 For anyone who is interested in the UNIX philosophy, I would suggest reading this book as it has parts written by numerous people who were the original developers of the UNIX.	107
7.1 Generating a new turtlebot to be spawned.	151
7.3 A visual representation of the communication happening between the nodes <code>/turtlesim</code> and <code>/teleop</code>	157
7.4	158
7.5	159
9.1 A robot is comprised of numerous coordinate system as can be seen from this robot and needs to be constantly be kept in check which <code>tf2</code> allows [ros2iron2025].	221

List of Tables

1.1	The certainty matrix for the robot [nourbakhsh1995dervish].	33
2.1	Most popular distributions used according to distrowatch	58
3.1	Types of shells used in industry and academia. For reference, the authors computer uses zsh.	70
4.1	Octal Notation and their numerical meaning.	99
4.2	The value and their meaning using octal notation	99
6.1	Current types supported by Robot Operating System 2 (ROS) Humble.	128

Chapter 1

Mobile Robot Localisation

Table of Contents

1.1	Introduction	3
1.2	The problems of Noise and Aliasing	5
1.3	Localisation v. Hard-Coded Navigation	11
1.4	Representing Belief	14
1.5	Representing Maps	18
1.6	Probabilistic Map-Based Localisation	27
1.7	Other Examples of Localisation Methods	40
1.8	Building Maps	45

1.1 Introduction



Figure 1.1: Navigation is one if not the most demanding and complicated task in AMR. However a successful implementation will result in a versatile AMR which can find its way in unknown environments such as exploring other planets [UoW2023].

Navigation is one of, if not, the most challenging problem faced by an AMR and for the robot to be able to successfully navigate its environment, it requires four (4) functions:

Perception the robot must be able to interpret its sensors to extract meaningful data,

Localisation the robot must be able to determine its position within the environment,

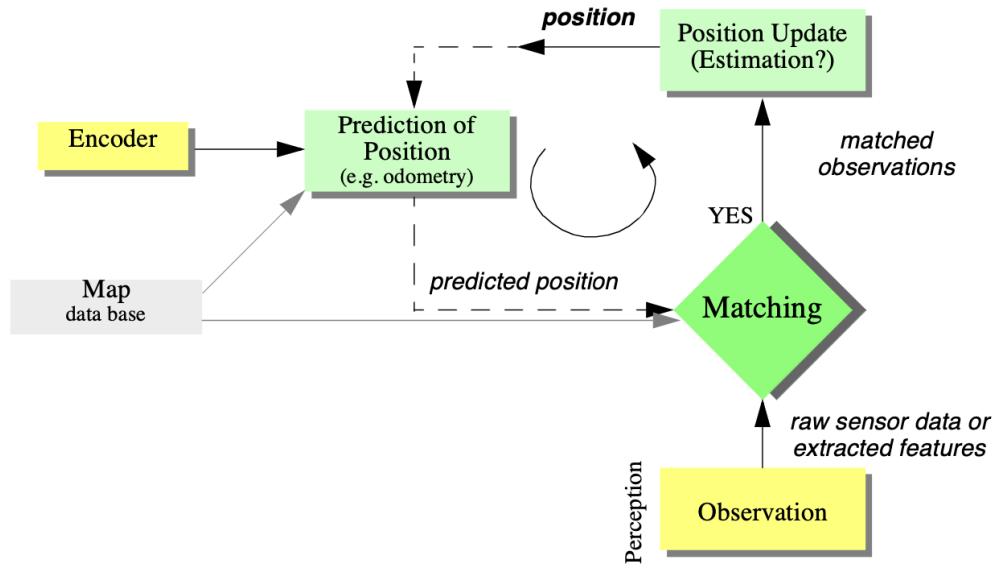


Figure 1.2: General schematic for mobile robot localisation.

Cognition the robot must be able to decide how to act to achieve its goals,

Motion control the robot must be able to modulate its motor outputs to achieve the desired trajectory.

Of these four (4) aforementioned components, localisation has received the greatest research attention in the past and, as a result, significant advances have been made on this front, presented in [**bailey2002mobile**], [**campbell2020localization**], and [**hodge2023sensors**]. In this chapter, we will explore the successful localisation methodologies and techniques used in academic research and industrial application [**jaradat2017challenges**].

The structure of the chapter is as follows:

- We will describe how sensor and effector uncertainty is responsible for the difficulties of localisation in Section 1.2,
- Then, in Section 1.3, we will have a look at the two (2) extreme approaches to dealing with the challenge of robot localisation [**filliat2003map**]:
 - Avoiding localisation altogether,
 - Performing explicit map-based localisation
- The remainder of the chapter discusses the question of representation, which we will have a look at different case studies of successful localisation systems using a variety of representations and techniques to achieve AMR localisation.

1.2 The problems of Noise and Aliasing

If one could attach an accurate Global Positioning System (GPS) sensor to an AMR, much of the localisation problem would be obviated. GPS would then inform the robot of its **exact** position and orientation, indoors and outdoors, so the answers to the questions,

Where am I?, Where am I going?, and, How should I get there? [leonard2012directed]

would **always** be immediately available.

Unfortunately, such a sensor is **NOT** currently practical.¹ The existing GPS network provides accuracy to within several m [**wing2005consumer**], which is still not the optimal accuracy for localising human-scale AMRs as well as miniature AMRs such as desk robots and the body-navigating nano-robots of the future.

In addition, GPS cannot function indoors or in obstructed areas and are therefore limited in their workspace. But, looking beyond the limitations of GPS, localisation implies more than knowing one's absolute position in the Earth's reference frame.

Consider a robot which is interacting with humans. This robot may need to identify its absolute position, but its relative position with respect to target humans is also equally important. Its localisation task can include:

- identifying humans using its sensor array [**bellotto2007people**],
- then computing its relative position to the humans.

Furthermore, during operation a robot will select a strategy for achieving its goals. If it intends to reach a particular location, then localisation may not be enough. The robot may need to acquire or build an environmental model,² which aids it in planning a path to the goal.

Localisation means more than simply determining an absolute pose in space. It means building a map, then identifying the robot's position relative to that map.

¹ Of course, this misleading statement as we have technology which allows the shrinking of errors down to cm using real-time kinematic positioning which is used to correct Global Navigation Satellite System (GNSS), which transmits the robot's location by longitude, latitude, altitude, and a timestamp [hodge2023sensors].

²i.e., a map representing 2D space if it is an indoor space which is level, or a 3D space if it is navigating rough terrain.

Clearly, the robot's sensors and effectors play an integral role in all the above forms of localisation. It is because of the inaccuracy and incompleteness of these sensors and effectors localisation poses difficult challenges.

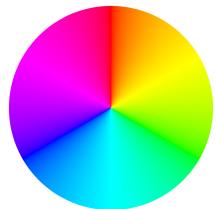
1.2.1 Sensor Noise

Sensors are the fundamental robot input for the process of perception, and therefore the degree to which sensors can discriminate world state is critical. Sensor noise produces a **limitation on the consistency of sensor readings** in the same environmental state and, therefore, on the number of

useful bits available from each sensor reading.

Often, the source of sensor noise problems is that some environmental features are not captured by the robot's representation and are thus overlooked.

³For example, this could be indoor office building, or a warehouse.



⁴One of the properties of a colour, defined as the degree to which a stimulus can be described as similar to or different from stimuli that are described as red, orange, yellow, green, blue, violet within certain theories of colour vision.

For example, a vision system used for indoor navigation³ may use the colour values detected by its colour Charge Coupled Device (CCD) camera. When the Sun is hidden by clouds, the illumination of the building's interior changes due to windows throughout the building. As a result, hue⁴ values are not constant. The colour CCD appears noisy from the robot's perspective as if subject to **random error**, and the hue values obtained from the CCD camera will be unusable, unless the robot is able to note the position of the Sun and clouds in its representation.

Illumination dependency is only one example of the apparent noise in a vision-based sensor system [**sridharan2009color**]. Picture jitter, signal gain, blooming and blurring are all additional sources of noise, potentially reducing the useful content of a colour video image.

Consider the noise level of ultrasonic range-measuring sensors, such as sonars, as we discussed previously. When a sonar transducer emits sound towards a relatively smooth and angled surface, much of the signal will coherently reflect away, failing to generate a return echo. Depending on the material characteristics, a small amount of energy may return nonetheless. When this level is close to the gain threshold of the sonar sensor, then the sonar will, at times, succeed and, at other times, fail to detect the object. From the robot's perspective, a virtually unchanged environmental state will result in two (2) different possible sonar readings:

one short, and one long which causes an nondeterministic behaviour.

⁵The propagation phenomenon resulting in signals reaching the receiver by two (2) or more paths. Causes of multipath can be atmospheric ducting, ionospheric reflection and refraction, and reflection from water bodies and terrestrial objects such as mountains and buildings.

The poor Signal-to-Noise Ratio (SNR) of a sonar sensor is further confounded by interference between multiple sonar emitters. Often, research robots have between 12 to 48 sonars on a single platform. In acoustically reflective environments, multipath interference⁵ is possible between the sonar emissions of one transducer and the echo detection circuitry of another transducer. The result can be dramatically large errors in ranging values due to a set of coincidental angles. Such errors occur rarely, less than 1% of the time, and are virtually random from the robot's perspective.

In conclusion, sensor noise reduces the useful information content of sensor readings. Clearly, the solution is to take multiple readings into account, employing temporal fusion or multi-sensor fusion⁶ to increase the overall information content of the robot's inputs.

1.2.2 Sensor Aliasing

Aliasing is the second major shortcoming of AMR sensors which cause them to give little information content, further amplifying the problem of **perception** and **localisation**.

The problem, known as sensor aliasing, is a phenomenon that humans seldom encounter. The human sensory system, particularly the visual system, tends to receive unique inputs in each unique local state within normal usage [williams1985aliasing]. In other words, every different place looks different. The power of this unique mapping is only apparent when one considers situations where this fails to hold.

Consider moving through an unfamiliar building that is completely dark. When the visual system sees only black, one's localisation system quickly degrades. Another useful example is that of a human-sized maze made from tall hedges. Such mazes have been created for centuries, and humans find them extremely difficult to solve without landmarks or clues because, without visual uniqueness, human localisation competence degrades rapidly.

In robots, the non-uniqueness of sensors readings, or sensor aliasing,⁷ is the norm and not the exception. Consider a narrow-beam rangefinder such as ultrasonic or infrared rangefinders. This sensor provides range information in a single direction without any additional data regarding material composition such as color, texture and hardness. Even for a robot with several such sensors in an array, there are a variety of environmental states that would trigger the same sensor values across the array. Formally, there is a many-to-one mapping from environmental states to the robot's perceptual inputs. Therefore, the robot's sensors cannot distinguish from among these many states.

A classical problem with sonar-based robots involves distinguishing between humans and inanimate objects in an indoor setting [blumrosen2014noncontact, sabatini1998method].

When facing an apparent obstacle in front of itself, should the robot say "Excuse me" because the obstacle may be a moving human, or should the robot plan a path around the object because it may be a cardboard box?

With sonar alone, these states are aliased and differentiation is impossible.



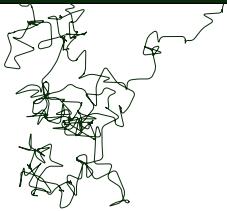
⁷Sensor aliasing in multiple types of sensors. One of the most apparent one is usually seen in digital images. For example, in the image above, due to low sampling, moire pattern starts to be seen [maksim2006].

The navigation problem due to sensor aliasing is that, even with noise-free sensors, the amount of information is generally insufficient to identify the robot's accurate position from a single sensor's reading. Therefore techniques needs to be employed by the robot programmer which base the robot's localisation on a series of readings and sufficient information to recover the robot's position over time.

1.2.3 Effector Noise

The challenges of localisation does NOT lie with sensor technologies alone. Just as robot sensors are noisy, limiting the information content of the signal, so do the robot effectors.

A single action taken by a AMR may have several different possible results, even though from the robot's point of view the initial state before the action was taken is well-known.



⁸An over-exaggerated example of effector noise where the motion is severely affected by the uncertainty caused by the deterministic error.

In short, AMR effectors introduce uncertainty about future state.⁸ The simple act of moving tends to increase the uncertainty of a AMR. There are, of course, exceptions. Using filters and predictive modelling, the motion can be carefully planned so as to minimise this effect, and indeed sometimes to actually result in more certainty. Furthermore, when the robot actions are taken in concert with careful interpretation of sensory feedback, it can compensate for the uncertainty introduced by noisy actions using the information provided by the sensors.

First, however, it is important to understand the precise nature of the effector noise that impacts AMR. It is important to note that, from the robot's point of view, this error in motion is viewed as error in the odometer, or the robot's inability to estimate its own position over time using knowledge of its kinematics and dynamics. The true source of error generally lies in an incomplete model of the environment.

For instance, the robot does NOT model the fact that the floor may be sloped, the wheels may slip, and a human may push the robot.

All of these unmodeled sources of error result in:

- inaccuracy between the physical motion of the robot,
- the intended motion of the robot, and the
- proprioceptive sensor estimates of motion.

⁹The process of calculating the current position of a moving object by using a previously determined position, or fix, and incorporating estimates of speed, heading (or direction or course), and elapsed time.

In odometry and dead reckoning⁹ the position update is based on proprioceptive sensors. The movement of the robot, sensed with wheel encoders and /or heading sensors is integrated to compute position. Because the sensor measurement errors are integrated, the position error accumulates over time. Thus the position has to be updated from time to time by other localisation mechanisms. Otherwise the robot is not able to maintain a meaningful position estimate in long run.

In the following we will concentrate on odometry based on the wheel sensor readings of a differential drive robot only [batavia2000path].¹⁰

There are many sources of odometric error, from environmental factors to resolution:

- Limited resolution during integration¹¹
- Misalignment of wheels causing deterministic error,
- Unequal wheel diameter, which again, causing deterministic error,
- Unequal floor contact, which can cause slipping during operation.

Some of the errors might be deterministic¹² (systematic). However, there are still a number of non-deterministic (random) errors which remain, leading to uncertainties in position estimation over time. From a geometric point of view one can classify the errors into three (3) types:

¹²To reiterate, deterministic errors are any errors which can be avoided and are generally caused by bad design or poorly calibrated sensors.

Range error Integrated path length of the robot movement, as in the sum of wheel motion.

Turn error Similar to range error, but for turns which are difference of the wheel motions.

Drift error difference in the error of the wheels leads to an error in the robot's angular orientation.

Over long periods of time, turn and drift errors far outweigh range errors, as their contribute to the overall position error is non-linear. Consider a robot, whose position is initially perfectly well-known, moving forward in a straight line along the x axis. The error in the y -position introduced by a move of d meters will have a component of $d \sin \Delta\theta$, which can be quite large as the angular error $\Delta\theta$ grows. Over time, as an AMR moves about the environment, the rotational error between its internal reference frame and its original reference frame grows quickly. As the robot moves away from the origin of these reference frames, the resulting linear error in position grows quite large. It is instructive to establish an error model for odometric accuracy and see how the errors propagate over time.

1.2.4 An Error Model for Odometric Position Estimation

Generally the pose (position) of a robot is represented by the vector

$$p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

For a differential drive robot the position can be estimated starting from a known position by integrating the movement (summing the incremental travel distances). For a discrete system with a fixed sampling interval Δt the incremental travel distances ($\Delta x; \Delta y; \Delta\theta$) are:

$$\begin{aligned} \Delta x &= \Delta s \cos(\theta + \Delta\theta/2) & \Delta y &= \Delta s \sin(\theta + \Delta\theta/2) \\ \Delta\theta &= \frac{\Delta s_r - \Delta s_l}{b} & \text{and} & \Delta s = \frac{\Delta s_r + \Delta s_l}{2} \end{aligned}$$

As we discussed earlier, odometric position updates can give only a very rough estimate of the actual position. Due to integration errors of the uncertainties of p and the motion errors during the incremental motion (s ; s) the position error based on odometry integration r_l grows with time. In the next step we will establish an error model for the integrated position p' to obtain the of the odometric position estimate. To do so, we assume that at the covariance matrix starting point the initial covariance matrix is known. For the motion increment p (s ; s) we assume the following covariance matrix : r_l where s and s are the distances travelled by each wheel, and k_r, k_l are error constants r_l representing the non-deterministic parameters of the motor drive and the wheel-floor interaction. As you can see in equation (5.8) we made the following assumption

- The two (2) errors of the individually driven wheels are independent.¹³
- The errors are proportional to the absolute value of the travelled distances ($\delta s_r; \delta s_l$).

¹³If there is more knowledge regarding the actual robot kinematics, the correlation terms of the covariance matrix could also be used.

These assumptions, while **NOT** perfect in the slightest, are nevertheless suitable and will therefore be used for the further development of the error model.

The **motion errors** are due to unprecise movement because of deformation of wheel, slippage, unequal floor, errors in encoders, ...

The values for the error constants k_r and k_l depend on the robot and the environment and should be experimentally determined by performing and analysing representative movements.

If we assume that ρ and $\Delta_{rl} = (\delta s_r; \delta s_l)$ are **uncorrelated** and the derivation of f (equ. (5.7)) can be reasonably approximated by the first-order Taylor expansion (linearization) we conclude, using the error propagation law (see section 4.2.3):

$$\Sigma_p f = \nabla_\rho f \cdot \Sigma_p \cdot \nabla_\rho f^T + \nabla_{\delta_{rl}} f \cdot \Sigma_{ffl} \cdot \nabla_{\delta_{rl}} f^T$$

The covariance matrix is, of course, always given by the $p p'$

1.3 Localisation v. Hard-Coded Navigation

Fig. 1.3 depicts a standard indoor environment an AMR is set to navigate. Now, suppose an AMR in question must deliver messages between two (2) specific rooms in this environment:

These are rooms A and B.

In creating a navigation system for this task, it is clear the AMR will need sensors and a motion control system. Sensors are required to avoid hitting moving obstacles such as humans, and some motion control system is required so that the robot can actively move.

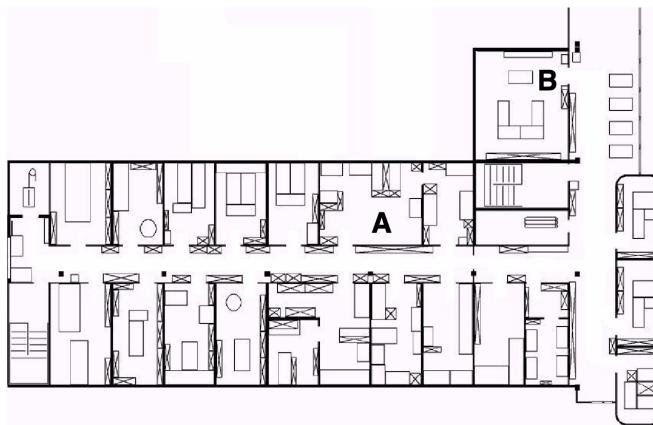


Figure 1.3: A sample environment.

It is less evident, however, whether or not this AMR will require a localisation system. Localisation may seem mandatory to successfully navigate between the two (2) rooms. It is through localising on a map, after all, which the robot can hope to recover its position and detect when it has arrived at the goal location. It is true that, at the least, the robot must have a way of detecting the goal location. However, explicit localisation with reference to a map is **NOT** the only strategy that qualifies as a goal detector.

An alternative, adopted by the behaviour-based community, suggests that, since sensors and effectors are noisy and information-limited, one should **avoid** creating a geometric map for localisation. Instead, they suggest designing sets of behaviours which together result in the **desired robot motion**.

In its essence, this approach avoids explicit reasoning about localisation and position, and therefore generally avoids explicit path planning as well.

This technique is based on a idea that, there exists a procedural solution to the particular navigation problem at hand. For example, in **Fig.** 1.3, the behavioralist approach to navigating from Room A to Room B might be to design a left-wall-following behavior and a detector for Room B that is triggered by some unique queue in Room B, such as the color of the carpet. Then, the robot can reach Room B by engaging the left wall follower with the Room B detector as the termination condition for the program.

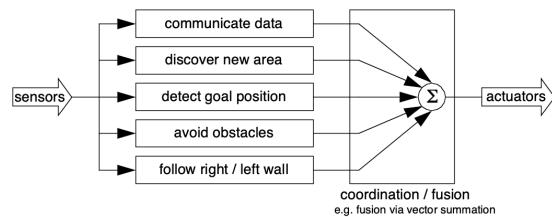


Figure 1.4: An Architecture for Behavior-based Navigation

The architecture of this solution to a specific navigation problem is shown in **Fig. 1.4**. The key advantage of this method is that, when possible, it may be implemented very quickly for a single environment with a small number of goal positions. It suffers from some disadvantages, however.

- The method does not directly scale to other environments or to larger environments. Often, the navigation code is location-specific, and the same degree of coding and debugging is required to move the robot to a new environment.
- The underlying procedures, such as left-wall-follow, must be carefully designed to produce the desired behaviour. This task may be time-consuming and is heavily dependent on the specific robot hardware and environmental characteristics.
- A behaviour-based system may have multiple active behaviors at any one time. Even when individual behaviours are tuned to optimise performance, this fusion and rapid switching between multiple behaviors can negate that fine-tuning. Often, the addition of each new incremental behavior forces the robot designer to re-tune all of the existing behaviors again to ensure that the new interactions with the freshly introduced behavior are all stable

In contrast to the behaviour-based approach, the map-based approach includes both localisation and cognition modules shown in **Fig. 1.5**.

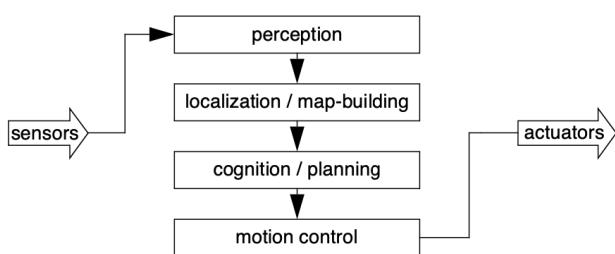


Figure 1.5: An Architecture for Map-based (or model-based) Navigation

In map-based navigation, the robot **explicitly** attempts to localise by collecting sensor data, then updating some belief about its position with respect to a map of the environment. The key advantages of the map-based approach for navigation are as follows:

- The explicit, map-based concept of position makes the system's belief about position transparent.

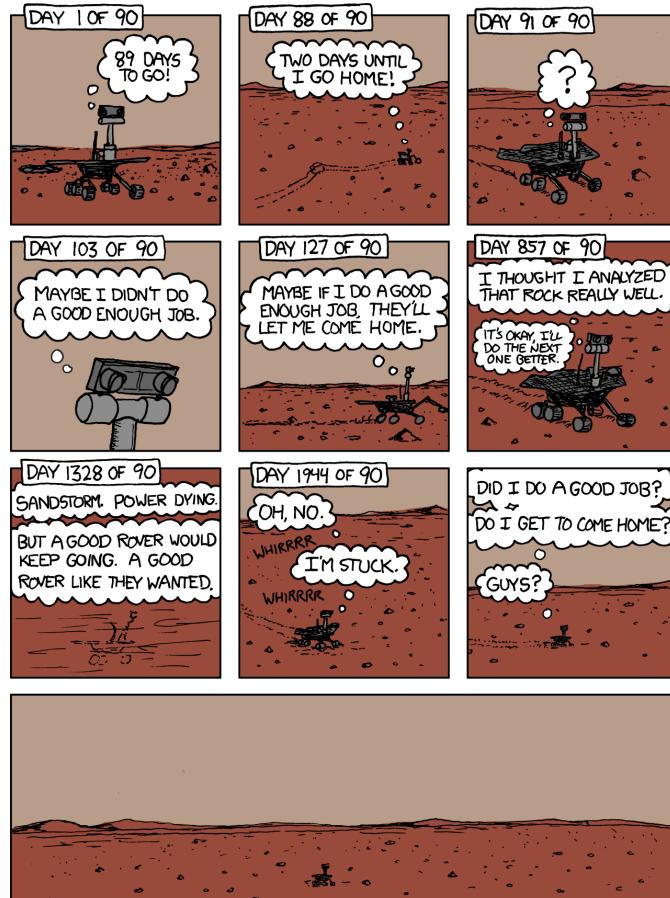
ently available to the human operators.

- The existence of the map itself represents a medium for communication between human and robot as the human can simply give the robot a new map if the robot goes to a new environment.
- The map, if created by the robot, can be used by humans as well, achieving two uses.

The map-based approach will require more up-front development effort to create a navigating AMR. The hope is that the development effort results in an architecture which can successfully map and navigate a variety of environments, thereby compensating for the up-front design cost over time.

Of course the primary risk of the map-based approach is that an internal representation, rather than the real world itself, is being constructed and trusted by the robot. If that model diverges from reality,¹⁴ then the robot's behaviour may be undesirable at best or wrong at worst, even if the raw sensor values of the robot are only transiently incorrect.

In the remainder of this chapter, we focus on a discussion of map-based approaches and, specifically, the localisation component of these techniques. These approaches are particularly appropriate for study given their significant recent successes in enabling AMR to navigate a variety of environments, from academic research buildings to factory floors and museums around the world.



¹⁴As in if the robot gets the wrong idea about its environment and draws the wrong map.

Figure 1.6: On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown [xkcd2025].

1.4 Representing Belief

The fundamental issue which differentiates different types of map-based localisation systems is the issue of **representation**. There are two (2) specific concepts which the robot must represent, and each has its own unique possible solutions.

1. Representation of the environment,
2. The map.

What aspects of the environment are contained in this map? At what level of fidelity does the map represent the environment? These are the design questions for map representation.

The robot must also have a representation of its **belief** regarding its position on the map.

Does the robot identify a single unique position as its current position, or does it describe its position in terms of a set of possible positions? If multiple possible positions are expressed in a single belief, how are those multiple positions ranked, if at all?

These are the design questions for belief representation. Decisions along these two (2) design axes can result in varying levels of architectural complexity, computational complexity and overall localisation accuracy.

We will start by discussing belief representation. The first major branch in the classification of belief representation systems differentiates between single hypothesis and multiple hypothesis belief systems.

Single Former covers solutions in which the robot postulates its unique position,

Multiple Enables a AMR to describe the degree to which it is uncertain about its position.

A sampling of different belief and map representations is shown in figure 5.9.

1.4.1 Single Hypothesis Belief

The single hypothesis belief representation is the most direct possible postulation of an AMR's position [**reuter1999scan**].

Given some environmental map, the robot's belief about position is expressed as a single unique point on the map.

In **Fig.** 1.7, three (3) examples of a single hypothesis belief are shown using three different map representations of the same actual environment shown in **Fig.** 1.7a. In **Fig.** 1.7b, a single point is geometrically annotated as the robot's position in a continuous two-dimensional geometric map.

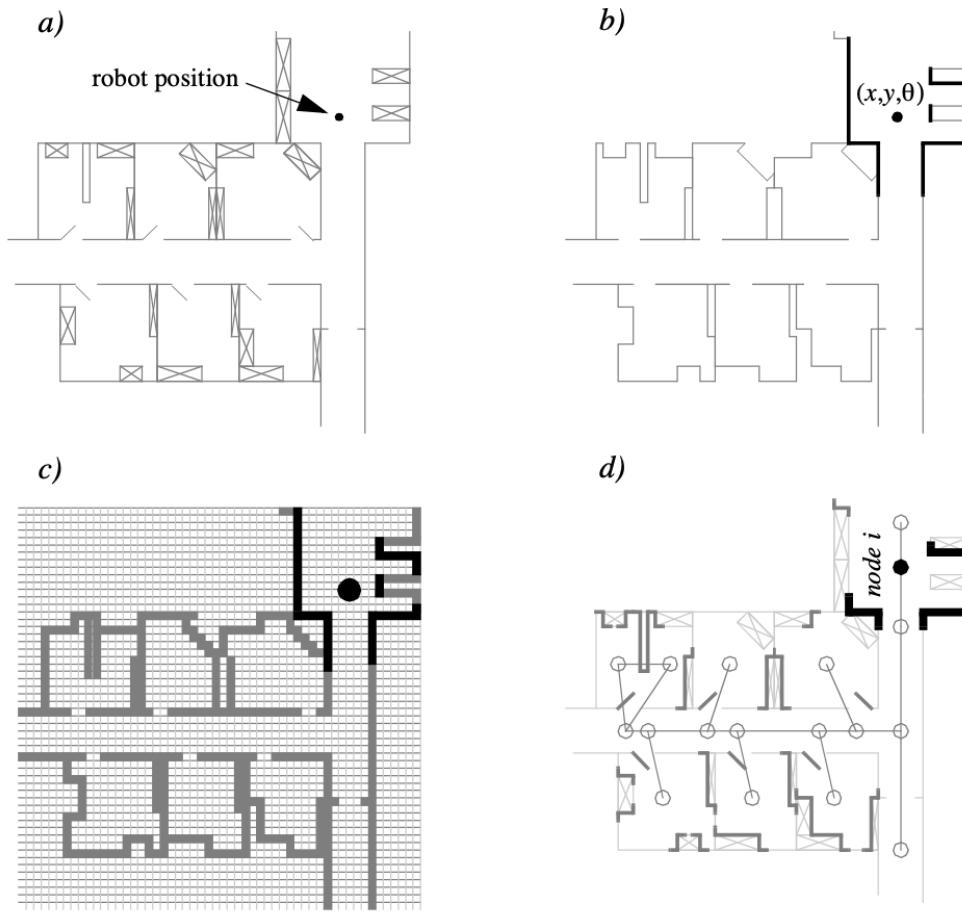


Figure 1.7: The three (3) examples of single hypotheses of position using different map representation. **a)** real map with walls, doors and furniture **b)** line-based map -> around 100 lines with two parameters **c)** occupancy grid based map -> around 3000 grid cells sizing 50x50 cm **d)** topological map using line features (Z/S-lines) and doors -> around 50 features and 18 nodes

In **Fig. 1.7c**, the map is a discrete, tessellated map, and the position is noted at the same level of fidelity as the map cell size. In **Fig. 1.7d**, the map is not geometrical at all but abstract and topological. In this case, the single hypothesis of position involves identifying a single node i in the topological graph as the robot's position.

The principal advantage of the single hypothesis representation of position stems from the fact that, given a unique belief, there is **no position ambiguity**. The unambiguous nature of this representation facilitates decision-making at the robot's cognitive level (e.g. path planning). The robot can simply assume that its belief is correct, and can then select its future actions based on its unique position.

Just as decision-making is facilitated by a single-position hypothesis, so updating the robot's belief regarding position is also facilitated, since the single position must be updated by definition to a new, single position. The challenge with this position update approach, which ultimately is the principal disadvantage of single-hypothesis representation, is that robot motion often induces uncertainty due to effectory and sensory noise.

Forcing the position update process to always generate a single hypothesis of position is challenging and, often, impossible.

1.4.2 Multiple Hypothesis Belief

In the case of multiple hypothesis beliefs regarding position, the robot tracks **NOT** just a single possible position but a possibly **infinite set of positions**. In one simple example originating in the work of Jean-Claude Latombe [5, 89], the robot's position is described in terms of a convex polygon positioned in a two-dimensional map of the environment.

This multiple hypothesis representation communicates the set of possible robot positions geometrically, with no preference ordering over the positions. Each point in the map is simply either contained by the polygon and, therefore, in the robot's belief set, or outside the polygon and thereby excluded. Mathematically, the position polygon serves to partition the space of possible robot positions. Such a polygonal representation of the multiple hypothesis belief can apply to a continuous, geometric map of the environment or, alternatively, to a tessellated, discrete approximation to the continuous environment.

It may be useful, however, to incorporate some ordering on the possible robot positions, capturing the fact that some robot positions are likelier than others. A strategy for representing a continuous multiple hypothesis belief state along with a preference ordering over possible positions is to model the belief as a mathematical distribution. For example, [42,47] note the robot's position belief using an X,Y point in the two-dimensional environment as the mean μ plus a standard deviation parameter σ , thereby defining a Gaussian distribution. The intended interpretation is that the distribution at each position represents the probability assigned to the robot being at that location. This representation is particularly amenable to mathematically defined tracking functions, such as the Kalman Filter, that are designed to operate efficiently on Gaussian distributions.

An alternative is to represent the set of possible robot positions, not using a single Gaussian probability density function, but using discrete markers for each possible position. In this case, each possible robot position is individually noted along with a confidence or probability parameter (See Fig. (5.11)). In the case of a highly tessellated map this can result in thousands or even tens of thousands of possible robot positions in a single belief state.

The key advantage of the multiple hypothesis representation is that the robot can explicitly maintain uncertainty regarding its position. If the robot only acquires partial information regarding position from its sensors and effectors, that information can conceptually be incorporated in an updated belief.

A more subtle advantage of this approach revolves around the robot's ability to explicitly measure its own degree of uncertainty regarding position. This advantage is the key to a class of localisation and navigation solutions in which the robot not only reasons about reaching a particular goal, but reasons about the future trajectory of its own belief state. For instance, a robot may choose paths

that minimise its future position uncertainty. An example of this approach is [90], in which the robot plans a path from point A to B that takes it near a series of landmarks in order to mitigate localisation difficulties. This type of explicit reasoning about the effect that trajectories will have on the quality of localisation requires a multiple hypothesis representation.

One of the fundamental disadvantages of the multiple hypothesis approaches involves decision-making. If the robot represents its position as a region or set of possible positions, then how shall it decide what to do next? Figure 5.11 provides an example. At position 3, the robot's belief state is distributed among 5 hallways separately. If the goal of the robot is to travel down one particular hallway, then given this belief state what action should the robot choose?

The challenge occurs because some of the robot's possible positions imply a motion trajectory that is inconsistent with some of its other possible positions. One approach that we will see in the case studies below is to assume, for decision-making purposes, that the robot is physically at the most probable location in its belief state, then to choose a path based on that current position. But this approach demands that each possible position have an associated probability.

In general, the right approach to such a decision-making problems would be to decide on trajectories that eliminate the ambiguity explicitly. But this leads us to the second major disadvantage of the multiple hypothesis approaches. In the most general case, they can be computationally very expensive. When one reasons in a three dimensional space of discrete possible positions, the number of possible belief states in the single hypothesis case is limited to the number of possible positions in the 3D world. Consider this number to be N . When one moves to an arbitrary multiple hypothesis representation, then the number of possible belief states is the power set of N , which is far larger: 2^N . Thus explicit reasoning about the possible trajectory of the belief state over time quickly becomes computationally untenable as the size of the environment grows. There are, however, specific forms of multiple hypothesis representations that are somewhat more constrained, thereby avoiding the computational explosion while allowing a limited type of multiple hypothesis belief. For example, if one assumes a Gaussian distribution of probability centered at a single position, then the problem of representation and tracking of belief becomes equivalent to Kalman Filtering, a straightforward mathematical process described below. Alternatively, a highly tessellated map representation combined with a limit of 10 possible positions in the belief state, results in a discrete update cycle that is, at worst, only 10x more computationally expensive than single hypothesis belief update.

In conclusion, the most critical benefit of the multiple hypothesis belief state is the ability to maintain a sense of position while explicitly annotating the robot's uncertainty about its own position. This powerful representation has enabled robots with limited sensory information to navigate robustly in an array of environments, as we shall see in the case studies below.

1.5 Representing Maps

The problem of representing the environment in which an AMR moves is a dual of the problem of representing the robot's possible position or positions. Decisions made regarding the environmental representation can have impact on the choices available for robot position representation.

Often the fidelity of the position representation is bounded by the fidelity of the map.

There are three (3) fundamental relationships which must be understood when choosing a particular map representation:

1. The precision of the map must appropriately match the precision with which the robot needs to achieve its goals.
2. The precision of the map and the type of features represented must match the precision and data types returned by the robot's sensors.
3. The complexity of the map representation has direct impact on the computational complexity of reasoning about mapping, localisation and navigation.

Using the aforementioned criteria, we identify and discuss critical design choices in creating a map representation. Each such choice has great impact on the relationships, and on the resulting robot localisation architecture. As we will see, the choice of possible map representations is broad, if not expansive. Selecting an appropriate representation requires understanding all of the trade-offs inherent in that choice as well as understanding the specific context in which a particular AMR implementation must perform localisation.

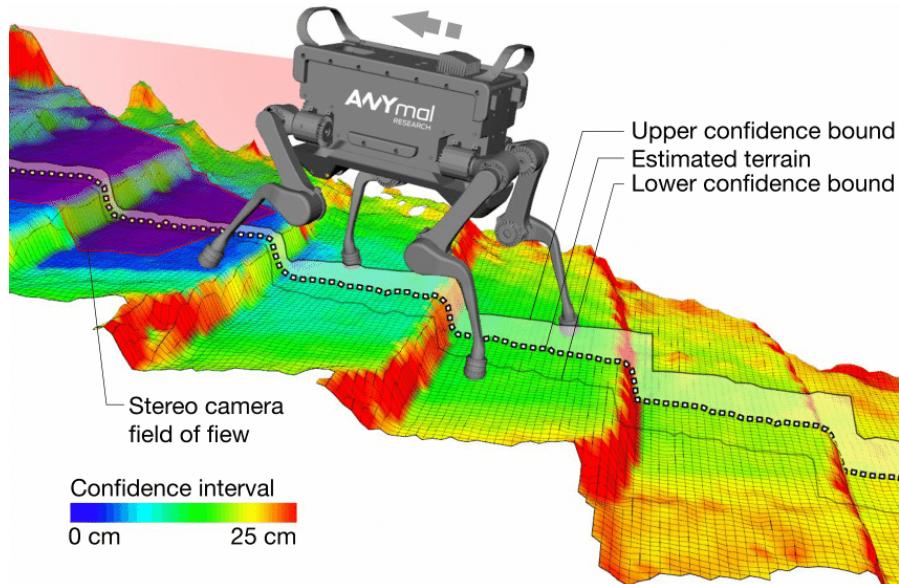


Figure 1.8: The presented robot-centric mapping framework enables mobile robots to create consistent elevation maps of the terrain. Mapping does not necessarily need to be done only in 2D as robots which will be used in outdoor environment would need the height of the map as well [fankhauser2018probabilistic].

1.5.1 Continuous Representation

A continuous-valued map is one method for **exact** decomposition of the environment. The position of environmental features can be mapped precisely in continuous space.

AMR implementations to date use continuous maps only in two (2) dimensional representations, as increasing the number of dimensions can result in high computational load on the AMR navigation computer.

A common approach is to combine the exactness of a continuous representation with the compactness of the closed world assumption. This means that one assumes the representation will specify all environmental objects in the map, and that any area in the map which is devoid of objects has no objects in the corresponding portion of the environment. Therefore, the total storage needed in the map is proportional to the density of objects in the environment, and a sparse environment can be represented by a low-memory map.

One example of such a representation, shown in **Fig. 1.9**, is a 2D representation in which polygons represent all obstacles in a continuous-valued coordinate space. This is similar to the method used by Latombe [5, 113] and others to represent environments for AMR path planning techniques. In the case of [5, 113], most of the experiments are in fact simulations run exclusively within the computer's memory. Therefore, no real effort would have been expended to attempt to use sets of polygons to describe a real-world environment, such as a park or office building.

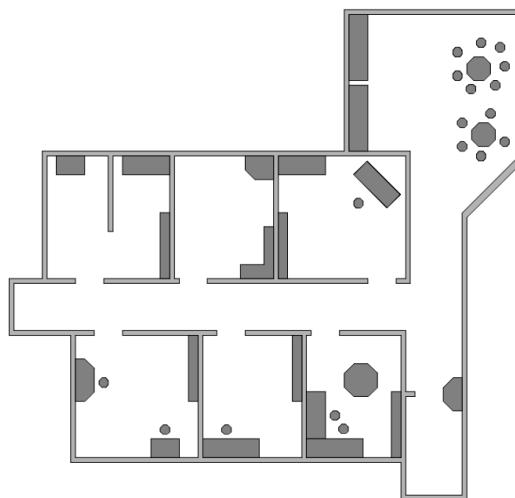


Figure 1.9: A continuous representation using polygons as environmental obstacles.

In other work in which real environments must be captured by the maps, there seems to be a trend towards **selectivity** and **abstraction**. The human map-maker tends to capture on the map, for localisation purposes, only objects that can be detected by the robot's sensors and, furthermore, only a subset of the features of real-world objects.

It should be immediately apparent that geometric maps can capably represent the physical locations of objects without referring to their texture, colour, elasticity, or any other such secondary features that do not relate directly to position and space.

In addition to this level of abstraction, an AMR map can further reduce memory usage by capturing only **aspects of object geometry** which are **immediately relevant** to localisation. For example all objects may be approximated using very simple convex polygons,¹⁵ sacrificing map felicity for the sake of computational speed.

¹⁵A convex polygon is any shape that has all interior angles that measure less than 180 degrees

One excellent example involves **line extraction**. Many indoor AMR rely upon laser range-finding devices to recover distance readings to nearby objects. Such robots can automatically extract best-fit lines from the dense range data provided by thousands of points of laser strikes. Given such a line extraction sensor, an appropriate continuous mapping approach is to populate the map with a set of infinite lines. The continuous nature of the map guarantees that lines can be positioned at arbitrary positions in the plane and at arbitrary angles. The abstraction of real environmental objects such as walls and intersections captures only the information in the map representation that matches the type of information recovered by the AMR's rangefinding sensor.

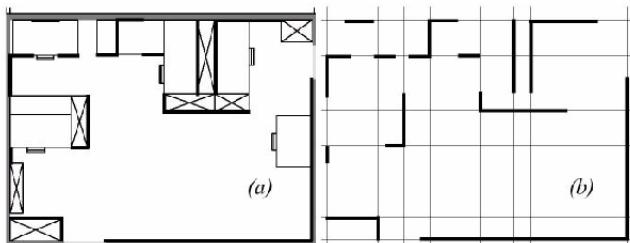


Figure 1.10: Example of a continuous-valued line representation of EPFL. left: real map right: representation with a set of infinite lines.

Fig. 1.10 shows a map of an indoor environment at EPFL using a continuous line representation. Note that the only environmental features captured by the map are straight lines, such as those found at corners and along walls. This represents not only a sampling of the real world of richer features, but also a simplification, for an actual wall may have texture and relief that is not captured by the mapped line. The impact of continuous map representations on position representation is primarily positive. In the case of single hypothesis position representation, that position may be specified as any continuous-valued point in the coordinate space, and therefore extremely high accuracy is possible. In the case of multiple hypothesis position representation, the continuous map enables two types of multiple position representation. In one case, the possible robot position may be depicted as a geometric shape in the hyperplane, such that the robot is known to be within the bounds of that shape. This is shown in Figure 5.30, in which the position of the robot is depicted by an oval bounding area. Yet, the continuous representation does not disallow representation of position in the form of a discrete set of possible positions. For instance, in [111] the robot position belief state is captured by sampling nine continuous-valued positions from within a region near the robot's best known position. This algorithm captures, within a continuous space, a discrete sampling of possible robot positions. In summary, the key advantage of a continuous map representation is

the potential for high accuracy and expressiveness with respect to the environmental configuration as well as the robot position within that environment. The danger of a continuous representation is that the map may be computationally costly. But this danger can be tempered by employing abstraction and capturing only the most relevant environmental features. Together with the use of the closed world assumption, these techniques can enable a continuous-valued map to be no more costly, and sometimes even less costly, than a standard discrete representation.

1.5.2 Decomposition Methods

In previous section, we discussed one method of simplification, in which the continuous map representation contains a set of infinite lines which approximate real-world environmental lines based on a two-dimensional slice of the world.

Basically this transformation from the real world to the map representation is a filter that removes all non-straight data and furthermore extends line segment data into infinite lines that require fewer parameters.

A more dramatic form of simplification is abstraction:

a general decomposition and selection of environmental features.

In this section, we explore decomposition as applied in its more extreme forms to the question of map representation. Why would one radically decompose the real environment during the design of a map representation? The immediate disadvantage of decomposition and abstraction is the loss of fidelity between the map and the real world. Both qualitatively, in terms of overall structure, and quantitatively, in terms of geometric precision, a highly abstract map does not compare favourably to a high-fidelity map.

Despite this disadvantage, decomposition and abstraction may be useful if the abstraction can be planned carefully so as to capture the relevant, useful features of the world while discarding all other features. The advantage of this approach is that the map representation can potentially be minimised. Furthermore, if the decomposition is hierarchical, such as in a pyramid of recursive abstraction, then reasoning and planning with respect to the map representation may be computationally far superior to planning in a fully detailed world model.

A standard, lossless form of opportunistic decomposition is termed exact cell decomposition. This method, introduced by [5], achieves decomposition by selecting boundaries between discrete cells based on geometric criticality.

Figure 5.14 depicts an exact decomposition of a planar workspace populated by polygonal obstacles. The map representation tessellates the space into areas of free space. The representation can be extremely compact because each such area is actually stored as a single node, shown in the graph at the bottom of Figure 5.14.

The underlying assumption behind this decomposition is that the particular position of a robot within each area of free space does not matter. What matters is the robot's ability to traverse from each area of free space to the adjacent areas. Therefore, as with other representations we will see, the resulting graph captures the adjacency of map locales. If indeed the assumptions are valid and the robot does not care about its precise position within a single area, then this can be an effective representation that nonetheless captures the connectivity of the environment.

Such an exact decomposition is not always appropriate. Exact decomposition is a function of the particular environment obstacles and free space. If this information is expensive to collect or even unknown, then such an approach is not feasible.

An alternative is fixed decomposition, in which the world is tessellated, transforming the continuous real environment into a discrete approximation for the map. Such a transformation is demonstrated in Figure 5.15, which depicts what happens to obstacle-filled and free areas during this transformation. The key disadvantage of this approach stems from its inexact nature. It is possible for narrow passageways to be lost during such a transformation, as shown in Figure 5.15. Formally this means that fixed decomposition is sound but not complete. Yet another approach is adaptive cell decomposition as presented in Figure 5.16.

The concept of fixed decomposition is extremely popular in AMRics; it is perhaps the single most common map representation technique currently utilised. One very popular

version of fixed decomposition is known as the occupancy grid representation [91]. In an occupancy grid, the environment is represented by a discrete grid, where each cell is either filled (part of an obstacle) or empty (part of free space). This method is of particular value when a robot is equipped with range-based sensors because the range values of each sensor, combined with the absolute position of the robot, can be used directly to update the filled/ empty value of each cell. In the occupancy grid, each cell may have a counter, whereby the value 0 indicates that the cell has not been "hit" by any ranging measurements and, therefore, it is likely free space. As the number of ranging strikes increases, the cell's value is incremented and, above a certain threshold, the cell is deemed to be an obstacle. By discounting the values of cells over time, both hysteresis and the possibility of transient obstacles can be represented using this occupancy grid approach. Figure 5.17 depicts an occupancy grid representation in which the darkness of each cell is proportional to the value of its counter. One commercial robot that uses a standard occupancy grid for mapping and navigation is the Cye robot [112].

There remain two main disadvantages of the occupancy grid approach. First, the size of the map in robot memory grows with the size of the environment and, if a small cell size is used, this size can quickly become untenable. This occupancy grid approach is not compatible with the closed world assumption, which enabled continuous representations to have potentially very small memory requirements in large, sparse environments. In contrast, the occupancy grid must have memory set aside for every cell in the matrix. Furthermore, any fixed decomposition method such as this imposes a geometric grid on the world *a priori*, regardless of the environmental details. This can be inappropriate in cases where geometry is not the most salient feature of the environment. For these

reasons, an alternative, called topological decomposition, has been the subject of some exploration in AMRics. Topological approaches avoid direct measurement of geometric environmental qualities, instead concentrating on characteristics of the environment that are most relevant to the robot for localisation. Formally, a topological representation is a graph that specifies two things: nodes and the connectivity between those nodes. Insofar as a topological representation is intended for the use of a AMR, nodes are used to denote areas in the world and arcs are used to denote adjacency of pairs of nodes. When an arc connects two nodes, then the robot can traverse from one node to the other without requiring traversal of any other intermediary node. Adjacency is clearly at the heart of the topological approach, just as adjacency in a cell decomposition representation maps to geometric adjacency in the real world. However, the topological approach diverges in that the nodes are not of fixed size nor even specifications of free space. Instead, nodes document an area based on any sensor discriminant such that the robot can recognise entry and exit of the node. Figure 5.18 depicts a topological representation of a set of hallways and offices in an indoor

environment. In this case, the robot is assumed to have an intersection detector, perhaps using sonar and vision to find intersections between halls and between halls and rooms. Note that nodes capture geometric space and arcs in this representation simply represent connectivity. Another example of topological representation is the work of Dudek [49], in which the goal is to create a AMR that can capture the most interesting aspects of an area for human consumption. The nodes in Dudek's representation are visually striking locales rather than route intersections. In order to navigate using a topological map robustly, a robot must satisfy two constraints. First, it must have a means for detecting its current position in terms of the nodes of the topological graph. Second, it must have a means for traveling between nodes using robot motion. The node sizes and particular dimensions must be optimised to match the sensory discrimination of the AMR hardware. This ability to "tune" the representation to the robot's particular sensors can be an important advantage of the topological approach. However, as the map representation drifts further away from true geometry, the expressiveness of the representation for accurately and precisely describing a robot position is lost. Therein lies the compromise between the discrete cell-based map representations and the topological representations. Interestingly, the continuous map representation has the potential to be both compact like a topological representation and precise as with all direct geometric representations. Yet, a chief motivation of the topological approach is that the environment may contain important non-geometric features - features that have no ranging relevance but are useful for localisation. In Chapter 4 we described such whole-image vision-based features. In contrast to these whole-image feature extractors, often spatially localised landmarks are artificially placed in an environment to impose a particular visual-topological connectivity upon the environment. In effect, the artificial landmark can impose artificial structure. Examples of working systems operating with this landmark-based strategy have also demonstrated success. Latombe's landmark-based navigation research 89 has been implemented on real-world indoor AMRs that employ paper landmarks attached to the ceiling as the locally observable features. Chips the museum robot is another robot that uses man-made landmarks to obviate the localisation problem. In this case, a bright pink square serves as a landmark with dimensions and color signature that would be hard to accidentally reproduce in a museum environment [88]. One such museum landmark is shown in Figure (5.19). In summary, range is clearly not the only measurable and useful

environmental value for a AMR. This is particularly true due to the advent of color vision as well as laser rangefinding, which provides reflectance information in addition to range information. Choosing a map representation for a particular AMR requires first understanding the sensors available on the AMR and second understanding the AMR's functional requirements (e.g. required goal precision and accuracy).

1.5.3 Current Challenges

Previous section describe major design decisions with regards to map representation choices. There are, however, fundamental real-world features which AMR map representations do not work as well. These continue to be the subject of open research, and several such challenges are described below.

The real world is **dynamic**. As AMRs come to work and move in the same spaces as humans, they will encounter:

- moving people,
- cars,
- strollers, and
- transient obstacles.

This is particularly true when one considers a home setting with which domestic robots will someday need to contend.

The map representations described previously do not, in general, have **explicit methods** for identifying and distinguishing between permanent obstacles (e.g. walls, doorways, etc.) and transient obstacles (e.g., humans, shipping packages, etc.). The current state of the art in terms of AMR sensors is partly to blame for this shortcoming. Although vision research is rapidly advancing, robust sensors that discriminate between moving animals and static structures from a moving reference frame are not yet available. Furthermore, estimating the motion vector of transient objects remains a research problem.

Usually, the assumption behind the above map representations is that all objects on the map are effectively **static**. Partial success can be achieved by discounting mapped objects over time. For example, occupancy grid techniques can be more robust to dynamic settings by introducing temporal discounting, effectively treating transient obstacles as noise. The more challenging process of map creation is particularly fragile to environment dynamics; most mapping techniques generally require that the environment be free of moving objects during the mapping process. One exception to this limitation involves topological representations. Because precise geometry is not important, transient objects have little effect on the mapping or localisation process, subject to the critical constraint that the transient objects must not change the topological connectivity of the environment. Still, neither the occupancy grid representation nor a topological approach is actively recognizing and

representing transient objects as distinct from both sensor error and permanent map features.

As vision sensing provides more robust and more informative content regarding the transience and motion details of objects in the world, researchers will in time propose representations that make use of that information. A classic example involves occlusion by human crowds. Museum tour guide robots¹⁶ generally suffer from an extreme amount of occlusion. If the robot's sensing suite is located along the robot's body, then the robot is effectively blind when a group of human visitors completely surrounds the robot. This is because its map contains only environment features that are, at that point, fully hidden from the robot's sensors by the wall of people. In the best case, the robot should recognise its occlusion and make no effort to localise using these invalid sensor readings. In the worst case, the robot will localise with the fully occluded data, and will update its location incorrectly. A vision sensor that can discriminate the local conditions of the robot (e.g. we are surrounded by people) can help eliminate this error mode.

A second open challenge in AMR localisation involves the traversal of open spaces. Existing localisation techniques generally depend on local measures such as range, thereby demanding environments that are somewhat densely filled with objects that the sensors can detect and measure. Wide open spaces such as parking lots, fields of grass and indoor open-spaces such as those found in convention centres or expos pose a difficulty for such systems due to their relative sparseness. Indeed, when populated with humans, the challenge is exacerbated because any mapped objects are almost certain to be occluded from view by the people.

Once again, more recent technologies provide some hope for overcoming these limitations. Both vision and state-of-the-art laser range-finding devices offer outdoor performance with ranges of up to a hundred meters and more. Of course, GPS performs even better. Such long-range sensing may be required for robots to localise using distant features.

This trend teases out a hidden assumption underlying most topological map representations. Usually, topological representations make assumptions regarding spatial locality:

a node contains objects and features that are themselves within that node.

The process of map creation therefore involves making nodes which are, in their own self-contained way, recognizable by virtue of the objects contained within the node. Therefore, in an indoor environment, each room can be a separate node. This is a reasonable assumption as each room will have a layout and a set of belongings that are **unique** to that room.

However, consider the outdoor world of a wide-open park.

Where should a single node end and the next node begin?

The answer is unclear as objects which are far away from the current node, or position, can give information for the localisation process. For example, the hump of a hill at the horizon, the position of a river in the valley and the trajectory of the Sun all are non-local features that have great bearing on one's ability to infer current position.



¹⁶An Example of a museum tour guide robot used in the National Museum of Korea [yonhap2018].

The spatial locality assumption is violated and, instead, replaced by a visibility criterion:

the node or cell may need a mechanism for representing objects that are measurable and visible from that cell.

Once again, as sensors and outdoor locomotion mechanisms improve, there will be greater urgency to solve problems associated with localisation in wide-open settings, with and without GPS-type global localisation sensors.¹⁷

We end this section with one final open challenge that represents one of the fundamental academic research questions of robotics: **sensor fusion**.

Information

Sensor Fusion

A variety of measurement types are possible using off-the-shelf robot sensors, including heat, range, acoustic and light-based reflectivity, color, texture, friction, etc. Sensor fusion is a research topic closely related to map representation. Just as a map must embody an environment in sufficient detail for a robot to perform localisation and reasoning, sensor fusion demands a representation of the world that is sufficiently general and expressive that a variety of sensor types can have their data correlated appropriately, strengthening the resulting percepts well beyond that of any individual sensor's readings.

An implementation example implementation of sensor fusion to date is that of neural network classifier. Using this technique, any number and any type of sensor values may be jointly combined in a network that will use whatever means necessary to optimise its classification accuracy. For the AMR that must use a human-readable internal map representation, no equally general sensor fusion scheme has yet been born. It is reasonable to expect that, when the sensor fusion problem is solved, integration of a large number of disparate sensor types may easily result in sufficient discriminatory power for robots to achieve real-world navigation, even in wide-open and dynamic circumstances such as a public square filled with people.

1.6 Probabilistic Map-Based Localisation

1.6.1 Introduction

As stated previously, multiple hypothesis position representation is advantageous because the robot can explicitly track its own beliefs regarding its possible positions in the environment. Ideally, the robot's belief state will change, over time, as is consistent with its motor outputs and perceptual inputs. One geometric approach to multiple hypothesis representation, mentioned earlier, involves identifying the possible positions of the robot by specifying a polygon in the environmental representation [113]. This method does not provide any indication of the relative chances between various possible robot positions. Probabilistic techniques differ from this because they explicitly identify probabilities with the possible robot positions, and for this reason these methods have been the focus of recent research. In the following sections we present two classes of probabilistic localisation. The first class, Markov localisation, uses an explicitly specified probability distribution across all possible robots positions. The second method, Kalman filter localisation, uses a Gaussian probability density representation of robot position and scan matching for localisation. Unlike Markov localisation, Kalman filter localisation does not independently consider each possible pose in the robot's configuration space. Interestingly, the Kalman filter localization process results from the Markov localisation axioms if the robot's position uncertainty is assumed to have a Gaussian form [28 page 43-44]. Before discussing each method in detail, we present the general robot localisation problem and solution strategy. Consider a AMR moving in a known environment. As it starts to move, say from a precisely known location, it can keep track of its motion using odometry. Due to odometry uncertainty, after some movement the robot will become very uncertain about its position (see section 5.2.4). To keep position uncertainty from growing unbounded, the robot must localise itself in relation to its environment map. To localise, the robot might use its on-board sensors (ultrasonic, range sensor, vision) to make observations of its environment. The information provided by the robot's odometry, plus the information provided by such exteroceptive observations can be combined to enable the robot to localise as well as possible with respect to its map. The processes of updating based on proprioceptive sensor values and exteroceptive sensor values are often separated logically, leading to a general two-step process for robot position update. Action update represents the application of some action model Act to the AMR's proprioceptive encoder measurements o and prior belief state s to yield a new belief s' representing the robot's belief about its current position. Note that throughout this chapter we will assume that the robot's proprioceptive encoder measurements are used as the best possible measure of its actions over time. If, for instance, a differential drive robot had motors without encoders connected to its wheels and employed open-loop control, then instead of encoder measurements the robot's highly uncertain estimates of wheel spin would need to be incorporated. We ignore such cases and therefore have a simple formula:

$$s'_t = \text{Act}(o_t, s_{t-1}) \quad (1.1)$$

Perception update represents the application of some perception model See to the AMR's exteroceptive sensor inputs i and updated belief state s' to yield a refined belief s'' representing the

robot's current position:

$$s_t = \text{See}\left(i_t, s'_{t-1}\right) \quad (1.2)$$

The perception model See and sometimes the action model Act are abstract functions of both the map and the robot's physical configuration.¹⁸

¹⁸such as sensors and their positions, kinematics, etc.

In general, the action update process **contributes uncertainty** to the robot's belief about position:

encoders have error and therefore motion is somewhat nondeterministic.

In contrast, perception update generally **refines** the belief state. Sensor measurements, when compared to the robot's environmental model, tend to provide clues regarding the robot's possible position.

In the case of Markov localisation, the robot's belief state is usually represented as separate probability assignments for every possible robot pose in its map. The action update and perception update processes must update the probability of every cell in this case. Kalman filter localisation represents the robot's belief state using a single, well-defined Gaussian probability density function, and therefore retains just a μ and σ parameterisation of the robot's belief about position with respect to the map. Updating the parameters of the Gaussian distribution is all that is required. This fundamental difference in the representation of belief state leads to the following advantages and disadvantages of the two (2) methods, as presented in [**brock1999high**]:

- Markov localization allows for localization starting from any unknown position and can thus recover from ambiguous situations because the robot can track multiple, completely disparate possible positions. However, to update the probability of all positions within the whole state space at any time requires a discrete representation of the space (grid). The required memory and computational power can thus limit precision and map size.
- Kalman filter localization tracks the robot from an initially known position and is inherently both precise and efficient. In particular, Kalman filter localization can be used in continuous world representations. However, if the uncertainty of the robot becomes too large (e.g. due to a robot collision with an object) and thus not truly unimodal, the Kalman filter can fail to capture the multitude of possible robot positions and can become irrevocably lost.

Improvements are achieved or proposed by either only updating the state space of interest within the Markov approach [**borensteinfast**] or by combining both methods to create a hybrid localization system [**brock1999high**].

We will now look at them in great detail.

1.6.2 Markov Localisation

Markov localization tracks the robot's belief state using an arbitrary probability density function to represent the robot's position. In practice, all known Markov localization systems implement this generic belief representation by first tessellating the robot configuration space into a finite, discrete number of possible robot poses in the map. In actual applications, the number of possible poses can range from several hundred positions to millions of positions.

Given such a generic conception of robot position, a powerful update mechanism is required that can compute the belief state that results when new information (e.g. encoder values and sensor values) is incorporated into a prior belief state with arbitrary probability density. The solution is born out of probability theory, and so the next section describes the foundations of probability theory that apply to this problem, notably Bayes formula. Then, two subsequent subsections provide case studies, one robot implementing a simple feature-driven topological representation of the environment [**brooks1986robust**], and the other using a geometric grid-based map [**borensteinfast**].

Application of Probability for Localisation

Given a discrete representation of robot positions, to express a belief state we wish to assign to each possible robot position a probability that the robot is indeed at that position.

From probability theory we use the term $P(A)$ to denote the probability that A is true. This is also called the prior probability of A because it measures the probability that A is true independent of any additional knowledge we may have.

For example we can use $P(r_t = l)$ to denote the prior probability that the robot r is at position l at time t .

In practice, we wish to compute the probability of each individual robot position given the encoder and sensor evidence the robot has collected. For this, we use the term $P(A|B)$ to denote the conditional probability of A given that we know B .

For example, we use $P(r_t = l|i_t)$ to denote the probability that the robot is at position l given that the robot's sensor inputs i .

The question is,

how can a term such as $P(r_t = l|i_t)$ be simplified to its constituent parts so that it can be computed?

The answer lies in the product rule, which states:

$$P(A \wedge B) = P(A|B) P(B) \quad (1.3)$$

¹⁹To simplify notation we will be using the wedge (\wedge) symbol to denote AND, and the vee (\vee) symbol to denote OR.

The equation given in Eq. (1.3) is relatively straightforward, as the probability of both A and¹⁹ B being true is being related to B being true and the other being conditionally true. But you should be able to convince yourself that the alternate equation is equally correct:

$$P(A \wedge B) = P(B|A) P(A) \quad (1.4)$$

Using both Eq. (1.3) and Eq. (1.4) together, we can derive [Bayes formula](#) for computing $P(A|B)$:

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)} \quad (1.5)$$

We use Bayes rule to compute the robot's new belief state as a function of its sensory inputs and its former belief state. But to do this properly, we must recall the basic goal of the Markov localisation approach:

a discrete set of possible robot positions L are represented.

The belief state of the robot must assign a probability $P(r_t = l)$ for each location l in L .

The See function described in Eq. (1.2) expresses a mapping from a belief state and sensor input to a refined belief state. To do this, we must update the probability associated with each position l in L , and we can do this by directly applying Bayes formula to every such l .

In denoting this, we will stop representing the temporal index t for simplicity and will further use $P(l)$ to mean $P(r = l)$:

$$P(l|i) = \frac{P(i|l) P(l)}{P(i)} \quad (1.6)$$

The value of $P(l|i)$ is key to Eq. (1.6), and this probability of a sensor input at each robot position must be computed using some model. An obvious strategy would be to consult the robot's map, identifying the probability of particular sensor readings with each possible map position, given knowledge about the robot's sensor geometry and the mapped environment. The value of $P(l)$ is easy to recover in this case. It is simply the probability $P(r = l)$ associated with the belief state before the perceptual update process.

Finally, note that the denominator $P(i)$ does **NOT** depend upon l ; that is, as we apply Eq. (1.6) to all positions l in L , the denominator never varies.

Because it is effectively constant, in practice this denominator is usually dropped and, at the end of the perception update step, all probabilities in the belief state are re-normalized to sum at 1.0.

Now consider the Act function of Eq. (1.1). Act maps a former belief state and encoder measurement (i.e. robot action) to a new belief state. To compute the probability of position l in the new belief state, one must integrate over all the possible ways in which the robot may have reached l according

to the potential positions expressed in the former belief state. This is subtle but fundamentally important. The same location l can be reached from multiple source locations with the same encoder measurement o because the encoder measurement is uncertain. Temporal indices are required in this update equation:

$$P(l_t|o_t) = \int P(l_t|l'_{t-1}, o_t) P(l'_{t-1}) dl'_{t-1} \quad (1.7)$$

Thus, the total probability for a specific position l is built up from the individual contributions from every location l' in the former belief state given encoder measurement o . Equations 5.21 and 5.22 form the basis of Markov localization, and they incorporate the Markov assumption. Formally, this means that their output is a function only of the robot's previous state and its most recent actions (odometry) and perception. In a general, non-Markovian situation, the state of a system depends upon all of its history. After all, the value of a robot's sensors at time t do not really depend only on its position at time t . They depend to some degree on the trajectory of the robot over time; indeed on the entire history of the robot. For example, the robot could have experienced a serious collision recently that has biased the sensor's behavior. By the same token, the position of the robot at time t does not really depend only on its position at time $t-1$ and its odometric measurements. Due to its history of motion, one wheel may have worn more than the other, causing a left-turning bias over time that affects its current position. So the Markov assumption is, of course, not a valid assumption. However the Markov assumption greatly simplifies tracking, reasoning and planning and so it is an approximation that continues to be extremely popular in mobile robotics.

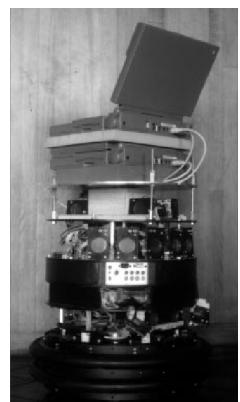
Application: Markov Localisation using a Topological Map

A straightforward application of Markov localization is possible when the robot's environment representation already provides an appropriate decomposition. This is the case when the environment representation is purely topological.

Consider a contest in which each robot is to receive a topological description of the environment. The description would describe only the connectivity of hallways and rooms, with no mention of geometric distance. In addition, this supplied map would be imperfect, containing several false arcs (e.g. a closed door). Such was the case for the 1994 AAAI National Robot Contest, at which each robot's mission was to use the supplied map and its own sensors to navigate from a chosen starting position to a target room.

Dervish²⁰, the winner of this contest, employed probabilistic Markov localization and used just this multiple hypothesis belief state over a topological environmental representation. We now describe Dervish as an example of a robot with a topological representation and a probabilistic localization algorithm.

Dervish, shown in Figure 5.20, includes a sonar arrangement custom-designed for the 1994 AAAI National Robot Contest. The environment in this contest consisted of a rectilinear indoor office space filled with real office furniture as obstacles. Traditional sonars are arranged radially around the robot in a ring. Robots with such sensor configurations are subject to both tripping over short objects below the ring and to decapitation by tall objects (such as ledges, shelves and tables) that



²⁰

are above the ring. Dervish's answer to this challenge was to arrange one pair of sonars diagonally upward to detect ledges and other overhangs. In addition, the diagonal sonar pair also proved to ably detect tables, enabling the robot to avoid wandering underneath tall tables. The remaining sonars were clustered in sets of sonars, such that each individual transducer in the set would be at a slightly varied angle to minimize specularity. Finally, two sonars near the robot's base were able to detect low obstacles such as paper cups on the floor.

We have already noted that the representation provided by the contest organizers was purely topological, noting the connectivity of hallways and rooms in the office environment. Thus, it would be appropriate to design Dervish's perceptual system to detect matching perceptual events: the detection and passage of connections between hallways and offices.

This abstract perceptual system was implemented by viewing the trajectory of sonar strikes to the left and right sides of Dervish over time. Interestingly, this perceptual system would use time alone and no concept of encoder value in order to trigger perceptual events. Thus, for instance, when the robot detects a 7 to 17 cm indentation in the width of the hallway for more than one second continuously, a closed door sensory event is triggered. If the sonar strikes jump well beyond 17 cm for more than one second, an open door sensory event triggers.

Sonars have a notoriously problematic error mode known as specular reflection: when the sonar unit strikes a flat surface at a shallow angle, the sound may reflect coherently away from the transducer, resulting in a large overestimate of range. Dervish was able to filter such potential noise by tracking its approximate angle in the hallway and completely suppressing sensor events when its angle to the hallway parallel exceeded 9 degrees. Interestingly, this would result in a conservative perceptual system that would easily miss features because of this suppression mechanism, particularly when the hallway is crowded with obstacles that Dervish must negotiate. Once again, the conservative nature of the perceptual system, and in particular its tendency to issue false negatives, would point to a probabilistic solution to the localization problem so that a complete trajectory of perceptual inputs could be considered.

Dervish's environment representation was a classical topological map, identical in abstraction and information to the map provided by the contest organizers. Figure 5.21 depicts a geometric representation of a typical office environment and the topological map for the same office environment. One can place nodes at each intersection and in each room, resulting in the case of figure 5.21 with four nodes total.

Once again, though, it is crucial that one maximize the information content of the representation based on the available percepts. This means reformulating the standard topological graph shown in Figure 5.21 so that transitions into and out of intersections may both be used for position updates. Figure 5.22 shows a modification of the topological map in which just this step has been taken. In this case, note that there are 7 nodes in contrast to 4. In order to represent a specific belief state, Dervish associated with each topological node n a probability that the robot is at a physical position within the boundaries of n : $p(r = n) \cdot t$. As will become clear below, the probabilistic update used by Dervish was approximate, therefore technically one should refer to the resulting values as likelihoods

rather than probabilities.

The perception update process for Dervish functions precisely as in Equation (5.21). Perceptual events are generated asynchronously, each time the feature extractor is able to recognize a large-scale feature (e.g. doorway, intersection) based on recent ultrasonic values. Each perceptual event consists of a percept-pair (a feature on one side of the robot or two features on both sides).

	Wall	Closed Door	Open Door	Open Hallway	Foyer
Nothing Detected	0.70	0.40	0.05	0.001	0.30
Closed Door Detected	0.30	0.60	0	0	0.05
Open Door Detected	0	0	0.90	0.10	0.15
Closed Hallway Detected	0	0	0.001	0.90	0.5

Table 1.1: The certainty matrix for the robot [nourbakhsh1995dervish].

Given a specific percept pair i , Equation (5.21) enables the likelihood of each possible position n to be updated using the formula:

$$P(n|i) = P(i|n) \quad (1.8)$$

The value of $p(n)$ is already available from the current belief state of Dervish, and so the challenge lies in computing $p(i|n)$. The key simplification for Dervish is based upon the realization that, because the feature extraction system only extracts 4 total features and because a node contains (on a single side) one of 5 total features, every possible combination of node type and extracted feature can be represented in a 4×5 table. Dervish's certainty matrix (shown in Table 5.1) is just this lookup table. Dervish makes the simplifying assumption that the performance of the feature detector (i.e. the probability that it is correct) is only a function of the feature extracted and the actual feature in the node. With this assumption in hand, we can populate the certainty matrix with confidence estimates for each possible pairing of perception and node type. For each of the five world features that the robot can encounter (wall, closed door, open door, open hallway and foyer) this matrix assigns a likelihood for each of the three one-sided percepts that the sensory system can issue. In addition, this matrix assigns a likelihood that the sensory system will fail to issue a perceptual event altogether (nothing detected).

For example, using the specific values in Table 5.1, if Dervish is next to an open hallway, the likelihood of mistakenly recognizing it as an open door is 0.10. This means that for any node n that is of type Open Hallway and for the sensor value $i=$ Open door, $p(i|n) = 0.10$. Together with a specific topological map, the certainty matrix enables straightforward computation of $p(i|n)$ during the perception update process.

For Dervish's particular sensory suite and for any specific environment it intends to navigate, humans generate a specific certainty matrix that loosely represents its perceptual confidence, along

with a global measure for the probability that any given door will be closed versus opened in the real world.

Recall that Dervish has no encoders and that perceptual events are triggered asynchronously by the feature extraction processes. Therefore, Dervish has no action update step as depicted by Equation (5.22). When the robot does detect a perceptual event, multiple perception update steps will need to be performed in order to update the likelihood of every possible robot position given Dervish's former belief state. This is because there is often a chance that the robot has traveled multiple topological nodes since its previous perceptual event (i.e. false negative errors). Formally, the perception update formula for Dervish is in reality a combination of the general form of action update and perception update. The likelihood of position n given perceptual event i is calculated as in Equation (5.22):

$$P(I_t|o_t) = \int P(I_t|I'_{t-1}, o_t) P(I'_{t-1}) dI'_{t-1} \quad (1.9)$$

The value of $p(n')$ denotes the likelihood of Dervish being at position n' as represented by Dervish's former belief state. The temporal subscript $t-i$ is used in lieu of $t-1$ because for each possible position n' the discrete topological distance from n' to n can vary depending on the specific topological map. The calculation of $p(n', i)$ is performed by multiplying the probability of generating perceptual event i at position n by the probability of having failed to generate perceptual events at all nodes between n' and n :

For example (figure 5.23), suppose that the robot has only two nonzero nodes in its belief state, 1-2, 2-3, with likelihoods associated with each possible position: $p(1-2) = 1.0$ and $p(2-3) = 0.2$. For simplicity assume the robot is facing East with certainty. Note that the likelihoods for nodes 1-2 and 2-3 do not sum to 1.0. These values are not formal probabilities, and so computational effort is minimized in Dervish by avoiding normalization altogether. Now suppose that a perceptual event is generated: the robot detects an open hallway on its left and an open door on its right simultaneously. State 2-3 will progress potentially to states 3, 3-4 and 4. But states 3 and 3-4 can be eliminated because the likelihood of detecting an open door when there is only wall is zero. The likelihood of reaching state 4 is the product of the initial likelihood for state 2-3, 0.2, the likelihood of not detecting anything at node 3, (a), and the likelihood of detecting a hallway on the left and a door on the right at node 4, (b). Note that we assume the likelihood of detecting nothing at node 3-4 is 1.0 (a simplifying approximation). (a) occurs only if Dervish fails to detect the door on its left at node 3 (either closed or open), $[(0.6)(0.4) + (1-0.6)(0.05)]$, and correctly detects nothing on its right, 0.7. (b) occurs if Dervish correctly identifies the open hallway on its left at node 4, 0.90, and mistakes the right hallway for an open door, 0.10. The final formula, $(0.2)[(0.6)(0.4)+(0.4)(0.05)](0.7)[(0.9)(0.1)]$, yields a likelihood of 0.003 for state 4. This is a partial result for $p(4)$ following from the prior belief state node 2-3. Turning to the other node in Dervish's prior belief state, 1-2 will potentially progress to states 2, 2-3, 3, 3-4 and 4. Again, states 2-3, 3 and 3-4 can all be eliminated since the likelihood of detecting an open door when a wall is present is zero. The likelihood of state 2 is the product of the prior likelihood for state 1-2, (1.0), the likelihood of detecting the door on the right as an open door, $[(0.6)(0) + (0.4)(0.9)]$, and the

likelihood of correctly detecting an open hallway to the left, 0.9. The likelihood for being at state 2 is then $(1.0)(0.4)(0.9)(0.9) = 0.3$. In addition, 1-2 progresses to state 4 with a certainty factor of -6 4.3 10 , which is added to the certainty factor above to bring the total for state 4 to 0.00328. Dervish would therefore track the new belief state to be 2, 4, assigning a very high likelihood to position 2 and a low likelihood to position 4. Empirically, Dervish's map representation and localization system have proven to be sufficient for navigation of four indoor office environments: the artificial office environment created explicitly for the 1994 National Conference on Artificial Intelligence; the psychology department, the history department and the computer science department at Stanford University. All of these experiments were run while providing Dervish with no notion of the distance between adjacent nodes in its topological map. It is a demonstration of the power of probabilistic localization that, in spite of the tremendous lack of action and encoder information, the robot is able to navigate several real-world office buildings successfully.

One open question remains with respect to Dervish's localization system. Dervish was not just a localizer but also a navigator. As with all multiple hypothesis systems, one must ask the question, how does the robot decide how to move, given that it has multiple possible robot positions in its representation? The technique employed by Dervish is a most common technique in the AMRics field: plan the robot's actions by assuming that the robot's actual position is its most likely node in the belief state. Generally, the most likely position is a good measure of the robot's actual world position. However, this technique has shortcomings when the highest and second highest most likely positions have similar values. In the case of Dervish, it nonetheless goes with the highest likelihood position at all times, save at one critical juncture. The robot's goal is to enter a target room and remain there. Therefore, from the point of view of its goal, it is critical that it finish navigating only when the robot has strong confidence in being at the correct final location. In this particular case, Dervish's execution module refuses to enter a room if the gap between the most likely position and the second likeliest position is below a preset threshold. In such a case, Dervish will actively plan a path that causes it to move further down the hallway in an attempt to collect more sensor data and thereby increase the relative likelihood of one position in the belief state. Although computationally unattractive, one can go further, imagining a planning system for robots such as Dervish for which one specifies a goal belief state rather than a goal position. The robot can then reason and plan in order to achieve a goal confidence level, thus explicitly taking into account not only robot position but also the measured likelihood of each position. An example of just such a procedure is the Sensory Uncertainty Field of Latombe [90], in which the robot must find a trajectory that reaches its goal while maximizing its localization confidence enroute.

1.6.3 Kalman Filter Localisation

The Markov localization model can represent any probability density function over robot position. This approach is very general but, due to its generality, inefficient. A successful alternative is to use a more compact representation of a specific class of probability densities. The Kalman filter does just this, and is an optimal recursive data processing algorithm. It incorporates all information,

regardless of precision, to estimate the current value of the variable of interest. A comprehensive introduction can be found in [46] and a more detailed treatment is presented in [28]. Figure 5.26 depicts the a general scheme of Kalman filter estimation, where the system has a control signal and system error sources as inputs. A measuring device enables measuring some system states with errors. The Kalman filter is a mathematical mechanism for producing an optimal estimate of the system state based on the knowledge of the system and the measuring device, the description of the system noise and measurement errors and the uncertainty in the dynamics models. Thus the Kalman filter fuses sensor signals and system knowledge in an optimal way. Optimality depends on the criteria chosen to evaluate the performance and on the assumptions. Within the Kalman filter theory the system is assumed to be linear and white with Gaussian noise. As we have discussed earlier, the assumption of Gaussian error is invalid for our AMR applications but, nevertheless, the results are extremely useful. In other engineering disciplines, the Gaussian error assumption has in some cases been shown to be quite accurate [46]. We begin with a subsection that introduces Kalman filter theory, then we present an application of that theory to the problem of AMR localization. Finally, the third subsection will present a case study of a AMR that navigates indoor spaces by virtue of Kalman filter localization.

A Gentle Introduction to Kalman Filter Theory

The **Kalman filter** method allows multiple measurements to be incorporated optimally into a single estimate of state. In demonstrating this, first we make the simplifying assumption that the state does **NOT** change²¹ between the acquisition of the first and second measurement.

²¹i.e., we are assuming the robot is stationary.

After presenting this static case, we can introduce dynamic prediction readily.

Static Estimation Let us assume we have taken two (2) measurements:

- one with an ultrasonic range sensor at time k , and
- one with a more precise laser range sensor at time $k + 1$.

Based on each measurement we are able to estimate the robot's position.

Such an estimate derived from the first sensor measurements is q_1 and the estimate of position based on the second measurement is q_2 .

As we know each measurement can be inaccurate, we wish to **modulate** these position estimates based on the **expected measurement error** from each sensor. Suppose we use two (2) variances (σ_1^2, σ_2^2) to predict the error associated with each measurement. We will assume a unimodal error distribution throughout the remainder of the Kalman filter approach, which gives us the two (2)

robot position estimates:

$$\hat{q}_1 = q_1 \quad \text{with variance} \quad \sigma_1^2, \quad (1.10)$$

$$\hat{q}_2 = q_2 \quad \text{with variance} \quad \sigma_2^2. \quad (1.11)$$

$$(1.12)$$

So now we have two (2) measurements available to estimate the robots position. The question we now have to answer is

How do we fuse these data to get the best estimate \hat{q} for the robot position?

We are assuming that there was no robot motion between time k and time $k + 1$, and therefore we can directly apply the same weighted least square technique:

$$S = \sum_{i=1}^n w_i (\hat{q} - q_i)^2 \quad (1.13)$$

with w being the weight of measurement i . To find the minimum error we set the derivative i of S equal to zero. which gives us:

$$\hat{q} = \frac{\sum_{i=1}^n w_i q_i}{\sum_{i=1}^n w_i} \quad (1.14)$$

Dynamic Estimation Following our previous model, we will now consider a robot which moves between successive sensor measurements. Suppose that the motion of the robot between times k and $k + 1$ is described by the velocity u and the noise w which represents the uncertainty of the actual velocity:

$$\frac{dx}{dt} = u + w \quad (1.15)$$

If we now start at time k , knowing the variance σ_k^2 of the robot position at this time and knowing the variance σ_w^2 of the motion, we obtain for the time k' just when the measurement is taken:

$$\hat{x}_{k'} = \hat{x}_k + u(t_{k+1} - t_k) \quad (1.16)$$

$$\hat{x}_{k'} = \hat{x}_k + u(t_{k+1} - t_k) \quad (1.17)$$

$$(1.18)$$

where ..

Kalman Filter Localisation

The Kalman filter is an optimal and efficient [sensor fusion](#) technique.

Application of the Kalman filter to localisation requires posing the robot localisation problem as a sensor fusion problem.

Recall that the basic probabilistic update of robot belief state can be segmented into two (2) phases:

- perception update, and
- action update

The fundamental difference between the Kalman filter approach and Markov localisation approach lies in the perception update process.

²²i.e., the robot's set of instantaneous sensor measurements.

In Markov localisation, the entire perception²² is used to update each possible robot position in the belief state individually using Bayes formula.

²³as in Dervish.

In some cases, the perception is abstract, having been produced by a feature extraction mechanism.²³ In other cases, as with Rhino, the perception consists of raw sensor readings.

By contrast, perception update using a Kalman filter is a **multi-step** process. The robot's total sensory input is treated, not as a monolithic whole, but as a set of extracted features which each relate to objects in the environment. Given a set of possible features, the Kalman filter is used to fuse the distance estimate from each feature to a matching object in the map. Instead of carrying out this matching process for many possible robot locations individually as in the Markov approach, the Kalman filter accomplishes the same probabilistic update by treating the whole, unimodal and Gaussian belief state at once. **Fig. 1.11** depicts the particular schematic for Kalman filter localisation.

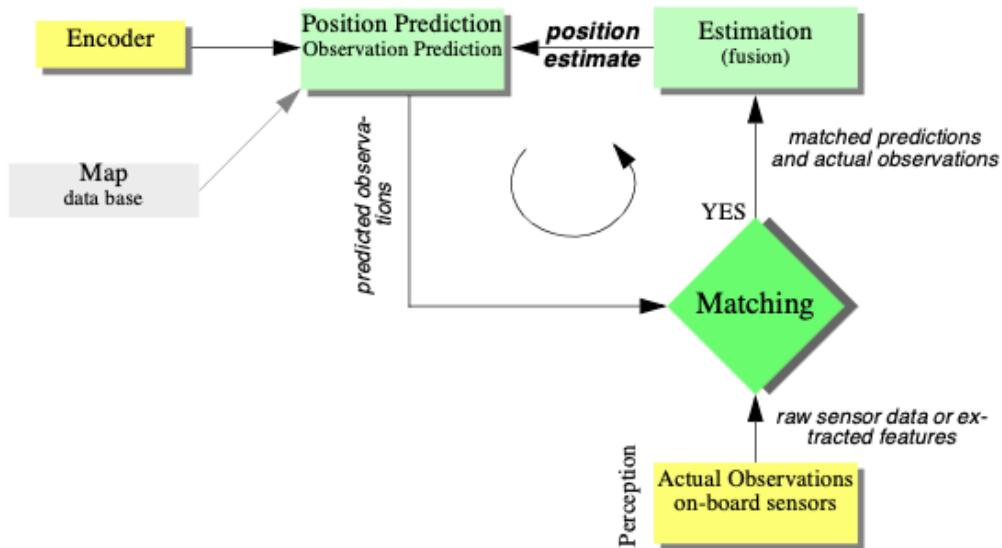


Figure 1.11: The schematic for the Kalman filter localisation

The first step is action update or position prediction, the straightforward application of a Gaussian error motion model to the robot's measured encoder travel. The robot then collects actual sensor data and extracts appropriate features²⁴ in the observation step. At the same time, based on its predicted position in the map, the robot generates a measurement prediction which identifies the features which the robot expects to find and the positions of those features. In matching the robot identifies the best pairings between the features actually extracted during observation and the

²⁴e.g. lines, doors, or even the value of a specific sensor

expected features due to measurement prediction. Finally, the Kalman filter can fuse the information provided by all of these matches in order to update the robot belief state in estimation.

1.7 Other Examples of Localisation Methods

Markov localisation and Kalman filter localisation have been two extremely popular strategies for research AMR systems navigating indoor environments. They have strong formal bases and therefore well-defined behavior. But there are a large number of other localisation techniques that have been used with varying degrees of success on commercial and research AMR platforms. We will not explore the space of all localisation systems in detail. Refer to surveys such as [4] for such information. There are, however, several categories of localisation techniques that deserve mention. Not surprisingly, many implementations of these techniques in commercial robotics employ modifications of the robot's environment, something that the Markov localisation and Kalman filter localisation communities eschew. In the following sections, we briefly identify the general strategy incorporated by each category and reference example systems, including as appropriate those that modify the environment and those that function without environmental modification.

1.7.1 Landmark-based Navigation

Landmarks are generally defined as **passive objects** in the environment which provide a high degree of localisation accuracy when they are within the robot's field of view. Mobile robots that make use of landmarks for localisation generally use artificial markers that have been placed by the robot's designers to make localisation easy.

The control system for a landmark-based navigator consists of two (2) discrete phases.

- When a landmark is in view, the robot localizes frequently and accurately, using action update and perception update to **track its position without cumulative error**.
- when the robot is in no landmark "zone", then only action update occurs, and the robot accumulates position uncertainty until the next landmark enters the robot's field of view.

The robot is thus effectively dead-reckoning from landmark zone to landmark zone. This in turn means the robot must consult its map carefully, ensuring that each motion between landmarks is sufficiently short, given its motion model, that it will be able to localize successfully upon reaching the next landmark.

Fig. 1.12 shows one instantiating of landmark-based localisation. The particular shape of the landmarks enables reliable and accurate pose estimation by the robot, which must travel using dead reckoning between the landmarks.

One key advantage of the landmark-based navigation approach is that a strong formal theory has been developed for this general system architecture [113]. In this work, the authors have shown precise assumptions and conditions which, when satisfied, guarantee that the robot will always be able to localize successfully. This work also led to a real-world demonstration of landmark-based localisation. Standard sheets of paper were placed on the ceiling of the Robotics Laboratory at



Figure 1.12: An illustration showing the object-level landmarks in blue-boxes. (a,b) shows two different indoor scenarios. The blue boxes represent the 3D object detection of object-level landmarks. The red dots indicate the nodes of the topological map. The yellow lines indicate the edges of the topological map. The green curve is the feasible navigation trajectory generated based on the proposed method [wang2022object].

Stanford University, each with a unique checkerboard pattern. A Nomadics 200 AMR was fitted with a monochrome CCD camera aimed vertically up at the ceiling. By recognizing the paper landmarks, which were placed approximately 2 meters apart, the robot was able to localize to within several centimeters, then move using dead-reckoning to another landmark zone.

The primary disadvantage of landmark-based navigation is that in general **it requires significant environmental modification**. Landmarks are local, and therefore a large number is usually required to cover a large factory area or research laboratory. For example, the Robotics Laboratory at Stanford made use of approximately 30 discrete landmarks, all affixed individually to the ceiling.

1.7.2 Globally Unique Localisation

The landmark-based navigation approach makes a strong general assumption:

when the landmark is in the robot's field of view, localisation is essentially perfect.

One way to reach the near perfect AMR localisation is to effectively enable such an assumption to be valid wherever the robot is located. It would be revolutionary the robot's sensors immediately identified its particular location, uniquely, and repeatedly.

Such a strategy for localisation is surely aggressive, but the question of whether it can be done is primarily a question of sensor technology software. Clearly, such a localisation system would need to use a sensor which collects a very large amount of information.

Since vision does indeed collect far more information than other sensors, it has been used as the sensor of choice in research towards globally unique localisation.

If humans were able to look at an individual picture and identify the robot's location in a well-known environment, then one could argue that the information for globally unique localisation does exist within the picture. It must simply be interpreted correctly.

One such approach has been attempted by several researchers and involves constructing one or more image histograms to represent the information content of an image stably (see for example Figure 4.51 and Section 4.3.2.2). A robot using such an image histogramming system has been shown to uniquely identify individual rooms in an office building as well as individual sidewalks in an outdoor environment. However, such a system is highly sensitive to external illumination and provides only a level of localisation resolution equal to the visual footprint of the camera optics.

The Angular histogram depicted in Figure 5.37 is another example in which the robot's sensor values are transformed into an identifier of location. However, due to the limited information content of sonar ranging strikes, it is likely that two places in the robot's environment may have angular histograms that are too similar to be differentiated successfully.

One way of attempting to gather sufficient sonar information for global localisation is to allow the robot time to gather a large amount of sonar data into a local evidence grid (i.e. occupancy grid) first, then match the local evidence grid with a global metric map of the environment. In [115] the researchers demonstrate such a system as able to localize on-thefly even as significant changes are made to the environment, degrading the fidelity of the map. Most interesting is that the local evidence grid represents information well enough that it can be used to correct and update the map over time, thereby leading to a localisation system that provides corrective feedback to the environment representation directly. This is similar in spirit to the idea of taking rejected observed features in the Kalman filter localisation algorithm and using them to create new features in the map.

A most promising, new method for globally unique localisation is called Mosaic-based localisation [114]. This fascinating approach takes advantage of an environmental feature that is rarely used by AMRs: fine-grained floor texture. This method succeeds primarily because of the recent ubiquity of very fast processors, very fast cameras and very large storage media.

The robot is fitted with a high-quality high-speed CCD camera pointed toward the floor, ideally situated between the robot's wheels and illuminated by a specialized light pattern off the camera axis to enhance floor texture. The robot begins by collecting images of the entire floor in the robot's workspace using this camera. Of course the memory requirements are significant, requiring a 10GB drive in order to store the complete image library of a 300 x 300 meter area. Once the complete image mosaic is stored, the robot can travel any trajectory on the floor while tracking its own position without difficulty. Localisation is performed by simply recording one image, performing action update, then performing perception update by matching the image to the mosaic database using simple techniques based on image database matching. The resulting performance has been impressive: such a robot has been shown to localize repeatedly with 1mm precision while moving at 25 km/hr. The key advantage of globally unique localisation is that, when these systems function correctly, they greatly simplify robot navigation. The robot can move to any point and will always be assured

of localizing by collecting a sensor scan. But the main disadvantage of globally unique localisation is that it is likely that this method will never offer a complete solution to the localisation problem. There will always be cases where local sensory information is truly ambiguous and, therefore, globally unique localisation using only current sensor information is unlikely to succeed. Humans often have excellent local positioning systems, particularly in non-repeating and well-known environments such as their homes. However, there are a number of environments in which such immediate localisation is challenging even for humans: consider hedge mazes and large new office buildings with repeating halls that are identical. Indeed, the mosaic-based localisation prototype described above encountered such a problem in its first implementation. The floor of the factory floor had been freshly painted and was thus devoid of sufficient micro-fractures to generate texture for correlation. Their solution was to modify the environment after all, painting random texture onto the factory floor.

1.7.3 Positioning Beacon systems

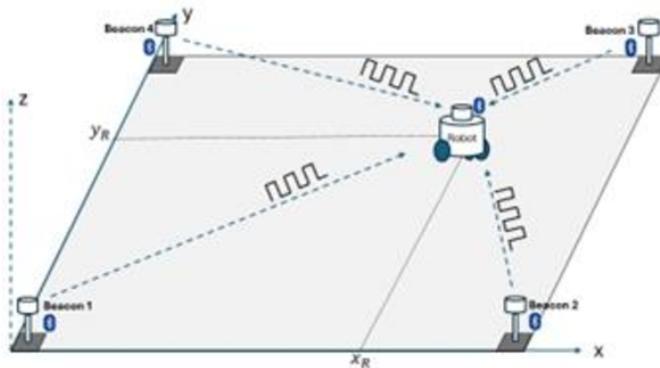
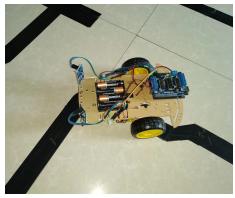


Figure 1.13

With most beacon systems, the design depicted depends foremost upon geometric principles to effect localisation. In this case the robots must know the positions of the two pinger units in the global coordinate frame in order to localize themselves to the global coordinate frame. A popular type of beacon system in industrial robotic applications is depicted in Figure 5.39. In this case beacons are retroreflective markers that can be easily detected by a AMR based on their reflection of energy back to the robot. Given known positions for the optical retroreflectors, a AMR can identify its position whenever it has three such beacons in sight simultaneously. Of course, a robot with encoders can localize over time as well, and does not need to measure its angle to all three beacons at the same instant. The advantage of such beacon-based systems is usually extremely high engineered reliability. By the same token, significant engineering usually surrounds the installation of such a system in a specific commercial setting. Therefore, moving the robot to a different factory floor will be both time-consuming and expensive. Usually, even changing the routes used by the robot will require serious re-engineering.

1.7.4 Route-Based Localisation

Even more reliable than beacon-based systems are route-based localisation strategies. In this case, the route of the robot is explicitly marked so that it can determine its position, not relative to some global coordinate frame, but relative to the specific path it is allowed to travel.²⁵ There are many techniques for marking such a route and the subsequent intersections.



²⁵A perfect example for these kind of localisation is the traditional line following robot. The robot does not need to know where it is as its only job is to make sure the line it is following is within its vision [AaaravG2018].

In all cases, one is effectively creating a railway system, except the railway system is somewhat more flexible and certainly more human-friendly actual rail.

For example, high UV-reflective, optically transparent paint can mark the route such that only the robot, using a specialized sensor, easily detects it. Alternatively, a guide wire buried underneath the hall can be detected using inductive coils located on the robot chassis.

In all such cases, the robot localisation problem is effectively trivialized by forcing the robot to always follow a prescribed path. While this may remove the **autonomous** part of AMR, there are industrial unmanned guided vehicles that do deviate briefly from their route in order to avoid obstacles. Nevertheless, the cost of this extreme reliability is obvious:

the robot is much more inflexible given such localisation means, and therefore any change to the robot's behavior requires significant engineering and time.

1.8 Building Maps

Humans are excellent navigators due to their remarkable ability to build cognitive maps [mcnamara1989subjective] which form the basis of spatial memory [chun1998contextual], [huang2023visual]. However, when it comes to AMR, we unfortunately need to be more hands on.

All of the localisation strategies we have discussed previously require active human effort to install the robot into a space. Artificial environmental modifications may be necessary to reduce ambiguity [meyer2011using]. Even if this is not so, a map of the environment must be created for the robot.

But a robot which localizes successfully has the right sensors for detecting the environment, and so the robot ought to build its own map.

This ambition goes to the heart of AMR. In prose, we can express our eventual goal as follows:

Starting from an arbitrary initial point, a AMR should be able to autonomously explore the environment with its on-board sensors, gain knowledge about it, interpret the scene, build an appropriate map and localize itself relative to this map.

While we have system which allows certain level of intelligence to robots, most applications require a connected network or a central node to achieve any autonomous action [hazik2022fleet], [xidias2022intelligent]. Accomplishing this goal purely using internal components in a robust is probably years away, but an important sub-goal is the invention of techniques for autonomous creation and modification of an environment map. Of course a AMR's sensors have only limited range, and so it must physically explore its environment to build such a map. So, the robot must not only create a map but it must do so while moving and localizing to explore the environment. This is often called the Simultaneous Localisation and Mapping (SLAM) problem,²⁶ arguably the most difficult problem specific to AMR systems.



Figure 1.14: 2005 DARPA Grand Challenge winner Stanley performed SLAM as part of its autonomous driving system [Kivaan2007].

The reason why SLAM is difficult is born precisely from the interaction between the robot's position updates as it localises and its mapping actions. If a AMR updates its position based on an observation

²⁶Computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. While this initially appears to be a chicken or the egg problem, there are several algorithms known to solve it in, at least approximately and in reasonable time for certain environments. Popular solutions include the particle filter, extended Kalman filter, covariance intersection, and GraphSLAM. SLAM algorithms are based on concepts in computational geometry and computer vision, and are used in robot navigation, robotic mapping and odometry for virtual reality or augmented reality.

of an imprecisely known feature, the resulting position estimate becomes correlated with the feature location estimate. Similarly, the map becomes correlated with the position estimate if an observation taken from an imprecisely known position is used to update or add a feature to the map.

For localisation the robot needs to know where the features are whereas for map building the robot needs to know where it is on the map.

The only path to a complete and optimal solution to this joint problem is to consider all the correlations between position estimation and feature location estimation. Such cross-correlated maps are called stochastic maps [hashimoto2002humanoid]. Unfortunately, implementing such an optimal solution is computationally prohibitive.

1.8.1 Stochastic Map Technique

Fig. 1.15 shows a general schematic incorporating map building and maintenance into the standard localisation loop depicted by Figure (5.29) during discussion of Kalman filter localisation [9]. The added arcs represent the additional flow of information that occurs when there is an imperfect match between observations and measurement predictions.

Unexpected observations will affect the creation of new features in the map whereas unobserved measurement predictions will affect the removal of features from the map. As discussed earlier, each specific prediction or observation has an unknown exact value and so it is represented by a distribution. The uncertainties of all of these quantities must be considered throughout this process.

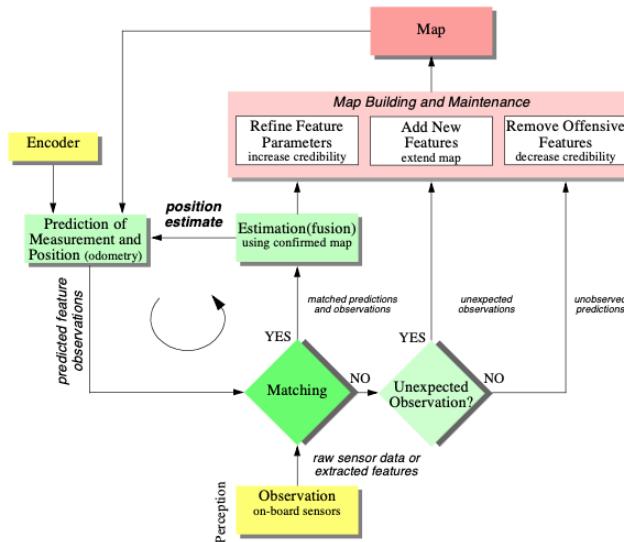


Figure 1.15: General schematic for concurrent localization and map building.

The new type of map we are creating not only has features in it as did previous maps, but it also has varying degrees of probability that each feature is indeed part of the environment.

We represent this new map M with a set n of probabilistic feature locations \hat{z}_t , each with the covariance matrix Σ_t and an associated **credibility factor** c_t between 0 and 1.

The purpose of c_t is to quantify the belief in the existence of the feature in the environment (see Fig. (5.41)):

$$M = \left\{ z_t, \Sigma_t, c_t \mid (1 \leq t \leq n) \right\} \quad (1.19)$$

In contrast to the map used for Kalman filter localisation previously, the map M is **NOT** assumed to be **precisely known** as it will be created by an uncertain robot over time. This is why the features \hat{z} are described with associated covariance matrices Σ_t .

Similar to Kalman filter localisation, the matching steps has three (3) outcomes in regard to measurement predictions and observations:

- matched prediction and observations,
- unexpected observations, and
- unobserved predictions

Localisation, or the position update of the robot, proceeds as before. However, the map is also updated now, using all three outcomes and complete propagation of all the correlated uncertainties.

The interesting concept in this modelling is the **credibility factor** c_t , which governs the likelihood that the mapped feature is indeed in the environment.

How should the robot's failure to match observed features to a particular map feature reduce that map feature's credibility?

How should the robot's success at matching a mapped feature increase the chance that the mapped feature is "correct?"

As an example, in [carroll2017transformation] the following function is proposed for calculating credibility:

$$c_t(k) = 1 - \exp \left(- \left(\frac{n_s}{a} - \frac{n_u}{b} \right) \right) \quad (1.20)$$

where a and b define the **learning** and **forgetting** rate and n_s and n_u are the number of matched and unobserved predictions up to time k , respectively. The update of the covariance matrix Σ_t building the feature positions and the robot's position are strongly correlated.

This forces us to use a stochastic map, in which all cross-correlations must be updated in each cycle.

The stochastic map consists of a stacked system state vector:

$$\mathbf{X} = [x_r(k) \quad x_1(k) \quad x_2(k) \quad \dots \quad x_n(k)]^T \quad (1.21)$$

and a system state covariance matrix:

$$\Sigma = \begin{bmatrix} C_{rr} & C_{r1} & \dots & C_{rn} \\ C_{1r} & C_{11} & \dots & C_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{nr} & C_{n1} & \dots & C_{nn} \end{bmatrix} \quad (1.22)$$

where the index r stands for the robot and the index i = 1 to n for the features in the map.

In contrast to localization based on an a priori accurate map, in the case of a stochastic map the cross-correlations must be maintained and updated as the robot is performing automatic map-building. During each localization cycle, the cross-correlations robot-to-feature and feature-to-robot are also updated. In short, this optimal approach requires every value in the map to depend on every other value, and therein lies the reason that such a complete solution to the automatic mapping problem is beyond the reach of even today's computational resources.

1.8.2 Other Mapping Techniques

The AMR research community has spent significant research effort on the problem of automatic mapping, and has demonstrating working systems in many environments without having solved the complete stochastic map problem described earlier.

This field of AMR research is extremely large, and this Lecture Book will **NOT** present a comprehensive survey of the field

Instead, let's look at the two (2) key considerations associated with automatic mapping, together with brief discussions of the approaches taken by several automatic mapping solutions to overcome these challenges.

Cyclic Environments

Possibly the single hardest challenge for automatic mapping to be conquered is to correctly map cyclic environments. The problem is simple:

²⁷Such as four (4) hallways that intersect to form a rectangle

Given an environment which has one or more loops or cycles,²⁷ create a globally consistent map for the whole environment.

This problem is hard because of the fundamental behavior of automatic mapping systems:

The maps they create are not perfect.

And, given any local imperfection, accumulating such imperfections over time can lead to arbitrarily large global errors between a map, at the macro level, and the real world, as shown in **Fig. 1.16**.

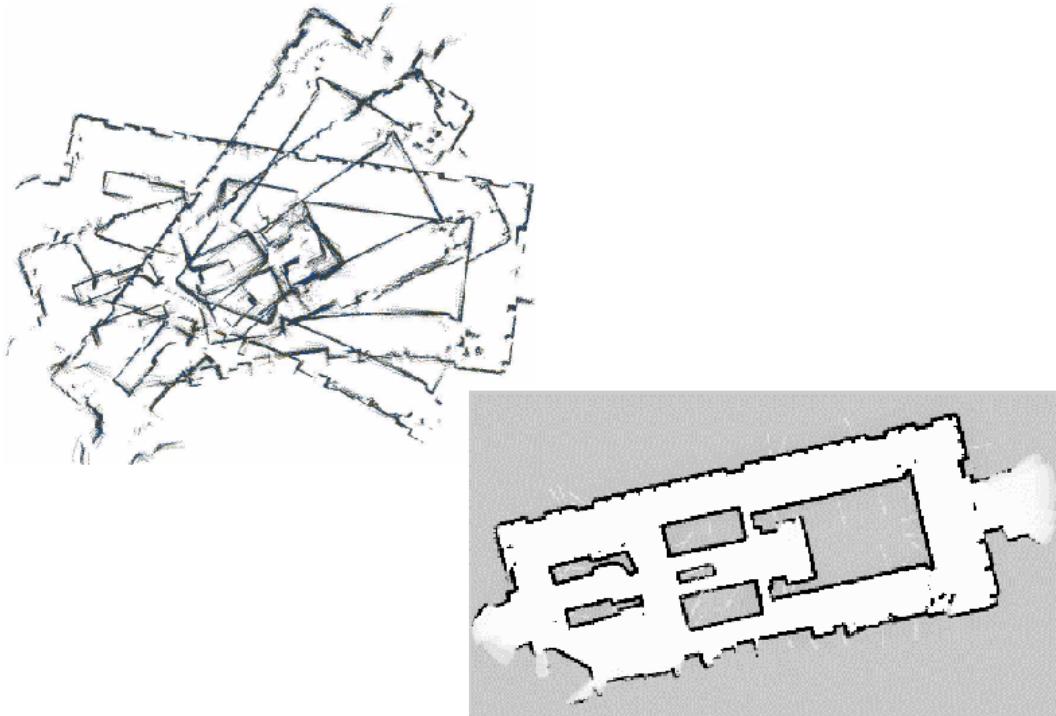


Figure 1.16: A naive, local mapping strategy with small local error leads to global maps that have a significant error, as demonstrated by this real-world run on the left. By applying topological correction, the grid map on the right is extracted [bruce2000fast].

Such global error is usually irrelevant for AMR localisation and navigation. After all, a warped map will still serve the robot perfectly well **so long as the local error is bounded**. However, an extremely large loop still eventually returns to the same spot, and the robot must be able to note this fact in its map. Therefore, global error does indeed matter in the case of cycles. In some of the earliest work attempting to solve the cyclic environment problem, [116] used a purely topological representation of the environment, reasoning that the topological representation only captures the most abstract, most important features and avoids a great deal of irrelevant detail. When the robot arrives at a topological node that could be the same as a previously visited and mapped node (e.g. similar distinguishing features), then the robot postulates that it has indeed returned to the same node. To check this hypothesis, the robot explicitly plans and moves to adjacent nodes to see if its perceptual readings are consistent with the cycle hypothesis.

With the recent popularity of metric maps such as fixed decomposition grid representations, the cycle detection strategy is not as straightforward. Two important features are found in most autonomous mapping systems that claim to solve the cycle detection problem:

- First, as with many recent systems, these mobile robots tend to accumulate recent perceptual history to [create small-scale local sub-maps \[nourbakhsh1997mobile\]](#). In this approach, each sub-map is treated as a singular sensor during the robot's position update. The advantage of this approach is two-fold.
 - As odometry is relatively accurate over small distances, the relative registration of features and raw sensor strikes in a local sub-map will be quite accurate.
 - The robot will have created a virtual sensor system with a significantly larger horizon than its actual sensor system's range. In a sense, this strategy at the very least defers the problem of very large cyclic environments by increasing the map scale that can be handled well by the robot.
- The second recent technique for dealing with cycle environments is in fact a [return to the topological representation](#). Some recent automatic mapping systems will attempt to identify cycles by associating a topology with the set of metric sub-maps, explicitly identifying the loops first at the topological level.

One could certainly imagine other augmentations based on known topological methods.

For example, the globally unique localisation methods described previously could be used to identify topological correctness.

Dynamic Environments

A second challenge extends **NOT** just to existing autonomous mapping solutions but even to the basic execution of the stochastic map approach.

All previously mentioned strategies tend to assume the environment is either [unchanging](#) or changes in ways that are [virtually insignificant](#). Such assumptions are certainly valid with respect to some environments, such as for example the computer science department of a university at 3:00 AM.

However, for many practical applications, this assumption is lacking at best. In the case of wide-open spaces that are popular gathering places for humans, there is rapid change in the freespace and a vast majority of sensor strikes represent detection of the transient humans rather than fixed surfaces such as the perimeter wall.

Another class of dynamic environments are spaces such as factory floors and warehouses, where the objects being stored redefine the topology of the pathways on a day-to-day basis as shipments are moved in and out.

²⁸In this context, salient means anything which is sticking out.

In all such dynamic environments, an automatic mapping system should capture the salient²⁸ objects detected by its sensors and, furthermore, the robot should have the flexibility to modify its map as

the position of these salient objects changes.

The subject of **continuous mapping**, or mapping of dynamic environments is to some degree a direct outgrowth of successful strategies for automatic mapping of unfamiliar environments.

For example, in the case of stochastic mapping using the credibility factor c_t mechanism, the credibility equation can continue to provide feedback regarding the probability of existence of various mapped features after the initial map creation process is ostensibly complete. Therefore, a mapping system can become a map-modifying system by simply continuing to operate.

This is most effective, of course, if the mapping system is real-time and incremental.

If map construction requires off-line global optimisation, then the desire to make small-grained, incremental adjustments to the map is more difficult to satisfy.

Earlier we stated that a mapping system should capture only the salient objects detected by its sensors. One common argument for handling the detection of, for instance, humans in the environment is that mechanisms such as c_t serve to be mapped in the first place.

The general solution to the problem of detecting salient features, however, requires a solution to the perception problem in general. When a robot's sensor system can reliably detect the difference between a wall and a human, using for example a vision system, then the problem of mapping in dynamic environments will become significantly more straightforward.

We have discussed just two important considerations for automatic mapping. There is still a great deal of research activity focusing on the general map building and localisation problem. This field is certain to produce significant new results in the next several years, and as the perceptual power of robots improves we expect the payoff to be greatest here.

Information _____ **DARPA Grand Challenge**

A prize competition for American autonomous vehicles, funded by the Defense Advanced Research Projects Agency (DARPA). The goal of the challenge is to further DARPA's mission to sponsor revolutionary, high-payoff research that bridges the gap between fundamental discoveries and military use. The initial DARPA Grand Challenge in 2004 was created to spur the development of technologies needed to create the first fully autonomous ground vehicles capable of completing a substantial off-road course within a limited time. The third event, the DARPA Urban Challenge in 2007, extended the initial Challenge to autonomous operation in a mock urban environment. The 2012 DARPA Robotics Challenge, focused on autonomous emergency-maintenance robots, and new Challenges are still being conceived. The DARPA Subterranean Challenge was tasked with building robotic teams to autonomously map, navigate, and search subterranean environments. Such teams could be useful in exploring hazardous areas and in search and rescue.



Figure 1.17: Stanford Racing and Victor Tango together at an intersection in the DARPA Urban Challenge Finals.

Chapter 2

Welcome to Linux

Table of Contents

2.1	Learning the Linux Command Line	53
2.2	Installation	59
2.3	Docker	60

2.1 Learning the Linux Command Line

Working with a text-based [Command Line Environment](#) (CLI), without a Graphical User Interface (GUI) can be intimidating at first glance, as most of us are accustomed to using a GUI. But understanding the command line environment will show how powerful and efficient it is.

```
1 echo "Hello, Linux!"                                C.R. 1  
1 Hello, Linux!                                         bash  
1  
1                                         text
```

Most senior programmers in the industry and veteran Linux [system administrators](#) will exclusively use Command-Line Interface (CLI) as their day to day interaction with the computer. The reason is, the GUI was designed for simplifying human interaction with computers rather than improving the computer's efficiency at doing tasks.

The goal of this chapter aims to introduce the fundamentals of working with the [Linux command line](#) using a very common shell called Bash as it will be important in the future when working with [ROS](#) (Robot Operating System) or in any future endeavour the reader may pursue in the fields related to computer science.

- Work on what the command line is and how it works,
- Look at working with files and folders,
- How Linux protects files from unauthorised access with permissions,
- Common commands to be familiar with and how to connect commands together with pipes,
- Introduction to some complex command line tasks.

This part of the lecture-book aims to give practical knowledge on working with the widely used [Bash](#) shell, in case you choose to extend your learning into user management, network configuration, programming and development, system administration, or if you catch the tinkerer-bug.

2.1.1 A Short History on Computer Interfaces

The CLI came from a form of dialogue by humans over [teleprinter](#) (TTY) machines, in which human operators remotely exchanged information instead of a human communicating with another human over a teleprinter. Early computer systems often used teleprinter machines as the means of interaction with a human operator.

The computer became one end of the human-to-human teleprinter model.

The mechanical teleprinter was then replaced by a [terminal](#), a keyboard and screen emulating the teleprinter. [Smart terminals](#) permitted additional functions, such as cursor movement over the entire screen, or local editing of data on the terminal for transmission to the computer.



Figure 2.2: Nokia Bell Labs Murray Hill, NJ ([Original](#))

As the microcomputer revolution replaced the traditional systems, hardware terminals were replaced by terminal emulators - Personal Computer (PC) software that interpreted terminal signals sent through the PC's serial ports. These were typically used to interface an organisation's new PC's with their existing mini- or mainframe computers, or to connect PC to PC. Some of these PCs were running Bulletin Board System software.

Early operating system CLIs were implemented as part of resident monitor programs, and could not easily be replaced. The first implementation of the [shell](#) as a replaceable component was part of the Multics time-sharing operating system. In 1964, MIT Computation Center staff member Louis Pouzin developed the RUNCOM tool for executing command scripts while allowing argument

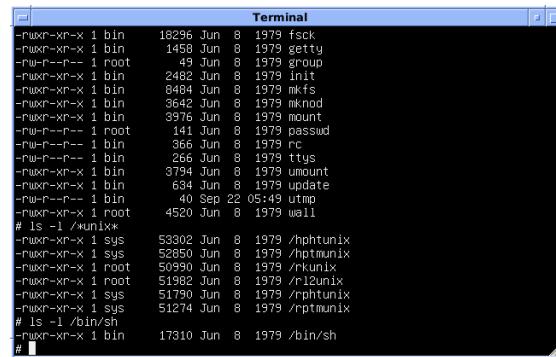


Figure 2.1: Hughes telegraph, an early (1855) teleprinter built by Siemens and Halske. The centrifugal governor to achieve synchronicity with the other end can be seen [Manske2009].

substitution.

Pouzin coined the term “shell” to describe the technique of using commands like a programming language, and wrote a paper about how to implement the idea in the Multics operating system. Pouzin returned to his native France in 1965, and the first Multics shell was developed by Glenda Schroeder. At Nokia Bell Labs headquarters the first Unix shell, the V6 shell, was developed by Ken Thompson in 1971 and was modelled after Schroeder’s Multics shell. The Bourne shell was introduced in 1977 as a replacement for the V6 shell. Although it is used as an interactive command interpreter, it was also intended as a scripting language and contains most of the features that are commonly considered to produce structured programs.

The Bourne shell led to the development of the KornShell ([ksh](#)), Almquist shell ([ash](#)), and the popular Bourne-again shell (or [bash](#)). Early microcomputers themselves were based on a CLI such as CP/M, DOS or AppleSoft BASIC. During the 1980s and 1990s, the introduction of the Apple Macintosh and of Microsoft Windows on PCs saw the command line interface as the primary user interface replaced by the Graphical User Interface. The command line remained available as an alternative user interface, often used by system administrators and other advanced users for system administration, computer programming and batch processing.



```

-rwxr-xr-x 1 bin    18296 Jun  8 1979 fsck
-rwxr-xr-x 1 bin    1458 Jun  8 1979 getty
-rw-r--r-- 1 root   49 Jun  8 1979 group
-rwxr-xr-x 1 bin    2482 Jun  8 1979 init
-rwxr-xr-x 1 bin    8484 Jun  8 1979 mkfs
-rwxr-xr-x 1 bin    3642 Jun  8 1979 mknod
-rwxr-xr-x 1 bin    3976 Jun  8 1979 mount
-rw-r--r-- 1 root   141 Jun  8 1979 passwd
-rw-r--r-- 1 bin    366 Jun  8 1979 rc
-rw-r--r-- 1 bin    266 Jun  8 1979 ttys
-rwxr-xr-x 1 bin    3794 Jun  8 1979 umount
-rwxr-xr-x 1 bin    634 Jun  8 1979 update
-rw-r--r-- 1 bin    40 Sep 22 05:49 utmp
-rwxr-xr-x 1 root   4520 Jun  8 1979 wall
# ls -l */unix/*
-rwxr-xr-x 1 sys    53302 Jun  8 1979 /phtunix
-rwxr-xr-x 1 sys    52850 Jun  8 1979 /phtmunix
-rwxr-xr-x 1 root   50990 Jun  8 1979 /rkunix
-rwxr-xr-x 1 root   51982 Jun  8 1979 /rl2unix
-rwxr-xr-x 1 sys    51790 Jun  8 1979 /rphunix
-rwxr-xr-x 1 sys    51274 Jun  8 1979 /rptmunix
# ls -l /bin/sh
-rwxr-xr-x 1 bin    17310 Jun  8 1979 /bin/sh

```

Figure 2.3: Bourne shell interaction on Version 7 Unix ([Original](#)).

Shells in other Operating Systems

Windows In November 2006, Microsoft released version 1.0 of Windows PowerShell, which combined features of traditional Unix shells with their proprietary object-oriented .NET Framework. MinGW and Cygwin are open-source packages for Windows that offer a Unix-like CLI. Microsoft provides MKS Inc.’s ksh implementation MKS Korn shell for Windows through their Services for UNIX add-on.

Macintosh Since 2001, the Macintosh operating system macOS has been based on a Unix-like operating system called Darwin. On these computers, users can access a Unix-like CLI by running the terminal emulator program called Terminal, or by remotely logging into the machine using [ssh](#). Z shell is the default shell for macOS¹ with [bash](#), [tcsh](#), and the KornShell also provided.

¹This was implemented as of macOS Catalina.

Before macOS Catalina, bash was the default shell.

2.1.2 Linux is a Nutshell

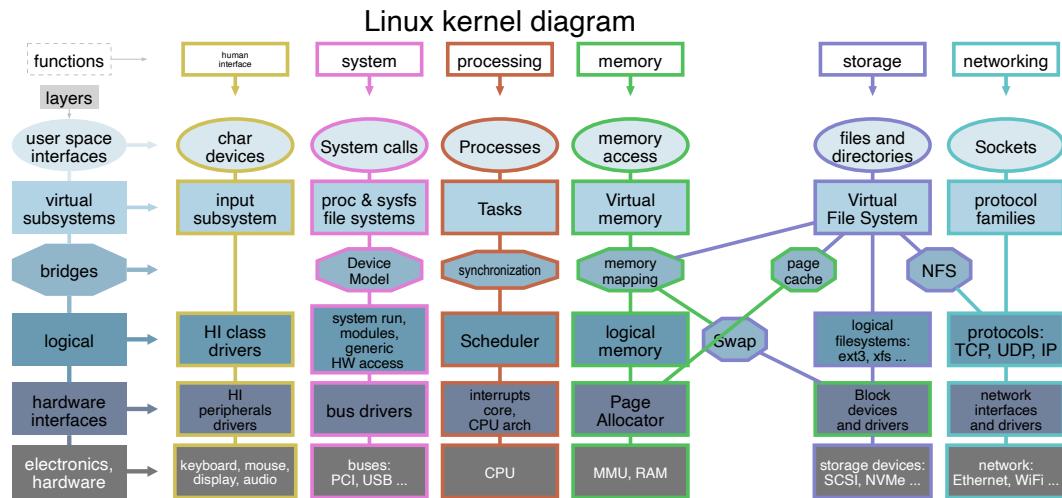


Figure 2.4: The kernel mapping of the Linux operating system.

A Brief Description of What Linux Does

Linux is a general purpose computer operating system, originally released in 1991 by Linus Torvalds and began as a personal project of him [[koren2006study](#)]. It was to create a new **free** operating system kernel which the resulting kernel has been marked by constant growth throughout its history.²

Linux is defined by its kernel, called the **Linux kernel**, which is the core component of the system. This kernel interacts with the computer hardware to allow software and other hardware to exchange information, which you can see in **Fig. 2.4**.



²The MINIX logo. There were alternative OSs on the market such as MINIX but it was under a proprietary license which was later became open-source in 2000. This was one of the reasons why Linux was attempted in the first place. To create a truly open-source implementation of UNIX.

Imagine the kernel as the middle-man between your software and the hardware. This allows you to write a program without worrying too much about what the hardware is.

As Linux is an open-source project and is probably one of the greatest collaborative software work in history, it has a rich history. It was inspired by MINIX which, in turn, was inspired by UNIX with UNIX being the first **portable** operating system ever designed [[bach1986design](#)] as it was mostly written with the C programming language [[johnson1978unix](#)].

Open Source v. Closed Source

In programming there are two (2) main approaches when it comes to sharing code:

- It can be closed source, which means, you are not allowed to edit the code the program is running on,
- Open source which you are free to edit and share the code as you see fit.

Linux is based on a philosophy of software and operating systems being **free**.

Software should be free of cost and freely modifiable.

The software license which allows this, in the case of the Linux kernel, is called the GNU General Public License.³ This emphasis of freedom, both, of cost and modification has helped Linux to become popular for many different applications and purposes from tinkering programming to being used in massive databases of major companies.

Linux has popped up everywhere from the majority of the servers that run web services we all use, to super computers, to Wi-Fi routers, in cars, mobile phones, and everywhere in between. Odds are that you are closed to a device that uses some part of the Linux kernel. In the midst of all these different kinds of Linux installations, the most important distinction you'll need to be aware of is one of the genealogy of Linux.

³are a series of widely used free software licenses, or copyleft licenses, that guarantee end users the freedoms to run, study, share, or modify the software.

2.1.3 Linux Distributions

While the Linux kernel is more or less the same across nearly all installations of Linux, the software that surrounds the kernel that provides capabilities like *software package management*, *control of services*, and the *location of configuration files* differs between them. Many of the tools that come packaged with Linux come from the GNU Project and aren't actually a part of Linux and, taken together, the combination of the kernel and these common tools is often referred to as **GNU Linux**. Different groups of software and configuration choices that are maintained by individuals or groups of people are called distributions, or distro's. Most major distributions of Linux fall into categories based on the original distribution from which they were derived. These are:⁴

Depending the readers future work or study area, it is likely to end up learning to use the command line on a system that inherits from one of these distributions. Most likely, it will be a distribution derived from Debian or Red Hat. Linux Mint, Ubuntu, Elementary OS, and Kali Linux are all derived from Debian. CentOS, Fedora, and Red Hat Enterprise Linux are derived from Red Hat.

⁴The entire family history of linux can be viewed in the [Distribution Timeline](#)

The history of all of these different distributions of Linux is beyond the scope of this document. But, what this means at its core is the need to be aware of what system is in use and the need to adapt to account for differences in distributions. As we begin working with Linux, through the command line, it will be apparent, most of what can be done is the same across the major distributions.

Distribution	Advantages	Disadvantages
Linux Mint	Superb collection of custom tools developed in-house, hundreds of user-friendly enhancements, inclusion of multimedia codecs, open to users' suggestions	The project does not issue security advisories
Ubuntu	Fixed release cycle and support period; long-term support (LTS) variants with five years of security updates; novice-friendly; wealth of documentation, both official and user-contributed	Lacks compatibility with Debian; frequent major changes tend to drive some users away; non-LTS releases come with only nine months of security support
Arch Linux	Excellent software management infrastructure; unparalleled customisation and tweaking options; superb on-line documentation	Occasional instability and risk of breakdown
Gentoo	Highly flexible, endlessly customizable, able to use a range of compile-time configurations, init systems and run on many architectures	Requires a higher degree of knowledge to use, upgrading packages via source can be time consuming
Slackware Linux	Considered highly stable, clean and largely bug-free, strong adherence to UNIX principles	Limited number of officially supported applications; conservative in terms of base package selection; complex upgrade procedure
Debian	Very stable; remarkable quality control; includes over 30,000 software packages; supports more processor architectures than any other Linux distribution	Conservative - due to its support for many processor architectures, newer technologies are not always included; slow release cycle (one stable release every 2 - 3 years); discussions on developer mailing lists and blogs can be uncultured at times
Fedor	Highly innovative; outstanding security features; large number of supported packages; strict adherence to the free software philosophy; availability of live spins featuring many popular desktop environments	Fedor's priorities tend to lean towards enterprise features, rather than desktop usability; some bleeding edge features, such as switching early to KDE 4 and GNOME 3, occasionally alienate some desktop users
openSUSE	Comprehensive and intuitive configuration tool; large repository of software packages, excellent web site infrastructure and printed documentation, Btrfs with boot environments by default	Its resource-heavy desktop setup and graphical utilities are sometimes seen as "bloated and slow"
Red Hat	Long-term, commercial support of ten years or more. Stability.	Lacks latest Linux technologies; small software repositories; licensing restrictions
FreeBSD	Fast and stable; availability of over 24,000 software applications (or "ports") for installation; very good documentation; native ZFS support and boot environments	Tends to lag behind Linux in terms of support for new and exotic hardware, limited availability of commercial applications; lacks graphical configuration tools

Table 2.1: Most popular distributions used according to [distrowatch](#).

2.2 Installation

There are a wide variety of ways of installing Linux on a computer. These include:

Creating a virtual environment

Allows you to install Linux within your primary Operating System (OS). There are great many benefits to this approach as it allows you to test an experimental OS without affecting your primary setup. It also allows you to run simple applications without leaving your primary OS and can have various inter-operability options such as shared folder, and shared network settings.

The disadvantages include some hit to performance as both the host and virtual OS have to share the same pool of resources. In addition, graphically virtual ram limits are set to 256 MB.

Creating a partition on your computer

and install it alongside your primary OS. This option is generally done by most people who know what they are doing as they generally have one or two software where there is no alternative on Linux and therefore would like to keep their primary OS for those specific software.

Using a container

One of the more popular option in recent times. The way it works is similar to that of using a virtual machine but heavily stripped one. Basically a container houses enough components to house an application. Think of running Linux per application instead of a full blown os.

There are many merits and demerits to using an option and in this lecture we will focus on building a container image for both Linux programming and ROS application



Figure 2.5: The docker logo

2.3 Docker

Docker is an open-source platform that has completely changed the way we develop, deploy, and use apps. The application development lifecycle is a dynamic process, and developers are always looking for ways to make it more efficient. Docker enables developers to package their work and all of its dependencies into standardised units called containers by utilizing containerisation technology.

By separating apps from the underlying infrastructure, these lightweight containers provide reliable performance and functionality in a variety of environments. Because of this, Docker is a game-changer for developers because it frees them up to concentrate on creating amazing software rather than handling difficult infrastructure.

Regardless of your level of experience, Docker provides an extensive feature set and a strong toolset that can greatly enhance your development process. In this tutorial, we will provide you with a thorough understanding of Docker, going over its main features, advantages, and ways to use it to develop, launch, and distribute apps more quickly and easily.



⁵The logo of the podman software.

Docker is not the only containerisation software available as there is also podman⁵ which is almost compatible and is developed by RHEL.

Information

Docker v. Podman

The biggest difference is the underlying architecture each is built on. Docker heavily relies on a daemon, while Podman is daemonless. Think of a daemon as a process that runs in the background on the host OS. In Docker's case, its daemon is responsible for managing Docker objects⁶ and communicating with other systems. To run its daemon, Docker uses a package called `dockerd`. Daemons typically require root-level access to the machine they run on. This lends itself to **security vulnerabilities**. If a bad actor can get access to a daemon, they now have access to the entire machine.

Podman's daemonless architecture comes with a few benefits. Since running daemons almost always requires root privileges, a daemonless architecture can be thought of as "rootless." This means that users who don't have system-level access to the machine their containers are running on can still use Podman which isn't always the case with Docker. Instead of a daemon, Podman uses a Linux package known as `systemd`. Since `systemd` is native to the Linux operating system, Podman is often considered more "light-weight" than

Docker and will usually see faster container spin-up times than when using Docker.

2.3.1 Dockerfile

A Dockerfile is a text document in which you can lay down all the instructions you want for an image to be created.

- The first entry in the file specifies the base image, which is a pre-made image containing all the dependencies you need for your application.
- Then, there are commands you can send to the Dockerfile to install additional software, copy files, or run scripts.

The result is a Docker image:

a self-sufficient, executable file with all the information needed to run an application.

Dockerfiles are an easy way to create and deploy applications. They help in creating an environment **consistently reproducibly**, and in an easier way.

A Dockerfile is used to create new custom images prepared individually according to specific needs. For instance, a Docker image can have a particular version of a web server or, for example, a database server, or in this case run an entire Linux OS with ROS installed.

For the preparation of the lecture the following Dockerfile is written which you can see in snippets below, which is a great way to start explaining how the document works:

```
1 # Declare the ubuntu version
2 FROM ubuntu:jammy-20250404
```

C.R. 2

dockerfile

Here we are declaring a base image. A base image is a bare-bones OS and/or application in which we build our software on. In this case it is an Ubuntu jammy jellyfish (22.04).

```
1 # Define the target-platform and the current maintainer
2 ARG TARGETPLATFORM
3 LABEL maintainer="dtm@mci4me.at"
```

C.R. 3

dockerfile

We define an **ARG** which is a variable. We set a **TARGETPLATFORM** which we can use to change the architecture of the install. **LABEL** is used to give a meta-data information which in this case is a simple email information of the current maintainer.

```
1 # Execute the following command as string
2 SHELL ["/bin/bash", "-c"]
```

C.R. 4

dockerfile

Next we configure the shell of the container to use `bash` and process the given input as a string and `NOT` as a command.

```

1 # Upgrade Ubuntu Jammy and remove downloaded list of packages
2 RUN apt-get update -q && \
3     DEBIAN_FRONTEND=noninteractive apt-get upgrade -y && \
4     apt-get autoclean && \
5     apt-get autoremove && \
6     rm -rf /var/lib/apt/lists/*

```

C.R. 5
dockerfile

We now start with building the OS. We start with declaring an update of the OS and ask it to do it non-interactively. Once it updates the system, we ask it to remove the repo-list to save up on space.

```

1 # Install Ubuntu Mate desktop and remove downloaded list of packages
2 RUN apt-get update -q && \
3     DEBIAN_FRONTEND=noninteractive apt-get install -y \
4         ubuntu-mate-desktop && \
5     apt-get autoclean && \
6     apt-get autoremove && \
7     rm -rf /var/lib/apt/lists/*

```

C.R. 6
dockerfile

To make our programming easier and creating a more friendly environment, we shall install a GUI. There are a wide variety of desktop environments for use in Linux, such as GNOME, KDE, Xfce but for our application we shall use mate which is the default environment for Linux Mint.

```

1 # Add important packages
2 RUN apt-get update && \
3     DEBIAN_FRONTEND=noninteractive apt-get install -y \
4         tigervnc-standalone-server tigervnc-common \
5         supervisor wget curl gosu git sudo python3-pip tini nano \
6         build-essential vim sudo lsb-release locales info \
7         bash-completion tzdata emacs \
8         dos2unix && \
9     apt-get autoclean && \
10    apt-get autoremove && \
11    rm -rf /var/lib/apt/lists/*

```

C.R. 7
dockerfile

We start now with installing software on top of the OS. Remember, we are installing a bare bones system which means it is stripped from all the software suites we take for granted.

```

1 # Install noVNC and Websockify
2 RUN git clone \
3     https://github.com/AtsushiSaito/noVNC.git \
4     -b add_clipboard_support /usr/lib/novnc
5
6 RUN pip install git+https://github.com/novnc/websockify.git@v0.10.0
7 RUN ln -s /usr/lib/novnc/vnc.html /usr/lib/novnc/index.html
8
9 # Set remote resize function enabled by default

```

C.R. 8
dockerfile

```

10 RUN sed -i \
11   "s/UI.initSetting('resize', 'off');/UI.initSetting('resize', 'remote');/g" \
12   /usr/lib/novnc/app/ui.js
13
14 # Disable auto update and crash report
15 RUN sed -i 's/Prompt=.*?Prompt=never/' /etc/update-manager/release-upgrades
16 RUN sed -i 's/enabled=1/enabled=0/g' /etc/default/apport

```

C.R. 9
dockerfile

This section is all about installing a Virtual Network Computing (VNC) which is a graphical desktop-sharing system which allows remote controlling of another computer. It transmits the keyboard and mouse input from one computer to another, relaying the graphical-screen updates, over a network. We also do some text manipulation to fix some glitches.

```

1 # Install Firefox and its configuration
2 RUN DEBIAN_FRONTEND=noninteractive add-apt-repository ppa:mozillateam/ppa -y && \
3   echo 'Package: *' > /etc/apt/preferences.d/mozilla-firefox && \
4   echo 'Pin: release o=LP-PPA-mozillateam' \
5   > /etc/apt/preferences.d/mozilla-firefox && \
6   echo 'Pin-Priority: 1001' >> /etc/apt/preferences.d/mozilla-firefox && \
7   apt-get update -q && \
8   apt-get install -y \
9   firefox && \
10  apt-get autoclean && \
11  apt-get autoremove && \
12  rm -rf /var/lib/apt/lists/*
13

```

C.R. 10
dockerfile

To aid in easy navigation and searching information, we shall also install Firefox into this docker container as well.

```

1 # Install VSCode for people who are accosstomed to VSCode but
2 # prefer to keep it open-source
3 RUN wget https://gitlab.com/paulcarrotv/vscodium-deb-rpm-repo/raw/master/pub.gpg \
4   -O /usr/share/keyrings/vscodium-archive-keyring.asc && \
5   echo 'deb [ signed-by=/usr/share/keyrings/vscodium-archive-keyring.asc ] \
6   https://paulcarrotv.gitlab.io/vscodium-deb-rpm-repo/debs vscodium main' \
7   | tee /etc/apt/sources.list.d/vscodium.list && \
8   apt-get update -q && \
9   apt-get install -y codium && \
10  apt-get autoclean && \
11  apt-get autoremove && \
12  rm -rf /var/lib/apt/lists/*
13

```

C.R. 11
dockerfile

To allow us to do easy programming, we shall install VSCodium, an open-source implementation of VSCode.

```

1 # Install ROS Humble version
2 ENV ROS_DISTRO=humble
3
4 # Install Desktop version
5 ARG INSTALL_PACKAGE=desktop
6
7 RUN apt-get update -q && \
8     apt-get install -y curl gnupg2 lsb-release && \
9     curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
10    -o /usr/share/keyrings/ros-archive-keyring.gpg && \
11    echo "deb [arch=$(dpkg --print-architecture) \
12      signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
13      http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | tee \
14      /etc/apt/sources.list.d/ros2.list > /dev/null && \
15      apt-get update -q && \
16      apt-get install -y ros-${ROS_DISTRO}-$({INSTALL_PACKAGE}) \
17      python3-argcomplete \
18      python3-colcon-common-extensions \
19      python3-rosdep python3-vcstool && \
20      rosdep init && \
21      rm -rf /var/lib/apt/lists/*
22
23 RUN rosdep update
24
25 # Install simulation package only on amd64
26 RUN if [ "$TARGETPLATFORM" = "linux/amd64" ]; then \
27     apt-get update -q && \
28     apt-get install -y \
29     ros-${ROS_DISTRO}-gazebo-ros-pkgs \
30     ros-${ROS_DISTRO}-ros-sign && \
31     rm -rf /var/lib/apt/lists/*; \
32 fi

```

C.R. 12

dockerfile

Now that everything is installed, we start the installation of ROS and its dependencies. Here we define `ENV` variable which allows us set which version of ROS to install.

```

1 # Download the Linux Tutorial file from repo
2 ARG ZIPFILE="https://github.com/dTmC0945/L-MCI-BSc-Mobile-Robotics/raw/refs/heads/main/\
3   datasets/linux-tutorials.zip"
4
5 # Create some user directories to simulate a desktop environment
6 RUN mkdir -p \
7     /home/ubuntu/Downloads \
8     /home/ubuntu/Desktop
9
10 # Download the tutorial files to their correct place
11 RUN cd "/home/ubuntu/Desktop" && \
12     wget -O "linux.zip" "$ZIPFILE" && \
13     unzip "linux.zip" && \
14     rm "linux.zip"

```

C.R. 13

dockerfile

```

16 # Enable apt-get completion after running `apt-get update` in the container
17 RUN rm /etc/apt/apt.conf.d/docker-clean
18
19 # Copy the entrypoint.sh into the image
20 COPY ./entrypoint.sh /
21
22 # Convert file for linux compatibility
23 RUN dos2unix /entrypoint.sh
24 ENTRYPOINT [ "/bin/bash", "-c", "/entrypoint.sh" ]
25
26 # Define user and password
27 ENV USER=ubuntu
28 ENV PASSWD=ubuntu

```

C.R. 14
dockerfile

Finally, we download some additional files from a repo which will be a folder containing tutorial files for use in learning Linux. We then unzip it to Desktop and remove the image. To finish off we define a user called **ubuntu** with a password of **ubuntu**.

As mentioned, a Dockerfile is a text document which includes all the different steps and instructions on how to build a Docker image. In a general sense, the main elements described in the Dockerfile are:

- the base image,
- required dependencies, and
- commands to execute application deployment within a container.

Let's have a look in detail as to what is going on under-the-hood. Here we will have a look at command used in the file. For a more detailed explanation of what is going on, please have a look at the file.

FROM

This instruction sets the base image on which the new image is going to be built upon. It is usually the first instruction in a Dockerfile.

In this case we are downloading the official **ubuntu:jammy-20250404** version from the docker repository.

ARG These define variables used during the building of the image.

Here, we define a variable called **TARGETPLATFORM** to control the architecture installed on the OS

LABEL

Allows writing meta-data information to the docker image. This could be the maintainer of the code or the version as an example.

RUN

This will be an instruction that will be executed for running the commands inside the container while building. It is typically used to install an application, update libraries, or do general setup.

COPY

Allows us to copy a file from the host to the image. In this case we are copying a special file called `entrypoint.sh` which sets up the container with various configurations.

ENV

Sets up environmental parameters inside the image.

To build this image we need to use the CLI, move to the directory where both `Dockerfile` and `entrypoint.sh` is present and run the following code to build image.

```
1 docker build . -t mci:ros2 -f Dockerfile
```

C.R. 15
bash

2.3.2 Running the Container

Now we built the image, we now need to create a container. The following command will create a new container from an image.

This command will create a container every time it is invoked.

An image is a read-only, self-contained template containing instructions for building a Docker container, like a blueprint for a building. Container is a running instance of that image, like the building itself, and is a fully isolated environment for running applications. Images are used to create containers, and multiple containers can be created from the same image.

```
1 docker run \
2   --volume ~/Documents/docker-documents:/home/ubuntu/Desktop/Host \
3   --publish 6080:80 \
4   --name="ros2linux" \
5   --security-opt seccomp=unconfined \
6   --shm-size=512m \
7   mci:ros2
```

C.R. 16
bash

Let's have a look at all the options given here and understand what's going on:

docker run

Our main command. It `runs` a command in a new container, pulling the image if needed and starting the container. Of course, in our file it is already written so when this is executed, there won't be any additional downloads needed.

--volume

Binds volume between the host and the computer. Here we introduce a path from our host computer `~ > Documents > docker-documents` and link it to the docker container `root > home > ubuntu > Desktop > Host` to allow us to share files between host and docker container.

The left side of the path may need to be adjusted for your computer.

--publish

Publish a container's port(s) to the host. Here we are allowing access of the port 6080 (TCP) to the host computer which is used by noVNC.

--name

Adds a specific name to a container. If this option is not set a random will be given.

--security-opt

Additional security options. Here we are passing no additional security options. This is done due to the requirements by noVNC.

--shm-size

the amount of shared memory allotted to a docker container. A temporary file storage filesystem using Random Access Memory (RAM) for storing files.

Once the container is created, please use the following to close it properly.

```
1 docker container stop "ros2linux"
```

C.R. 17
bash

To re-run the container and continue where left off, use the following.

```
1 docker container start "ros2linux"
```

C.R. 18
bash

Chapter 3

Command Line Fundamentals

Table of Contents

3.1	Introduction	69
3.2	The Structure of Commands	72
3.3	Helpful Keyboard Shortcuts for the Terminal	75
3.4	When you need help with Commands	77
3.5	Additional Information	81

3.1 Introduction

In this day and age, it takes a certain level of skill to be alien to technology and as everyone has computers in their pockets, the interactions with them is almost uncountable. However, these interactions are done through what is called a GUI. Devices running on Windows, MacOS, iOS, and Android all use this interface to interact with the user.

i.e., when clicked on an icon, the close, minimize and maximize buttons on the windows etc..

It must be stressed as these visual components are all for the benefit of the user. While these greatly simplify tasks like photo editing or video creation, some applications just completely omit the use of GUI and instead use a simpler version of it called CLI. This is especially true for servers¹, embedded applications and in many other areas where either **memory is limited** or efficiency of the computer (e.g., such as limiting the CPU load etc.) is highly desired. Server software, utilities, and other programs usually only need some text-based information to do what they do. Many of these programs run on a server in a data centre somewhere without a monitor so the overhead of a GUI is completely **unnecessary**.

¹A server is a software or hardware offering a service to a user, usually referred to as client. As an example, a hardware server is a shared computer on a network, usually powerful and housed in a data centre.

UNIX	Windows
Bourne shell (sh)	COMMAND.COM, default in Windows 9x and provided for DOS compatibility in 32-bit versions of NT-based Windows via NTVDM.
Almquist shell (ash)	
Debian Almquist shell (dash)	
Bash (Unix shell) (bash)	
Korn shell (ksh)	cmd.exe , the default command-line interpreter of the Windows NT-family
Z Shell (zsh)	
C shell (csh)	Recovery Console
TENEX C shell (tcsh)	Windows PowerShell, based on .NET Framework
Ch shell (ch)	PowerShell, based on .NET Core
Emacs shell (eshell)	Hamilton C shell, a clone of the Unix C shell
Friendly interactive shell (fish)	
Powershell (pwsh)	4NT, a clone of CMD.EXE.
rc shell (rc)	Take Command, a newer incarnation of 4NT
Stand-alone shell (sash)	
Scheme Shell (scsh)	

Table 3.1: Types of shells used in industry and academia. For reference, the authors computer uses zsh.

One way we interact with these programs that don't have a GUI is through the CLI. This is a text-based interface where the commands to execute are typed and all actions are shown as text on a terminal screen, whether it is updating a software or moving files around. The environment we use is called a shell, or command-line interpreter, and there are many shells out there.

A list of Shells that can be encountered in industry and academia can be seen in **Table 3.1**.

The command-line interpreter was one of the earliest ways of interacting with the general-purpose computer, starting in 1971 with the Thompson shell for UNIX². As UNIX evolved and came to be replaced in many capacities by Linux, the shell environments evolved and improved as well.

Bash, or the Bourne-again shell is one of the most widely-used shells and odds are, it's the one to be encountered in industry or in academic work. Bash is the shell that comes enabled by default with most of the popular Linux distributions. It's also available on macOS³ and in Windows with the Windows subsystem for Linux.

²A family of multitasking, multi-user computer operating systems that derive from the original

AT&T Unix, whose development started in 1969. It is considered one of the most groundbreaking software ever designed.

³newer versions have [zsh](#) shell instead but they are designed to be compatible.

The author of this work also uses [zsh](#) as his main driver.

In this document, Bash will be used. However, the reader is encouraged to explore some of the other shells out there once a working foundation in Bash is achieved.

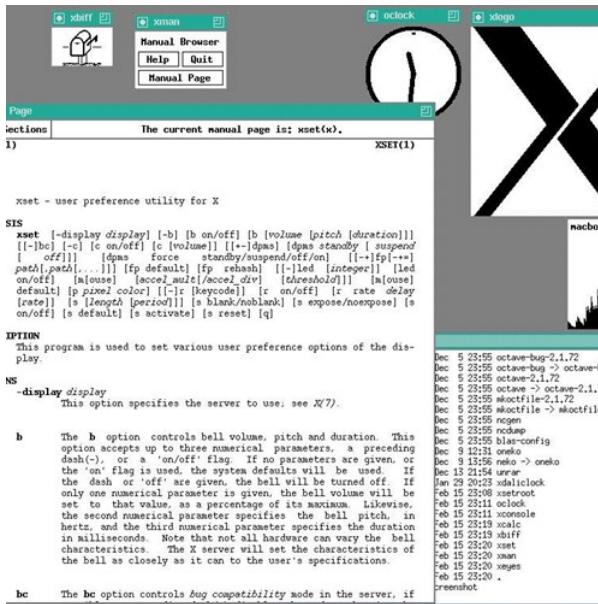


Figure 3.1: A graphical interface from the late 1980s, which features a TUI window for a man page, a shaped window (oclock) as well as several iconified windows. In the lower right we can see a terminal emulator running a Unix shell, in which the user can type commands as if they were sitting at a terminal. - *From Wikipedia*

3.2 The Structure of Commands

There are a few concepts and principles which needs to be understood to be a productive member of the CLI family. Before jumping into using commands though, have a look at how command line statements are structured with the following:

```
1  command [-flag(s)] [-option(s) [value]] [argument(s)]
```

C.R. 1

bash

This is the general form. The pattern is **command**, **options**, and then **arguments**. Here's a couple of common commands you'll see with options and arguments that are used with them.

```
1  ls -l /tmp
2  cd /usr/local
3  cat /etc/passwd
```

C.R. 2

bash

The details of what the aforementioned commands do will be the focus in the future. I just want to show you the structure of what we'll be working with before we get into what these actually do. Depending on the current action, you might just have a command or a command and one or more options or just a command with one or more arguments.

But there will always be a command.

Command is the **minimum** thing which can be done with a CLI. Think of it as the atom of any action you can take. The command is the program you're running or the action you're taking. To give

command to a UNIX system, type the name of the command, along with any associated information, such as a filename, and press the **Return** key.

The typed line is called the command line and UNIX uses a special program, called the shell or the command line interpreter, mentioned in the previous section, to interpret what you have typed into what you want to do.

The components of the command line are:

1. the command,
2. any options required by the command,
3. the command's arguments.⁴

⁴This is optional as some commands just don't have any options.

3.2.1 Some Rules Regarding the Syntax

Since the introduction of UNIX System V, Release 3 (released 1983), any new commands must obey a particular syntax governed by the following rules:

- Command names must be between 2 and 9 characters in length,
- Command names must be comprised of lowercase characters and digits,
- Option names must be one character in length,
- All options are preceded by a hyphen (-),
- Options without arguments may be grouped after the hyphen,
- The first option argument, following an option, must be preceded by white space,
i.e., `-o sfile` is valid but `-osfile` is **illegal**.
- Option arguments are **not optional**,
- If an option takes more than one argument then they must be separated by commas with no spaces, or if spaces are used the string must be included in double quotes (").
i.e., both are acceptable: `-f past,next` and `-f "past now next"`.
- All options must precede other arguments on the command line,
- A double hyphen -- may be used to indicate the end of the option list,
- The order of the options are order independent,

- The order of arguments may be important,
- A single hyphen - is used to mean standard input.

Options **must** come after the command and before arguments. Options **should not** appear after the main argument(s). However, some options can have their own arguments! Historically, UNIX commands have been fairly standard in the way that they use options but there are variations.

Bear in mind that commands established before System V, Release 3, do not conform to all of the above rules.

3.3 Helpful Keyboard Shortcuts for the Terminal

Before we moving on to more specific commands and get into CLI programming, there's a few other helpful things to know about working at the Command Line. The first is **Tab completion**, a wonderful feature of the Bash shell, and is also included in many others. This feature let's you skip typing out a whole file name or folder name when you're working at the Command Line.

When you're working in the command line it looks at all the information it has so far and makes a guess about what you mean. For example, I can type `ls -l De` and press `Tab`, and it completes the line with `Desktop`. Now type `ls -l Do` and nothing would happen when I press `Tab`. That's because `Tab` doesn't have one clear suggestion to return. As it can be either `Documents` or `Downloads`. However, pressing `Tab` again should give you a suggestion of which items can be completed to.

For reference, in the following page, there is a table for most useful keyboard shortcut for Linux Bash.

	Shortcut	Action
Navigation	<code>Ctrl + A</code>	Go to the beginning of the line.
	<code>Ctrl + E</code>	Go to the end of the line.
	<code>Alt + F</code>	Move the cursor forward one word.
	<code>Alt + B</code>	Move the cursor back one word.
	<code>Ctrl + F</code>	Move the cursor forward one character.
	<code>Ctrl + B</code>	Move the cursor back one character.
	<code>Ctrl + X</code>	Toggle between the current cursor position and the beginning of the line.
	<code>Ctrl + A</code>	Undo! (That's an underscore, so you'll need to use <code>Shift</code> as well.).
Editing	<code>Ctrl + X</code>	Edit the current command in your \$EDITOR.
	<code>Alt + D</code>	Delete the word after the cursor.
	<code>Alt</code>	Delete the word before the cursor.
	<code>Ctrl + D</code>	Delete the character beneath the cursor.
	<code>Ctrl + H</code>	Delete the character before the cursor (like backspace).
	<code>Ctrl + K</code>	Cut the line after the cursor to the clipboard.
	<code>Ctrl + U</code>	Cut the line before the cursor to the clipboard.
	<code>Ctrl + D</code>	Cut the word after the cursor to the clipboard.
	<code>Ctrl + W</code>	Cut the word before the cursor to the clipboard.
	<code>Ctrl + Y</code>	Paste the last item to be cut.
	<code>Ctrl + L</code>	Clear the entire screen (like the clear command).
	<code>Ctrl + Z</code>	Place the currently running process into a suspended background process.
Processes	<code>Ctrl + C</code>	Kill the currently running process by sending the SIGINT signal.
	<code>Ctrl + D</code>	Exit the current shell.
	<code>Return</code>	Exit a stalled SSH session.
	<code>Ctrl + R</code>	Bring up the history search..
	<code>Ctrl + G</code>	Exit the history search.
History	<code>Ctrl + P</code>	See the previous command in the history.
	<code>Ctrl + N</code>	See the next command in the history.

3.4 When you need help with Commands

If you ever see an experienced Linux user typing away at the command line in blazing speeds it can seem like memorising the ins and outs of commands and options is the only way to be productive and understand what's going on. But everybody starts somewhere, and even experienced command-line users don't memorize everything.

In the world of programming, it's not practical to try to memorise all of the syntax and options of command-line tools. Of course, it's important to remember the basics, but while you're getting started, you only need to remember a few commands. The first one is `man`, which stands for the **manual pages**.

```

1 MAN(1)           Manual pager utils      MAN(1)      text
2
3 NAME
4   man - an interface to the system reference manuals
5
6 SYNOPSIS
7   man [man options] [[section] page ...] ...
8   man -k [apropos options] regexp ...
9   man -K [man options] [section] term ...
10  man -f [whatis options] page ...

```

A `man` page⁵ is a form of software documentation found on UNIX and Unix-like operating systems. Topics covered include programs, system libraries, system calls, and sometimes local system details.

⁵Stands for short for manual page.

Think of the man pages as a technical reference book for your Linux distribution⁶. If you know the name of a command, you can find out a wealth of information about what it does, what options it provides or what arguments it takes. To look up something in the manual pages, type `man`, followed by a command you want to learn. Open up the Terminal by **[Ctrl]** + **[Alt]** + **[T]**. Earlier, you saw the command `ls`, so let's look that up. Type `man ls` and press **Return**.

⁶i.e., Ubuntu, Mint

Some distributions or application specific installation of Linux remove the `man` pages to save up on space. In these system one must first do `unminimize` to install `man` pages.

```

C.R. 3
1 man ls | head -10                                bash
2
3 LS(1)           User Commands      LS(1)      text
4
5 NAME
6   ls - list directory contents
7
8 SYNOPSIS
9   ls [OPTION]... [FILE]...

```

```
9 | DESCRIPTION  
10 | List information about the FILEs (the current directory by default).  
text
```

⁷Please ignore the head - 10, we will have a look at it later.

Here, you can see some information about the `ls` command⁷. You can see that it's for listing directory contents and in the synopsis section you get a quick overview of how to use the command. In this case it is `ls [OPTION]... [FILE]...`. We write `ls` followed by any of the options we need, and the file or folder path we want to use.

⁸for example `[OPTION]` and `[FILE]` The terms in square brackets⁸ are optional. This basically means **you don't have to use these** for the command to work. You can just use the `ls` command by itself to see the default output of listing the directory. Here, below the description header, there is a bit more detailed information about the command, including its default behaviour and usage notes, and below, is a listing of the options that the command takes.

```
1 | Usage: ls [OPTION]... [FILE]...  
2 | List information about the FILEs (the current directory by default).  
3 | Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.  
4 |  
5 | Mandatory arguments to long options are mandatory for short options too.  
6 | -a, --all           do not ignore entries starting with .  
7 | -A, --almost-all   do not list implied . and ..  
8 | --author          with -l, print the author of each file  
9 | -b, --escape       print C-style escapes for nongraphic characters  
10 | --block-size=SIZE with -l, scale sizes by SIZE when printing them;  
text
```

There are a lot of ways to use the `man` pages efficiently and is a powerful tool when you need to find what can a command do. There are other ways to learn about a command. Most of commands also have an option called `help`, which provides a brief amount of information about them. However, they usually refer you to the manual pages for more detailed documentation. Therefore, `help` will give you a brief information compared to the `man` command.

You can see if a command you're using has this feature available by typing `--help` after the command.

```
1 | ls --help | head -10  
C.R. 4  
bash
```



```
1 | Usage: ls [OPTION]... [FILE]...  
2 | List information about the FILEs (the current directory by default).  
3 | Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.  
4 |  
5 | Mandatory arguments to long options are mandatory for short options too.  
6 | -a, --all           do not ignore entries starting with .  
7 | -A, --almost-all   do not list implied . and ..  
8 | --author          with -l, print the author of each file  
9 | -b, --escape       print C-style escapes for nongraphic characters  
10 | --block-size=SIZE with -l, scale sizes by SIZE when printing them;  
text
```

Here you can scroll up and down to have a look at some of the information. There is another command that's useful when you're working in Bash, and that's just `help` by itself.

Information

The `help` Command

Displays information about shell built-in commands.

```

1 help | head -10                                         C.R. 5
2
3
4
5
6
7
8
9
10
```

GNU bash, version 5.2.21(1)-release (aarch64-unknown-linux-gnu) text
These shell commands are defined internally. Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

job_spec [&] history [-c] [-d offset] [n] or hist>
((expression)) if COMMANDS; then COMMANDS; [elif C>

As we get into working with the Bash shell, the `help` tool can act as a handy reminder for the syntax of some Bash specific commands.

But what if you don't know the name of a command you are looking for?

In that case, you can use another program called `apropos` which searches a list of commands and their descriptions for text you provide as an argument.

Information

The `apropos` Command

helps users find any command using its `man` pages.

So if you wanted to find out what can list things, I could type `apropos list` and see a number of results that match that word.

```

1 apropos list | head -10                                         C.R. 6
2
3
4
5
```

port-contents(1) - List the files installed by a given port text
port-dependents(1), port-rdependents(1) - List ports that depend on a given (installed)
 ↳ port
port-deps(1), port-rdeps(1) - Display a dependency listing for the given port(s)
port-distfiles(1) - Print a list of distribution files for a port
port-echo(1) - Print the list of ports the argument expands to

```
6 port-installed(1)      - List installed versions of a given port, or all installed ports
7 port-list(1)           - List available ports
8 port-outdated(1)       - List outdated ports
9 port-variants(1)        - Print a list of variants with descriptions provided by a port
10 AllPlanes(3), BlackPixel(3), WhitePixel(3), ConnectionNumber(3), DefaultColormap(3),
    ↳ DefaultDepth(3), XListDepths(3), DefaultGC(3), DefaultRootWindow(3),
    ↳ DefaultScreenOfDisplay(3), DefaultScreen(3), DefaultVisual(3), DisplayCells(3),
    ↳ DisplayPlanes(3), DisplayString(3), XMaxRequestSize(3), XExtendedMaxRequestSize(3),
    ↳ LastKnownRequestProcessed(3), NextRequest(3), ProtocolVersion(3), ProtocolRevision(3),
    ↳ QLength(3), RootWindow(3), ScreenCount(3), ScreenOfDisplay(3), ServerVendor(3),
    ↳ VendorRelease(3) - Display macros and functions
```

Here's the command that can list directory contents we were looking for.

```
1 ls(1)                  - list directory contents
```

text

Searching for commands this way can be time-consuming, but if you know what you need to do but not the command to do it, [apropos](#) is very helpful and powerful.

3.5 Additional Information

3.5.1 Use Tab completion on the Shell

If you do not know the exact name of a command, then you can make use of tab completion. To use this action, launch the terminal by pressing **Ctrl** + **Alt** + **T** or just click on the terminal icon in the task bar. Just type the command name that you know in the terminal and then press **Tab** twice. For example, if we can't remember **man**, we can write **ma** and can choose one of the option the Bash shell presents us.

```
1 ~$ ls
C.R. 7
bash

1 macptopbm      make-ssl-cert
2 mag            mako-render
3 mailmail3       man
4 make           mandb
5 make4ht         manpath
6 makeconv        man-recode
7 makedtx        mapfile
8 make-first-existing-target mapsrn
9 makeglossaries match_parens
10 makeglossaries-lite mathspic
11 makeindex      mattrib
12 makejvf        mawk
```

3.5.2 The info command

Some commands do not have their manuals written or they are either **incomplete**. To get help with those commands, we use **info**. To use this command, launch the terminal by pressing **Ctrl** + **Alt** + **T** or just click on the terminal icon in the task bar. Just type **info** in the terminal and with a space, type the name of the command whose manual does not exist and press **Return**.

```
1 info ls | head -10
C.R. 8
bash

1 File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directory listing
2
3 10.1 ls: List directory contents
4 =====
5
6 The ls program lists information about files (of any type, including
7 directories). Options and file arguments can be intermixed arbitrarily,
8 as usual. Later options override earlier options that are incompatible.
9
```

¹⁰

For non-option command-line arguments that are directories, by

text

⁹A mostly a plain text transliteration of the Texinfo source, with the addition of a few control characters to separate nodes and provide navigational information, designed by the NU project.

¹⁰1 for commands, 2 for system calls, etc...

The **info** command reads documentation in the info format⁹. It will give detailed information for a command when compared with the man page. The pages are made using the Texinfo tools which can link with other pages, create menus, and easy navigation.

Information

Man v. Info

Man pages are the UNIX traditional way of distributing documentation about programs. The term “man page” itself is short for “manual page”, as they correspond to the pages of the printed manual; the man pages “sections”¹⁰ correspond to sections in the full UNIX manual. Support is still there if you want to print a man page to paper, although this is rarely done these days, and the sheer number of man pages make it just impossible to bind them all into a single book.

In the early '90s, the GNU project decided that “man” documentation system was outdated, and wrote the info command to replace it: info has basic hyperlinking features and a simpler markup language to use (compared to the troff¹¹ system used for man pages). In addition, GNU advocates against the use of man pages at all and contends that complex software systems should have complete and comprehensive documentation rather than just a set of short man pages.

In the end, the form in which you get documentation depends on the internal policies of the project that provided the software in the first place – there is no globally accepted standard.

3.5.3 The whatis command

This command is used with another command just to show a one liner usage of the latter command from its manual. It's a quick way of knowing the usage of a command without going through the whole manual.

whatis command in Linux is used to get a one-line manual page description. In Linux, each manual page has some sort of description within it. So, this command search for the manual pages names and show the manual page description of the specified filename or argument.

To use this command, launch the terminal by pressing **Ctrl** + **Alt** + **T** or just click on the terminal icon in the task bar. Just type **whatis** in the terminal and after a space, type the name of the command whose one liner description you want (for example **ls**) and then press **Return**.

¹ **whatis ls | head -10**

C.R. 9

bash

```
1 dcmcjpls(1)           - Encode DICOM file to JPEG-LS transfer syntax      text
2 dcmdjpls(1)           - Decode JPEG-LS compressed DICOM file
3 gdircolors(1), dircolors(1) - color setup for ls
4 gls(1), ls(1)         - list directory contents
5 gdircolors(1), dircolors(1) - color setup for ls
6 git-ls-files(1)        - Show information about files in the index and the working tree
7 git-ls-remote(1)       - List references in a remote repository
8 git-ls-tree(1)         - List the contents of a tree object
9 git-mktree(1)          - Build a tree-object from ls-tree formatted text
10 gls(1), ls(1)         - list directory contents
```


Chapter 4

Working with Files and Folders

Table of Contents

4.1	Introduction	85
4.2	A Detailed Look in ls Command	90
4.3	Creating and Removing Folders	92
4.4	Move, Copy and Delete Files and Folders	94
4.5	Role to Users and sudo	96
4.6	File Permissions	98
4.7	Hard and Symbolic Links	101
4.8	The Linux File System	104
4.9	Common Command-Line Tools and Tasks	107
4.10	Advanced Topics	111

4.1 Introduction

If you've ever worked with computers for any amount of time, you would probably be familiar with the concept of **files** and folders.¹ Files are a collection of information representing photos, documents, source code, databases and all kinds of other things.

They can be thought as the basic unit of data storage we work with a GUI. That's still pretty much the same in the CLI as well. There are two (2) commands that needs explaining. These are called **file** and **stat**. Both these commands can look at a file and learn some things about it.

¹I mean this is in hope that you are familiar otherwise we might have a problem.

- The first one, **file** will generally be able to tell what kind of file you're asking about.

- If a file's name isn't clear or if it doesn't have an extension, sometimes it can be tricky to figure out what exactly it is.

- Using `file`, will give you some insight into whether something is an archive or an executable file or say, a text file or other kind of document.

■ The second one, `stat`, on the other-hand, tells you some extended information about a file.

Information

The `file` Command

Determines the type of a file. It identifies file types by examining their content rather than their file extensions.

Information

The `stat` Command

Provide detailed statistics about files and file systems. It displays crucial information such as file size, permissions, ownership, and timestamps.

To have a quick test, lets have a file called `sample.txt` with the contents of the following:

```
1 It was the best of times, it was the worst of times, it was the age of wisdom,
2 it was the age of foolishness, it was the epoch of belief, it was the epoch of
3 incredulity, it was the season of light, it was the season of darkness, it was
4 the spring of hope, it was the winter of despair.
```

C.R. 1
text

Now while we now what it contains, let's assume we don't. To see what kind of formatting this file contains we run the `file` command:

```
1 cd ~/Downloads || exit &&
2     file sample.txt
```

C.R. 2
bash

```
1 sample.txt: ASCII text
```

text

As we can see this command tells us the file has an `ASCII` formatting which means it is generally supported by almost all computers without the need of additional text encoding.² To get more information about the file we invoke the `stat` command:

```
1 cd ~/Downloads || exit &&
2     stat sample.txt
```

C.R. 3
bash

```
16777232 85091780 -rw-r--r-- 1 danielmcguiness staff 0 287 "Mar 4 19:05:27 2025" \
"Mar 4 19:05:15 2025" "Mar 4 19:05:16 2025" "Mar 4 19:05:15 2025" 4096 8 0 sample.txt
```

As we can see, we have a bit more information about the file, regarding its user, the date in which it was modified, the size and more. As we'll see when we look at the `ls` command, some of this is

²an acronym for American Standard Code for Information Interchange, is a character encoding standard for representing a particular set of 95¹ (English language focused)² printable and 33 control characters - a total of 128 code points. The set of available punctuation had¹ significant impact on the² syntax of computer languages and text markup. ASCII hugely influenced the design of character sets used by modern computers; for example, the first 128 code points of Unicode are the same as ASCII.

available there. These commands can be helpful to know about if you come across an unknown file. In the graphical environment³, we can navigate around these files and folders with the mouse, seeing how they're organized and finding out information about them. We can do the same thing in a CLI⁴.

³i.e., GUI

⁴In a much faster, but more unforgiving way.

In the file browser, we can navigate to the [Linux Tutorials](#) file. From the Home folder, you can click on Desktop. There's the file. In this graphical interface, we can see pretty easily what folder you are working in. Over here in the Terminal, we get a clue about what folder we're working in on the prompt. The tilda (~) the character, right here, means your home folder.

To match up with where the file browser is, the Linux Tutorials folder, you'll need to navigate into the Desktop and then into Linux Tutorials. To do that, use the `cd` command which stands for **change directory** (for more information try typing `man cd` in your terminal window). Start by typing the path that we want to go to. Type `De` and then press `Tab` to auto complete, since Bash knows what's available. Right now, nothing else in your Home folder should start with De except Desktop. Then press `Return` to run that command. Since we've navigated to a different folder, the prompt on your terminal window should change.

Now, it says tilde slash Desktop (`~/Desktop`), indicating that the present working directory is the documents residing inside of the `/home` folder. You can also find that out by typing `pwd` no your terminal window, for print working directory.

That shows the full path, or absolute path of a folder where you are currently working. An absolute path starts from the root of the file system, the highest level of the structure where files are stored. Inside of the root, the home folders for users are stored in the `/home` folder, and then my user's home folder is represented by your user name.

Inside that is documents but we need to go one folder deeper to get inside the [Linux Tutorials](#) folder. Write `cd Linux Tutorials` and press `Return`, but we get an error. You can see here that Bash thinks that we're trying to get into the folder called just Linux. That's because `cd` has interpreted Linux Tutorials, as two words, two separate arguments, because there's a space in between the words. You have to tell Bash that the **space is part of the name**, not a separator between two arguments or commands.

There are two ways to do this. The first way is to put the string of text inside quotes (" "), but the more common thing you'll see is to just escape a special characters. In this case the space between Linux and Tutorials. To let Bash know that the space is part of the folder name, not a break in the command, we type a back slash (\) in front of it. Escaping a character means that it's treated literally instead of having any other special meaning.

That works for one character at a time. If we had two spaces in there, we need to escape each space character individually. So, again type [`cd space Linux\ Tutorial`] and press `Enter`. Now when we type [`pwd`], we can see where we are.

```
1 ls
C.R. 4
bash
```

```
1 Books Data.txt Folder Poem.txt
```

text

Now that we're inside the tutorial files folder, Type `ls` again to see what we've got.

```
1 ls
```

C.R. 5

bash

```
1 Books Data.txt Folder Poem.txt
```

text

Let's take a look inside the `Books` folder. Write `ls` and this time I'll use the `-R` option to list folders recursively and add `Books/` here on the end.

```
1 ls -R Books/
```

C.R. 6

bash

```
1 Books/:
```

text

```
2 Classics Fantasy Literature Music Poetry Sci-Fi Science
```

```
3
```

```
4 Books/Classics:
```

```
5 'Charles Dickens' 'Herman Mellvile' 'Jane Austen'
```

```
6
```

```
7 'Books/Classics/Charles Dickens':
```

```
8
```

```
9 'Books/Classics/Herman Mellvile':
```

```
10
```

```
11 'Books/Classics/Jane Austen':
```

```
12
```

```
13 Books/Fantasy:
```

```
14 'Brandon Sanderson' 'G. R. R. Martin' 'J.R.R. Tolkien' 'Robert Jordan'
```

```
15
```

```
16 'Books/Fantasy/Brandon Sanderson':
```

```
17
```

```
18 'Books/Fantasy/G. R. R. Martin':
```

```
19
```

```
20 'Books/Fantasy/J.R.R. Tolkien':
```

```
21 Unfinished_LotR_Sequel.txt
```

```
22
```

```
23 'Books/Fantasy/Robert Jordan':
```

```
24
```

```
25 Books/Literature:
```

```
26 American English Greek Turkish
```

```
27
```

```
28 Books/Literature/American:
```

```
29
```

```
30 Books/Literature/English:
```

```
31
```

```
32 Books/Literature/Greek:
```

```
33
```

```
34 Books/Literature/Turkish:
```

```
35
```

```
36 Books/Music:
```

```
37  
38 Books/Poetry:  
39  
40 Books/Sci-Fi:  
41 'Arthur C. Clarke' 'Frank Herbert' 'Isaac Asimov'  
42  
43 'Books/Sci-Fi/Arthur C. Clarke':  
44  
45 'Books/Sci-Fi/Frank Herbert':  
46  
47 'Books/Sci-Fi/Isaac Asimov':  
48  
49 Books/Science:  
50 Biology Chemistry Physics  
51  
52 Books/Science/Biology:  
53  
54 Books/Science/Chemistry:  
55  
56 Books/Science/Physics:
```

Now we can see what's inside all of the folders inside `/Books`. Your recursive options means when `ls` comes across a folder, it steps inside and looks around, listing anything inside the folder. If it comes across another folder inside that folder, it does the same thing, steps inside, looks around and reports back. This is a helpful way of exploring a whole structure of folders.

4.2 A Detailed Look in ls Command

As the `ls` command is one of the more useful commands, it is worth looking into it in a bit more detail. As discussed, `ls` is a command in the CLI which allows us to lists the contents of a directory.

```
1 cd "home/ubuntu/Desktop/Linux Tutorials" && ls  
1 Books Data.txt Folder Poem.txt
```

This command is highly useful as it's **short** and has an **easily changeable output**. The output itself is pretty useful, and therefore it's worth taking some time to understand what it shows.

`ls`, just by itself gives a list, and depending on the environment, the items might have some colour or they might not. The colouration is helpful but it's not critical to use in `ls`. If it is not visible, try running `ls` with the dash dash colour equals always option (`--color=always`).

Let's go to the file Linux Tutorials and look around. As we have changed the path in our previous code, and as long as we are using the same terminal window, our path should still be in [Linux Tutorials](#). However if it is not the case, please Write `cd Desktop/Linux\ Tutorials/`. Once we are in the directory, write the following:

```
1 ls -l                                C.R. 8  
2  
3 total 20                               bash  
4 drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 22 2020 Books  
5 -rw-rw-r-- 1 ubuntu ubuntu   102 Nov 20 2020 Data.txt  
6 drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 20 2020 Folder  
7 -rw-rw-r-- 1 ubuntu ubuntu   592 Nov 18 2020 Poem.txt  
8
```

Here, the `-l` is used to see more information about the files within the directory. The first column on the left shows whether an item is:

- A folder or a directory which will be shown with a `d`,
 - A link which will be shown with a `l`,
 - A file, which is a dash (`-`). This means that the attribute is missing or offset.

If the output shows colours, folders will generally be blue text. Links will generally be light blue text and files will generally be grey or they'll be black or white depending on the background colour of your terminal.

The next set of columns show a representation of the permissions on the file. What different kinds of users are allowed to do with the file. Further to the right, we see the owner of the file, and the

group setting of the file. Then we see the size of the file in bytes, which can be a little bit easier to read with a `-h` option. Clear the screen and use `ls -lh`.

```
1 ls -lh                                         C.R. 9  
2  
3 total 20K                                     bash  
4 drwxrwxr-x 1 ubuntu ubuntu 4.0K Nov 22 2020 Books  
5 -rw-rw-r-- 1 ubuntu ubuntu 102 Nov 20 2020 Data.txt  
6 drwxrwxr-x 1 ubuntu ubuntu 4.0K Nov 20 2020 Folder  
7 -rw-rw-r-- 1 ubuntu ubuntu 592 Nov 18 2020 Poem.txt  
8
```

This calls out the size post fixes. `k` for kilo, `m` for mega, `g` for giga, and `t` for terabytes. Then there's the date and time that the file was modified. Finally the file name, or in the case of a link, which we'll explore a bit later. `ls` with its wide variety of options, give you a whole lot of helpful information about folders and files, so it's a good command to know about.

4.3 Creating and Removing Folders

Sometimes, we'll need to create folders to organise our files, and sometimes we'll need to remove folders as well. Sometimes it is good to know how to do it using the CLI. Currently this is the content of [Linux Tutorials](#). There is a single folder and a text file.

First, create a new folder or directory here in [Linux Tutorials](#) folder with `mkdir` which stands for make directory, and give it a name. Let's call it `Folder`, to keep it simple.

```
1 mkdir Folder
```

C.R. 10

bash

If we observe the file content, we can see the folder has appeared. We can also see the content of this directory with the command `ls -l`. There is the `Folder` we have just created.

```
1 ls -l
```

C.R. 11

bash

```
1 total 20
2 drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 22 2020 Books
3 -rw-rw-r-- 1 ubuntu ubuntu 102 Nov 20 2020 Data.txt
4 drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 20 2020 Folder
5 -rw-rw-r-- 1 ubuntu ubuntu 592 Nov 18 2020 Poem.txt
```

text

If we put a name after `mkdir`, it assumes we want to create a folder inside of the current working directory. In this case, this would be [Linux Tutorials](#). We can also specify a path outside the current folder or a folder deeper inside the working folder.

For example, Add a new folder inside our Books folder for Poetry. For that we have to write `mkdir Books/Lyrics`.

```
1 mkdir Books/Lyrics
```

C.R. 12

bash

What if we wanted to create a subfolder within a new folder? For example, let's say we want to make a Beethoven folder inside a Music folder. Instead of creating a Music folder and then creating a Beethoven folder inside of it, we can do it all at once using the `-p` option for `mkdir`. This option creates any parent folders that are needed, so in this case, it'll create the Music folder for us and then create the Beethoven folder after that. Write `mkdir -p Books/Music/Beethoven`.

```
1 mkdir -p Books/Music/Beethoven
```

C.R. 13

bash

Here are the newly created files. This is a versatile command. We can also remove empty directories using the `rmdir` command for remove directory. Let's go remove the Beethoven folder that we just created. For that, we have to write `rmdir Books/Music/Beethoven` and now if we take a look inside the Music folder, it's empty. One thing to keep in mind about removing folders this way is

that in order to remove a folder, it has to be empty. That means it's a little more tedious to remove a large folder structure with rmdir.

4.4 Move, Copy and Delete Files and Folders

It's pretty common to need to move, copy, and delete files in our day to day GUI activities, and this is also true when you are working using a CLI. Some experienced command line users would even argue that it is more efficient and effective to do file management using CLI than GUI as GUI might take more time with mouse movements.

In light of this, the first command for you to learn using in CLI file management to you here is `cp` which stands for copy. Remember if you don't remember or know how a command options work use the `man` command followed by the command you want to know more about (i.e., `man cp`.)

Information

The `cp` Command

A command in Unix and Unix-like operating systems for copying files and directories.

Let's make a duplicate copy of our `Poem.txt` file. As we can see, we are at the right directory to do any duplication.

```
1 ls -l                                         C.R. 14  
bash  
  
1 total 20  
2 drwxrwxr-x 1 ubuntu ubuntu 4096 May 19 06:07 Books  
3 -rw-rw-r-- 1 ubuntu ubuntu 102 Nov 20 2020 Data.txt  
4 drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 20 2020 Folder  
5 -rw-rw-r-- 1 ubuntu ubuntu 592 Nov 18 2020 Poem.txt
```

⁵if you don't have a terminal window open you can open one using  [Ctrl] + [Alt] + [T]

```
1 cp Poem.txt Poem_Copy.txt                  C.R. 15  
bash
```

Press [Return], and then take a look at the contents of this folder again with `ls -h`. Here we can see, there's the original `Poem.txt` file, and here's `Poem_Copy.txt`. You can also copy a file to a different path. For example, we can copy our `Poem.txt` file into our `/Poetry` folder inside the `/Books` folder. To do that, write `cp Poem.txt`, and then use [Tab] completion to get to `Books/Poetry`.

```
1 cp Poem.txt Books/Poetry                   C.R. 16  
bash
```

If we list the folder, we can see that the file's been copied there.

```
1 cd Books/Poetry && ls -h                 C.R. 17  
bash
```

```
1 Poem.txt
```

text

Let's take a look at moving a file rather than copying it. In principle, the move command has two (2) uses:

1. You can use it to move files between folders,
2. You can also use it to rename files.

The command for move is `mv`, so using this command let's move the `Poem_Copy.txt` to the `/Books` subfolder inside the `/Linux Tutorial` folder.

```
1 cd - && mv Poem_Copy.txt Books/
```

C.R. 18
bash

And we can check that the file's in that folder. And then we can see that it's no longer in the original folder by using `ls` here. It is also possible, as we mentioned before, to rename files with the `mv` command.

Information

The `mv` command

Moves one or more files or directories from one place to another.

Let's try it. Let's rename the `Poem_Copy.txt` to `Poem_Duplicate.txt` using the `mv` command.

```
1 cd Books && mv Poem_Copy.txt Poem_Duplicate.txt
```

C.R. 19
bash

```
1 rm -rf ./
```

C.R. 20
bash

```
1 ls
```

C.R. 21
bash

4.5 Role to Users and sudo

Linux is a multi-user environment. Now, what this means is that multiple users can use an operating system. This is a concept we're familiar with nowadays but was a new idea decades ago when Unix came on the scene when it was mostly used by specialized engineers and programmers. In principle I can have a user, someone else can have a user in the same operating system, but our files are kept separate in our individual home folders.

We can create files that only one or another user can access. At the command line, we can switch between users with the `su` command, which is variously referred to as set user, switch user, or substitute user. To use `su`, we write the command followed by the name of the user we want to switch to. Probably the most common use of switching users at the command line is to do some system administration tasks. There are two basic user roles in Linux. There's the **normal user** and the **superuser**. The difference here is one of privilege.

The normal user can modify, create, delete, and move their own files, but they can't make changes to the system. They can't install software, they can't make changes to system files, and generally speaking, they can't browse other users' home folders. The superuser, which is called root, can make changes to the system. It can install software, it can start and stop services, and so on. Normal users can be granted the ability to temporarily use root's power through a command called `sudo`. It's uncommon and it's really bad practice to log into the root user directly to do normal work. In fact, on many systems, the root user is actually disabled and can't be logged into.

You only want to borrow root's power when you really need it, so let's take a look at that. Let's try to see what's inside root's home folder, which is located at the root of the drive. These are two different meanings of the word root, which can be a little confusing. Remember, when we're talking about a file system, the root is at the highest level of the organizational structure and that's represented by a single slash. When we're talking about accessing users, root is the superuser. You could probably draw some parallels between levels in a hierarchy, but just keep in mind there's two different meanings for the word root on Linux. Let's see what happens when I write `ls /root`, and I see I'm denied permission.

```
1 ls /root                                C.R. 22  
                                bash  
  
1 ls: cannot open directory '/root': Permission denied      text
```

We need to use the `sudo` command to gain root's privileges to see inside there. This command

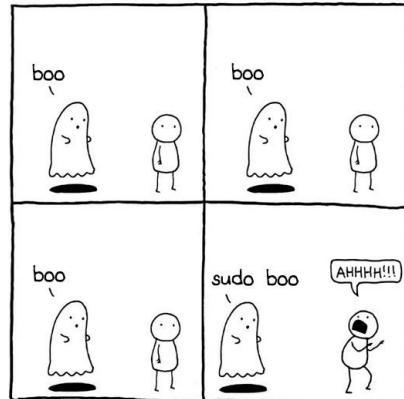


Figure 4.1: Beware of the sudo ghost.

basically tells the system to run whatever command is after it with superuser privileges instead of the normal user's privileges. So, write `sudo ls /root`. I'm prompted for my password. This is a good sign that the computer understood that I want superuser privileges.

Information _____ **Removing password from the system**

While it is not recommended, Linux gives you options to do anything and this includes **removing** user password from the computer. This would allow you to run `sudo` commands without ever being prompted for a password. You start first use the `passwd` command to delete the password of the user (which is you)

```
sudo passwd -d username
```

After entering your username in place of `username`, if you see **password expiry information changed** in the output, the password has been deleted successfully and now you can do all superuser actions without the need of a password.

4.6 File Permissions

At a first glance, file permissions can seem rather cryptic as these were devised when every key stroke mattered. We've seen them before when listing files in a directory but it's not immediately clear what they mean.

For example, `rwxr-xr-x` might not make any sense right now, but hopefully after this section you will have a working understanding of the file permission system in Linux. The sequence of letters breaks down into three (3) groups:

1. The First represents the user, or the owner of the file,
2. Second group of three represents the group that owns the file,
3. Third group represents all other users not in the group that owns the file.

Each of the groups of three breaks down into three individual letters, which stand for

Read someone can see the contents of a file but not modify it,

Write someone can make a change to the file, but not read the contents,

eXecute someone can run the file, for example, a program or script, without loading it into another program first.

There are a couple of other letters you may see and hear, but R, W, and X will take care of what we need to do for now.

We can change the permissions of a file using the `chmod` command which changes the file mode bits on a file, and there are two (2) ways to do it.

1. use **octal notation**, which uses three values to represent read, write, and execute.
2. use **symbolic notation**, which uses a shorthand for user, group, others, and all, an operator, and a list of permissions to change.

We'll take a look at both, starting with the octal notation.

Information

The `chmod` Command

A shell command for changing access permissions and special mode flags of files (including special files such as directories). The name is short for change mode where mode refers to the permissions and flags collectively

Read	4	Write	2	Execute	1
------	---	-------	---	---------	---

Table 4.1: Octal Notation and their numerical meaning.

Octal Notation

If you ever have worked with Linux or macOS or any other UNIX based OS you may have seen commands like `chmod 777`, or `chmod 644`, and similar things. The way we arrive at those numbers is by assigning `read`, `write`, and `execute` each a different value. Which can be seen in **Table 4.1**.

This notation makes it easy to represent various states of these three (3) values with just a single digit. So if a user can read, write and execute, that comes out to seven (7), four plus two plus one (**4+2+1**). If the group can only read and execute, that comes out to five (5), four plus one (**4+1**).

With this system and a some basic maths, it's impossible to be ambiguous about the permissions the user, group or others have.

If you don't feel like doing the maths, you can make up a table like you in **Table 4.2**.

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111
Mode	---	--x	-w-	-wx	r--	r-x	rw-	rwx

Table 4.2: The value and their meaning using octal notation

To view the privileges a file has one simply has to write

Symbolic Notation

The symbolic way of representing permissions is a more approachable method to a lot of people, because instead of setting numbers for each value, you can add or remove a permission by letter. User is represented by the letter `u`, group by `g`, others by `o`, and changing all of the values is represented by `a`. If you leave off a prefix, `chmod` applies your change to all values. There are three operators here you can use:

Plus (+)	adds whichever permission you specify to what's already there
Minus (-)	removes whatever is there
Equals (=)	sign resets the permissions to only whatever value you specify

For example, to set user permissions to read, write and execute, we need to use:

```
1 chmod u+rwx
```

C.R. 23

bash

If we wanted to set group (**g**) permissions to only read (**r**), then we use:

```
1 chmod g=r
```

C.R. 24

bash

```
1 chmod u-rwx
```

C.R. 25

bash

We can line up the octal and symbolic values and see what the results are. In octal, **777** is the same as saying **a+rwx**. **755** is the same as saying **u+rwx, g=rx, o=rx**. You can see the symbolic notation is a bit longer, but it contains more information and context, so it's a little easier to work with. The nice thing about symbolic notation is that it's a little easier to make changes, since you're specifying what to change rather than what octal value to use. Using octal notation is kind of like using the = operator in symbolic notation all the time. Saying whatever was there before, now it's this value, rather than add read or remove execute.

4.7 Hard and Symbolic Links

It is time to look at a special kind of file on the Linux system called **link**. These are basically files that are **references** to other files, and they're used to avoid having multiple copies of the same file in different places.⁶ You keep one file in a well-known location and then add a little **pointer** or a **link** to other places you want that file to appear to be.

As you're learning about the CLI you may not have a need to create links, but it's important to know what they are when you come across them, and can show their usefulness as you are developing more complex applications.

⁶It is always in your best interest to minimise the number of copies a file has as the maintenance of all these files would not be possible after some time.

To put it simply, there are two (2) kinds of links:

hard links point to data on the disk

Soft or symbolic links point to a file on the disk.

It's kind of a subtle difference but it changes how the resulting links work. Let's take a look at soft links or symbolic links quickly.

4.7.1 Symbolic Links

We can create a symbolic link with the **ln** command and a **-s** option:

Information

The **ln** Command

Primarily used to create links for files in Linux, effectively allowing one file to reference another. Doing so allows you to manage files more efficiently without creating duplicates, making this command crucial for optimizing storage and managing files in Unix-like operating systems.

1. the name of the source file, (i.e., `novel.txt`)
2. the file we want to make a link to, (i.e., `writing.txt`)
3. the name of the link I want to create.

To create a link to `novel.txt` we create a file called `writing.txt` and link it. Now the `writing.txt` file is a link to the `novel.txt` file. If you were to Look at the contents of `writing.txt`, you would see the contents of the original file, and editing the `writing.txt` file means editing the original as well. Think of `writing.txt` not as a file, but a pointer⁷ to the original one.

⁷In this case it is very similar to that of a pointer in C as the main idea is the new symbolic link is pointing to the original file.

It's important to know that this kind of link is **relative**, that is if you move the link somewhere else on the file system the system won't be able to reference the original file any more and if you move the original file, the link will break as well, because the system will be told to look at a particular path for the linked file and it won't be there any more.

Hard Links

You can create a hard link by leaving off the dash `-s` option. If we write `ln text.txt`, this will create a hard link to `text.txt`. A hard link appears to be a regular file in a file listing but it's also just a pointer to the original file or more specifically it's a pointer to the data that the original file references.

One of its major advantage is hard links **can be moved around the file system and it doesn't matter if the original file is moved**, as a hard link points to the underlying data for a file instead of the file itself. In fact, every file on your system is a hard link to its underlying data.

Hard links and soft links both have their uses depending on the applications you have in mind.

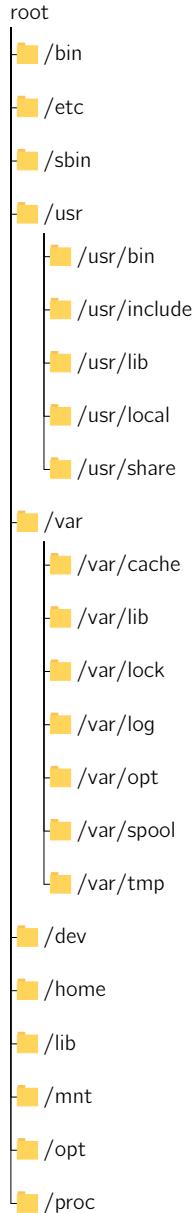
Below is a quick guide to the options the `ln` command has.

Command	Description
<code>--backup</code>	make a backup of each existing destination file
<code>-b</code>	like <code>--backup</code> but does not accept an argument
<code>-d</code>	allow the superuser to attempt to hard link directories (note: will probably fail due to system restrictions, even for the superuser)
<code>-f</code>	remove existing destination files
<code>-i</code>	prompt whether to remove destinations
<code>-L</code>	dereference TARGETs that are symbolic links
<code>-n</code>	treat LINK_NAME as a normal file if it is a symbolic link to a directory
<code>-P</code>	make hard links directly to symbolic links
<code>-r</code>	create symbolic links relative to link location
<code>-s</code>	make symbolic links instead of hard links
<code>-S</code>	override the usual backup suffix
<code>-t</code>	specify the DIRECTORY in which to create the links
<code>-T</code>	treat LINK_NAME as a normal file always
<code>-v</code>	print name of each linked file

4.8 The Linux File System

It makes sense to explore the Linux file system from a terminal window (i.e., CLI) as it has better tools to show the map of Linux's directory tree.

From top to bottom, the directories you are:⁸



- /bin contains binaries, that is, some of the applications and programs, such as `bash`, `cat`, `chmod` (For more information on binary files, please have a look [here](#).) You will find the `ls` program mentioned above in this directory, as well as other basic tools for making and removing files and directories, moving them around, etc.. There are more bin directories in other parts of the file system tree, but we'll be talking about those in a minute.
- /boot contains files required for starting the OS. Messing up one of the files here, may cause Linux to malfunction. Superuser privileges are needed to edit/change files here.
- /dev contains device files. Many of these are generated at boot time or even on the fly. For example, plugging a new webcam or a USB drive into the computer, will create a new device entry in this directory.
- /etc comes from the UNIX operating system, meaning "et cetera" (meaning "and other similar things") as it was a dumping ground for system files administrators were not sure where else to put. Nowadays, it would be more appropriate to say that etc stands for "Everything to configure", as it contains most, if not all system-wide configuration files. For example, the files that contain the name of your system, the users and their passwords, the names of machines on your network and when and where the partitions on your hard disks should be mounted are all in here.
- /lib stores libraries which are files containing code that applications use. They contain code snippets applications use to control peripherals, or send files to the hard disk for example. There are more `lib` directories scattered around the file system, but this one, the one hanging directly off of `/` is special in that, among other things, contains the all-important kernel modules. The kernel modules are drivers that make things like the video card, sound card, Wi-Fi, printer, etc.
- /home contains users' personal directories.
- /media where external storage will be automatically mounted when it is plugged in and being accessed. As opposed to most of the other items on this list, `/media` did not originate in 1970s, mainly because inserting and detecting storage (USB hard disks, SD cards, external SSDs, etc.) while a computer is running, is relatively new.
- /mnt where mount storage devices or partitions are manually mounted which is not used often nowadays.
- /opt here compiled programs (i.e., non-system) are stored. Applications will end up in the `/opt/bin`

⁸A Visual description of the linux file system

directory and libraries in the `/opt/lib` directory.

/proc virtual like `/dev`, contains information about the computer, such as information about the CPU and the kernel Linux is running on. As with `/dev`, the files and directories are generated when needed as the system is running and things change therefore don't save your documents here.

/root the home directory of the superuser (also known as the "Administrator") of the system. It is separate from the rest of the users' home directories and it is not meant to be tampered.

/run System processes use it to store temporary data for their own reasons. Similar to `/root` and `/boot`, it is best this folder is left alone.

/sbin similar to `/bin`, but contains applications only the superuser (hence the initial s) needs. Application here can be used with the `sudo` command. `/sbin` contains tools that can install stuff, delete stuff and format stuff.

/usr Originally where users' home directories were kept. However, now `/home` is where users kept their stuff as we saw above. These days, `/usr` contains a mish-mash of directories which in turn contains: applications, libraries, documentation, wallpapers, icons, and a long list of other stuff that need to be shared by applications and services. You will also find `/bin`, `/sbin` and `/lib` directories in `/usr`.

Information [/usr/bin v. /bin](#)

Not much nowadays. Originally, the `/bin` directory would contain basic commands, like `ls`, `mv` and `rm`; the bare minimum to run and maintain a system whereas `/usr/bin` would contain stuff the users would install and run to use the system as a work station, things like word processors, web browsers, and other apps. Many modern Linux distributions put everything into `/usr/bin` and have `/bin` point to `/usr/bin` just in case.

/srv contains data for servers. When running a web server, HTML files for sites would go into `/srv/http` (or `/srv/www`), or running an FTP (File Transfer Protocol) server, files would go into `/srv/ftp`.

/sys virtual directory like `/proc` and `/dev`, containing information from connected devices.

/tmp contains temporary files, usually placed there by running applications. The files and directories often (not always) contain data that an application doesn't need right now, but may need later on. `/tmp` also can store users' temporary files as it is one of the few directories hanging off `/` that can be used without superuser.

/var originally named because its contents was deemed variable, in that it changed frequently. Today it is a bit of a misnomer because there are many other directories that also contain data that changes frequently, especially the virtual directories. Be that as it may, `/var` contains things

like logs in the `/var/log` sub-directories. Logs are files that register events that happen on the system. If something fails in the kernel, it will be logged in a file in `/var/log`; If someone tries to break into the computer from outside, the firewall will also log the attempt here.

4.9 Common Command-Line Tools and Tasks

4.9.1 The UNIX Philosophy

Starting exploring command line tools, it's important to understand the principle behind many of the programs we'll be looking at. That principle, often called the [UNIX philosophy](#), originated by Ken Thompson, is a set of cultural norms and philosophical approaches to minimalist, modular software development.

Generally, these are:

- Small is beautiful,
- Make each program do one thing well,
- Build a prototype as soon as possible,
- Choose portability over efficiency,
- Store data in flat text files,
- Use software leverage to your advantage,
- Use shell scripts to increase leverage and portability,
- Avoid captive user interfaces,
- Make every program a filter.

In a nutshell, this philosophy emphasizes tools [shouldn't try to do too much](#). We don't want a tool which reads files and separates some of the text into another file and renames that file and compresses it into an archive when it's done, or tries to do everything that anyone can possibly want to do. While this may sound convenient, you have to consider there would be a lot of possible bugs and glitches of these sub-actions interacting with each other under this complex command. Therefore, we want one tool and one tool only to do each of those tasks, so we can use those specialised tools in any way we want to.

Of course, there are many applications that include many features and that's fine. However, those applications are beyond the scope of this lecture. We're talking about the standard set of command line tools that can be configured to work together in an incredible number of ways.

Jack of all trades, master of none is not encouraged in programming.

To get real work done, quality is needed; tools dedicated to a specific task working together easily.

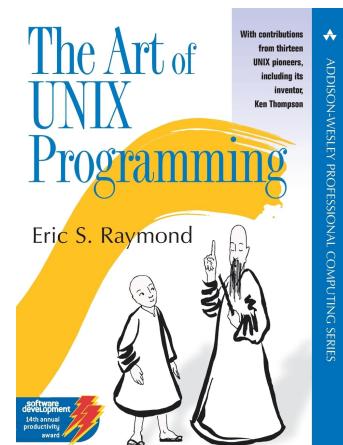


Figure 4.2: For anyone who is interested in the UNIX philosophy, I would suggest reading this book as it has parts written by numerous people who were the original developers of the UNIX.

Think of an assembly line where one machine does one task and then passes the product onto the next specialized machine, rather than one complicated robot doing many different tasks on the same item.

The point here is not that we have one multifunction generalist program. We want to be able to incorporate the right tools into doing a task as flexibly as possible, and as we'll see in a little bit, this philosophy underlies a lot of how you work at the command line.

You will use one program to read text from a file, then send it to a program that filters certain text. Then the output of that program gets processed, so that it doesn't have duplicate lines and then the result of that will get written back to a file.

Modularity and flexibility are features, not limitations, of working at the command line. However, just because programmers strive for simplicity in their programming, it doesn't mean they can't have Easter eggs in them. In the terminal just type in:

```
1 apt help
```

C.R. 26

bash

APT (Advanced Packaging Tools) is used to install updates and utilities and has Super Cow Powers.

```
1 apt 2.7.14 (arm64)
2 Usage: apt [options] command
3
4 apt is a commandline package manager and provides commands for
5 searching and managing as well as querying information about packages.
6 It provides the same functionality as the specialized APT tools,
7 like apt-get and apt-cache, but enables options more suitable for
8 interactive use by default.
9
10 Most used commands:
11   list - list packages based on package names
12   search - search in package descriptions
13   show - show package details
14   install - install packages
15   reinstall - reinstall packages
16   remove - remove packages
17   autoremove - automatically remove all unused packages
18   update - update list of available packages
19   upgrade - upgrade the system by installing/upgrading packages
20   full-upgrade - upgrade the system by removing/installing/upgrading packages
21   edit-sources - edit the source information file
22   satisfy - satisfy dependency strings
23
24 See apt(8) for more information about the available commands.
25 Configuration options and syntax is detailed in apt.conf(5).
26 Information about how to configure sources can be found in sources.list(5).
27 Package and version choices can be expressed via apt_preferences(5).
28 Security details are available in apt-secure(8).
29
```

This APT has Super Cow Powers.

4.9.2 Connecting Commands with Pipes

At the command line, we use pipes (`|`) to take the output of one command and send it to another. Think of commands as little processing nodes which do one particular thing and pipes as connections between those nodes. Searching on the internet should give you a good idea of where to find it on your keyboard depending on the type, if you need to. We type this character in between commands that we want to be *piped* together. Throughout the course, put a space on either side of it so it's easier to see, but it doesn't need to have spaces. Take a look at using pipes at the command line. To do this, we need to know a few more commands. The first is `echo`, which prints out whatever you give it. For example, write `echo "hello"`, and that works as promised.

```
1 echo "hello"                                C.R. 27
                                         bash
1 hello                                         text
```

Now, write that command again and this time add a pipe character to send the output to the command `wc` for word count. And here, instead of the output from `echo`, we see the output of the `wc` program responding to the input from the `echo` command.

```
1 echo "hello" | wc                           C.R. 28
                                         bash
1 1      1      6                            text
```

What `wc` is telling here is that there is one line of text, one word, and six characters. To change the output type `echo "hello, world! from the command-line interface"` and pipe that to `wc`. That's one line, six words, 45 characters.

```
1 echo "hello, world! from the command-line interface" | wc
                                         C.R. 29
                                         bash
1 1      6      46                            text
```

As can be seen the sentence contain 45 characters but 46 is printed as `wc` counts an invisible character at the end of the string called a new line in addition to the characters we sent. A command can be piped to any other command, and usually it'll do something whether it is useful or not.

4.9.3 Viewing Text Files with `cat`, `head`, `tail`, and `less`

The majority of tasks we will be working with at the command line will involve text files or text output. Therefore, it's important to know the following commands to check out the contents of text files. The first is called `cat`, stands for **concatenate**.

Information**The cat Command**

A standard utility which reads files sequentially, writing them to standard output.

Basically when programmers talk about concatenation, they mean sticking two or more things together, and `cat` can do exactly do just that. But more often than not, it is used to print the contents of a file to the screen. It's also helpful to get the contents of a text file into a series of piped commands. Depending on the operating system, there will be different files available to you. Normally, as an administrator we tend to use `cat` to look at a log file or something similar. But here, for the sake of simplicity, we will use a simple text file inside the Linux Tutorial Folder. The `Poem.txt` file. For the curious, the poem is "*Stopping by Woods on a Snowy Evening*" by Robert Frost ([a](#)).

4.10 Advanced Topics

4.10.1 Find Linux Distribution and Kernel Information

Up until now, almost everything we've done has been distribution independent.

That is, it hasn't mattered if you're running CentOS, Fedora, Ubuntu, or another distribution of Linux. But it's good to know what environment you're working with, in case you need to make some changes to the system or to install software. If you find yourself in an environment that you don't know about, it's pretty easy to figure out what distribution you're using. This information is kept in files inside the /etc folder. What it's called specifically varies by distro, but we can use a wildcard to target the names of these files and see what's inside them. First, let's take a look at what these files are.

Let's write the following

```
1 ls -lah /etc/*release                                C.R. 30
2
3
4 -rw-r--r-- 1 root root 104 Feb  5 16:08 /etc/lsb-release      bash
5 lrwxrwxrwx 1 root root  21 Feb  5 16:08 /etc/os-release -> ../usr/lib/os-release    text
```

In my case, I have two (2) files here, lsb-release and os-release, which is a link to another file in /usr/lib.

Let's see what information's in there.

To do that, we'll type cat /etc/*release.

```
1 cat /etc/*release                                C.R. 31
2
3
4 DISTRO_ID=Ubuntu
5 DISTRO_RELEASE=24.04
6 DISTRO_CODENAME=noble
7 DISTRO_DESCRIPTION="Ubuntu 24.04.2 LTS"
8 PRETTY_NAME="Ubuntu 24.04.2 LTS"
9 NAME="Ubuntu"
10 VERSION_ID="24.04"
11 VERSION="24.04.2 LTS (Noble Numbat)"
12 VERSION_CODENAME=noble
13 ID=ubuntu
14 ID_LIKE=debian
15 HOME_URL="https://www.ubuntu.com/"
16 SUPPORT_URL="https://help.ubuntu.com/"
17 BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
18 PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
```

```
16 UBUNTU_CODENAME=noble          text  
17 LOGO=ubuntu-logo
```

Which lists the contents of all of the files in the /etc folder that end with the word release.

On different distributions, there'll be different numbers and names of files that match this wildcard, but they'll contain the information we need.

Here I can see that I'm using Ubuntu, version 20.04 LTS (Long Term Service), Focal Fossa.

On other systems, we'd see slightly different information here.

Another important piece of information to know about a system is what version of the kernel you're using.

You can find that information with the uname command, Using the dash a (-a) option to show all the information.

```
1  uname -a                         C.R. 32  
                                bash  
  
1  Linux 807fe6353460 6.10.14-linuxkit #1 SMP Fri Nov 29 17:22:03 UTC 2024 aarch64 aarch64 text  
→  aarch64 GNU/Linux
```

This shows the type of system, the host name, the version of the kernel, when it was built, the architecture of the system and so on.

This kind of information can be helpful if you're troubleshooting something.

Again, if you've set up a system, chances are good you know what kind of software it's running.

But if for some reason you don't, now we've seen how to figure it out.

4.10.2 Find System Hardware and Disk Information

It is important to finding out some information about the system you're working with. If you're using a physical computer, or a virtual machine you set up for yourself, you have some knowledge about the hardware it has, like how much RAM (Random Access Memory) it has, what kind of CPU (Central Processing Unit) it has, and how much hard drive space there is. But if you're working on a remote system or you are an administrator of a machine you have yet to have working knowledge of, it can be helpful to get a sense of what your resources are and what hardware system has.

First, let's find out how much RAM this machine has. To do this, use the `free` command with the `-h` option, which gives us values in human readable numbers.

```
1 free -h
C.R. 33
bash

1      total        used        free       shared  buff/cache   available
2 Mem:    7.7Gi     479Mi     7.1Gi     1.5Mi     294Mi    7.2Gi
3 Swap:   1.0Gi        0B     1.0Gi
```

Here, under total memory, we can see that this machine has two gigabytes of memory. Next, let's take a look at what our processor resources are. There is a file in the /proc directory called cpuinfo, so let's take a look at that. To access this information write `cat /proc/cpuinfo`.

```
1 cat /proc/cpuinfo | head -8
C.R. 34
bash

1 processor      : 0
2 BogoMIPS       : 48.00
3 Features       : fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid
   ↳ asimdrdm jscvt fcma lrcpc dcpop sha3 asimddp sha512 asimdfhm dit uscat ilrcpc flagm
   ↳ ssbs sb paca pacg dcopdp flagm2 fint
4 CPU implementer : 0x61
5 CPU architecture: 8
6 CPU variant    : 0x0
7 CPU part       : 0x000
8 CPU revision   : 0
```

There is a lot of information here. Scroll up a little bit and I can see that I'm using an Intel Xeon processor at 3.5 gigahertz (GHz). And under CPU cores, I can see that this machine has four (4) CPU core. I can also find out how much space is taken up and how much is available on the system's hard drive. For that, use the df command with the -h option, again, to show human readable sizes.

```
1 df -h
C.R. 35
bash

1 Filesystem      Size  Used Avail Use% Mounted on
2 overlay         59G   32G   24G  58% /
3 tmpfs           64M    0    64M  0% /dev
4 shm              512M   0   512M  0% /dev/shm
5 /dev/vda1        59G   32G   24G  58% /etc/hosts
6 /run/host_mark/Users  461G  338G  124G  74% /home/ubuntu/Desktop/Host
7 tmpfs            3.9G   0    3.9G  0% /proc/scsi
8 tmpfs            3.9G   0    3.9G  0% /sys/firmware
```

This shows space across a few different volumes, but the most interesting one to me is slash (/) or root, since that's where my user data is, and it's where you are likely to be taking up space if I installed software. The rest of these are managed by the system, so you don't need to worry about those. You can also use the du command to see how much space files and folders take up on your system. Let's have a look at how much space is taken up across my whole system. I'll write sudo du / -hd1 I have to use sudo, because my user can't see into all of the folders at the root of the drive.

Then there is the du command for disk usage, and then slash (/), which is the level I want to start from, right at the root. The dash h (-h) option gives me sizes in human readable formats, kilobytes, megabytes, gigabytes, and so on, and the d option shows the du command what level of detail to show. In this case, I'm giving it the argument of one (1), meaning just show me one level d, the first level away from the root, adding everything up within each of those folders. Let's take a look

Chapter 5

Installation

Table of Contents

5.1 ROS 2 Humble Hawksbill	115
5.2 Auto-Install Script	119

5.1 ROS 2 Humble Hawksbill

5.1.1 Introduction

This section is written and edited for students who wish to install ROS on their own computers which has Linux¹ on it. For student who will use Docker containers, you can skip this chapter.

Before we tackle the wonderful world of robot programming, we must first install all the essential tools we need to make sure it works. Below are the instructions on how to do it.

Please read the instructions first and then apply the commands as required by the instructions. This is not just a one-time warning and should be carefully noted as Linux while powerful can be unforgiving if one does **NOT** take good care and attention to what they type to the terminal.

¹Of course, it is assumed the student has Ubuntu 22.04 installed. If any other system is installed such as Arch, or NixOs, then I would like to use the following text printed out when a user uninstalls their boot-loader from their computer: "Bailing out, you are on your own. Good Luck."

5.1.2 Setting up the Local

Make sure you have a locale which supports UTF-8.² If you are in a minimal environment (such as a docker container), the locale may be something minimal like Portable Operating System Interface

²A character encoding standard used for electronic communication. Defined by the [Unicode Standard](#), the name is derived from Unicode Transformation Format - 8-bit. Almost every web page is stored in UTF-8.

(POSIX). We test with the following settings. However, it should be fine if you're using a different UTF-8 supported locale.

To start with installation, if you are using a GUI open up your terminal (**Ctrl** + **Alt** + **T**).

```
1  locale # check for UTF-8                                C.R. 1  
2  
3  sudo apt update && sudo apt install locales           bash  
4  sudo locale-gen en_US en_US.UTF-8  
5  sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8  
6  export LANG=en_US.UTF-8  
7  
8  locale # verify settings
```

Information

The POSIX Standard

POSIX is a set of standard OS interfaces based on the Unix operating system. The most recent POSIX specifications IEEE Std 1003.1-2024 defines a standard interface and environment that can be used by an ³such as the shell. OS to provide access to POSIX-compliant applications. The standard also defines a command interpreter³ and common utility programs. POSIX supports application portability at the source code level so applications can be built to run on any POSIX-compliant OS.

5.1.3 Setting Up the Source Files

Before we can begin working with ROS, we must install all the necessary files and dependencies required. For these lectures we will install ROS [Humble Hawksbill](#) which is currently available for [Ubuntu Jammy](#) (22.04 LTS).

While currently there are more up-to-date version of ROS available such as Jammy, due to its long term support and established compatibility, we will be using ROS Humble.

It is **heavily** recommended to be on Ubuntu 22.04 LTS as ROS Humble is only officially supported on this version. It is possible to compile ROS on other Ubuntu or Linux distributions, however, you need to compile the binaries yourself.

We will need to add the ROS `apt` repository to your system.

Information

The Advanced Package Manager

A free-software user interface that works with core libraries to handle the installation and removal of software on Debian and Debian-based Linux distributions. It is just another package manager.

First ensure that the Ubuntu Universe⁴ repository is enabled.

```
1 sudo apt install software-properties-common
2 sudo add-apt-repository universe
```

C.R. 2
bash

⁴A snapshot of the free, open-source, and Linux world. It houses almost every piece of open-source software, all built from a range of public sources

Now add the ROS GPG key with apt.

Information

GPG

A hybrid-encryption software which uses a combination of conventional symmetric-key cryptography for speed, and public-key cryptography for ease of secure key exchange, typically by using the recipient's public key to encrypt a session key which is used only once.

```
1 sudo apt update && sudo apt install curl -y
2 sudo curl -sSL \
3   https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
4   -o /usr/share/keyrings/ros-archive-keyring.gpg
```

C.R. 3
bash

Then add the repository to your [sources list](#). This list contains all the paths your system will scan when it is looking for a package or to look for updates.

```
1 echo "deb [arch=$(dpkg --print-architecture) \
2   signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
3   http://packages.ros.org/ros2/ubuntu \
4   $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | \
5   sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

C.R. 4
bash

Information

The echo Command

A command that outputs the strings that are passed to it as arguments. It is a useful command to let the user know what process is going on with the current running program or allows the insertion of text to other files.

5.1.4 Install ROS 2 Packages

Update your [apt](#) repository caches after setting up the repositories as it may **NOT** have the latest version on it. The ROS packages are built on frequently updated Ubuntu systems.

It is always recommended that you ensure your system is up to date before installing new packages.

```
1 sudo apt update && sudo apt upgrade
```

C.R. 5
bash

Once updated you have a few options on which type to install (1st is recommended)

Desktop Install It install all the necessary components and software packages such as ROS, RViz, demos, tutorials.

```
sudo apt install ros-humble-desktop
```

Development tools Compilers and other tools to build ROS packages.

```
sudo apt install ros-dev-tools
```

This setup may take some time as the file download size is around 2 GB.

5.1.5 Setting Up the Environment

Once the software is installed we still need to tell our shell where the software is so we can call it in our session. For this we need to **source** a script. Set up your environment by sourcing the following file in your terminal ([Ctrl] + [Alt] + [T]).

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

C.R. 6
bash

Alternatively, if you prefer to automate this action, simply type the following code in your terminal:

```
1 echo "source /opt/ros/humble/setup.bash" >> "~/.bashrc"
```

C.R. 7
bash

The aforementioned command simply writes the command to the last line of a document called **.bashrc** which is a script that runs whenever you open a new terminal window.

The **.bashrc** file is a script file that's executed when a user logs in. The file itself contains a series of configurations for the terminal session. This includes setting up or enabling: colouring, completion, shell history, command aliases, and more.

It is a hidden file and simple **ls** command won't show the file.

5.2 Auto-Install Script

As a bonus, if you have read the document before doing copy and pasting, you can use the `rosinstall.sh` provided to you to automatically install everything required for this course.

```

1 #!/bin/bash                                         C.R. 8
2
3 # First check your Ubuntu Version
4 # For maximum compatibility with ROS it needs to be 22.04 LTS
5
6 # Creating log for troubleshooting
7 echo "##### BEGIN ATTEMPT #####" >install_log.txt
8
9 echo "Welcome to ROS 2 Automated Installation"
10 echo ""
11 echo ""
12
13 # Accessing the Ubuntu version using AWK and piping it to grep for Regex
14 version=$(awk "/VERSION_ID/" IGNORECASE=1 /etc/*release |
15     grep -Eo "[[:digit:]]+([.][[:digit:]])?"
16 )
17
18 # Checks version for ROS Compliance
19 if [[ "${version}" == *"22.04"* ]]; then
20     echo "Version is supported."
21     sleep 1
22     echo "Continuing installation..."
23     sleep 1
24 else
25     echo "Your version: ${version}, What is needed: 22.04"
26     echo "I am sorry but your version is not supported."
27     echo "This install script will terminate"
28     exit
29
30 fi
31
32 echo ""
33 echo "Installing UTF-8 Compliance ..."
34
35 {
36     locale # check for UTF-8
37
38     sudo apt update
39     sudo apt install locales
40     sudo locale-gen en_US en_US.UTF-8
41     sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
42     export LANG=en_US.UTF-8
43
44     locale # verify settings
45 } &>install_log.txt
46
47

```

```
48 echo ""
49 echo "Enabling Ubuntu Universe Repositories..."
50
51 {
52     sudo apt install software-properties-common
53     echo | sudo add-apt-repository universe
54 } &>install_log.txt
55
56 echo ""
57 echo "Adding ROS 2 GPG Keys ..."
58
59 {
60     sudo apt update
61     sudo apt install curl -y
62     sudo curl -sSL \
63         https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
64         -o /usr/share/keyrings/ros-archive-keyring.gpg
65 } &>install_log.txt
66
67 echo ""
68 echo "Adding ROS 2 to repository ..."
69
70 {
71     echo "deb [arch=$(dpkg --print-architecture) \
72 signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
73 http://packages.ros.org/ros2/ubuntu \
74 $(. /etc/os-release && echo $UBUNTU_CODENAME) main" \
75     | sudo tee /etc/apt/sources.list.d/ros2.list >/dev/null
76 } &>install_log.txt
77
78 echo ""
79 echo "Getting Updates ..."
80
81 {
82     sudo apt update
83     echo yes | sudo apt upgrade
84 } &>install_log.txt
85
86 echo ""
87 echo "Installing ROS ..."
88
89 {
90     echo yes | sudo apt install ros-humble-desktop
91     yes | sudo apt install ros-dev-tools
92 } &>install_log.txt
93
94 {
95     sudo apt install dbus-x11
96 } &>install_log.txt
97
98 echo ""
99 echo "Sourcing ROS file ..."
100 sleep 1
```

```
101
102 echo "source /opt/ros/humble/setup.bash" >~/.bashrc
103
104 echo ""
105 echo "Removing unnecessary files ..."
106 sleep 1
107 {
108     yes | sudo apt autoremove
109 } &>install_log.txt
```

C.R. 10

bash

Chapter 6

ROS Concepts

Table of Contents

6.1	Introduction	123
6.2	Publisher and Subscriber Architecture	124
6.3	Nodes - The Building Blocks	125
6.4	The Discovery Process	126
6.5	Communication Between Nodes	127
6.6	Topics	133
6.7	Services	135
6.8	Actions	137
6.9	Parameters	139
6.10	Working with Command Line	142
6.11	Launch File	144

6.1 Introduction

Before diving into the vast world of ROS, it is worth looking at some concepts and ideas which are used. Most of the concept we will cover will be relatively high-level and therefore are here to give you more of an idea of the cogs and gears working in the system.

In a nutshell, ROS is a middle-ware¹ software based on a **strongly-typed, anonymous publish/-subscribe** mechanism which allows for message passing between different processes.

At the heart of any ROS system is the ROS graph where the ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate.

¹a software that facilitates communication and data management between applications, systems, and databases.

6.2 Publisher and Subscriber Architecture

A common design pattern in used [concurrency programming](#) is the [producer-consumer architecture](#), where;

- One or more threads or processes act as a producer, where it adds elements to some shared data structure,
- One or more other threads act as a consumer where it removes items from that structure and does something with them.

To demonstrate, imagine the producer is preparing coffee for the customers. This makes the customers, consumers. As the barista makes coffee, it creates a [queue](#) of coffees waiting to be consumed by the customers. Queues operate on a simple principle called a First In First Out (). This means items are removed in the same order that they're put into the queue. The first item added, in this case coffee, will be the first item to be removed. So when a customer wants to consume another cup of coffee, they'll grab one from the end of the line because it's been in the queue the longest. The coffee represent elements of data for the consumer thread to process or perhaps packaged tasks for the consumer to execute.

An important factor to consider is the need of a protection to make sure the producer won't try to add data to the queue when it's full and the consumer won't try to remove data from an empty buffer. Some programming languages may include implementations of a queue that's considered thread-safe and handles all of these challenges under the hood, so we don't have to, but some languages does not include that support. Therefore we need to use the combination of a mutex (mutual exclusion) and condition variables to implement your own thread-safe, synchronised queue.

We may run into scenarios where the producer cannot be paused if the queue fills up, for example, when it is an external source of streaming data that we can't slow down, so it's important to consider the rate at which items are produced and consumed from the queue. If the consumer can't keep up with production, then we face a [buffer overflow](#), and we'll lose data. Some programming languages offer implementations of unbounded queues, which are implemented using linked lists to have an advertised unlimited capacity. But keep in mind, even those will be limited by the amount of physical memory in the computer. The rate at which the producer is adding items may not always be consistent. For example, in network applications, data might arrive in bursts of network packets, which fill the queue quickly. But if those bursts occur infrequently, the consumer has time to catch up between bursts.

We should consider the average rate at which items are produced and consumed as we want the average rate of production to be less than the average rate of consumption. But if more steps were required to process this data, then we could expand our simple producer-consumer setup into a pipeline of tasks. A pipeline consists of a chain of processing elements arranged so that the output of each element is the input to the next one. It's basically a series of producer-consumer pairs connected together with some sort of buffer, like a queue, between each consecutive element.

6.3 Nodes - The Building Blocks

A node is an actor in the overall ROS graph, which uses a client library² to communicate with other nodes.

²This could either be `rospack` or `rosdep` depending on the chosen language.

Nodes can also communicate with other nodes within the same process, in a different process, or on a different machine and are considered the **unit of computation** in a ROS graph as each node should do one logical thing.

Nodes can either **publish** to named topics to deliver data to other nodes, or **subscribe** to named topics to get data from other nodes. They can also act as a service client to have another node perform a computation on their behalf, or as a service server to provide functionality to other nodes.

For long-running calculations, a node can act as an action client to have another node perform it on their behalf, or as an action server to provide functionality to other nodes. Nodes can provide configurable parameters to change behaviour during run-time.

Nodes are often a complex combination of publishers, subscribers, service servers, service clients, action servers, and action clients, all at the same time.

An advantage of using nodes is that it allows **language agnostic programming**. It means we can write one node in Python, another node in C++, and both can communicate without any problem. In addition they provide a **great fault tolerance**. As nodes only communicate through ROS, they are not directly linked. If one node crashes, it will not make the other nodes crash.

For all these actions to work, a node needs to know where others are. Therefore, to achieve this we use a method called a **distributed discovery process**.

6.4 The Discovery Process

Discovery of nodes happens **automatically** through the underlying middle-ware of ROS. The sequence of operations can be summarised as follows:

1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain.³ Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.
2. Nodes **periodically** announce their presence so connections can be made with new-found entities, even after the initial discovery period.
3. Nodes advertise to other nodes when they go offline.

⁴In this context it means the description or measurement of the overall performance of a service, such as a telephony or computer network, or a cloud computing service, particularly the performance seen by the users of the network.¹

Nodes will only establish connections with others having compatible Quality of Service.⁴

As an example to see what's going on let's used the built-in example of **talker-listener**. As the name implies, one node advertises the presence and the other listens.

1 . ~/ros2_humble/install/local_setup.bash
2 ros2 run demo_nodes_cpp talker

C.R. 1
bash

1 [INFO] [1748930700.789182596] [talker]: Publishing: 'Hello World: 1'
2 [INFO] [1748930701.785043679] [talker]: Publishing: 'Hello World: 2'
3 [INFO] [1748930702.785231597] [talker]: Publishing: 'Hello World: 3'
4 [INFO] [1748930703.785897764] [talker]: Publishing: 'Hello World: 4'

text

Now it is running in one, open another terminal window, and type:

1 . ~/ros2_humble/install/local_setup.bash
2 ros2 run demo_nodes_py listener

C.R. 2
bash

1 [INFO] [1748930759.815469762] [listener]: I heard: [Hello World: 60]
2 [INFO] [1748930760.787999512] [listener]: I heard: [Hello World: 61]
3 [INFO] [1748930761.787663179] [listener]: I heard: [Hello World: 62]
4 [INFO] [1748930762.787127847] [listener]: I heard: [Hello World: 63]

text

Running the C++ talker node in one terminal will **publish** messages on a topic, and the Python listener node running in another terminal will **subscribe** to messages on the same topic.

We should see that these nodes discover each other automatically, and begin to exchange messages.

6.5 Communication Between Nodes

6.5.1 Description

ROS applications generally communicate through interfaces of one of three (3) types:

1. topics,
2. services,
3. actions.

ROS uses a simplified description language⁵ to describe these interfaces. This description allows simplification for ROS tools to automatically generate source code for the interface type in several target languages.⁶ In this document we will describe the supported types:

⁵interface definition language.

⁶i.e., Python, C++

msg simple text files describing the fields of a ROS message. They are used to generate source code for messages in different languages.

srv describes a service and composed of two (2) parts:

- a request, and
- a response.

The request and response are **message declarations**.

action describes actions and composed of three (3) parts:

- a goal,
- a result, and
- a feedback.

Each part is a **message declaration itself**.

6.5.2 Messages

Messages are a way for a ROS node to **send data** on the network to other ROS nodes, **without expecting any responses**.

As an example, let's say a ROS node reads temperature data from a sensor. It can then **publish** that data on the ROS network using a **Temperature** message. Other nodes on the ROS network can **subscribe** to that data and receive the **Temperature** message.

Messages are described and defined in `.msg` files in the `msg/` directory of a ROS package.

`.msg` files are composed of two (2) parts:

Fields

Information about a data type and its name

Constants

Similar to other languages, define an `unchangeable` variable.

Fields

Each field consists of a **type** and a **name**, separated by a space. The syntax for this would be the following:

```

1  fieldtype1 fieldname1
2  fieldtype2 fieldname2
3  fieldtype3 fieldname3

```

C.R. 3
text

For example, lets say we want to define a 32-bit integer and a string variable. For that we would have to write the following:

```

1  int32 my_int
2  string my_string

```

C.R. 4
text

Field Types As with most statically-typed languages we need to define what type of variables we are working. In ROS, the fields type can be one of two (2) things:

- A built-in type,
- names of Message descriptions defined on their own, such as `geometry_msgs/PoseStamped`

The following is a table of all currently built-in variable types.

Table 6.1: Current types supported by ROS Humble.

Type name	C++	Python	DDS type
<code>bool</code>	<code>bool</code>	<code>builtins.bool</code>	<code>boolean</code>
<code>byte</code>	<code>uint8_t</code>	<code>builtins.bytes*</code>	<code>octet</code>
<code>char</code>	<code>char</code>	<code>builtins.int*</code>	<code>char</code>
<code>float32</code>	<code>float</code>	<code>builtins.float*</code>	<code>float</code>

Continued on next page

Table 6.1: Current types supported by ROS Humble. (Continued)

Type name	C++	Python	DDS type
float64	double	builtins.float*	double
int8	int8_t	builtins.int*	octet
uint8	uint8_t	builtins.int*	octet
int16	int16_t	builtins.int*	short
uint16	uint16_t	builtins.int*	unsigned short
int32	int32_t	builtins.int*	long
uint32	uint32_t	builtins.int*	unsigned long
int64	int64_t	builtins.int*	long long
uint64	uint64_t	builtins.int*	unsigned long long
string	std::string	builtins.str	string
wstring	std::u16string	builtins.str	wstring

In addition, array-like feature are also supported

Type name	C++	Python	DDS type
static array	std::array<T, N>	builtins.list*	T[N]
unbounded dynamic array	std::vector	builtins.list	sequence
bounded dynamic array	custom_class<T, N>	builtins.list*	sequence<T, N>
bounded string	std::string	builtins.str*	string

All types which are more permissive than their ROS definition enforce the ROS constraints in range and length by software.

Example of message definition using arrays and bounded types:

```

1 int32[] unbounded_integer_array
2 int32[5] five_integers_array
3 int32[<=5] up_to_five_integers_array
4
5 string string_of_unbounded_size
6 string<=10 up_to_ten_characters_string
7
8 string[<=5] up_to_five_unbounded_strings
9 string<=10[] unbounded_array_of_strings_up_to_ten_characters_each
10 string<=10[<=5] up_to_five_strings_up_to_ten_characters_each

```

C.R. 5

text

Naming Conventions There are some restrictions in how these variables are named and therefore are worth of mention. Field names **must be lowercase alphanumeric characters with underscores for separating words**. They must start with an alphabetic character, and they must **NOT** end with an underscore or have two (2) consecutive underscores.

⁷types NOT present in the built-in-types which applies to all nested messages.

Default Values Default values can be set to any field in the message type. Currently default values are **NOT** supported for string arrays and complex types.⁷

Defining a default value is done by adding a **third** element to the field definition line. For example:

```
1 fieldtype fieldname fielddefaultvalue
```

C.R. 6
text

An example implementation would be:

```
1 uint8 x 42
2 int16 y -2000
3 string full_name "John Doe"
4 int32[] samples [-200, -100, 0, 100, 200]
```

C.R. 7
text

For example, in the first line we define a 8-bit unsigned integer (**uint8**), call it **x**, and give it a default value of 42.

String values must be defined in ' or " quotes and currently string values are **NOT** escaped.

Constants Each constant definition is like a field description with a **default** value, except that this **value can never be changed programmatically**. This value assignment is indicated by use of an equal (=) sign. The syntax is:

```
1 constanttype CONSTANTNAME=constantvalue
```

C.R. 8
text

For example:

```
1 int32 X=123
2 int32 Y=-123
3 string FOO="foo"
4 string EXAMPLE='bar'
```

C.R. 9
text

Constants names have to be **UPPERCASE**.

Services

Services are a request/response communication, where the client (requester) is waiting for the server (responder) to make a short computation and return a result. Services are described and defined in `.srv` files in the `srv` directory of a ROS package.

A service description file consists of a request and a response msg type, separated by `---`. Any two `.msg` files concatenated with a `---` are a legal service description. Here is a very simple example of a service that takes in a string and returns a string:

```
1 uint32 request
2 ---
3 uint32 response
```

C.R. 10
text

We can of course get much more complicated.⁸

```
1 uint32 a
2 uint32 b
3 ---
4 uint32 sum
```

C.R. 11
text

⁸if we want to refer to a message from the same package we must **NOT** mention the package name.

We cannot embed another service inside of a service.

Actions

Actions are a long-running request/response communication, where the action client (**requester**) is waiting for the action server (**the responder**) to take some action and return a result. In contrast to services, actions can be long-running,⁹ provide feedback while they are happening, and can be interrupted.

⁹This could be many seconds up to minutes.

Action definitions have the following form:

```
1 <request_type> <request_fieldname>
2 ---
3 <response_type> <response_fieldname>
4 ---
5 <feedback_type> <feedback_fieldname>
```

C.R. 12
text

Similar to services, the request fields are before and the response fields are after the first triple-dash (`---`), respectively. There is also a third set of fields after the second triple-dash, which is the fields to be sent when sending feedback. There can be:

- arbitrary numbers of request fields (including zero),
- arbitrary numbers of response fields (including zero), and

- arbitrary numbers of feedback fields (including zero).

The `<request_type>`, `<response_type>`, and `<feedback_type>` follow all of the same rules as the `<type>` for a message.

The `<request_fieldname>`, `<response_fieldname>`, and `<feedback_fieldname>` follow all of the same rules as the `<fieldname>` for a message.

As an example, the [Fibonacci](#) action definition contains the following:

```
1 int32 order
2 ---
3 int32[] sequence
4 ---
5 int32[] sequence
```

C.R. 13
text

This is an action definition where the action client is sending a single `int32` field representing the number of Fibonacci steps to take, and expecting the action server to produce an array of `int32` containing the complete steps. Along the way, the action server may also provide an intermediate array of `int32` contains the steps accomplished up until a certain point.

6.6 Topics

Topics are one of the three (3) primary styles of interfaces provided by ROS. Topics **should be used for continuous data streams**, like sensor data, robot state, etc. As stated earlier, ROS is a **strongly-typed, anonymous** publish/subscribe system. Let's break down that sentence and explain it a bit more.

6.6.1 Publisher - Subscriber Architecture

A publish/subscribe system is one in which there are:

1. producers of data (publishers),
2. consumers of data (subscribers).

The publishers and subscribers know how to contact each other through the concept of a **topic**, which is a common name so that the entities can find each other.

When we create a publisher, we must also give it a string which is the name of the topic and the same goes for the subscriber.

Any publishers and subscribers that are on the same topic name can **directly communicate with each other**. There may be zero or more publishers and zero or more subscribers on any particular topic. When data is published to the topic by any of the publishers, all subscribers in the system will receive the data.

This system is also known as a **bus** as it resembles a bus bar from used in power engineering. This concept of a bus is part of what makes ROS a powerful and flexible system. Publishers and subscribers can come and go as needed, meaning that debugging and introspection are natural extensions to the system.

As an example, if we want to record data, we can use the `ros2 bag record` command. Under the hood, `ros2 bag record` creates a new subscriber to whatever topic we tell it, without interrupting the flow of data to the other parts of the system.

6.6.2 Anonymity

Another fact mentioned in the introduction is that ROS is **anonymous**. This means that when a subscriber gets a piece of data, it doesn't generally know or care which publisher originally sent it.¹⁰ The benefit to this architecture is that publishers and subscribers can be swapped out at will without affecting the rest of the system.

¹⁰If needed though, it can find out.

6.6.3 Strongly-Typed

As a last point, the publish/subscribe system is **strongly-typed**. That has two (2) meanings in this context:

- The types of each field in a ROS message are typed, and that type is **enforced** at various levels.
- For instance, if the ROS message contains:

```
1  uint32 field1  
2  string field2
```

C.R. 14
text

- Then the code will ensure the **field1** is always an **unsigned integer** and **field2** is always a **string**.
- The semantics of each field are **well-defined**.
 - There is no automated mechanism to ensure this, but all of the core ROS types have strong semantics associated with them.
 - For instance, the IMU message contains a 3-dimensional vector for the measured angular velocity, and each of the dimensions is specified to be in radians/second. Other interpretations should **NOT** be placed into the message.

6.7 Services

In ROS, a service refers to a remote procedure call. In other words,

a node can make a remote procedure call to another node which will do a computation and return a result.

This structure is reflected in how a service message definition looks:

```

1 uint32 request
2 ---
3 uint32 response

```

C.R. 15

text

In ROS, services are expected to return quickly, as the client is generally waiting on the result. **Services should never be used for longer running processes**, in particular processes that might need to be pre-empted for exceptional situations.

If we have a service that will be doing a long-running computation, consider using an **action**.

Services are identified by a service name, which looks much like a topic name.¹¹ A service consists of two (2) parts:

- the service server,
- the service client.

¹¹but is in a different namespace.

6.7.1 Service Server

A service server is the entity that will accept a remote procedure request, and perform some computation on it. For instance, suppose the ROS message contains the following:

```

1 uint32 a
2 uint32 b
3 ---
4 uint32 sum

```

C.R. 16

text

The service server would be the entity that receives this message, adds **a** and **b** together, and returns the **sum**.

There should only ever be one (1) service server per service name. It is undefined which service server will receive client requests in the case of multiple service servers on the same service name.

Service Client

A service client is an entity that will request a remote service server to perform a computation on its behalf. Following the previous example, the service client is the entity that creates the initial message containing `a` and `b`, and waits for the service server to compute the `sum` and return the result.

Unlike service server, there can be a number of service clients using the same service name.

6.8 Actions

In ROS, an action refers to a [long-running remote procedure call with feedback](#) and the ability to cancel or pre-empt the goal. As an example, the high-level state machine running a robot may call an action to tell the navigation subsystem to travel to a way point, which may take several seconds, or even minutes to do. Along the way, the navigation subsystem can provide feedback on how far along it is, and the high-level state machine has the option to cancel or preempt the travel to that way point.

This structure is reflected in how an action message definition looks:

```

1 <request_type> <request_fieldname>          C.R. 17
2 ---                                         text
3 <response_type> <response_fieldname>
4 --- 
5 <feedback_type> <feedback_fieldname>
```

In ROS, actions are expected to be long running procedures, as there is overhead in setting up and monitoring the connection.

If we need a short running remote procedure call, consider using a service instead.

Actions are identified by an [action name](#), which looks much like a topic name (but is in a different namespace) and consists of two parts:

1. the action server,
2. the action client.

6.8.1 Action Server

The action server is the entity that will accept the remote procedure request and perform some procedure on it. It is also responsible for sending out feedback as the action progresses and should react to cancellation/preemption requests.

For instance, consider an action to calculate the Fibonacci sequence with the following interface we looked at previously:

```

1 int32 order          C.R. 18
2 ---                                         text
3 int32[] sequence
4 --- 
5 int32[] sequence
```

The action server is the entity that receives this message, starts calculating the sequence up to order (providing feedback along the way), and finally returns a full result in sequence.

There should only ever be one action server per action name. It is undefined which action server will receive client requests in the case of multiple action servers on the same action name.

6.8.2 Action Client

An action client is an entity that will **request** a remote action server to perform a procedure on its behalf. Following the previous example, the action client is the entity that creates the initial message containing the order, and waits for the action server to compute the sequence and return it (with feedback along the way).

Unlike action server, there can be a number of action clients using the same action name.

6.9 Parameters

Parameters in ROS are associated with [individual nodes](#). Parameters are used to configure nodes at startup, and during runtime [without changing the code](#). The lifetime of a parameter is tied to the lifetime of the node.¹² Parameters are addressed by **node name**, **node namespace**, **parameter name**, and **parameter namespace**.

¹²though the node could implement some sort of persistence to reload values after restart.

Providing a parameter namespace is optional.

Each parameter consists of a **key**, a **value**, and a **descriptor**. The key is a string and the value is one of the following types:

`bool, int64, float64, string, byte[], bool[], int64[], float64[] or string[]`.

By default all descriptors are empty, but can contain parameter descriptions, value ranges, type information, and additional constraints.

6.9.1 A Detailed Look

Declaring Parameters

By default, a node needs to declare all of the parameters that it will accept during its lifetime.

We do this so the type and name of the parameters are well-defined at node startup time, which reduces the chances of misconfiguration later on.

For some types of nodes, [not all of the parameters will be known ahead of time](#). In these cases, the node can be instantiated with `allow_undeclared_parameters` set to `true`, which will allow parameters to be get and set on the node even if they haven't been declared.

Types of Parameters

Each parameter on a ROS node has one of the pre-defined parameter types as mentioned.

By default, any attempts to change the type of a declared parameter at runtime [will fail](#). This prevents common mistakes, such as putting a boolean value into an integer parameter.

If a parameter needs to be multiple different types, and the code using the parameter can handle it, this default behaviour can be changed. When the parameter is declared, it should be declared using a `ParameterDescriptor` with the `dynamic_typing` member variable set to `true`.

Parameter Callbacks

A ROS node can register two (2) different types of callbacks to be informed when changes are happening to parameters. Both of the callbacks are **optional**.

1. The first is known as a **set parameter** callback, and can be set by calling from the node Application Programming Interface (API),¹³ `add_on_set_parameters_callback`. The callback is passed a list of immutable **Parameter** objects, and returns an `rcl_interfaces>msg>SetParametersResult`.

The primary purpose of **set parameter** callback is to give the user the ability to inspect the upcoming change to the parameter and explicitly reject the change.

It is important to make sure that **set parameter** callbacks have no side-effects. Since multiple set parameter callbacks can be chained, there is no way for an individual callback to know if a later callback will reject the update. If the individual callback were to make changes to the class it is in, for instance, it may get out-of-sync with the actual parameter. To get a callback after a parameter has been successfully changed, we may need to look into the lower option.

2. The second type of callback is known as an **on parameter event** callback, and can be set by calling `on_parameter_event` from the parameter client APIs. The callback is passed an `rcl_interfaces>msg>ParameterEvent` object, and returns nothing. This callback will be called after all parameters in the input event have been declared, changed, or deleted.

The primary purpose of **on parameter event** callback is to give the user the ability to react to changes from parameters that have successfully been accepted.

6.9.2 Parameter Interaction

ROS 2 nodes can perform parameter operations through node APIs. External processes can perform parameter operations via parameter services that are created by default when a node is instantiated.

The services that are created by default are:

`node_name>describe_parameters`: Uses service type `rcl_interfaces>srv>DescribeParameters`. Given a list of parameter names, returns a list of descriptors associated with the parameters.

`node_name>get_parameter_types`: Uses service type `rcl_interfaces>srv>GetParameterTypes`. Given a list of parameter names, returns a list of parameter types associated with the parameters.

`node_name>get_parameters`: Uses service type `rcl_interfaces>srv>GetParameters`. Given a list of parameter names, returns a list of parameter values associated with the parameters.

`node_name >> list_parameters`: Uses service type `rcl_interfaces >> srv >> ListParameters`. Given an optional list of parameter prefixes, returns a list of the available parameters with that prefix. If the prefixes are empty, returns all parameters.

`node_name >> set_parameters`: Uses service type `rcl_interfaces >> srv >> SetParameters`. Given a list of parameter names and values, attempts to set the parameters on the node. Returns a list of results from trying to set each parameter; some of them may have succeeded and some may have failed.

`node_name >> set_parameters_atomically`: Uses service type `rcl_interfaces >> srv >> SetParametersAtomically`. Given a list of parameter names and values, attempts to set the parameters on the node. Returns a single result from trying to set all parameters, so if one failed, all of them failed.

Some additional information regarding parameters are as follows:

Setting Initial Parameters Values when a Node is Running Initial parameter values can be set when running the node either through individual command-line arguments, or through YAML files.

Information

The .yaml Format

A human-readable data serialization language. It is commonly used for configuration files and in applications where data is being stored or transmitted. Stands for *YAML Ain't Markup Language*.

Setting Initial Parameters Values when a Node is Launching Initial parameter values can also be set when running the node through the ROS 2 launch facility.

Manipulating parameter values at runtime The `ros2 param` command is the general way to interact with parameters for nodes that are already running. `ros2 param` uses the parameter service API as described above to perform the various operations.

6.10 Working with Command Line

ROS 2 includes a suite of command-line tools for introspecting a ROS 2 system.

The main entry point for the tools is the command `ros2`, which itself has various sub-commands for introspecting and working with nodes, topics, services, and more.

To see all available sub-commands run:

```
1 ros2 --help                                C.R. 19
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

usage: ros2 [-h] [--use-python-default-buffering] Call `ros2 <command> -h` for more text
→ detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

options:

- h, --help show this help message and exit
- use-python-default-buffering
 - Do not force line buffering in stdout and instead use the python
 - default buffering, which might be affected by
 - PYTHONUNBUFFERED/-u and depends on whatever stdout is
 - interactive or
 - not

Commands:

- action Various action related sub-commands
- bag Various rosbag related sub-commands
- component Various component related sub-commands
- daemon Various daemon related sub-commands
- doctor Check ROS setup and other potential issues
- interface Show information about ROS interfaces
- launch Run a launch file
- lifecycle Various lifecycle related sub-commands
- multicast Various multicast related sub-commands
- node Various node related sub-commands
- param Various param related sub-commands
- pkg Various package related sub-commands
- run Run a package specific executable
- security Various security related sub-commands
- service Various service related sub-commands
- topic Various topic related sub-commands
- wtf Use `wtf` as alias to `doctor`

Call `ros2 <command> -h` for more detailed usage.

ROS 2 uses a distributed discovery process for nodes to connect to each other. As this process purposefully does not use a centralized discovery mechanism, it can take time for ROS nodes to discover all other participants in the ROS graph. Because of this, there is a

long-running daemon in the background that stores information about the ROS graph to provide faster responses to queries, e.g. the list of node names.

The daemon is automatically started when the relevant command-line tools are used for the first time. We can run `ros2 daemon --help` for more options for interacting with the daemon.

6.11 Launch File

A ROS 2 system typically consists of many nodes running across many different processes (and even different machines). While it is possible to run each of these nodes separately, it gets cumbersome quite quickly.

The launch system in ROS 2 is meant to automate the running of many nodes with a single command. It helps the user describe the configuration of their system and then executes it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them, and ROS-specific conventions which make it easy to reuse components throughout the system by giving them each a different configuration. It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

All of the above is specified in a launch file, which can be written in Python, XML, or YAML. This launch file can then be run using the `ros2 launch` command, and all of the nodes specified will be run.

Chapter 7

Command Line Tools

Table of Contents

7.1	Setting the Environment	145
7.2	Turtles and Graphs	148
7.3	A Deeper Look into Nodes	153
7.4	Working with Topics	156
7.5	Working with Services	161
7.6	Working with Parameters	165
7.7	A Practical Look into Actions	169
7.8	Launching Nodes	173

7.1 Setting the Environment

ROS relies on the notion of combining workspaces using the shell environment.¹ The “workspace” is a ROS term for the location on our system where we’re developing with ROS.

¹In this case, of course, it is our bash shell.

- the core ROS workspace is called the underlay,
- and any subsequent local workspaces are called overlays.

When developing with ROS, we will typically have several workspaces active concurrently.

Combining workspaces makes developing against different versions of ROS, or against different sets of packages, easier. It also allows the installation of several ROS distributions² on the same computer and switching between them. This is accomplished by sourcing setup files every time we open a new shell, or by adding the source command to our shell startup script once. Without sourcing the setup files, we won’t be able to access ROS commands, or find or use ROS packages.

²or “distros”, e.g. Dashing and Eloquent.

In other words, we won't be able to use ROS. To source, simply type the following:

```
1 # Replace ".bash" with your shell if you're not using bash
2 # Possible values are: setup.bash, setup.sh, setup.zsh
3 source /opt/ros/humble/setup.bash
```

C.R. 1
bash

The exact command depends on where we installed ROS. Of course, if we have installed ROS using a docker container, this section can be skipped.

Sourcing the Script If we don't want to have to source the setup file every time we open a new shell, then we can add the command to our shell startup script:³

³For our case it is .bashrc

```
1 # Replace ".bash" with your shell if you're not using bash
2 echo "source /opt/ros/humble/setup.bash" » ~/.bashrc
```

C.R. 2
bash

Checking Environment Variables Sourcing ROS setup files will set several environment variables necessary for operating ROS. If we ever have problems finding or using our ROS packages, make sure that our environment is properly set up using the following command:

```
1 printenv | grep -i ROS
```

C.R. 3
bash

Information

The `printenv` Command

Displays the values of environment variables.

Using this command let's check all variables like `ROS_DISTRO` and `ROS_VERSION` are set.

```
1 ROS_VERSION=2
2 ROS_PYTHON_VERSION=3
3 AMENT_PREFIX_PATH=/opt/ros/humble
4 PYTHONPATH=/opt/ros/humble/lib/python3.10/site-packages:/opt/ros/humble/local/lib/
   ↳ python3.10/dist-packages
5 LD_LIBRARY_PATH=/opt/ros/humble/opt/rviz_ogre_vendor/lib:/opt/ros/humble/lib/aarch64-
   ↳ linux-gnu:/opt/ros/humble/lib
6 ROS_LOCALHOST_ONLY=0
7 PATH=/opt/ros/humble/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
8 ROS_DISTRO=humble
```

text

If the environment variables are **NOT** set correctly, it might be worthwhile to reinstall ROS.

The ROS Domain ID Variable Once we have determined a **unique integer** for our group of ROS nodes, we can set the environment variable with the following command:

```
1 export ROS_DOMAIN_ID=<your_domain_id>
```

C.R. 4

bash

To maintain this setting between shell sessions, we can also add the command to our shell startup script `.bashrc`:

```
1 echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bashrc
```

C.R. 5

bash

Changing the Localhost Variable By default, ROS communication is **NOT** limited to localhost.⁴

`ROS_LOCALHOST_ONLY` environment variable allows us to limit ROS communication to localhost only.

This means our ROS system, and its topics, services, and actions will **NOT** be visible to other computers on the local network.

Using `ROS_LOCALHOST_ONLY` is helpful in certain settings, such as classrooms, where multiple robots may publish to the same topic causing strange behaviors. We can set the environment variable with the following command:

```
1 export ROS_LOCALHOST_ONLY=1
```

C.R. 6

bash

Of course, to maintain it across different shell session we can write this command to our `.bashrc`:

```
1 echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

C.R. 7

bash

⁴Localhost is a form of hostname, meaning the specific computer that the program is running on. It is employed as a method of connecting to services on the network on the physical host machine without the services of an outer network. When using "localhost", we are connected to our computer or the node that is addressed by the IP address 127. Often used for testing and development, it lets developers run and test Web applications, or any other network service, locally before putting them on a live server. To summarize, it can be mentioned that localhost is the loopback network interface.

7.2 Turtles and Graphs

Turtlesim is a great introduction to ROS as it is a lightweight simulator and easy to work on. It illustrates what ROS does at the most basic level to give us an idea of what we will do with a real robot or a robot simulation later on.

The `ros2` tool is how the user manages, introspects, and interacts with a ROS system. It supports multiple commands which target different aspects of the system and its operation.

One might use it to start a node, set a parameter, listen to a topic, and many more.

The `ros2` tool is part of the core ROS installation and should already be installed. This can easily be tested by typing `ros2` into a terminal.

The second tool is `rqt`, a GUI tool for ROS.

Everything done in `rqt` can be done using CLI, but `rqt` provides a more user-friendly way to manipulate ROS elements.

We will work together to understand the fundamental concept which build the core of ROS, like nodes, topics, and services. All of these concepts will be elaborated on later.

For now, we will simply set up the tools and get a feel for them.

Installing the Required Packages In case of **NOT** having a configured docker container which was discussed in detail previously, please use the following commands to install all the necessary packages.

```
1 sudo apt update  
2 sudo apt install ros-humble-turtlesim
```

C.R. 8
bash

```
1 0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
```

text

To check if the package is installed, run the following command in the terminal:

```
1 ros2 pkg executables turtlesim
```

C.R. 9
bash

which should return a list of turtlesim's executable options:

```
1 turtlesim draw_square  
2 turtlesim mimic  
3 turtlesim turtle_teleop_key  
4 turtlesim turtlesim_node
```

text

Installing and Starting Turtlesim Let's begin the tutorial by starting turtlesim:

```
1 ros2 run turtlesim turtlesim_node
```

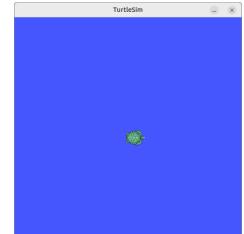
C.R. 10

bash

Under the command, we will see messages from the node. There we can see the default turtle's name and the coordinates where it spawns.

```
1 QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-ubuntu'          text
2 [INFO] [1748967100.900464593] [turtlesim]: Starting turtlesim with node name /turtlesim
3 [INFO] [1748967100.908647551] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445],
   ~ y=[5.544445], theta=[0.000000]
```

The simulator window should appear, with a random turtle in the centre.⁵



Making the Turtle Move Around Now, open a new terminal and if it is NOT automatic, source ROS again. Now we will run a new node to control the turtle in the first node:

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 11

bash

```
1 Reading from keyboard
2 -----
3 Use arrow keys to move the turtle.
4 Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
5 'Q' to quit.
```

⁵Here we can see the turtlebot in its natural habitat, the turtlesim environment. The type of the turtle changes every time a new turtlesim environment is called. All these turtles represent a version of ROS.

At this point we should have three (3) windows open:

- a terminal running `turtlesim_node`,
- a terminal running `turtle_teleop_key`, and
- the turtlesim window itself.

Let's arrange these windows so we can see the turtlesim window, but also have the terminal running `turtle_teleop_key` active so that we can control the turtle in turtlesim using the arrow keys on the keyboard. It will move around the screen, using its attached "pen" to draw the path it followed so far.

Pressing an arrow key will only cause the turtle to move a short distance and then stop. This is because, realistically, we wouldn't want a robot to continue carrying on an instruction if, for example, the operator lost the connection to the robot.

We can see the nodes, and their associated topics, services, and actions, using the list sub-commands of the respective commands:

```

1 echo "~~ Node information ~~" &&
2   ros2 node list &&
3   echo "~~ Topic information ~~" &&
4   ros2 topic list &&
5   echo "~~ Service information ~~" &&
6   ros2 service list &&
7   echo "~~ Action information ~~" &&
8   ros2 action list
9
10
11 ~~ Node information ~~
12 /teleop_turtle
13 /turtlesim
14 ~~ Topic information ~~
15 /parameter_events
16 /rosout
17 /turtle1/cmd_vel
18 /turtle1/color_sensor
19 /turtle1/pose
20 ~~ Service information ~~
21 /clear
22 /kill
23 /reset
24 /spawn
25 /teleop_turtle/describe_parameters
26 /teleop_turtle/get_parameter_types
27 /teleop_turtle/get_parameters
28 /teleop_turtle/list_parameters
29 /teleop_turtle/set_parameters
30 /teleop_turtle/set_parameters_atomically
31 /turtle1/set_pen
32 /turtle1/teleport_absolute
33 /turtle1/teleport_relative
34 /turtlesim/describe_parameters
35 /turtlesim/get_parameter_types
36 /turtlesim/get_parameters
37 /turtlesim/list_parameters
38 /turtlesim/set_parameters
39 /turtlesim/set_parameters_atomically
40 ~~ Action information ~~
41 /turtle1/rotate_absolute

```

Don't worry about what this all means as we will have a detailed look into each of them in a later part.

Running rqt `rqt` is a GUI framework which implements various tools and interfaces in the form of plugins. If it is not installed on our system please run the following commands:

```

1 sudo apt update
2 sudo apt install '~nros-humble-rqt*'

```

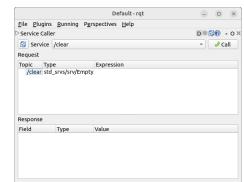
Once installed, running it is pretty straightforward:

```
1 rqt
```

C.R. 14
bash

When running `rqt` for the first time, the window will be blank. This is normal. To see what is currently going on just select `Plugins > Services > Service Caller` from the menu bar at the top.⁶

Use the refresh button to the left of the Service dropdown list to ensure all the services of our turtlesim node are available. Once refreshed, click on the Service dropdown list to see services belonging to `turtlesim`, and select the `/spawn` service.



⁶The view of `rqt` with only turtlesim and teleop running.

Working the the spawn service Let's use `rqt` to call the `/spawn` service. We can guess from its name that `/spawn` will create another turtle in the turtlesim window.

Give the new turtle a unique name, like `turtle2`, by double-clicking between the empty single quotes in the Expression column. We can see that this expression corresponds to the value of name and is of type string.

Next enter some valid coordinates at which to spawn the new turtle, like `x = 1.0` and `y = 1.0`.

If we try to spawn a new turtle with the same name as an existing turtle, like the default `turtle1`, we will get an error message in the terminal running `turtlesim_node`.

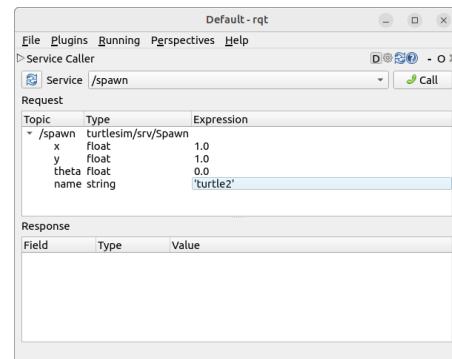
To spawn `turtle2`, we then need to call the service by clicking the Call button on the upper right side of the `rqt` window.

If the service call was successful, we should see a new turtle, again with a random design. spawn at the coordinates we input for `x` and `y`.

If we refresh the service list in `rqt`, we will also see that now there are services related to the new turtle, `turtle2`, in addition to `turtle1`.

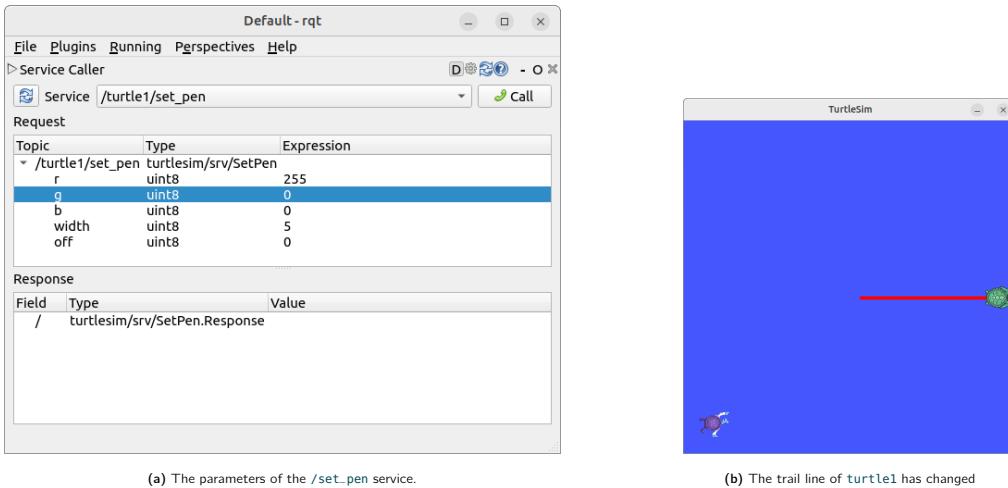
Changing the Trail Parameters Now let's give `turtle1` a unique pen using the `/set_pen` service:

The values for `r`, `g`, and `b`, which are between 0 and 255, set the colour of the pen `turtle1` draws with, and width sets the thickness of the line.



To have `turtle1` draw with a distinct red line, change the value of `r` to 255, and the value of width to 5.

Don't forget to call the service after updating the values.



If we were to return to the terminal where `turtle_teleop_key` is running and press the arrow keys, we will see `turtle1`'s pen has changed and also noticed that there's no way to move `turtle2`. That's because there is no teleop node for `turtle2`.

Remapping Controls We need a second teleop node in order to control `turtle2`. However, if we try to run the same command as before, we will notice that this one also controls `turtle1`. The way to change this behavior is by remapping the `cmd_vel` topic.

To do that, In a new terminal, source ROS, and run:

```
1 ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

C.R. 15

bash

Now, we can move `turtle2` when this terminal is active, and `turtle1` when the other terminal running `turtle_teleop_key` is active.

7.3 A Deeper Look into Nodes

Each node in ROS should be responsible for a **single, modular purpose**.⁷ Each node can send and receive data from other nodes using:

- topics,
- services,
- actions, or
- parameters.

⁷e.g., controlling the wheel motors or publishing the sensor data from a laser range-finder.

A full robotic system is comprised of many interconnected nodes working in unison. In ROS, for example, a single executable⁸ can contain one or more nodes.

⁸C++ program, Python program, etc.

ros2 run Now let's put all this knowledge into motion. The command `ros2 run` launches an executable from a package.

```
1 ros2 run <package_name> <executable_name>
```

C.R. 16

bash

To run turtlesim, open a new terminal (`Ctrl` + `Alt` + `T`), and enter the following command:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 17

bash

Here, the package name is turtlesim and the executable name is `turtlesim_node`. We still don't know the node name, however.

ros2 node list We can find node names by using `ros2 node list`, which will show us the names of all running nodes. This is especially useful when we want to interact with a node, or when we have a system running many nodes and need to keep track of them.

Open a new terminal while turtlesim is **still running in the other one**, and enter the following command. The terminal will return the node name:

```
1 ros2 node list
```

C.R. 18

bash

```
1 /turtlesim
```

text

Open another new terminal and start the teleop node with the command:

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 19

bash

Here, we are referring to the turtlesim package again, but this time we target the executable named `turtle_teleop_key`.

Return to the terminal where we ran `ros2 node list` and run it again. We will now see the names of two active nodes:

```
1 ros2 node list                                C.R. 20  
2  
1 /turtlesim                                         bash  
2 /teleop_turtle                                     text
```

Remapping a Node Remapping allows us to reassign default node properties:

like node name, topic names, service names, etc., to custom values.

Previously, we have used remapping on `turtle_teleop_key` to change the `cmd_vel` topic and target `turtle2`.

Now, let's reassign the name of our `/turtlesim` node. In a new terminal, run the following command:

```
1 ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle      C.R. 21  
2  
3
```

Since we're calling `ros2 run` on turtlesim again, another turtlesim window will open. However, now if we return to the terminal where we ran `ros2 node list`, and run it again, we will see three (3) node names:

```
1 /my_turtle                                         text  
2 /turtlesim                                         text  
3 /teleop_turtle                                     text
```

ros2 node info Now that we know the names of our nodes, we can access more information about them with:

```
1 ros2 node info <node_name>                      C.R. 22  
2  
3
```

To examine our latest node, `my_turtle`, run the following command:

```
1 ros2 node info /my_turtle                         C.R. 23  
2  
3
```

```

1 /my_turtle                                         text
2   Subscribers:
3     /parameter_events: rcl_interfaces/msg/ParameterEvent
4     /turtle1/cmd_vel: geometry_msgs/msg/Twist
5   Publishers:
6     /parameter_events: rcl_interfaces/msg/ParameterEvent
7     /rosout: rcl_interfaces/msg/Log
8     /turtle1/color_sensor: turtlesim/msg/Color
9     /turtle1/pose: turtlesim/msg/Pose
10  Service Servers:
11    /clear: std_srvs/srv/Empty
12    /kill: turtlesim/srv/Kill
13    /my_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
14    /my_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
15    /my_turtle/get_parameters: rcl_interfaces/srv/GetParameters
16    /my_turtle/list_parameters: rcl_interfaces/srv/ListParameters
17    /my_turtle/set_parameters: rcl_interfaces/srv/SetParameters
18    /my_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
19    /reset: std_srvs/srv/Empty
20    /spawn: turtlesim/srv/Spawn
21    /turtle1/set_pen: turtlesim/srv/SetPen
22    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
23    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
24  Service Clients:
25
26  Action Servers:
27    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
28  Action Clients:
```

`ros2 node info` returns a list of subscribers, publishers, services, and actions.⁹

Now try running the same command on the `/teleop_turtle` node, and see how its connections differ from `my_turtle`.

⁹i.e., the ROS graph connections that interact with that node.

7.4 Working with Topics

ROS breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics.

Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

To begin we start with a fresh new terminal (`[Ctrl] + [Alt] + [T]`) and type the following command:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 24
bash

And of course we would like to control this turtle so we need to add the `teleop` node as well.

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 25
bash

Graphing the Topics Throughout this tutorial, we will use `rqt_graph` to visualize the changing nodes and topics, as well as the connections between them.

The turtlesim tutorial tells us how to install `rqt` and all its plugins, including `rqt_graph`.

To run `rqt_graph`, open a new terminal and enter the command:

```
1 rqt_graph
```

C.R. 26
bash

You should see the above nodes and topic, as well as two (2) actions around the periphery of the graph. If we hover your mouse over the topic in the centre, we'll see the color highlighting.

The graph is depicting how the `/turtlesim` node and the `/teleop_turtle` node are communicating with each other over a topic. The `/teleop_turtle` node is publishing data¹⁰ to the `/turtle1/cmd_vel` topic, and the `/turtlesim` node is subscribed to that topic to receive the data.

The highlighting feature of `rqt_graph` is very helpful when examining more complex systems with many nodes and topics connected in many different ways.

¹⁰the keystrokes you enter to move the turtle around

ros2 topic list Running the `ros2 topic list` command in a new terminal will return a list of all the topics currently active in the system:

```
1 ros2 topic list
```

C.R. 27

bash

```
1 /parameter_events
2 /rosout
3 /turtle1/cmd_vel
4 /turtle1/color_sensor
5 /turtle1/pose
```

text

Alternatively, running `ros2 topic list -t` will return the same list of topics, this time with the topic type appended in brackets:

```
1 ros2 topic list -t
```

C.R. 28

bash

```
1 /parameter_events [rcl_interfaces/msg/ParameterEvent]
2 /rosout [rcl_interfaces/msg/Log]
3 /turtle1/cmd_vel [geometry_msgs/msg/Twist]
4 /turtle1/color_sensor [turtlesim/msg/Color]
5 /turtle1/pose [turtlesim/msg/Pose]
```

text

These attributes, particularly the type, are how nodes know they're talking about the same information as it moves over topics.

If you're wondering where all these topics are in `rqt_graph`, you can uncheck all the boxes under Hide:

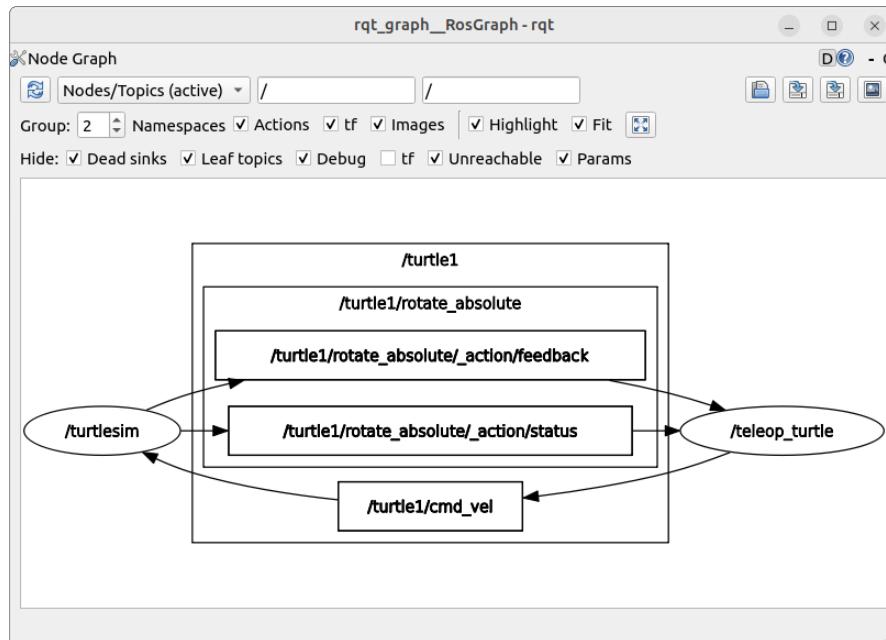


Figure 7.3: A visual representation of the communication happening between the nodes `/turtlesim` and `/teleop_turtle`.

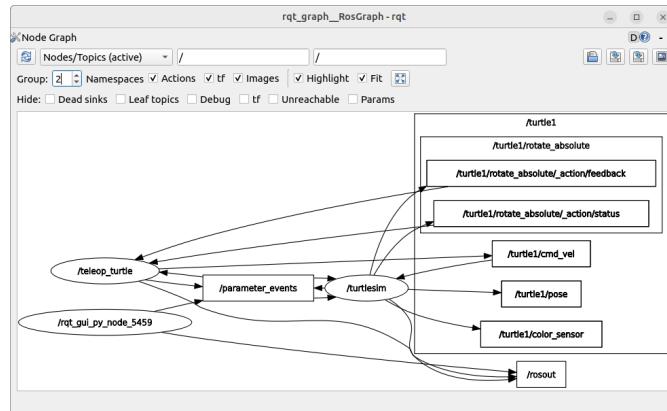


Figure 7.4

ros2 topic echo To see the data being published on a topic, use:

```
1 ros2 topic echo <topic_name>
```

C.R. 29
bash

Since we know that `/teleop_turtle` publishes data to `/turtlesim` over the `/turtle1/cmd_vel` topic, let's use echo to introspect that topic:

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

C.R. 30
bash

At first, this command won't return any data. That's because it's waiting for `/teleop_turtle` to publish something.

Return to the terminal where `turtle_teleop_key` is running and use the arrows to move the turtle around. Watch the terminal where your echo is running at the same time, and you'll see position data being published for every movement you make:

```
1 linear:
2   x: 2.0
3   y: 0.0
4   z: 0.0
5 angular:
6   x: 0.0
7   y: 0.0
8   z: 0.0
9   ---
```

text

Now return to `rqt_graph` and uncheck the Debug box.

`/_ros2cli_5707` is the node created by the echo command we just ran (the number might be different). Now you can see that the publisher is publishing data over the `cmd_vel` topic, and two (2) subscribers are subscribed to it.

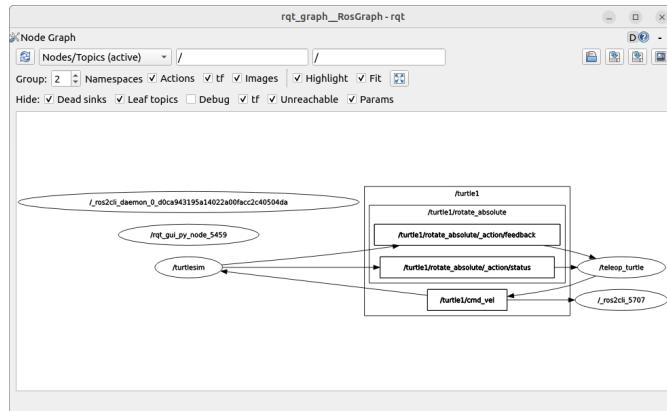


Figure 7.5

ros2 topic info Topics don't have to only be one-to-one communication; they can be one-to-many, many-to-one, or many-to-many.

Another way to look at this is running:

```
1 ros2 topic info /turtle1/cmd_vel                                C.R. 31
                                                               bash

1 Type: geometry_msgs/msg/Twist                                 text
2 Publisher count: 1
3 Subscription count: 2
```

ros2 interface show Nodes send data over topics using messages. Publishers and subscribers must send and receive the same type of message to communicate.

The topic types we saw earlier after running `ros2 topic list -t` let us know what message type is used on each topic. Recall that the `cmd_vel` topic has the type:

```
1 geometry_msgs/msg/Twist                                     C.R. 32
                                                               bash
```

This means that in the package `geometry_msgs` there is a msg called Twist.

Now we can run `ros2 interface show <msg_type>` on this type to learn its details. Specifically, what structure of data the message expects.

```
1 ros2 interface show geometry_msgs/msg/Twist                  C.R. 33
                                                               bash
```

Which will return

```
1 # This expresses velocity in free space broken into its linear and angular parts.      text
2   Vector3 linear
3     float64 x
4     float64 y
5     float64 z
6   Vector3 angular
7     float64 x
8     float64 y
9     float64 z
```

This tells you that the /turtlesim node is expecting a message with two vectors, linear and angular, of three elements each. If you recall the data we saw `/teleop_turtle` passing to /turtlesim with the echo command, it's in the same structure:

```
1 linear:                                         text
2   x: 2.0
3   y: 0.0
4   z: 0.0
5 angular:
6   x: 0.0
7   y: 0.0
8   z: 0.0
9   ---
```

7.5 Working with Services

As a brief review, services are yet another method of communication for nodes in the ROS graph. Services are based on a call-and-response model versus the publisher-subscriber model of topics. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.

To begin we start with a fresh new terminal (**[Ctrl] + [Alt] + [T]**) and type the following command to start the `/turtlesim` and `/teleop_turtle` nodes:

```
1 ros2 run turtlesim turtlesim_node
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 34

bash

C.R. 35

bash

ros2 service list Running the `ros2 service list` command in a new terminal will return a list of all the services currently active in the system:

```
1 ros2 service list
1 /clear
2 /kill
3 /reset
4 /spawn
5 /teleop_turtle/describe_parameters
6 /teleop_turtle/get_parameter_types
7 /teleop_turtle/get_parameters
8 /teleop_turtle/list_parameters
9 /teleop_turtle/set_parameters
10 /teleop_turtle/set_parameters_atomically
11 /turtle1/set_pen
12 /turtle1/teleport_absolute
13 /turtle1/teleport_relative
14 /turtlesim/describe_parameters
15 /turtlesim/get_parameter_types
16 /turtlesim/get_parameters
17 /turtlesim/list_parameters
18 /turtlesim/set_parameters
19 /turtlesim/set_parameters_atomically
```

C.R. 36

bash

text

Looking at this output we will see both nodes have the same six (6) services with parameters in their names. Nearly every node in ROS has these infrastructure services that parameters are built off of.

For now, let's focus on the turtlesim-specific services:

■ `/clear,`

- `/kill`
- `/reset`
- `/spawn`,
- `/turtle1/set_pen`,
- `/turtle1/teleport_absolute`, and
- `/turtle1/teleport_relative`.

We have interacted some of these services using `rqt` previously.

ros2 service type Services have types which describe how the `request` and `response` data of a service is structured. Service types are defined similarly to topic types, except service types have two (2) parts:

- one message for request,
- another one for response.

To find out the type of a service, use the command:

```
1 ros2 service type <service_name>
```

C.R. 37
bash

Let's take a look at turtlesim's `/clear` service. In a brand new terminal, let's enter the command:

```
1 ros2 service type /clear
```

C.R. 38
bash

```
1 std_srvs/srv/Empty
```

text

In this context, the `Empty` type means the service call sends no data when making a request and receives no data when receiving a response.

ros2 service list -t To see the types of all the active services at the same time, we can append the `--show-types` option, abbreviated as `-t`, to the list command:

```
1 ros2 service list -t
```

C.R. 39
bash

```
1 /clear [std_srvs/srv/Empty]
2 /kill [turtlesim/srv/Kill]
3 /reset [std_srvs/srv/Empty]
```

text

```

4 /spawn [turtlesim/srv/Spawn]
5 ...
6 /turtle1/set_pen [turtlesim/srv/SetPen]
7 /turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
8 /turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
9 ...

```

text

ros2 service find If we want to find all the services of a specific type, we can use the command, which the syntax is as follows:

```

1 ros2 service find <type_name>

```

C.R. 40

bash

For example, we can find all the `Empty` typed services like this:

```

1 ros2 service find std_srvs/srv/Empty

```

C.R. 41

bash

```

1 /clear
2 /reset

```

text

ros2 interface show We can call services from the command line, but first we need to know the structure of the input arguments. The syntax of the interface command is as follows:

```

1 ros2 interface show <type_name>

```

C.R. 42

bash

Try this on the `/clear` service's type, `Empty`:

```

1 ros2 interface show std_srvs/srv/Empty

```

C.R. 43

bash

```

1 ---

```

text

The `---` separates the request structure (above) from the response structure (below). But, as we learned earlier, the `Empty` type doesn't send or receive any data. So, naturally, its structure is blank.

Let's introspect a service with a type that sends and receives data, like `/spawn`. From the results of `ros2 service list -t`, we know `/spawn`'s type is `turtlesim/srv/Spawn`.

To see the request and response arguments of the `/spawn` service, run the command:

```

1 ros2 interface show turtlesim/srv/Spawn

```

C.R. 44

bash

```
1 float32 x                                         text
2 float32 y
3 float32 theta
4 string name # Optional. A unique name will be created and returned if this is empty
5 ---
6 string name
```

The information above the `---` line tells us the arguments needed to call `/spawn`. `x`, `y` and `theta` determine the 2D pose of the spawned turtle, and `name` is clearly optional.

The information below the line isn't something we need to know in this case, but it can help us understand the data type of the response we get from the call.

ros2 service call Now that we know what a service type is, how to find a service's type, and how to find the structure of that type's arguments, we can call a service using:

```
1 ros2 service call <service_name> <service_type> <arguments>
```

C.R. 45
bash

The `<arguments>` part is optional. For example, we know that `Empty` typed services don't have any arguments:

```
1 ros2 service call /clear std_srvs/srv/Empty
```

C.R. 46
bash

This command will clear the turtlesim window of any lines our turtle has drawn. Now let's spawn a new turtle by calling `/spawn` and setting arguments. Input `<arguments>` in a service call from the command-line need to be in YAML syntax.

Enter the command:

```
1 ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```

C.R. 47
bash

```
1 requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='')
```

2

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

We will get this method-style view of what's happening, and then the service response.

Our turtlesim window will update with the newly spawned turtle right away:

7.6 Working with Parameters

To give a brief refreshment, a parameter is a [configuration value](#) of a node. Think of parameters as node settings or its configuration file. A node can store parameters as integers, floats, booleans, strings, and lists.

In ROS, each node maintains its own parameters.

To begin we start with a fresh new terminal ([\[Ctrl\]](#) + [\[Alt\]](#) + [\[T\]](#)) and type the following command:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 48

bash

And of course we would like to control this turtle so we need to add the [teleop](#) node as well.

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 49

bash

ros2 param list Now that both of them are running [at the same time](#), lets look at all the parameters in this system

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 50

bash

```
1 /teleop_turtle:
2   qos_overrides./parameter_events.publisher.depth
3   qos_overrides./parameter_events.publisher.duration
4   qos_overrides./parameter_events.publisher.history
5   qos_overrides./parameter_events.publisher.reliability
6   scale_angular
7   scale_linear
8   use_sim_time
9 /turtlesim:
10  background_b
11  background_g
12  background_r
13  qos_overrides./parameter_events.publisher.depth
14  qos_overrides./parameter_events.publisher.duration
15  qos_overrides./parameter_events.publisher.history
16  qos_overrides./parameter_events.publisher.reliability
17  use_sim_time
```

text

Every node has the parameter `use_sim_time`. It is **NOT** unique to [turtlesim](#).

Based on their names, it looks like [/turtlesim](#) parameters determine the background color of the turtlesim window using RGB color values.

ros2 param get To determine a type of the parameter we are trying to understand let's use `ros2 param get`. The following is the syntax of the command

```
1 ros2 param get <node_name> <parameter_name>
```

C.R. 51

bash

Now, let's use this command to determine they type and the value of the parameter:

```
1 Integer value is: 86
```

text

It seems the `background_g` parameter holds an integer value. If we were to run the same command on `background_r` and `background_b`, we will get the values 69 and 255, respectively.

ros2 param set Time to change these parameters and play around. To change a parameter's value at runtime, use the command:

```
1 ros2 param set <node_name> <parameter_name> <value>
```

C.R. 52

bash

Let's do something simple and try to change the `/turtlesim` node's background:

```
1 ros2 param set /turtlesim background_r 150
```

C.R. 53

bash

```
1 Set parameter successful
```

text



The background of our turtlesim window should change colours.¹¹ It is worth mentioning that, setting parameters with the set command will only change them in our `current` session, **NOT** permanently. However, we can save our settings and reload them the next time we want to start a node.

ros2 param dump For this we need to dump this parameters to some other file for later use. Let's have a look at the `ros2 param dump` command. This allows us to view all of a node's current parameter values:

```
1 ros2 param dump <node_name>
```

C.R. 54

bash

The command prints to the standard output (`stdout`) by default but we can also redirect the parameter values into a file to save them for later.

To save our current configuration of `/turtlesim` parameters into the file `turtlesim.yaml`, enter the command:

```
1 ros2 param dump /turtlesim > turtlesim.yaml
```

C.R. 55

bash

In the current working directory we will find a new file. If we were to open this file, we'll see the following content:

```
1 /turtlesim:C.R. 56  
2   ros_parameters:yaml  
3     background_b: 255  
4     background_g: 86  
5     background_r: 150  
6     qos_overrides:  
7       /parameter_events:  
8         publisher:  
9           depth: 1000  
10          durability: volatile  
11          history: keep_last  
12          reliability: reliable  
13         use_sim_time: false
```

Dumping parameters comes in handy if we want to reload the node with the same parameters in the future.

ros2 param load Once the parameters are dumped into a file, we can load them to a currently running node using the command:

```
1 ros2 param load <node_name> <parameter_file>C.R. 57  
bash
```

To load the `turtlesim.yaml` file we just generated with `ros2 param dump` into `/turtlesim` node parameters, we just have to enter the command:

```
1 ros2 param load /turtlesim turtlesim.yamlC.R. 58  
bash
```

```
1 Set parameter background_b successfultext  
2 Set parameter background_g successful  
3 Set parameter background_r successful  
4 Set parameter qos_overrides./parameter_events.publisher.depth failed: parameter  
   ↳ 'qos_overrides./parameter_events.publisher.depth' cannot be set because it is read-only  
5 Set parameter qos_overrides./parameter_events.publisher.durability failed: parameter  
   ↳ 'qos_overrides./parameter_events.publisher.durability' cannot be set because it is  
   ↳ read-only  
6 Set parameter qos_overrides./parameter_events.publisher.history failed: parameter  
   ↳ 'qos_overrides./parameter_events.publisher.history' cannot be set because it is  
   ↳ read-only  
7 Set parameter qos_overrides./parameter_events.publisher.reliability failed: parameter  
   ↳ 'qos_overrides./parameter_events.publisher.reliability' cannot be set because it is  
   ↳ read-only  
8 Set parameter use_sim_time successful
```

Read-only parameters can only be modified at startup and not afterwards, that is why there are some warnings for the `qos_overrides` parameters.

Loading Parameters on Startup To start the same node using our saved parameter values, use:

```
1 ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

C.R. 59
bash

This is the same command we always use to start turtlesim, with the added flags `--ros-args` and `--params-file`, followed by the file we want to load.

To test this out, stop our running turtlesim node, and try reloading it with our saved parameters, using:

```
1 ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

C.R. 60
bash

The turtlesim window should appear as usual, but with the purple background we set earlier.

When a parameter file is used at node startup, all parameters, including the read-only ones, will be updated.

7.7 A Practical Look into Actions

Actions are one of the communication types in ROS and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model (described in the topics tutorial). An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.

To begin we start with a fresh new terminal (`Ctrl + Alt + T`) and type the following command:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 61

bash

And of course we would like to control this turtle so we need to add the `teleop` node as well.

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 62

bash

When we launch the `/teleop_turtle` node, we will see the following message in our terminal:

```
1 Use arrow keys to move the turtle.
2 Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
```

text

Let’s focus on the second line, which corresponds to an action. Pay attention to the terminal where the `/turtlesim` node is running. Each time we press one of these keys, we are sending a goal to an action server that is part of the `/turtlesim` node. The goal is to rotate the turtle to face a particular direction. A message relaying the result of the goal should display once the turtle completes its rotation:

```
1 [INFO] [turtlesim]: Rotation goal completed successfully
```

text

The F key will cancel a goal mid-execution.

Try pressing the C key, and then pressing the F key before the turtle can complete its rotation. In the terminal where the `/turtlesim` node is running, we will see the message:

```
1 [INFO] [turtlesim]: Rotation goal canceled
```

text

Not only can the client-side (our input in the teleop) stop a goal, but the server-side (the `/turtlesim` node) can as well. When the server-side chooses to stop processing a goal, it is said to “abort” the goal.

Try hitting the D key, then the G key before the first rotation can complete. In the terminal where the `/turtlesim` node is running, we will see the message:

```
1 [WARN] [turtlesim]: Rotation goal received before a previous goal finished.          text
2                 Aborting previous goal
```

This action server chose to abort the first goal because it got a new one. It could have chosen something else, like reject the new goal or execute the second goal after the first one finished. Don't assume every action server will choose to abort the current goal when it gets a new one.

ros2 node info To see the list of actions a node provides, `/turtlesim` in this case, open a new terminal and run the command:

```
1 ros2 node info /turtlesim                                         C.R. 63
                                                               bash

1 /turtlesim                                                       text
2 Subscribers:
3   /parameter_events: rcl_interfaces/msg/ParameterEvent
4   /turtle1/cmd_vel: geometry_msgs/msg/Twist
5 Publishers:
6   /parameter_events: rcl_interfaces/msg/ParameterEvent
7   /rosout: rcl_interfaces/msg/Log
8   /turtle1/color_sensor: turtlesim/msg/Color
9   /turtle1/pose: turtlesim/msg/Pose
10 Service Servers:
11   /clear: std_srvs/srv/Empty
12   /kill: turtlesim/srv/Kill
13   /reset: std_srvs/srv/Empty
14   /spawn: turtlesim/srv/Spawn
15   /turtle1/set_pen: turtlesim/srv/SetPen
16   /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
17   /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
18   /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
19   /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
20   /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
21   /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
22   /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
23   /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
24 Service Clients:
25
26 Action Servers:
27   /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
28 Action Clients:
```

The command returns a list of `/turtlesim`'s subscribers, publishers, services, action servers and action clients.

Notice that the `/turtle1/rotate_absolute` action for `/turtlesim` is under Action Servers. This

means /turtlesim responds to and provides feedback for the `/turtle1/rotate_absolute` action.

The `/teleop_turtle` node has the name `/turtle1/rotate_absolute` under Action Clients meaning that it sends goals for that action name. To see that, run:

```
1 ros2 node info /teleop_turtle                                         C.R. 64
bash

1 /teleop_turtle                                                       text
2 Subscribers:
3   /parameter_events: rcl_interfaces/msg/ParameterEvent
4 Publishers:
5   /parameter_events: rcl_interfaces/msg/ParameterEvent
6   /rosout: rcl_interfaces/msg/Log
7   /turtle1/cmd_vel: geometry_msgs/msg/Twist
8 Service Servers:
9   /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
10  /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
11  /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
12  /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
13  /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
14  /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
15 Service Clients:
16
17 Action Servers:
18
19 Action Clients:
20   /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

ros2 action list To identify all the actions in the ROS graph, run the command:

```
1 ros2 action list                                                 C.R. 65
bash

1 /turtle1/rotate_absolute                                         text
```

This is the only action in the ROS graph right now. It controls the turtle's rotation, as we saw earlier. We also already know that there is one action client (part of `/teleop_turtle`) and one action server (part of `/turtlesim`) for this action from using the `ros2 node info <node_name>` command.

ros2 action list -t Actions have types, similar to topics and services. To find `/turtle1/rotate_absolute`'s type, run the command:

```
1 ros2 action list -t                                              C.R. 66
bash

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

In brackets to the right of each action name (in this case only `/turtle1/rotate_absolute`) is the action type, turtlesim/action/RotateAbsolute. We will need this when we want to execute an action from the command line or from code.

7.8 Launching Nodes

Up to now, we have been opening new terminals for every new node we ran. As we create more complex systems with more and more nodes running simultaneously, opening terminals and reentering configuration details becomes tedious.

Launch files allow us to start up and configure a number of executables containing ROS nodes simultaneously.

Running a single launch file with the ros2 launch command will start up our entire system¹² at once.

¹²which includes all nodes and their configurations

```
1 ros2 launch turtlesim multisim.launch.py
```

C.R. 67
bash

This command will run the following launch file:

```
1 from launch import LaunchDescription
2 import launch_ros.actions
3
4
5 def generate_launch_description():
6     return LaunchDescription([
7         launch_ros.actions.Node(
8             namespace='turtlesim1', package='turtlesim',
9             executable='turtlesim_node', output='screen'),
10        launch_ros.actions.Node(
11            namespace='turtlesim2', package='turtlesim',
12            executable='turtlesim_node', output='screen'),
13    ])

```

C.R. 68
python

The launch file above is written in Python, but we can also use XML and YAML to create launch files. We can see a comparison of these different ROS launch formats in Using XML, YAML, and Python for ROS Launch Files.

If this is working well, then we should see two turtlesim environments popping up. This will run two (2) turtlesim nodes.

Controlling the Nodes Now that these nodes are running, we can control them like any other ROS nodes. For example, we can make the turtles drive in opposite directions by opening up two additional terminals and running the following commands:

In a second terminal:

```
1 ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

C.R. 69
bash

In a third terminal:

```
1 ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"
```

C.R. 70

After running these commands, we should see the two turtles in their respective environments spinning like a record.

Chapter 8

Client Libraries

Table of Contents

8.1	Getting Started with Colcon	175
8.2	Creating a Workspace	179
8.3	Creating a Package	183
8.4	Writing a Simple Publisher & Subscriber	184
8.5	Writing a Simple Service and Client	191
8.6	Creating Custom msg and srv Files	197
8.7	Using Parameters in a Class	205
8.8	Managing Dependencies	211
8.9	Creating an Action	213
8.10	Writing an Action Server and Client	215
8.11	Writing a Launch File	216

8.1 Getting Started with Colcon

Now we have learned previously all the essential concepts required for us to get a good introduction to ROS and now we can begin to go deeper into understanding how packages are written and how all previously defined concepts and ideas can be implemented in code. In this light, let us start with looking at the compiler for ROS: [colcon](#)

a meta build tool to improve the workflow of building, testing and using multiple software packages [[Debianpackageros2025](#)].

The first thing to define is [colcon](#) is an iteration on the ROS build tools:

[catkin_make](#), [catkin_make_isolated](#), [catkin_tools](#) and [ament_tools](#).

¹Of course, if we have installed ROS with a Dockerfile and followed the lecture material, we don't have to do this step.

However, this step is nevertheless here in case the document is followed non-linearly.²

We begin by installing colcon.¹

```
1 sudo apt install python3-colcon-common-extensions
2
3 Reading package lists... 0%
4 Reading package lists... 0%
5 Reading package lists... 7%
6 Reading package lists... Done
7 Building dependency tree... 0%
8 Building dependency tree... 0%
9 Building dependency tree... 50%
10 Building dependency tree... 50%
11 Building dependency tree... Done
12 Reading state information... 0%
13 Reading state information... 0%
14 Reading state information... Done
python3-colcon-common-extensions is already the newest version (0.3.0-100).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

C.R. 1
bash

text

The Structure of a Package A ROS workspace is a directory with a **particular structure**. Commonly there is a `src` subdirectory.² Within this subdirectory is where the source code (`src`) of ROS packages will be located.

²This kind of directory usually exists for development of applications such as C or C++.

When this directory is created, it will be empty.

The primary job of `colcon` is to create the structure for, compile and build these source files. By default it will create the following hierarchy of the `src` directory:

build Where intermediate files are stored. For each package a subfolder will be created in which e.g., CMake³ is being invoked.

install Where each package will be installed to. By default each package will be installed into a separate subdirectory.

log Contains various logging information about each `colcon` invocation, for use in error-checking and debugging purposes.

For any student who worked with ROS 1 and `catkin`, there is no `devel` directory.

⁴The name of the directory is up to the user.

To get started we first create a directory (`ros2_ws`) to contain our workspace:⁴

```
1 mkdir -p ~/ros2_ws/src
2 cd ~/ros2_ws
```

C.R. 2
bash

At this moment if we were to `ls` into our directory we will only see one (1) folder which is `src`, which is to be expected as it is created just a second ago.

```
1 ls
C.R. 3
bash

1 src
text
```

Now let's populate our newly created environment with some tutorial files from the official ROS repo:

```
1 git clone https://github.com/ros2/examples src/examples -b humble
C.R. 4
bash

1 Cloning into 'src/examples'...
2 remote: Enumerating objects: 9987, done.
3 remote: Counting objects: 100% (2546/2546), done.
4 remote: Compressing objects: 100% (358/358), done.
5 remote: Total 9987 (delta 2376), reused 2195 (delta 2188), pack-reused 7441 (from 3)
6 Receiving objects: 100% (9987/9987), 1.56 MiB | 4.28 MiB/s, done.
7 Resolving deltas: 100% (7250/7250), done.
```

Sourcing an Underlay It is important that we've sourced the environment for an existing ROS installation which will provide our workspace with the necessary build dependencies for the example packages. This is achieved by sourcing the setup script provided by a binary installation or a source installation.⁵

⁵i.e. another, colcon workspace.

We call this environment an **underlay**.

Our workspace,  **ros2_ws**, will be an **overlay** on top of the existing ROS installation.

In general, it is recommended to use an overlay when we plan to iterate on a small number of packages, rather than putting all of our packages into the same workspace.

Building the Workspace In the root of the workspace, run **colcon build**. Since build types such as **ament_cmake** do **NOT** support the concept of the devel space and require the package to be installed, colcon supports the option **--symlink-install**. This allows the installed files to be changed by changing the files in the source space⁶ for faster iteration.

⁶e.g., Python files or other non-compiled resources

```
1 colcon build --symlink-install --executor sequential
C.R. 5
bash
```

The way we are currently building is **NOT** the official way as we had to add **--executor sequential** option. We are adding this as by default, the colcon processes/compiles packages parallel to speed up the compilation time. If we were to have ROS installed on native hardware rather than a docker container we may not have needed this additional option.

After the build is finished, we should see the **build**, **install**, and **log** directories:

```
1 ls -l                                         C.R. 6  
2  
3 drwxr-xr-x 24 ubuntu ubuntu 4096 Jun  6 13:58 build  
4 drwxr-xr-x 24 ubuntu ubuntu 4096 Jun  6 13:58 install  
5 drwxr-xr-x  4 ubuntu ubuntu 4096 Jun  6 13:57 log  
6 drwxr-xr-x  3 ubuntu ubuntu 4096 Jun  6 13:54 src  
7  
8
```

To run test on built packages we can run `colcon test`

Sourcing the New Package When `colcon` has completed building successfully, the output will be in the `install` directory. Before we can use any of the installed executables or libraries, we will have to add them to our path and library paths. `colcon` will have generated bash files in the `install` directory to help set up the environment. These files will add all of the required elements to our path and library paths as well as provide any bash or shell commands exported by packages.

```
1 source install/setup.bash                         C.R. 7  
2  
3
```

It is time to test out what we have built. Let's open up a new terminal window side by side and run the following two (2) commands:

```
1 ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function          C.R. 8  
2  
3 ros2 run examples_rclcpp_minimal_publisher publisher_member_function           C.R. 9  
4
```

If everything has worked well, we should see messages from the publisher and subscriber with numbers incrementing.

Information

Overlay v. Underlay

As a final clarification on these two (2) concepts, let's look at them in a bit more detail:

- An **underlay** is the core ROS installation that provides the foundational packages and environment for your ROS development. In our case it is Humble.
- An **overlay** is the secondary workspace where we can add new packages without interfering with the existing ROS 2 workspace that we're extending.

8.2 Creating a Workspace

A workspace is a directory containing ROS packages. Before using ROS, it's necessary to source our ROS installation workspace in the terminal we plan to work in. This makes ROS's packages available for you to use in that terminal.

We also have the option of sourcing an "overlay", which is a secondary workspace where we can add new packages without interfering with the existing ROS workspace that we're extending, or "underlay".

Our underlay must contain the dependencies of all the packages in our overlay.

Packages in our overlay will override packages in the underlay.

It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.

Sourcing the Environment Our main ROS installation will be our underlay for this section.⁷

Depending on how we installed ROS, either from source or binaries, and which platform we're on, our exact source command will vary:

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

C.R. 10

bash

⁷Keep in mind that an underlay does **NOT** necessarily have to be the main ROS installation.

Creating a New Directory Best practice is to create a **new directory for every new workspace**.

The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace. Let's choose the directory name **ros2_ws**, for "development workspace":

```
1 mkdir -p ~/ros2_ws/src
2 cd ~/ros2_ws/src
```

C.R. 11

bash

Another best practice is to put any packages in our workspace into the **src** directory. The above code creates a **src** directory inside **ros2_ws** and then navigates into it.

Cloning a Sample Repo Ensure we're still in the **ros2_ws/src** directory before we clone.

In the following sections, we will create our own packages, but for now we will practice putting a workspace together using existing packages. A repo can have multiple branches. We need to check out the one that targets our installed ROS distro. When we clone this repo, add the **-b** argument followed by that branch.

In the **ros2_ws/src** directory, run the following command:

```
1 git clone https://github.com/ros/ros_tutorials.git -b humble
```

C.R. 12

bash

Now `ros_tutorials` is cloned in our workspace. The `ros_tutorials` repository contains the `turtlesim` package, which we'll use in this section. The other packages in this repository are not built because they contain a `COLCON_IGNORE` file.

So far we have populated our workspace with a sample package, but it isn't a fully-functional workspace yet as we need to resolve the dependencies first and then build the workspace.

Resolving Dependencies Before building the workspace, we need to resolve the package dependencies. It is possible we may have all the dependencies already, but best practice is to check for dependencies every time we clone. We wouldn't want a build to fail after a long wait only to realize that we have missing dependencies.

From the root of our workspace (`ros2_ws`), run the following command:

If we're still in the `src` directory with the `ros_tutorials` clone, make sure to run `cd ..` to move back up to the workspace (`ros2_ws`).

```
1 cd ..
2 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 13

bash

If we already have all our dependencies, the console will return:

```
1 #All required rosdeps installed successfully
```

text

⁸We will learn more about packages in the following section. Packages declare their dependencies in the `package.xml` file.⁸ This command walks through those declarations and installs the ones that are missing.

Building the Workspace From the root of our workspace (`ros2_ws`), we can now build our packages using the command:

```
1 colcon build
```

C.R. 14

bash

```
1 Starting >> turtlesim
2 Finished << turtlesim [5.49s]
3
4 Summary: 1 package finished [5.58s]
```

text

There are some useful arguments for `colcon build` which are as follows:

--packages-up-to

builds the package we want, plus all its dependencies, but not the whole workspace (saves time)

--symlink-install

saves us from having to rebuild every time we tweak python scripts

--event-handlers console_direct+

shows console output while building (can otherwise be found in the log directory)

--executor sequential

processes the packages one by one instead of using parallelism

Once the build is finished, enter the command in the workspace root (`~/ros2_ws`). We will see that colcon has created new directories:

```
1 ls
C.R. 15
bash

1 build install log src
text
```

The install directory is where our workspace's setup files are, which we can use to source our overlay.

Sourcing our Overlay Before sourcing the overlay, it is very important that we open a new terminal, separate from the one where we built the workspace. Sourcing an overlay in the same terminal where we built, or likewise building where an overlay is sourced, may create complex issues.

In the new terminal, source our main ROS 2 environment as the “underlay”, so we can build the overlay “on top of” it:

```
1 source /opt/ros/humble/setup.bash
C.R. 16
bash
```

Time to go into the root of our workspace:

```
1 cd ~/ros2_ws
C.R. 17
bash
```

In the root, source our overlay:

```
1 source install/local_setup.bash
C.R. 18
bash
```

Sourcing the `local_setup` of the overlay will only add the packages available in the overlay to our environment. `setup` sources the overlay as well as the underlay it was created in, allowing us to utilise both workspaces. So, sourcing our main ROS installation's setup and then the `ros2_ws` overlay's `local_setup`, like we just did, is the same as just sourcing

📁 `ros2_ws`'s setup, because that includes the environment of its underlay.

Now we can run the `turtlesim` package from the overlay:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 19
bash

But how can you tell that this is the overlay turtlesim running, and not your main installation's turtlesim?

Let's modify turtlesim in the overlay so you can see the effects:

- We can modify and rebuild packages in the overlay separately from the underlay.
- The overlay takes precedence over the underlay.

Modifying the Overlay You can modify turtlesim in your overlay by editing the `title bar` on the turtlesim window. To do this, locate the `📁 turtle_frame.cpp` file in `📁 ros2_ws › src › ros_tutorials › turtlesim › src`. Open `📁 turtle_frame.cpp` with your preferred text editor.

Find the function `setTitle("TurtleSim");`, change the value "TurtleSim" to "MyTurtleSim", and save the file.

Return to the first terminal where you ran colcon build earlier and run it again.

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 20
bash

Return to the second terminal (where the overlay is sourced) and run turtlesim again:

Even though your main ROS 2 environment was sourced in this terminal earlier, the overlay of your `📁 ros2_ws` environment takes precedence over the contents of the underlay.

To see that your underlay is still intact, open a brand new terminal and source only your ROS 2 installation. Run turtlesim again:

```
1 ros2 run turtlesim turtlesim_node
```

C.R. 21
bash

You can see that modifications in the overlay did not actually affect anything in the underlay.

8.3 Creating a Package

A package is an organizational unit for our ROS code. If we want to be able to install our code or share it with others, then we'll need it organized in a package. With packages, we can release our ROS work and allow others to build and use it easily.

Package creation in ROS uses ament as its build system and colcon as its build tool. We can create a package using either CMake or Python, which are officially supported, though other build types do exist.

The Anatomy of a Package ROS Python and CMake packages each have their own minimum required contents:

package.xml file containing meta information about the package

resource/<package_name> marker file for the package

setup.cfg is required when a package has executables, so ros2 run can find them

setup.py containing instructions for how to install the package

<package_name> a directory with the same name as our package, used by ROS tools to find our package, contains **__init__.py**

The simplest possible package may have a file structure that looks like:

8.4 Writing a Simple Publisher & Subscriber

In this exercise, we will create `nodes` which passes information in the form of string messages to each other over a `topic`. The example we will use here is a simple “talker” and “listener” system where one publishes data and the other subscribes to the topic so it can receive that data.

Creating a ROS Packages To begin, we open a new terminal window and navigate to our previously created `ros2_ws`.

Recall that packages should be created in the `src` directory, not the root `/` of the workspace.

So naturally, navigate into the `ros2_ws/src` directory, and run the package creation command:

```
1 ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub
```

C.R. 22
bash

If everything works well, our terminal will return a message verifying the creation of our package `py_pubsub` and all its necessary files and folders.

8.4.1 Writing the Publisher Node

Now that we have a package template, please navigate into `ros2_ws/src/py_pubsub/py_pubsub`. To get started, download the example `talker code` by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/_  
→ minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py
```

C.R. 23
bash

Here the `wget` command basically access the file in a web-server and then downloads it to the current working directory. If the code works successfully, there will be a new file named `publisher_member_function.py` adjacent to `__init__.py`.

Now let's open the code and see what is going on under the hood. The following is the code in full with snippets and detailed explanation to follow:

```
1 import rclpy  
2 from rclpy.node import Node  
3  
4 from std_msgs.msg import String  
5  
6 class MinimalPublisher(Node):  
7  
8
```

C.R. 24
python

```

9  def __init__(self):
10     super().__init__('minimal_publisher')
11     self.publisher_ = self.create_publisher(String, 'topic', 10)
12     timer_period = 0.5 # seconds
13     self.timer = self.create_timer(timer_period, self.timer_callback)
14     self.i = 0
15
16 def timer_callback(self):
17     msg = String()
18     msg.data = 'Hello World: %d' % self.i
19     self.publisher_.publish(msg)
20     self.get_logger().info('Publishing: "%s"' % msg.data)
21     self.i += 1
22
23
24 def main(args=None):
25     rclpy.init(args=args)
26
27     minimal_publisher = MinimalPublisher()
28
29     rclpy.spin(minimal_publisher)
30
31     # Destroy the node explicitly
32     # (optional - otherwise it will be done automatically
33     # when the garbage collector destroys the node object)
34     minimal_publisher.destroy_node()
35
36     rclpy.shutdown()
37
38 if __name__ == '__main__':
39     main()

```

C.R. 25
python

Deconstructing the Code The first lines of code after the comments import `rclpy` so its `Node` class can be used.⁹

⁹As a reminder, the `rclpy` is the ROS library written for Python.

```

1 import rclpy
2 from rclpy.node import Node

```

C.R. 26
python

The next statement imports the built-in string message type which the node uses to structure the data it passes on the topic.

```

1 from std_msgs.msg import String

```

C.R. 27
python

These aforementioned lines represent the node's `dependencies`. Recall that dependencies have to be added to `package.xml`, which we will have a look at in just a little bit. Next, the `MinimalPublisher` class is created, which inherits¹⁰ from `Node`.

¹⁰or is a subclass of

```
1 class MinimalPublisher(Node):
```

C.R. 28

python

Following is the definition of the class's constructor. `super().__init__` calls the `Node` class's constructor and gives it our node name, in this case `minimal_publisher`.

¹¹which is imported from the `std_msgs.msg` module

`create_publisher` declares the node publishes messages of type `String`,¹¹ over a topic named `topic`, and that the “queue size” is `10`.

Queue size is a required QoS (quality of service) setting which limits the amount of queued messages if a subscriber is **NOT** receiving them fast enough.

Next, a timer is created with a callback to execute every `0.5` seconds. `self.i` is a counter used in the callback.

```
1 def __init__(self):
2     super().__init__('minimal_publisher')
3     self.publisher_ = self.create_publisher(String, 'topic', 10)
4     timer_period = 0.5 # seconds
5     self.timer = self.create_timer(timer_period, self.timer_callback)
6     self.i = 0
```

C.R. 29

python

`timer_callback` creates a message with the counter value appended, and publishes it to the console with `get_logger().info`.

```
1 def timer_callback(self):
2     msg = String()
3     msg.data = 'Hello World: %d' % self.i
4     self.publisher_.publish(msg)
5     self.get_logger().info('Publishing: "%s"' % msg.data)
6     self.i += 1
```

C.R. 30

python

Lastly, the main function is defined.

```
1 def main(args=None):
2     rclpy.init(args=args)
3
4     minimal_publisher = MinimalPublisher()
5
6     rclpy.spin(minimal_publisher)
7
8     # Destroy the node explicitly
9     # (optional - otherwise it will be done automatically
10    # when the garbage collector destroys the node object)
11    minimal_publisher.destroy_node()
12    rclpy.shutdown()
```

C.R. 31

python

First the `rclpy` library is initialized, then the node is created, and then it “spins” the node so its callbacks are called.

Adding Dependencies Navigate one level back to the `ros2_ws/src/py_pubsub` directory, where the `setup.py`, `setup.cfg`, and `package.xml` files have been created for us. Open `package.xml` with our favourite text editor.¹²

As mentioned previously, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```
1 <description>Examples of minimal publisher/subscriber using rclpy</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>
```

C.R. 32
xml

¹²This could of course be emacs, or something which is not emacs so we can see why emacs is better.

After the lines above, add the following dependencies corresponding to our node's import statements:¹³

```
1 <exec_depend>rclpy</exec_depend>
2 <exec_depend>std_msgs</exec_depend>
```

C.R. 33
xml

¹³Remember, we need to let ROS know the dependencies required by the python script.

This declares the package needs `rclpy` and `std_msgs` when its code is executed.

Adding An Entry Point Given we have sorted our package manifesto, we need to configure our python code. Open the `setup.py` file. Again, match the `maintainer`, `maintainer_email`, `description` and `license` fields to our `package.xml`:

```
1 maintainer='YourName',
2 maintainer_email='you@email.com',
3 description='Examples of minimal publisher/subscriber using rclpy',
4 license='Apache License 2.0',
```

C.R. 34
python

Add the following line within the `console_scripts` brackets of the `entry_points` field:

```
1 entry_points={
2     'console_scripts': [
3         'talker = py_pubsub.publisher_member_function:main',
4     ],
5 },
```

C.R. 35
python

Checking the Configuration File The contents of the `setup.cfg` file should be correctly populated automatically, like so:

```
1 [develop]
2 script_dir=$base/lib/py_pubsub
3 [install]
4 install_scripts=$base/lib/py_pubsub
```

C.R. 36
cfg

This code is simply telling setuptools to put our executables in lib, because ros2 run will look for them there. We could build our package now, source the local setup files, and run it, but let's create the subscriber node first so we can see the full system at work.

8.4.2 Writing the Subscriber Node

Return to `ros2_ws/src/py_pubsub/py_pubsub` to create the next node. Enter the following code in our terminal:

```
1 wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/_           C.R. 37
   ↳ minimal_subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py      bash
```

Now the directory should have these files:

`__init__.py`, `publisher_member_function.py` and `subscriber_member_function.py`

Examining the Code Open the `subscriber_member_function.py` with our preferred text editor.

```
1 import rclpy
2 from rclpy.node import Node
3
4 from std_msgs.msg import String
5
6
7 class MinimalSubscriber(Node):
8
9     def __init__(self):
10         super().__init__('minimal_subscriber')
11         self.subscription = self.create_subscription(
12             String,
13             'topic',
14             self.listener_callback,
15             10)
16         self.subscription # prevent unused variable warning
17
18     def listener_callback(self, msg):
19         self.get_logger().info('I heard: "%s"' % msg.data)
20
21
22 def main(args=None):
23     rclpy.init(args=args)
24
25     minimal_subscriber = MinimalSubscriber()
26
27     rclpy.spin(minimal_subscriber)
28
```

```

29     # Destroy the node explicitly
30     # (optional - otherwise it will be done automatically
31     # when the garbage collector destroys the node object)
32     minimal_subscriber.destroy_node()
33     rclpy.shutdown()

34

35

36 if __name__ == '__main__':
37     main()

```

C.R. 39
python

The subscriber node's code is nearly identical to the publisher's. The constructor creates a subscriber with the same arguments as the publisher.

the topic name and message type used by the publisher and subscriber must match to allow them to communicate.

```

1 self.subscription = self.create_subscription(
2     String,
3     'topic',
4     self.listener_callback,
5     10)

```

C.R. 40
python

The subscriber's constructor and callback don't include any timer definition, because it doesn't need one. Its callback gets called as soon as it receives a message.

The callback definition simply prints an info message to the console, along with the data it received. Recall that the publisher defines

```

1 def listener_callback(self, msg):
2     self.get_logger().info('I heard: "%s"' % msg.data)

```

C.R. 41
python

The main definition is almost exactly the same, replacing the creation and spinning of the publisher with the subscriber.

```

1 minimal_subscriber = MinimalSubscriber()
2
3 rclpy.spin(minimal_subscriber)

```

C.R. 42
python

Since this node has the same dependencies as the publisher, there's nothing new to add to package.xml. The setup.cfg file can also remain untouched.

Adding an Entry Point Reopen setup.py and add the entry point for the subscriber node below the publisher's entry point. The `entry_points` field should now look like this:

```
1 entry_points={  
2     'console_scripts': [  
3         'talker = py_pubsub.publisher_member_function:main',  
4         'listener = py_pubsub.subscriber_member_function:main',  
5     ],  
6 }
```

C.R. 43
python

8.4.3 Building and Running

We likely already have the `rclpy` and `std_msgs` packages installed as part of our ROS system. It's good practice to run `rosdep` in the root of our workspace (`ros2_ws`) to check for missing dependencies before building:

8.5 Writing a Simple Service and Client

When nodes communicate using services, the node which sends a request for data is called the **client** node, and the one that responds to the request is the **service** node. The structure of the request and response is determined by a `.srv` file.

The example used here is a simple integer addition system;

one node requests the sum of two (2) integers, and the other responds with the result.

Now let's write our implementation.

Creating a New Package Let's begin by opening a new terminal and source our ROS installation so `ros2` commands will work. Once done, please navigate into the `ros2_ws` directory created in previously.

Remember that, packages should be created in the `src` directory and **NOT** the root of the workspace. Navigate into `ros2_ws/src` and create a new package:

```
1 ros2 pkg create \
2   --build-type ament_python \
3   --license Apache-2.0 py_srvcli \
4   --dependencies rclpy example_interfaces
```

C.R. 44
bash

Our terminal will return a message verifying the creation of our package `py_srvcli` and all its necessary files and folders.

The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml`. `example_interfaces` is the package that includes the `.srv` file we will need to structure our requests and responses:

```
1 int64 a
2 int64 b
3 ---
4 int64 sum
```

C.R. 45
xml

The first two lines are the parameters of the request, and below the dashes is the response.

Updating Package Manifesto Because we used the `--dependencies` option during package creation, we don't have to manually add dependencies to `package.xml`.

As always, though, make sure to add the description, maintainer email and name, and license information to `package.xml` as a good open-source developer.

```
1 <description>Python client server tutorial</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>
```

C.R. 46
xml

Updating the Configuration Add the same information to the `setup.py` file for the `maintainer`, `maintainer_email`, `description` and license fields:

```
1 maintainer='Your Name',
2 maintainer_email='you@email.com',
3 description='Python client server tutorial',
4 license='Apache License 2.0',
```

C.R. 47
python

8.5.1 Writing the Service Node

Once we are sure we are inside the `ros2_ws/src/py_srvcli/py_srvcli` directory, we then create a new file called `service_member_function.py` and paste the following code within:

```
1 from example_interfaces.srv import AddTwoInts
2
3 import rclpy
4 from rclpy.node import Node
5
6
7 class MinimalService(Node):
8
9     def __init__(self):
10         super().__init__('minimal_service')
11         self.srv = self.create_service(AddTwoInts, 'add_two_ints',
12                                     self.add_two_ints_callback)
13
14     def add_two_ints_callback(self, request, response):
15         response.sum = request.a + request.b
16         self.get_logger().info('Incoming request\na: %d b: %d' %
17                               (request.a, request.b))
18
19         return response
20
21     def main():
22         rclpy.init()
23
24         minimal_service = MinimalService()
25
26         rclpy.spin(minimal_service)
27
28         rclpy.shutdown()
```

C.R. 48
python

```

29
30     if __name__ == '__main__':
31         main()

```

C.R. 49
python

Let's look at the code in more detail and what what is going on.

Examining the Code The first `import` statement imports the `AddTwoInts` service type from the `example_interfaces` package. The following import statement imports the ROS Python client library (`rclpy`), and specifically the `Node` class.

```

1  from example_interfaces.srv import AddTwoInts
2
3  import rclpy
4  from rclpy.node import Node

```

C.R. 50
python

The `MinimalService` class constructor initializes the node with the name `minimal_service`. Then, it creates a service and defines the type, name, and callback.

```

1  def __init__(self):
2      super().__init__('minimal_service')
3      self.srv = self.create_service(AddTwoInts, 'add_two_ints',
4                                     self.add_two_ints_callback)

```

C.R. 51
python

The definition of the service callback receives the request data, sums it, and returns the sum as a response.

```

1  def add_two_ints_callback(self, request, response):
2      response.sum = request.a + request.b
3      self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))
4
5      return response

```

C.R. 52
python

Finally, the main class initializes the ROS Python client library, instantiates the `MinimalService` class to create the service node and spins the node to handle callbacks.

Adding an Entry Point To allow the `ros2 run` command to run our node, we must add the entry point to `setup.py`¹⁴. To make this happen, all we have to do this add the following line between the '`console_scripts`' brackets:

```

1  'service = py_srvcli.service_member_function:main',

```

C.R. 53
python

¹⁴This is located in the `ros2_ws/src/py_srvcli` directory.

8.5.2 Writing the Client Node

Once we are inside the `ros2_ws/src/py_srvcli/py_srvcli` directory, create a new file called `client_member_function.py` and paste the following code within:

```

1 import sys
2
3 from example_interfaces.srv import AddTwoInts
4 import rclpy
5 from rclpy.node import Node
6
7
8 class MinimalClientAsync(Node):
9
10     def __init__(self):
11         super().__init__('minimal_client_async')
12         self.cli = self.create_client(AddTwoInts, 'add_two_ints')
13         while not self.cli.wait_for_service(timeout_sec=1.0):
14             self.get_logger().info('service not available, waiting again...')
15         self.req = AddTwoInts.Request()
16
17     def send_request(self, a, b):
18         self.req.a = a
19         self.req.b = b
20         return self.cli.call_async(self.req)
21
22
23 def main():
24     rclpy.init()
25
26     minimal_client = MinimalClientAsync()
27     future = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
28     rclpy.spin_until_future_complete(minimal_client, future)
29     response = future.result()
30     minimal_client.get_logger().info(
31         'Result of add_two_ints: for %d + %d = %d' %
32         (int(sys.argv[1]), int(sys.argv[2]), response.sum))
33
34     minimal_client.destroy_node()
35     rclpy.shutdown()
36
37
38 if __name__ == '__main__':
39     main()

```

C.R. 54

python

Examining the Code As with the service code, we first `import` the necessary libraries.

```

1 import sys
2
3 from example_interfaces.srv import AddTwoInts

```

C.R. 55

python

```
4 import rclpy
5 from rclpy.node import Node
```

C.R. 56
python

The `MinimalClientAsync` class constructor initializes the node with the name `minimal_client_async`. The constructor definition creates a client with the same type and name as the service node. The type and name must match for the client and service to be able to communicate. The while loop in the constructor checks if a service matching the type and name of the client is available once a second. Finally it creates a new `AddTwoInts` request object.

```
1 def __init__(self):
2     super().__init__('minimal_client_async')
3     self.cli = self.create_client(AddTwoInts, 'add_two_ints')
4     while not self.cli.wait_for_service(timeout_sec=1.0):
5         self.get_logger().info('service not available, waiting again...')
6     self.req = AddTwoInts.Request()
```

C.R. 57
python

Below the constructor is the `send_request` method, which will send the request and spin until it receives the response or fails.

```
1 def send_request(self, a, b):
2     self.req.a = a
3     self.req.b = b
4     return self.cli.call_async(self.req)
```

C.R. 58
python

Finally we have the `main` method, which constructs a `MinimalClientAsync` object, sends the request using the passed-in command-line arguments, calls `rclpy.spin_until_future_complete` to wait for the result, and logs the results.

Adding an Entry Point Similar to the service node, we also have to add an `entry point` to be able to run the client node from the command-line. The `entry_points` field of our `setup.py` file should look like this:

```
1 entry_points={
2     'console_scripts': [
3         'service = py_srvcli.service_member_function:main',
4         'client = py_srvcli.client_member_function:main',
5         ],
6 }
```

C.R. 59
python

It's good practice to run `rosdep` in the root of our workspace (`ros2_ws`) to check for missing dependencies before building:

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 60
python

Navigate back to the root of our workspace, ros2_ws, and build our new package:

```
1 colcon build --packages-select py_srvcli
```

C.R. 61

python

Open a new terminal, navigate to ros2_ws, and source the setup files:

```
1 source install/setup.bash
```

C.R. 62

python

Now run the service node:

```
1 ros2 run py_srvcli service
```

C.R. 63

python

The node will wait for the client's request.

Open another terminal and source the setup files from inside ros2_ws again. Start the client node, followed by any two integers separated by a space. If we chose 2 and 3, for example, the client would receive a response like this:

```
1 ros2 run py_srvcli client 2 3
```

C.R. 64

python

```
1 [INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

text

Return to the terminal where our service node is running. We will see that it published log messages when it received the request:

```
1 [INFO] [minimal_service]: Incoming request
2 a: 2 b: 3
```

text

8.6 Creating Custom msg and srv Files

Previously, we utilised message and service interfaces to learn about:

- topics,
- services,
- simple publisher/subscriber, and
- service/client nodes.

It is worth noting, the interfaces we used previously were predefined in those cases.

While it's good practice to use predefined interface definitions, time will eventually come, where we will need to define our own messages and services sometimes as well. Here, we will have a look at the simplest method of creating custom interface definitions.

Creating a New Package Here, we will be creating our custom `.msg` and `.srv` files in their own package, and then utilising them in a separate package.

It is worth stressing, that both packages should be in the same workspace.

As we will use the pub/sub and service/client packages we created previously, make sure we are in the same workspace as those packages (`ros2_ws/src`), and then run the following command to create a new package:

```
1 ros2 pkg create --build-type ament_cmake --license Apache-2.0 tutorial_interfaces
```

C.R. 65
bash

`tutorial_interfaces` is the **name of the new package**. Note that it is, and can only be, an `ament_cmake` package,¹⁵ but this doesn't restrict in which type of packages we can use our messages and services. We can create our own custom interfaces in an `ament_cmake` package, and then use it in a C++ or Python node, which will be covered later.

¹⁵It is basically a build system for CMake based packages in ROS. It is a set of scripts enhancing CMake and adding convenience functionality for package authors.

The `.msg` and `.srv` files are required to be placed in directories called `msg` and `srv` respectively.

We can create the directories in `ros2_ws/src/tutorial_interfaces` using:

```
1 mkdir msg srv
```

C.R. 66
bash

8.6.1 Creating Custom Definitions

msg Definition In the `tutorial_interfaces>msg` directory we just created, make a new file called `Num.msg` with one line of code declaring its data structure:

```
1 int64 num
```

C.R. 67

text

This is a **custom message** which transfers a single 64-bit integer called `num`.

In addition, in the `tutorial_interfaces>msg` directory we've just created, make a new file called `Sphere.msg` with the following content:

```
1 geometry_msgs/Point center
2 float64 radius
```

C.R. 68

bash

This custom message uses a message from another message package. As we can see from the first line in the message, it is `geometry_msgs/Point`.

srv Definition Let's go back in the `tutorial_interfaces/srv` directory we've just created a few moments ago and make a new file called `AddThreeInts.srv` with the following request and response structure:

```
1 int64 a
2 int64 b
3 int64 c
4 ---
5 int64 sum
```

C.R. 69

bash

This is our custom service which requests three (3) integers aptly named `a`, `b`, and `c`, and responds with an integer called `sum`.

¹⁶This can be either C++ or Python.

CMakeLists To convert the interfaces we defined into language-specific code¹⁶ so that they can be used in those languages, let's add the following lines to our `CMakeLists.txt`:

```
1 find_package(geometry_msgs REQUIRED)
2 find_package(rosidl_default_generators REQUIRED)
3
4 rosidl_generate_interfaces(${PROJECT_NAME}
5   "msg/Num.msg"
6   "msg/Sphere.msg"
7   "srv/AddThreeInts.srv"
8   DEPENDENCIES geometry_msgs # Add packages that above messages
9   #depend on, in this case geometry_msgs for Sphere.msg
10 )
```

C.R. 70

cmake

The first argument (library name) in the `rosidl_generate_interfaces` must start with the name of the package, e.g., simply `${PROJECT_NAME}` or `${PROJECT_NAME}_suffix`.

Package.xml Because the interfaces rely on `rosidl_default_generators` for generating language-specific code, we need to declare a build tool dependency on it. `rosidl_default_runtime` is a runtime or execution-stage dependency, needed to be able to use the interfaces later.

The `rosidl_interface_packages` is the name of the dependency group which our package, `tutorial_interfaces`, should be associated with, declared using the `<member_of_group>` tag.

Add the following lines within the `<package>` element of `package.xml`:

```
1 <depend>geometry_msgs</depend>
2 <buildtool_depend>rosidl_default_generators</buildtool_depend>
3 <exec_depend>rosidl_default_runtime</exec_depend>
4 <member_of_group>rosidl_interface_packages</member_of_group>
```

C.R. 71

xml

Please pay attention to the last line where we have added the new tag.

Building the Package Now that all the parts of our custom interfaces package are in place, we can finally build the package. In the root of our workspace (`~/ros2_ws`), please run the following command:

```
1 colcon build --packages-select tutorial_interfaces
```

C.R. 72

bash

Now the interfaces will be discoverable by other ROS packages.

Confirming the Creation In a new terminal, let's run the following command from within our workspace (`ros2_ws`) to source it if it is required:

```
1 source install/setup.bash
```

C.R. 73

bash

Now we can confirm our interface creation has worked by using the `ros2 interface show` command. The output we see in our terminal should look similar to the following:

```
1 ros2 interface show tutorial_interfaces/msg/Num
```

C.R. 74

bash

```
1 int64 num
```

text

```
1 ros2 interface show tutorial_interfaces/msg/Sphere
```

C.R. 75

bash

```
1 geometry_msgs/Point center                                     text
2   float64 x
3   float64 y
4   float64 z
5 float64 radius

1 ros2 interface show tutorial_interfaces/srv/AddThreeInts          C.R. 76
2

1 int64 a
2 int64 b
3 int64 c
4 ---
5 int64 sum
```

8.6.2 Testing the Newly Built Interfaces

Time to see our new interface in action. A few simple modifications to the nodes, `CMakeLists.txt` and `package.xml` files will allow we to use our new interfaces.

Publisher and Subscriber System: Publisher Code

```
1 import rclpy
2 from rclpy.node import Node
3
4 from tutorial_interfaces.msg import Num                         # CHANGE
5
6
7 class MinimalPublisher(Node):
8
9     def __init__(self):
10         super().__init__('minimal_publisher')
11         self.publisher_ = self.create_publisher(Num, 'topic', 10)  # CHANGE
12         timer_period = 0.5
13         self.timer = self.create_timer(timer_period, self.timer_callback)
14         self.i = 0
15
16     def timer_callback(self):
17         msg = Num()                                              # CHANGE
18         msg.num = self.i                                         # CHANGE
19         self.publisher_.publish(msg)
20         self.get_logger().info('Publishing: "%d"' % msg.num)    # CHANGE
21         self.i += 1
22
23
24 def main(args=None):
```

```

25 rclpy.init(args=args)
26
27 minimal_publisher = MinimalPublisher()
28
29 rclpy.spin(minimal_publisher)
30
31 minimal_publisher.destroy_node()
32 rclpy.shutdown()
33
34
35 if __name__ == '__main__':
36     main()

```

C.R. 78
python

Publisher and Subscriber System: Subscriber Code

```

1 import rclpy
2 from rclpy.node import Node
3
4 from tutorial_interfaces.msg import Num           # CHANGE
5
6
7 class MinimalSubscriber(Node):
8
9     def __init__(self):
10         super().__init__('minimal_subscriber')
11         self.subscription = self.create_subscription(
12             Num,                                         # CHANGE
13             'topic',
14             self.listener_callback,
15             10)
16         self.subscription
17
18     def listener_callback(self, msg):
19         self.get_logger().info('I heard: "%d"' % msg.num) # CHANGE
20
21
22 def main(args=None):
23     rclpy.init(args=args)
24
25     minimal_subscriber = MinimalSubscriber()
26
27     rclpy.spin(minimal_subscriber)
28
29     minimal_subscriber.destroy_node()
30     rclpy.shutdown()
31
32
33 if __name__ == '__main__':
34     main()

```

C.R. 79
python

We also need to edit our `package.xml` file to make all the previous python code to work.

```
1 <exec_depend>tutorial_interfaces</exec_depend>
```

C.R. 80
xml

Once we have done the necessary changes let's build our package and execute it.

```
1 colcon build --packages-select py_pubsub
```

C.R. 81
bash

Then open two new terminals, source `ros2_ws` in each, and run:

```
1 ros2 run py_pubsub talker
```

C.R. 82
bash

```
1 ros2 run py_pubsub talker
```

C.R. 83
bash

Since `Num.msg` relays only an integer, the talker should only be publishing integer values, as opposed to the string it published previously:

```
1 [INFO] [minimal_publisher]: Publishing: '0'
2 [INFO] [minimal_publisher]: Publishing: '1'
3 [INFO] [minimal_publisher]: Publishing: '2'
```

text

Service Client System

With a few modifications to the service/client package created previously, we can see `AddThreeInts.srv` in action. Since we'll be changing the original two (2) integer request srv to a three (3) integer request srv, the output will be slightly different.

Service

```
1 from tutorial_interfaces.srv import AddThreeInts # CHANGE
2
3 import rclpy
4 from rclpy.node import Node
5
6
7 class MinimalService(Node):
8
9     def __init__(self):
10         super().__init__('minimal_service')
11         self.srv = self.create_service(AddThreeInts,
12                                       'add_three_ints',
13                                       self.add_three_ints_callback) # CHANGE
```

C.R. 84
python

```

15     def add_three_ints_callback(self, request, response):
16         response.sum = request.a + request.b + request.c # CHANGE
17         self.get_logger()\
18             .info('Incoming request\na: %d b: %d c: %d' % (request.a, request.b,
19                                         request.c)) # CHANGE
20
21     return response
22
23 def main(args=None):
24     rclpy.init(args=args)
25
26     minimal_service = MinimalService()
27
28     rclpy.spin(minimal_service)
29
30     rclpy.shutdown()
31
32 if __name__ == '__main__':
33     main()

```

C.R. 85
python

Client

```

1 from tutorial_interfaces.srv import AddThreeInts           # CHANGE python
2 import sys
3 import rclpy
4 from rclpy.node import Node
5
6
7 class MinimalClientAsync(Node):
8
9     def __init__(self):
10         super().__init__('minimal_client_async')
11         self.cli = self.create_client(AddThreeInts, 'add_three_ints') # CHANGE
12         while not self.cli.wait_for_service(timeout_sec=1.0):
13             self.get_logger().info('service not available, waiting again...')
14         self.req = AddThreeInts.Request() # CHANGE
15
16     def send_request(self):
17         self.req.a = int(sys.argv[1])
18         self.req.b = int(sys.argv[2])
19         self.req.c = int(sys.argv[3]) # CHANGE
20         self.future = self.cli.call_async(self.req)
21
22
23 def main(args=None):
24     rclpy.init(args=args)
25
26     minimal_client = MinimalClientAsync()
27     minimal_client.send_request()
28

```

C.R. 86
python

```

29     while rclpy.ok():
30         rclpy.spin_once(minimal_client)
31         if minimal_client.future.done():
32             try:
33                 response = minimal_client.future.result()
34             except Exception as e:
35                 minimal_client.get_logger().info(
36                     'Service call failed %r' % (e,))
37             else:
38                 minimal_client.get_logger().info(
39                     'Result of add_three_ints: for %d + %d + %d = %d' %
40                     # CHANGE
41                     (minimal_client.req.a, minimal_client.req.b, minimal_client.req.c,
42                     response.sum)) # CHANGE
43             break
44
45
46
47 if __name__ == '__main__':
48     main()

```

C.R. 87

python

To add these features to our package we need to let `package.xml` know, which for that we need to write:

```

1 <exec_depend>tutorial_interfaces</exec_depend>

```

C.R. 88

xml

After making the above edits and saving all the changes, build the package:

Then open two new terminals, source  `ros2_ws` in each, and run:

```

1 ros2 run py_srvcli service

```

C.R. 89

bash

```

1 ros2 run py_srvcli client 2 3 1

```

C.R. 90

bash

8.7 Using Parameters in a Class

When making our own nodes we will sometimes need to add parameters that can be set from the launch file.

This tutorial will show we how to create those parameters in a Python class, and how to set them in a launch file.

Creating a Package Open a new terminal and source our ROS installation so that ros2 commands will work.

Follow these instructions to create a new workspace named `ros2_ws`.

Recall that packages should be created in the `src` directory, not the root of the workspace. Navigate into `ros2_ws/src` and create a new package:

```
1 ros2 pkg create --build-type ament_python \
2   --license Apache-2.0 python_parameters \
3   --dependencies rclpy
```

C.R. 91

bash

Our terminal will return a message verifying the creation of our package `python_parameters` and all its necessary files and folders.

The `--dependencies` argument will automatically add the necessary dependency lines to `package.xml` and `CMakeLists.txt`.

Updating the Package Manifesto Because we used the `--dependencies` option during package creation, we don't have to manually add dependencies to `package.xml` or `CMakeLists.txt`.

As always, though, make sure to add the description, maintainer email and name, and license information to `package.xml`.

```
1 <description>Python parameter tutorial</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>
```

C.R. 92

xml

Writing our Python Code Inside the `ros2_ws/src/python_parameters/python_parameters` directory, create a new file called `python_parameters_node.py` and paste the following code within:

```
1 import rclpy
2 import rclpy.node
3
```

C.R. 93

python

```

4   class MinimalParam(rclpy.node.Node):
5       def __init__(self):
6           super().__init__('minimal_param_node')
7
7       self.declare_parameter('my_parameter', 'world')
8
9       self.timer = self.create_timer(1, self.timer_callback)
10
11
12   def timer_callback(self):
13       my_param = self.get_parameter('my_parameter').get_parameter_value().string_value
14
15       self.get_logger().info('Hello %s!' % my_param)
16
17       my_new_param = rclpy.parameter.Parameter(
18           'my_parameter',
19           rclpy.Parameter.Type.STRING,
20           'world'
21       )
22       all_new_parameters = [my_new_param]
23       self.set_parameters(all_new_parameters)
24
25
26   def main():
27       rclpy.init()
28       node = MinimalParam()
29       rclpy.spin(node)
30
31   if __name__ == '__main__':
32       main()

```

C.R. 94

python

Code Demystified The import statements at the top are used to import the package dependencies.

The next piece of code creates the class and the constructor. The line;

```
self.declare_parameter('my_parameter', 'world')
```

of the constructor creates a parameter with the name `my_parameter` and a default value of `world`. The parameter type is inferred from the default value, so in this case it would be set to a string type. Next the timer is initialized with a period of 1, which causes the `timer_callback` function to be executed once a second.

```

1   class MinimalParam(rclpy.node.Node):
2       def __init__(self):
3           super().__init__('minimal_param_node')
4
4       self.declare_parameter('my_parameter', 'world')
5
6       self.timer = self.create_timer(1, self.timer_callback)
7

```

C.R. 95

python

The first line of our `timer_callback` function gets the parameter `my_parameter` from the node,

and stores it in `my_param`. Next the `get_logger` function ensures the event is logged. The `set_parameters` function then sets the parameter `my_parameter` back to the default string value `world`. In the case that the user changed the parameter externally, this ensures it is always reset back to the original.

```
C.R. 96
python

1 def timer_callback(self):
2     my_param = self.get_parameter('my_parameter').get_parameter_value().string_value
3
4     self.get_logger().info('Hello %s!' % my_param)
5
6     my_new_param = rclpy.parameter.Parameter(
7         'my_parameter',
8         rclpy.Parameter.Type.STRING,
9         'world'
10    )
11    all_new_parameters = [my_new_param]
12    self.set_parameters(all_new_parameters)
```

Following the `timer_callback` is our main. Here ROS is initialized, an instance of the `MinimalParam` class is constructed, and `rclpy.spin` starts processing data from the node.

```
C.R. 97
python

1 def main():
2     rclpy.init()
3     node = MinimalParam()
4     rclpy.spin(node)
5
6 if __name__ == '__main__':
7     main()
```

(Optional) Adding a Parameter Descriptor Optionally, we can set a descriptor for the parameter. Descriptors allow us to specify a text description of the parameter and its constraints, like making it read-only, specifying a range, etc. For that to work, the `__init__` code has to be changed to:

```
C.R. 98
python

1 # ...
2
3 class MinimalParam(rclpy.node.Node):
4     def __init__(self):
5         super().__init__('minimal_param_node')
6
7         from rcl_interfaces.msg import ParameterDescriptor
8         my_parameter_descriptor = ParameterDescriptor(description='This parameter is
9             ↪ mine!')
10
11         self.declare_parameter('my_parameter', 'world', my_parameter_descriptor)
12
13         self.timer = self.create_timer(1, self.timer_callback)
```

The rest of the code remains the same. Once we run the node, we can then run `ros2 param describe`

/minimal_param_node my_parameter to see the type and description.

Adding an Entry Point Open the setup.py file. Again, match the maintainer, maintainer_email, description and license fields to our package.xml:

```
1 maintainer='YourName',
2 maintainer_email='you@email.com',
3 description='Python parameter tutorial',
4 license='Apache License 2.0',
```

C.R. 99
python

Add the following line within the console_scripts brackets of the entry_points field:

```
1 entry_points={
2     'console_scripts': [
3         'minimal_param_node = python_parameters.python_parameters_node:main',
4     ],
5 },
```

C.R. 100
python

Building and Running the Code It's good practice to run rosdep in the root of our workspace ( ros2_ws) to check for missing dependencies before building:

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 101
bash

Navigate back to the root of our workspace,  ros2_ws, and build our new package:

```
1 colcon build --packages-select python_parameters
```

C.R. 102
bash

Open a new terminal, navigate to  ros2_ws, and source the setup files:

```
1 source install/setup.bash
```

C.R. 103
bash

Now run the node. The terminal should return Hello world! every second:

```
1 ros2 run python_parameters minimal_param_node
```

C.R. 104
bash

```
1 [INFO] [parameter_node]: Hello world!
```

text

Now we can see the default value of our parameter, but we want to be able to set it ourselves. There are two ways to accomplish this.

Changing Output using the Console Make sure the node is running:

```
1 ros2 run python_parameters minimal_param_node
```

C.R. 105
bash

Open another terminal, source the setup files from inside  `ros2_ws` again, and enter the following line:

```
1 ros2 param list
```

C.R. 106
bash

There we will see the custom parameter `my_parameter`. To change it, simply run the following line in the console:

```
1 ros2 param set /minimal_param_node my_parameter earth
```

C.R. 107
bash

We know it went well if we get the output Set parameter successful. If we look at the other terminal, we should see the output change to `[INFO] [minimal_param_node]: Hello earth!`

Since the node afterwards set the parameter back to world, further outputs show:

`[INFO] [minimal_param_node]: Hello world!`

Changing using the Launch File We can also set parameters in a launch file, but first we will need to add a launch directory. Inside the `ros2_ws/src/python_parameters/` directory, create a new directory called `launch`. In there, create a new file called `python_parameters_launch.py`

```
1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4
5 def generate_launch_description():
6     return LaunchDescription([
7         Node(
8             package='python_parameters',
9             executable='minimal_param_node',
10            name='custom_minimal_param_node',
11            output='screen',
12            emulate_tty=True,
13            parameters=[
14                {'my_parameter': 'earth'}
15            ]
16        )
17    ])
```

C.R. 108
python

Here we can see that we set `my_parameter` to `earth` when we launch our node `parameter_node`. By adding the two lines below, we ensure our output is printed in our console.

```
1 output="screen",
2 emulate_tty=True,
```

C.R. 109
bash

Now open the setup.py file. Add the import statements to the top of the file, and the other new statement to the `data_files` parameter to include all launch files:

```
1 import os
2 from glob import glob
3 # ...
4
5 setup(
6     # ...
7     data_files=[
8         # ...
9         (os.path.join('share', package_name, 'launch'), glob('launch/*')),
10    ]
11 )
```

C.R. 110
bash

Open a console and navigate to the root of our workspace,  `ros2_ws`, and build our new package:

```
1 colcon build --packages-select python_parameters
```

C.R. 111
bash

Then source the setup files in a new terminal:

```
1 source install/setup.bash
```

C.R. 112
bash

Now run the node using the launch file we have just created. The terminal should return the following message the first time:

```
1 ros2 launch python_parameters python_parameters_launch.py
```

C.R. 113
bash

```
1 [INFO] [custom_minimal_param_node]: Hello earth!
```

text

Further outputs should show `[INFO] [minimal_param_node]: Hello world!` every second.

8.8 Managing Dependencies

8.8.1 Explaining Rosdep

rosdep is a dependency management utility that can work with packages and external libraries. It is a command-line utility for identifying and installing dependencies to build or install a package. rosdep is not a package manager in its own right; it is a meta-package manager that uses its own knowledge of the system and the dependencies to find the appropriate package to install on a particular platform. The actual installation is done using the system package manager (e.g. apt on Debian/Ubuntu, dnf on Fedora/RHEL, etc).

It is most often invoked before building a workspace, where it is used to install the dependencies of the packages within that workspace.

It has the ability to work over a single package or over a directory of packages (e.g. workspace).

While the name suggests it is for ROS, rosdep is semi-agnostic to ROS. We can utilize this powerful tool in non-ROS software projects by installing it as a standalone Python package. Successfully running rosdep relies on rosdep keys to be available, which can be downloaded from a public git repository with a few simple commands.

8.8.2 Explaining Pacakge Manifesto

The package.xml is the file in our software where rosdep finds the set of dependencies. It is important that the list of dependencies in the package.xml is complete and correct, which allows all of the tooling to determine the packages dependencies. Missing or incorrect dependencies can lead to users not being able to use our package, to packages in a workspace being built out-of-order, and to packages not being able to be released.

The dependencies in the package.xml file are generally referred to as "rosdep keys". These dependencies are manually populated in the package.xml file by the package's creators and should be an exhaustive list of any non-builtin libraries and packages it requires.

These are represented in the following tags (see REP-149 for the full specification):

<depend> These are dependencies that should be provided at both build time and run time for our package. For C++ packages, if in doubt, use this tag. Pure Python packages generally don't have a build phase, so should never use this and should use **<exec_depend>** instead.

<build_depend> If we only use a particular dependency for building our package, and not at execution time, we can use the **<build_depend>** tag.

With this type of dependency, an installed binary of our package does not require that particular

package to be installed.

However, that can create a problem if our package exports a header that includes a header from this dependency. In that case we also need a `<build_export_depend>`.

8.9 Creating an Action

We learned about actions previously in the Understanding actions tutorial. Like the other communication types and their respective interfaces (topics/msg and services/srv), we can also custom-define actions in our packages. This tutorial shows you how to define and build an action that we can use with the action server and action client we will write in the next tutorial.

Before we start, let's setup everything and make sure the requisites are in order:

```
1 mkdir -p ros2_ws/src # you can reuse an existing workspace
2 cd ros2_ws/src
3 ros2 pkg create action_tutorials_interfaces
```

C.R. 114

bash

Defining an Action Actions are defined in `.action` files of the form:

```
1 # Request
2 ---
3 # Result
4 ---
5 # Feedback
```

C.R. 115

text

An action definition is made up of three message definitions separated by `---`.

A request message is sent from an action client to an action server initiating a new goal.

A result message is sent from an action server to an action client when a goal is done.

Feedback messages are periodically sent from an action server to an action client with updates about a goal.

An instance of an action is typically referred to as a goal.

Say we want to define a new action “Fibonacci” for computing the Fibonacci sequence.

Create an action directory in our ROS package `action_tutorials_interfaces`:

```
1 cd action_tutorials_interfaces
2 mkdir action
```

C.R. 116

bash

Within the action directory, create a file called `Fibonacci.action` with the following contents:

```
1 int32 order
2 ---
3 int32[] sequence
4 ---
```

C.R. 117

text

```
5 int32[] partial_sequence
```

C.R. 118
text

The goal request is the order of the Fibonacci sequence we want to compute, the result is the final sequence, and the feedback is the `partial_sequence` computed so far.

Building an Action Before we can use the new Fibonacci action type in our code, we must pass the definition to the rosidl code generation pipeline.

This is accomplished by adding the following lines to our CMakeLists.txt before the `ament_package()` line, in the `action_tutorials_interfaces`:

```
1 find_package(rosidl_default_generators REQUIRED)
2
3 rosidl_generate_interfaces(${PROJECT_NAME}
4   "action/Fibonacci.action"
5 )
```

C.R. 119
cmake

We should also add the required dependencies to our package.xml:

```
1 <buildtool_depend>rosidl_default_generators</buildtool_depend>
2 <depend>action_msgs</depend>
3 <member_of_group>rosidl_interface_packages</member_of_group>
```

C.R. 120
xml

Note, we need to depend on `action_msgs` since action definitions include additional metadata (e.g. goal IDs).

We should now be able to build the package containing the Fibonacci action definition:

```
1 cd ~/ros2_ws # Change to the root of the workspace
2 colcon build # Build
```

C.R. 121
bash

We're done!

By convention, action types will be prefixed by their package name and the word action. So when we want to refer to our new action, it will have the full name `action_tutorials_interfaces/action/Fibonacci`.

We can check that our action built successfully with the command line tool:

```
1 . install/setup.bash # Source our workspace.
2 # Check that our action definition exists
3 ros2 interface show action_tutorials_interfaces/action/Fibonacci
```

C.R. 122
bash

We should see the Fibonacci action definition printed to the screen.

8.10 Writing an Action Server and Client

Actions are a form of asynchronous communication in ROS. Action clients send goal requests to action servers. Action servers send goal feedback and results to action clients.

Writing an Action Server Let's focus on writing an action server that computes the Fibonacci sequence using the action we created in the Creating an action tutorial.

Until now, we've created packages and used ros2 run to run our nodes. To keep things simple in this tutorial, however, we'll scope the action server to a single file. If we'd like to see what a complete package for the actions tutorials looks like, check out [action_tutorials](#).

Open a new file in our home directory, let's call it `fibonacci_action_server.py`, and add the following code:

```

1 import rclpy
2 from rclpy.action import ActionServer
3 from rclpy.node import Node
4
5 from action_tutorials_interfaces.action import Fibonacci
6
7
8 class FibonacciActionServer(Node):
9
10     def __init__(self):
11         super().__init__('fibonacci_action_server')
12         self._action_server = ActionServer(
13             self,
14             Fibonacci,
15             'fibonacci',
16             self.execute_callback)
17
18     def execute_callback(self, goal_handle):
19         self.get_logger().info('Executing goal...')
20         result = Fibonacci.Result()
21         return result
22
23
24     def main(args=None):
25         rclpy.init(args=args)
26
27         fibonacci_action_server = FibonacciActionServer()
28
29         rclpy.spin(fibonacci_action_server)
30
31
32 if __name__ == '__main__':
33     main()

```

8.11 Writing a Launch File

The launch system in ROS is responsible for helping the user describe the [configuration of their system](#) and then [execute it as described](#). The configuration of the system includes:

- what programs to run, where to run them,
- what arguments to pass them, and
- ROS-specific conventions which make it easy to reuse components throughout the system by giving them each a different configuration

It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

Launch files written in [XML](#), [YAML](#), or [Python](#) can start and stop different nodes as well as trigger and act on various events.

The package providing this framework is `launch_ros`, which uses the non-ROS-specific launch framework underneath.

So now we got the preliminary information out of the way, let us start with working on writing our own launch file.

Creating a Launch Directory We start by creating a new directory to store our launch files:

```
1 mkdir launch
```

C.R. 124
bash

Writing our Launch File Let's put together a ROS launch file using the `turtlesim` package and its executables. As mentioned previously, this can either be in XML, YAML, or Python. However for the sake of coherence, we shall only look at the Python version of it.

Please copy and paste the code into [launch>turtlesim_mimic_launch.py](#) file:

```
1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4
5 def generate_launch_description():
6     return LaunchDescription([
7         Node(
8             package='turtlesim',
9             namespace='turtlesim1',
10            executable='turtlesim_node',
```

C.R. 125
python

```

11         name='sim'
12     ),
13     Node(
14         package='turtlesim',
15         namespace='turtlesim2',
16         executable='turtlesim_node',
17         name='sim'
18     ),
19     Node(
20         package='turtlesim',
21         executable='mimic',
22         name='mimic',
23         remappings=[
24             ('/input/pose', '/turtlesim1/turtle1/pose'),
25             ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
26         ]
27     )
28 )

```

C.R. 126
python

Understanding the Launch File All of the launch files above are launching a system of three (3) nodes, all from the turtlesim package. The goal of the system is to launch two (2) turtlesim windows, and have one turtle mimic the movements of the other.

When launching the two (2) turtlesim nodes, the only difference between them is their `namespace` values. Unique namespaces allow the system to start two (2) nodes without node name or topic name conflicts. Both turtles in this system receive commands over the same topic and publish their pose over the same topic.

With unique namespaces, messages meant for different turtles can be distinguished.

The final node is also from the turtlesim package, but a different executable:

`mimic`.

This node has added configuration details in the form of remappings. `mimic`'s `input>pose` topic is remapped to `turtlesim1>turtle1>pose` and it's `output>cmd_vel` topic to `turtlesim2>turtle1>cmd_vel`.

This means `mimic` will subscribe to `turtlesim1>sim` pose topic and republish it for `turtlesim2>sim` velocity command topic to subscribe to.

In other words, `turtlesim2` will mimic `turtlesim1`'s movements.

Now to see what is going on with the launch file. These import statements pull in some Python launch modules.

```
1 from launch import LaunchDescription
2 from launch_ros.actions import Node
```

C.R. 127

python

Next, the launch description itself begins:

```
1 def generate_launch_description():
2     return LaunchDescription([
3         Node(
4             ])
```

C.R. 128

python

The first two actions in the launch description launch the two turtlesim windows:

```
1     Node(
2         package='turtlesim',
3         namespace='turtlesim1',
4         executable='turtlesim_node',
5         name='sim'
6     ),
7     Node(
8         package='turtlesim',
9         namespace='turtlesim2',
10        executable='turtlesim_node',
11        name='sim'
12    ),
```

C.R. 129

python

The final action launches the mimic node with the remaps:

```
1     Node(
2         package='turtlesim',
3         executable='mimic',
4         name='mimic',
5         remappings=[
6             ('/input/pose', '/turtlesim1/turtle1/pose'),
7             ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
8         ]
9     )
```

C.R. 130

python

Launching the Package To run the launch file created above, enter into the directory we created earlier and run the following command:

```
1 cd launch
2 ros2 launch turtlesim_mimic_launch.py
```

C.R. 131

bash

Two turtlesim windows will open, and we will see the following [INFO] messages telling us which nodes our launch file has started:

```

1 [INFO] [launch]: Default logging verbosity is set to INFO
2 [INFO] [turtlesim_node-1]: process started with pid [11714]
3 [INFO] [turtlesim_node-2]: process started with pid [11715]
4 [INFO] [mimic-3]: process started with pid [11716]

```

C.R. 132
text

To see the system in action, open a new terminal and run the ros2 topic pub command on the `turtlesim1` `>> turtle1` `>> cmd_vel` topic to get the first turtle moving:

```

1 ros2 topic pub -r 1 \
2   /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist \
3   "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}"

```

C.R. 133
bash

We will see both turtles following the same path.

Checking the Graph While the system is still running, open a new terminal and run `rqt_graph` to get a better idea of the relationship between the nodes in our launch file.

Run the command:

```
1 rqt_graph
```

C.R. 134
bash

A hidden node (the ros2 topic pub command we ran) is publishing data to the `turtlesim1` `>> turtle1` `>> cmd_vel` topic on the left, which the `turtlesim1` `>> sim` node is subscribed to. The rest of the graph shows what was described earlier: mimic is subscribed to `turtlesim1` `>> sim`'s pose topic, and publishes to `turtlesim2` `>> sim`'s velocity command topic.

Chapter 9

Transform Library

Table of Contents

9.1	A Gentle Introduction	221
9.2	Writing a Static Broadcaster	225
9.3	Writing a Listener	230
9.4	Adding a Frame	235
9.5	Writing a Broadcaster	242

9.1 A Gentle Introduction

`tf2` is the [transform library](#), which lets the user keep track of multiple coordinate frames over time. `tf2` maintains the relationship between coordinate frames in a tree structure buffered in time and lets the user transform points, vectors, etc., between any two (2) coordinate frames at any desired point in time.

A robotic system typically various types of 3D coordinate frames which change over time.

Examples include a world frame, base frame, gripper frame, head frame, etc.

The key use-case for `tf2` is to keep track of all these frames over time, and allows us to ask questions like:

- Where was the head frame relative to the world frame 5 seconds ago?

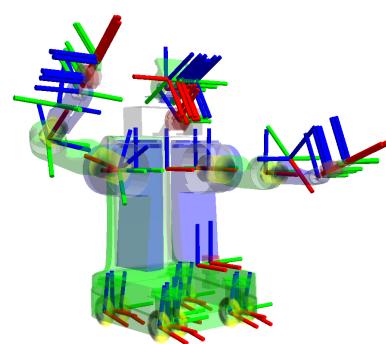


Figure 9.1: A robot is comprised of numerous coordinate systems as can be seen from this robot and needs to be constantly checked which `tf2` allows [ros2iron2025].

- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

`tf2` can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to **all** ROS components on any computer in the system. In addition, `tf2` can have every component in your distributed system build its own transform information database or have a central node that gathers and stores all transform information.

A Simple Test

To get started with the `tf2` library, we are going to run a demo and see some of the power of `tf2` in a multi-robot example using turtlesim, which we should already be familiar with.

Installing the Demo We will start by installing the demo package and its dependencies, like we did previously. This might have been already installed depending on the way ROS was installed, but it doesn't hurt to run the command again:

```
1 sudo apt-get install \
2   ros-humble-rviz2 \
3   ros-humble-turtle-tf2-py \
4   ros-humble-tf2-ros \
5   ros-humble-tf2-tools \
6   ros-humble-turtlesim
```

C.R. 1

bash

Now that we've installed the `turtle_tf2_py` tutorial package let's run the demo. First, open a **new terminal** and source your ROS installation so that `ros2` commands will work.¹ Then run the following command:

```
1 ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

C.R. 2

bash

You will see the turtlesim start with two (2) turtles. In the **second terminal** window type the following command:

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 3

bash

Once the turtlesim is started you can drive the central turtle around in the turtlesim using the keyboard arrow keys, select the second terminal window so that our keystrokes will be captured to drive the turtle.

You can see that one turtle continuously moves to follow the turtle you are driving around.

¹While the docker container setup has the source predefined, it is nevertheless a good practice to do this if you are working with multiple versions.

Looking at the Behaviour This demo is using the `tf2` library to create three (3) coordinate frames:

1. a world frame,
2. a `turtle1` frame, and
3. a `turtle2` frame.

In this tutorial, we use a `tf2 broadcaster` to publish the turtle coordinate frames and a `tf2 listener` to calculate the difference in the turtle frames and move one turtle to follow the other.

Useful Tools

It is time to see how `tf2` is being used to create this simple exercise. We can use `tf2_tools` to look at what `tf2` is doing behind the scenes.

Viewing the Frames If we want to see visually what is going on, we can use `view_frames` which creates a diagram of the frames being broadcast by `tf2` over ROS.

As of this writing `view_frames` only works on Linux.

```
1 ros2 run tf2_tools view_frames
1 Listening to tf data during 5 seconds...
2 Generating graph in frames.pdf file...
```

C.R. 4

bash

text

Here a `tf2` listener is listening to the frames which are being broadcast over ROS and drawing a tree of how the frames are connected.

If we want to view the tree, open the resulting  `frames.pdf` with a PDF viewer.

Here we can see three (3) frames that are broadcast by `tf2`:

`world`, `turtle1`, and `turtle2`.

The `world` frame is the parent of the `turtle1` and `turtle2` frames. `view_frames` also reports some diagnostic information about when the oldest and most recent frame transforms were received and how fast the `tf2` frame is published to `tf2` for debugging purposes.

Using Echo `tf2_echo` reports the transform between any two (2) frames broadcast over ROS. The syntax of this command is as follows:

```
1 ros2 run tf2_ros tf2_echo [source_frame] [target_frame]
```

C.R. 5

bash

Let's look at the transform of the `turtle2` frame with respect to `turtle1` frame which is equivalent to:

```
1 ros2 run tf2_ros tf2_echo turtle2 turtle1
```

C.R. 6

bash

```
1 At time 1683385337.850619099
2 - Translation: [2.157, 0.901, 0.000]
3 - Rotation: in Quaternion [0.000, 0.000, 0.172, 0.985]
4 - Rotation: in RPY (radian) [0.000, -0.000, 0.345]
5 - Rotation: in RPY (degree) [0.000, -0.000, 19.760]
6 - Matrix:
7   0.941 -0.338  0.000  2.157
8   0.338  0.941  0.000  0.901
9   0.000  0.000  1.000  0.000
10  0.000  0.000  0.000  1.000
11 At time 1683385338.841997774
12 - Translation: [1.256, 0.216, 0.000]
13 - Rotation: in Quaternion [0.000, 0.000, -0.016, 1.000]
14 - Rotation: in RPY (radian) [0.000, 0.000, -0.032]
15 - Rotation: in RPY (degree) [0.000, 0.000, -1.839]
16 - Matrix:
17   0.999  0.032  0.000  1.256
18   -0.032  0.999 -0.000  0.216
19   0.000  0.000  1.000  0.000
20   0.000  0.000  0.000  1.000
```

text

We will see the transform displayed as the `tf2_echo` listener receives the frames broadcast over ROS. As we drive our turtle around we will see the transform change as the two (2) turtles move relative to each other.

rviz2 and tf2 `rviz2` is a three-dimensional visualization platform in ROS. On the one hand, it can realize the graphical display of external information, and on the other hand, it can also release control information to the object through `rviz`, so as to realize the monitoring and control of the robot. It is also useful for examining `tf2` frames. Let's look at our turtle frames using `rviz2` by starting it with a configuration file using the `-d` option:

```
1 ros2 run rviz2 rviz2 -d \
2   $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz
```

C.R. 7

bash

In the side bar you will see the frames broadcasted by `tf2`. As you drive the turtle around you will see the frames move in `rviz`.

9.2 Writing a Static Broadcaster

Publishing static transforms is useful to define the relationship between a robot base and its sensors or non-moving parts.

For example, it is easiest to reason about laser scan measurements in a frame at the centre of the laser scanner.

In this section, we will be covering the basics of static transforms, which consists of two (2) parts:

- In the first part we will write code to publish static transforms to tf2.
- In the second part we will explain how to use the command-line `static_transform_publisher` executable tool in `tf2_ros`.

In the next two (2) sections we will write the code to **reproduce** the exercise we did at the beginning of the chapter.

Creating the Package As with almost all ROS applications, we first need to create a package will be used for this section and the following ones. The package called `learning_tf2_py` will depend on:

`geometry_msgs`, `python3-numpy`, `rclpy`, `tf2_ros_py`, and `turtlesim`

This package will be used in the following sections as well.

Open a new terminal and source your ROS installation so that `ros2` commands will work. Navigate to workspace's `src` folder and create a new package:

```
1 ros2 pkg create --build-type ament_python \
2   --license Apache-2.0 \
3   -- learning_tf2_py
```

C.R. 8
bash

Once the command has been executed, our terminal will return a message verifying the creation of our package `learning_tf2_py` and all its necessary files and folders.

Writing a Static Broadcaster Node We will start, of course, by first creating the source files. Within the `src` `learning_tf2_py` `learning_tf2_py` directory download the example static broadcaster code by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/ \
→ turtle_tf2_py/static_turtle_tf2_broadcaster.py
```

C.R. 9
bash

Now let's open the file called `static_turtle_tf2_broadcaster.py` and have a look at it.

Looking at the Code Now let's look at the code that is relevant to publishing the static turtle pose to `tf2`. The first lines import required packages. First we import the `TransformStamped` from the `geometry_msgs`, which provides us a template for the message that we will publish to the transformation tree.

```
1  from geometry_msgs.msg import TransformStamped
```

C.R. 10

python

Afterwards, `rclpy` is imported so its Node class can be used.

```
1  import rclpy
2  from rclpy.node import Node
```

C.R. 11

python

The `tf2_ros` package provides a `StaticTransformBroadcaster` to make the publishing of static transforms easy. To use the `StaticTransformBroadcaster`, we need to import it from the `tf2_ros` module.

```
1  from tf2_ros.static_transform_broadcaster import StaticTransformBroadcaster
```

C.R. 12

python

The `StaticFramePublisher` class constructor initializes the node with the name `static_turtle_tf2_broadcaster`. Then, `StaticTransformBroadcaster` is created, which will send one static transformation upon the startup.

```
1  self.tf_static_broadcaster = StaticTransformBroadcaster(self)
2  self.make_transforms(transformation)
```

C.R. 13

python

Here we create a `TransformStamped` object, which will be the message we will send over once populated. Before passing the actual transform values we need to give it the appropriate metadata.

- We need to give the transform being published a timestamp and well just stamp it with the current time, `self.get_clock().now()`
- Then we need to set the name of the parent frame of the link were creating, in this case `world`.
- Finally, we need to set the name of the child frame of the link were creating

```
1  t = TransformStamped()
2
3  t.header.stamp = self.get_clock().now().to_msg()
4  t.header.frame_id = 'world'
5  t.child_frame_id = transformation[1]
```

C.R. 14

python

Here we populate the 6D pose (translation and rotation) of the turtle.

```

1 t.transform.translation.x = float(transformation[2])
2 t.transform.translation.y = float(transformation[3])
3 t.transform.translation.z = float(transformation[4])
4 quat = quaternion_from_euler(
5     float(transformation[5]), float(transformation[6]), float(transformation[7]))
6 t.transform.rotation.x = quat[0]
7 t.transform.rotation.y = quat[1]
8 t.transform.rotation.z = quat[2]
9 t.transform.rotation.w = quat[3]

```

C.R. 15
python

Finally, we broadcast static transform using the `sendTransform()` function.

```

1 self.tf_static_broadcaster.sendTransform(t)

```

C.R. 16
python

Updating the Package Information Navigate one level back to the `src>learning_tf2_py` directory, where the `setup.py`, `setup.cfg`, and `package.xml` files have been created for you.

Open `package.xml` and make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

```

1 <description>Learning tf2 with rclpy</description>
2 <maintainer email="you@email.com">Your Name</maintainer>
3 <license>Apache License 2.0</license>

```

C.R. 17
xml

After the lines above, add the following dependencies corresponding to your nodes import statements:

```

1 <exec_depend>geometry_msgs</exec_depend>
2 <exec_depend>python3-numpy</exec_depend>
3 <exec_depend>rclpy</exec_depend>
4 <exec_depend>tf2_ros_py</exec_depend>
5 <exec_depend>turtlesim</exec_depend>

```

C.R. 18
xml

This declares the required `geometry_msgs`, `python3-numpy`, `rclpy`, `tf2_ros_py`, and `turtlesim` dependencies when its code is executed.

Adding an Entry Point To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py`. Add the following line between the '`console_scripts':` brackets:

```

1 'static_turtle_tf2_broadcaster = learning_tf2_py.static_turtle_tf2_broadcaster:main', python

```

C.R. 19

Build Its good practice to run `rosdep` in the root of your workspace to check for missing dependencies before building:

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 20
bash

Still in the root of your workspace, build your new package:

```
1 colcon build --packages-select learning_tf2_py
```

C.R. 21
bash

Open a new terminal, navigate to the root of your workspace, and source the setup files:

```
1 . install/setup.bash
```

C.R. 22
bash

Run Now run the `static_turtle_tf2_broadcaster` node:

```
1 ros2 run \
2     learning_tf2_py static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0
```

C.R. 23
bash

This sets a turtle pose broadcast for `mystaticturtle` to float 1 meter above the ground.

We can now check that the static transform has been published by echoing the `tf_static` topic If everything is well you should see a single static transform:

```
1 ros2 topic echo /tf_static
```

C.R. 24
bash

```
1 transforms:
2 - header:
3     stamp:
4         sec: 1622908754
5         nanosec: 208515730
6     frame_id: world
7     child_frame_id: mystaticturtle
8     transform:
9         translation:
10            x: 0.0
11            y: 0.0
12            z: 1.0
13         rotation:
14            x: 0.0
15            y: 0.0
16            z: 0.0
17            w: 1.0
```

text

The Proper way to Publish Static Transforms

This tutorial aimed to show how `StaticTransformBroadcaster` can be used to publish static transforms. In your real development process you shouldnt have to write this code yourself

and should use the dedicated `tf2_ros` tool to do so. `tf2_ros` provides an executable named `static_transform_publisher` that can be used either as a commandline tool or a node that you can add to your launchfiles.

The following command publishes a static coordinate transform to `tf2` resulting in a 1 meter offset in z and no rotation between the frames world and mystaticturtle. In ROS 2, roll/pitch/yaw refers to rotation in radians about the x/y/z-axis, respectively.

```
1 ros2 run tf2_ros \
2   static_transform_publisher \
3     --x 0 --y 0 --z 1 \
4     --yaw 0 --pitch 0 --roll 0 \
5     --frame-id world --child-frame-id mystaticturtle
```

C.R. 25
bash

The following command publishes the same static coordinate transform to `tf2`, but using quaternion representation for the rotation.

```
1 ros2 run tf2_ros \
2   static_transform_publisher \
3     --x 0 --y 0 --z 1 \
4     --qx 0 --qy 0 --qz 0 --qw 1 \
5     --frame-id world --child-frame-id mystaticturtle
```

C.R. 26
bash

`static_transform_publisher` is designed both as a command-line tool for manual use, as well as for use within launch files for setting static transforms. For example:

```
1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4
5 def generate_launch_description():
6   return LaunchDescription([
7     Node(
8       package='tf2_ros',
9       executable='static_transform_publisher',
10      arguments=[
11        '--x', '0', '--y', '0', '--z', '1',
12        '--yaw', '0', '--pitch', '0', '--roll',
13        '0', '--frame-id', 'world', '--child-frame-id', 'mystaticturtle']
14      ),
15    ])
```

C.R. 27
python

Note that all arguments except for `--frame-id` and `--child-frame-id` are optional; if a particular option isn't specified, then the identity will be assumed.

9.3 Writing a Listener

Writing the Listener Node Lets first create the source files. Go to the `learning_tf2_py` package we created in the previous tutorial. Inside the `src>learning_tf2_py>learning_tf2_py` directory download the example listener code by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/] C.R. 28  
↪ turtle_tf2_py/turtle_tf2_listener.py bash
```

Now open the file called `turtle_tf2_listener.py` using your preferred text editor.

```
1 import math C.R. 29  
2  
3 from geometry_msgs.msg import Twist python  
4  
5 import rclpy  
6 from rclpy.node import Node  
7  
8 from tf2_ros import TransformException  
9 from tf2_ros.buffer import Buffer  
10 from tf2_ros.transform_listener import TransformListener  
11  
12 from turtlesim.srv import Spawn  
13  
14  
15 class FrameListener(Node):  
16  
17     def __init__(self):  
18         super().__init__('turtle_tf2_frame_listener')  
19  
20         # Declare and acquire `target_frame` parameter  
21         self.target_frame = self.declare_parameter(  
22             'target_frame', 'turtle1').get_parameter_value().string_value  
23  
24         self.tf_buffer = Buffer()  
25         self.tf_listener = TransformListener(self.tf_buffer, self)  
26  
27         # Create a client to spawn a turtle  
28         self.spawner = self.create_client(Spawn, 'spawn')  
29         # Boolean values to store the information  
30         # if the service for spawning turtle is available  
31         self.turtle_spawning_service_ready = False  
32         # if the turtle was successfully spawned  
33         self.turtle_spawned = False  
34  
35         # Create turtle2 velocity publisher  
36         self.publisher = self.create_publisher(Twist, 'turtle2/cmd_vel', 1)  
37  
38         # Call on_timer function every second  
39         self.timer = self.create_timer(1.0, self.on_timer)
```

C.R. 30

python

```

40
41     def on_timer(self):
42         # Store frame names in variables that will be used to
43         # compute transformations
44         from_frame_rel = self.target_frame
45         to_frame_rel = 'turtle2'
46
47         if self.turtle_spawning_service_ready:
48             if self.turtle_spawned:
49                 # Look up for the transformation between target_frame and turtle2 frames
50                 # and send velocity commands for turtle2 to reach target_frame
51                 try:
52                     t = self.tf_buffer.lookup_transform(
53                         to_frame_rel,
54                         from_frame_rel,
55                         rclpy.time.Time())
56                 except TransformException as ex:
57                     self.get_logger().info(
58                         f'Could not transform {to_frame_rel} to {from_frame_rel}: {ex}')
59                 return
60
61                 msg = Twist()
62                 scale_rotation_rate = 1.0
63                 msg.angular.z = scale_rotation_rate * math.atan2(
64                     t.transform.translation.y,
65                     t.transform.translation.x)
66
67                 scale_forward_speed = 0.5
68                 msg.linear.x = scale_forward_speed * math.sqrt(
69                     t.transform.translation.x ** 2 +
70                     t.transform.translation.y ** 2)
71
72                 self.publisher.publish(msg)
73             else:
74                 if self.result.done():
75                     self.get_logger().info(
76                         f'Successfully spawned {self.result.result().name}')
77                     self.turtle_spawned = True
78                 else:
79                     self.get_logger().info('Spawn is not finished')
80             else:
81                 if self.spawner.service_is_ready():
82                     # Initialize request with turtle name and coordinates
83                     # Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
84                     request = Spawn.Request()
85                     request.name = 'turtle2'
86                     request.x = float(4)
87                     request.y = float(2)
88                     request.theta = float(0)
89                     # Call request
90                     self.result = self.spawner.call_async(request)
91                     self.turtle_spawning_service_ready = True
92             else:

```

```

93     # Check if the service is ready
94     self.get_logger().info('Service is not ready')
95
96
97 def main():
98     rclpy.init()
99     node = FrameListener()
100    try:
101        rclpy.spin(node)
102    except KeyboardInterrupt:
103        pass
104
105    rclpy.shutdown()

```

C.R. 31
python

Examining the Code Now, lets take a look at the code that is relevant to get access to frame transformations. The `tf2_ros` package provides an implementation of a `TransformListener` to help make the task of receiving transforms easier.

```

1 from tf2_ros.transform_listener import TransformListener

```

C.R. 32
python

Here, we create a `TransformListener` object. Once the listener is created, it starts receiving tf2 transformations over the wire, and buffers them for up to 10 seconds.

```

1 self.tf_listener = TransformListener(self.tf_buffer, self)

```

C.R. 33
python

Finally, we query the listener for a specific transformation. We call `lookup_transform` method with following arguments:

- Target frame
- Source frame
- The time at which we want to transform

Providing `rclpy.time.Time()` will just get us the latest available transform. All this is wrapped in a try-except block to handle possible exceptions.

Adding an Entry Point To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the [src>learning_tf2_py directory](#)).

Add the following line between the '`console_scripts`': brackets:

```

1 'turtle_tf2_listener = learning_tf2_py.turtle_tf2_listener:main',

```

C.R. 34
python

Updating the Launch Files Open the launch file called `turtle_tf2_demo.launch.py` in the `src>learning_tf2_py>launch` directory with your text editor, add two new nodes to the launch description, add a launch argument, and add the imports. The resulting file should look like:

```

C.R. 35
python

1 from launch import LaunchDescription
2 from launch.actions import DeclareLaunchArgument
3 from launch.substitutions import LaunchConfiguration
4
5 from launch_ros.actions import Node
6
7
8 def generate_launch_description():
9     return LaunchDescription([
10         Node(
11             package='turtlesim',
12             executable='turtlesim_node',
13             name='sim'
14         ),
15         Node(
16             package='learning_tf2_py',
17             executable='turtle_tf2_broadcaster',
18             name='broadcaster1',
19             parameters=[
20                 {'turtlename': 'turtle1'}
21             ]
22         ),
23         DeclareLaunchArgument(
24             'target_frame', default_value='turtle1',
25             description='Target frame name.'
26         ),
27         Node(
28             package='learning_tf2_py',
29             executable='turtle_tf2_broadcaster',
30             name='broadcaster2',
31             parameters=[
32                 {'turtlename': 'turtle2'}
33             ]
34         ),
35         Node(
36             package='learning_tf2_py',
37             executable='turtle_tf2_listener',
38             name='listener',
39             parameters=[
40                 {'target_frame': LaunchConfiguration('target_frame')}
41             ]
42         ),
43     ])

```

This will declare a `target_frame` launch argument, start a broadcaster for second turtle that we will spawn and listener that will subscribe to those transformations.

Build Run `rosdep` in the root of your workspace to check for missing dependencies.

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 36
bash

Still in the root of your workspace, build your package:

```
1 colcon build --packages-select learning_tf2_py
```

C.R. 37
bash

Open a new terminal, navigate to the root of your workspace, and source the setup files:

```
1 . install/setup.bash
```

C.R. 38
bash

Run Now you're ready to start your full turtle demo:

```
1 ros2 launch learning_tf2_py turtle_tf2_demo.launch.py
```

C.R. 39
bash

You should see the turtle sim with two turtles. In the second terminal window type the following command:

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 40
bash

To see if things work, simply drive around the first turtle using the arrow keys (make sure your terminal window is active, not your simulator window), and you'll see the second turtle following the first one!

9.4 Adding a Frame

Previously, we've recreated the turtle demo by writing a tf2 broadcaster and a tf2 listener. This tutorial will teach you how to add extra fixed and dynamic frames to the transformation tree. In fact, adding a frame in tf2 is very similar to creating the tf2 broadcaster, but this example will show you some additional features of tf2.

For many tasks related to transformations, it is easier to think inside a local frame. For example, it is easiest to reason about laser scan measurements in a frame at the center of the laser scanner. tf2 allows you to define a local frame for each sensor, link, or joint in your system. When transforming from one frame to another, tf2 will take care of all the hidden intermediate frame transformations that are introduced.

Viewing the Tree tf2 builds up a tree structure of frames and, thus, does not allow a closed loop in the frame structure. This means that a frame only has one single parent, but it can have multiple children. Currently, our tf2 tree contains three frames: world, turtle1 and turtle2. The two turtle frames are children of the world frame. If we want to add a new frame to tf2, one of the three existing frames needs to be the parent frame, and the new one will become its child frame.

Writing the Fixed Frame Broadcaster In our turtle example, well add a new frame carrot1, which will be the child of the turtle1. This frame will serve as the goal for the second turtle.

Lets first create the source files. Go to the `learning_tf2_py` package we created in the previous tutorials. Inside the `src/learning_tf2_py/learning_tf2_py` directory download the fixed frame broadcaster code by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/_ bash
  ↳ turtle_tf2_py/fixed_frame_tf2_broadcaster.py
```

C.R. 41

bash

Now open the file called `fixed_frame_tf2_broadcaster.py`.

```
1 from geometry_msgs.msg import TransformStamped C.R. 42
2
3 import rclpy
4 from rclpy.node import Node
5
6 from tf2_ros import TransformBroadcaster
7
8
9 class FixedFrameBroadcaster(Node):
10
11     def __init__(self):
12         super().__init__('fixed_frame_tf2_broadcaster')
13         self.tf_broadcaster = TransformBroadcaster(self)
```

python

```

14         self.timer = self.create_timer(0.1, self.broadcast_timer_callback)           C.R. 43
15
16     def broadcast_timer_callback(self):
17         t = TransformStamped()
18
19         t.header.stamp = self.get_clock().now().to_msg()
20         t.header.frame_id = 'turtle1'
21         t.child_frame_id = 'carrot1'
22         t.transform.translation.x = 0.0
23         t.transform.translation.y = 2.0
24         t.transform.translation.z = 0.0
25         t.transform.rotation.x = 0.0
26         t.transform.rotation.y = 0.0
27         t.transform.rotation.z = 0.0
28         t.transform.rotation.w = 1.0
29
30         self.tf_broadcaster.sendTransform(t)
31
32
33     def main():
34         rclpy.init()
35         node = FixedFrameBroadcaster()
36         try:
37             rclpy.spin(node)
38         except KeyboardInterrupt:
39             pass
40
41     rclpy.shutdown()

```

The code is very similar to the tf2 broadcaster tutorial example and the only difference is that the transform here does not change over time.

Examining the Code Lets take a look at the key lines in this piece of code. Here we create a new transform, from the parent turtle1 to the new child carrot1. The carrot1 frame is 2 meters offset in y axis in terms of the turtle1 frame.

```

1  t = TransformStamped()                                         C.R. 44
2
3  t.header.stamp = self.get_clock().now().to_msg()
4  t.header.frame_id = 'turtle1'
5  t.child_frame_id = 'carrot1'
6  t.transform.translation.x = 0.0
7  t.transform.translation.y = 2.0
8  t.transform.translation.z = 0.0

```

Adding an Entry Point To allow the ros2 run command to run your node, you must add the entry point to setup.py (located in the `src/learning_tf2_py` directory).

Add the following line between the '`console_scripts`': brackets:

```
1 'fixed_frame_tf2_broadcaster = learning_tf2_py.fixed_frame_tf2_broadcaster:main', C.R. 45
2 
```

python

Writing the Launch File Now lets create a launch file for this example. With your text editor, create a new file called `turtle_tf2_fixed_frame_demo.launch.py` in the `src > learning_tf2_py > launch` directory, and add the following lines:

```
1 from launch import LaunchDescription C.R. 46
2 from launch.actions import IncludeLaunchDescription
3 from launch.substitutions import PathJoinSubstitution
4 from launch_ros.actions import Node
5 from launch_ros.substitutions import FindPackageShare
6
7
8 def generate_launch_description():
9     return LaunchDescription([
10         IncludeLaunchDescription(
11             PathJoinSubstitution([
12                 FindPackageShare('learning_tf2_py'), 'launch',
13                 'turtle_tf2_demo.launch.py')
14         ),
15         Node(
16             package='learning_tf2_py',
17             executable='fixed_frame_tf2_broadcaster',
18             name='fixed_broadcaster',
19         ),
20     ])
21 
```

python

This launch file imports the required packages and then creates a `demo_nodes` variable that will store nodes that we created in the previous tutorials launch file.

The last part of the code will add our fixed carrot1 frame to the turtlesim world using our `fixed_frame_tf2_broadcaster` node.

```
1
2     Node(
3         package='learning_tf2_py',
4         executable='fixed_frame_tf2_broadcaster',
5         name='fixed_broadcaster',
6     ),
7 
```

C.R. 47
python

Build Run `rosdep` in the root of your workspace to check for missing dependencies.

```
1 rosdep install -i --from-path src --rosdistro humble -y C.R. 48
2 
```

bash

Still in the root of your workspace, build your package:

```
1 colcon build --packages-select learning_tf2_py
```

C.R. 49

bash

Open a new terminal, navigate to the root of your workspace, and source the setup files:

```
1 . install/setup.bash
```

C.R. 50

bash

Run Now you can start the turtle broadcaster demo:

```
1 ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo.launch.py
```

C.R. 51

bash

You should notice that the new `carrot1` frame appeared in the transformation tree.

If you drive the first turtle around, you should notice that the behavior didnt change from the previous tutorial, even though we added a new frame. Thats because adding an extra frame does not affect the other frames and our listener is still using the previously defined frames.

Therefore if we want our second turtle to follow the carrot instead of the first turtle, we need to change value of the `target_frame`. This can be done two ways. One way is to pass the `target_frame` argument to the launch file directly from the console:

```
1 ros2 launch learning_tf2_py \
2   turtle_tf2_fixed_frame_demo.launch.py \
3   target_frame:=carrot1
```

C.R. 52

bash

The second way is to update the launch file. To do so, open the `turtle_tf2_fixed_frame_demo.launch.py` file, and add the '`target_frame': 'carrot1'` parameter via `launch_arguments` argument.

```
1 def generate_launch_description():
2     demo_nodes = IncludeLaunchDescription(
3         ...
4         launch_arguments={'target_frame': 'carrot1'}).items(),
5     )
```

C.R. 53

python

Now rebuild the package, restart the `turtle_tf2_fixed_frame_demo.launch.py`, and youll see the second turtle following the carrot instead of the first turtle!

Writing the Dynamic Frame Broadcaster

The extra frame we published in this tutorial is a fixed frame that doesnt change over time in relation to the parent frame. However, if you want to publish a moving frame you can code the broadcaster to change the frame over time. Lets change our `carrot1` frame so that it changes relative to `turtle1` frame over time. Go to the `learning_tf2_py` package we created in the previous

tutorial. Inside the `src>learning_tf2_py>learning_tf2_py` directory download the dynamic frame broadcaster code by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/] bash
→ turtle_tf2_py/dynamic_frame_tf2_broadcaster.py
```

Now open the file called `dynamic_frame_tf2_broadcaster.py`:

```
1 import math
2
3 from geometry_msgs.msg import TransformStamped
4
5 import rclpy
6 from rclpy.node import Node
7
8 from tf2_ros import TransformBroadcaster
9
10
11 class DynamicFrameBroadcaster(Node):
12
13     def __init__(self):
14         super().__init__('dynamic_frame_tf2_broadcaster')
15         self.tf_broadcaster = TransformBroadcaster(self)
16         self.timer = self.create_timer(0.1, self.broadcast_timer_callback)
17
18     def broadcast_timer_callback(self):
19         seconds, _ = self.get_clock().now().seconds_nanoseconds()
20         x = seconds * math.pi
21
22         t = TransformStamped()
23         t.header.stamp = self.get_clock().now().to_msg()
24         t.header.frame_id = 'turtle1'
25         t.child_frame_id = 'carrot1'
26         t.transform.translation.x = 10 * math.sin(x)
27         t.transform.translation.y = 10 * math.cos(x)
28         t.transform.translation.z = 0.0
29         t.transform.rotation.x = 0.0
30         t.transform.rotation.y = 0.0
31         t.transform.rotation.z = 0.0
32         t.transform.rotation.w = 1.0
33
34         self.tf_broadcaster.sendTransform(t)
35
36
37 def main():
38     rclpy.init()
39     node = DynamicFrameBroadcaster()
40     try:
41         rclpy.spin(node)
42     except KeyboardInterrupt:
43         pass
```

45 `rclpy.shutdown()`

C.R. 56
python

Examining the Code Instead of a fixed definition of our x and y offsets, we are using the `sin()` and `cos()` functions on the current time so that the offset of `carrot1` is constantly changing.

```
1  seconds, _ = self.get_clock().now().seconds.nanoseconds()
2  x = seconds * math.pi
3  ...
4  t.transform.translation.x = 10 * math.sin(x)
5  t.transform.translation.y = 10 * math.cos(x)
```

C.R. 57
bash

Adding an Entry Point To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py` (located in the `src\learning_tf2_py` directory).

Add the following line between the '`console_scripts`' brackets:

1 `'dynamic_frame_tf2_broadcaster = learning_tf2_py.dynamic_frame_tf2_broadcaster:main'`, python

C.R. 58

Writing the Launch File To test this code, create a new launch file `turtle_tf2_dynamic_frame_demo.launch.py` in the `src\learning_tf2_py\launch` directory and paste the following code:

```
1  from launch import LaunchDescription
2  from launch.actions import IncludeLaunchDescription
3  from launch.substitutions import PathJoinSubstitution
4  from launch_ros.actions import Node
5  from launch_ros.substitutions import FindPackageShare
6
7
8  def generate_launch_description():
9      return LaunchDescription([
10         IncludeLaunchDescription(
11             PathJoinSubstitution([
12                 FindPackageShare('learning_tf2_py'), 'launch',
13                 'turtle_tf2_demo.launch.py']),
14             launch_arguments={'target_frame': 'carrot1}.items(),
15         ),
16         Node(
17             package='learning_tf2_py',
18             executable='dynamic_frame_tf2_broadcaster',
19             name='dynamic_broadcaster',
20         ),
21     ])
```

C.R. 59
python

Build Run `rosdep` in the root of your workspace to check for missing dependencies.

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 60
bash

Still in the root of your workspace, build your package:

```
1 colcon build --packages-select learning_tf2_py
```

C.R. 61
bash

Open a new terminal, navigate to the root of your workspace, and source the setup files:

```
1 . install/setup.bash
```

C.R. 62
bash

Run Now you can start the dynamic frame demo:

```
1 ros2 launch learning_tf2_py turtle_tf2_dynamic_frame_demo.launch.py
```

C.R. 63
bash

You should see that the second turtle is following the carrots position that is constantly changing.

9.5 Writing a Broadcaster

Writing the Broadcaster Node Lets first create the source files. Go to the `learning_tf2_py` package we created in the previous tutorial. Inside the `src>learning_tf2_py>learning_tf2_py` directory download the example broadcaster code by entering the following command:

```
1 wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/_      C.R. 64  
↪ turtle_tf2_py/turtle_tf2_broadcaster.py                                bash
```

Now open the file called `turtle_tf2_broadcaster.py` using your preferred text editor.

```
1 import math                                         C.R. 65  
2  
3 from geometry_msgs.msg import TransformStamped  
4  
5 import numpy as np  
6  
7 import rclpy  
8 from rclpy.node import Node  
9  
10 from tf2_ros import TransformBroadcaster  
11  
12 from turtlesim.msg import Pose  
13  
14  
15 def quaternion_from_euler(ai, aj, ak):  
16     ai /= 2.0  
17     aj /= 2.0  
18     ak /= 2.0  
19     ci = math.cos(ai)  
20     si = math.sin(ai)  
21     cj = math.cos(aj)  
22     sj = math.sin(aj)  
23     ck = math.cos(ak)  
24     sk = math.sin(ak)  
25     cc = ci*ck  
26     cs = ci*sk  
27     sc = si*ck  
28     ss = si*sk  
29  
30     q = np.empty((4, ))  
31     q[0] = cj*sc - sj*cs  
32     q[1] = cj*ss + sj*cc  
33     q[2] = cj*cs - sj*sc  
34     q[3] = cj*cc + sj*ss  
35  
36     return q  
37  
38  
39 class FramePublisher(Node):
```

C.R. 66

python

```

40
41     def __init__(self):
42         super().__init__('turtle_tf2_frame_publisher')
43
44         # Declare and acquire `turtlename` parameter
45         self.turtlename = self.declare_parameter(
46             'turtlename', 'turtle').get_parameter_value().string_value
47
48         # Initialize the transform broadcaster
49         self.tf_broadcaster = TransformBroadcaster(self)
50
51         # Subscribe to a turtle{1}{2}/pose topic and call handle_turtle_pose
52         # callback function on each message
53         self.subscription = self.create_subscription(
54             Pose,
55             f'/{self.turtlename}/pose',
56             self.handle_turtle_pose,
57             1)
58         self.subscription # prevent unused variable warning
59
60     def handle_turtle_pose(self, msg):
61         t = TransformStamped()
62
63         # Read message content and assign it to
64         # corresponding tf variables
65         t.header.stamp = self.get_clock().now().to_msg()
66         t.header.frame_id = 'world'
67         t.child_frame_id = self.turtlename
68
69         # Turtle only exists in 2D, thus we get x and y translation
70         # coordinates from the message and set the z coordinate to 0
71         t.transform.translation.x = msg.x
72         t.transform.translation.y = msg.y
73         t.transform.translation.z = 0.0
74
75         # For the same reason, turtle can only rotate around one axis
76         # and this why we set rotation in x and y to 0 and obtain
77         # rotation in z axis from the message
78         q = quaternion_from_euler(0, 0, msg.theta)
79         t.transform.rotation.x = q[0]
80         t.transform.rotation.y = q[1]
81         t.transform.rotation.z = q[2]
82         t.transform.rotation.w = q[3]
83
84         # Send the transformation
85         self.tf_broadcaster.sendTransform(t)
86
87
88     def main():
89         rclpy.init()
90         node = FramePublisher()
91         try:
92             rclpy.spin(node)

```

```

93     except KeyboardInterrupt:
94         pass
95
96     rclpy.shutdown()

```

C.R. 67

python

Examining the Code Now, lets take a look at the code that is relevant to publishing the turtle pose to tf2. Firstly, we define and acquire a single parameter `turtlename`, which specifies a turtle name, e.g. `turtle1` or `turtle2`.

```

1 self.turtlename = self.declare_parameter(
2     'turtlename', 'turtle').get_parameter_value().string_value

```

C.R. 68

python

Afterward, the node subscribes to topic `{self.turtlename}/pose` and runs function `handle_turtle_pose` on every incoming message.

```

1 self.subscription = self.create_subscription(
2     Pose,
3     f'/{self.turtlename}/pose',
4     self.handle_turtle_pose,
5     1)

```

C.R. 69

python

Now, we create a `TransformStamped` object and give it the appropriate metadata.

- We need to give the transform being published a timestamp, and well just stamp it with the current time by calling `self.get_clock().now()`. This will return the current time used by the Node.
- Then we need to set the name of the parent frame of the link were creating, in this case world.
- Finally, we need to set the name of the child node of the link were creating, in this case this is the name of the turtle itself.

The handler function for the turtle pose message broadcasts this turtles translation and rotation, and publishes it as a transform from frame world to frame `turtleX`.

```

1 t = TransformStamped()
2
3 # Read message content and assign it to
4 # corresponding tf variables
5 t.header.stamp = self.get_clock().now().to_msg()
6 t.header.frame_id = 'world'
7 t.child_frame_id = self.turtlename

```

C.R. 70

python

Here we copy the information from the 3D turtle pose into the 3D transform.

```

1 # Turtle only exists in 2D, thus we get x and y translation
2 # coordinates from the message and set the z coordinate to 0
3 t.transform.translation.x = msg.x
4 t.transform.translation.y = msg.y
5 t.transform.translation.z = 0.0
6
7 # For the same reason, turtle can only rotate around one axis
8 # and this why we set rotation in x and y to 0 and obtain
9 # rotation in z axis from the message
10 q = quaternion_from_euler(0, 0, msg.theta)
11 t.transform.rotation.x = q[0]
12 t.transform.rotation.y = q[1]
13 t.transform.rotation.z = q[2]
14 t.transform.rotation.w = q[3]

```

C.R. 71
python

Finally we take the transform that we constructed and pass it to the `sendTransform` method of the `TransformBroadcaster` that will take care of broadcasting.

```

1 # Send the transformation
2 self.tf_broadcaster.sendTransform(t)

```

C.R. 72
python

Adding an Entry Point To allow the `ros2 run` command to run your node, you must add the entry point to `setup.py`

Add the following line between the '`console_scripts`' brackets:

```

1 # Send the transformation
2 self.tf_broadcaster.sendTransform(t)

```

C.R. 73
python

Writing the Launch File Now create a launch file for this demo. Create a launch folder in the `src/learning_tf2_py` directory. With your text editor, create a new file called `turtle_tf2_demo.launch.py` in the launch folder, and add the following lines:

```

1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4
5 def generate_launch_description():
6     return LaunchDescription([
7         Node(
8             package='turtlesim',
9             executable='turtlesim_node',
10            name='sim'
11        ),
12        Node(
13            package='learning_tf2_py',

```

C.R. 74
python

```

14     executable='turtle_tf2_broadcaster',
15     name='broadcaster1',
16     parameters=[
17         {'turtlename': 'turtle1'}
18     ],
19 ),
20 ]
)

```

C.R. 75
python

Examining the Code First we import required modules from the launch and `launch_ros` packages. It should be noted that launch is a generic launching framework (not ROS 2 specific) and `launch_ros` has ROS 2 specific things, like nodes that we import here.

```

1 from launch import LaunchDescription
2 from launch_ros.actions import Node

```

C.R. 76
python

Now we run our nodes that start the turtlesim simulation and broadcast turtle1 state to the tf2 using our `turtle_tf2_broadcaster` node.

```

1     Node(
2         package='turtlesim',
3         executable='turtlesim_node',
4         name='sim'
5     ),
6     Node(
7         package='learning_tf2_py',
8         executable='turtle_tf2_broadcaster',
9         name='broadcaster1',
10        parameters=[
11            {'turtlename': 'turtle1'}
12        ],
13    )
)

```

C.R. 77
python

Adding Dependencies Navigate one level back to the `learning_tf2_py` directory, where the `setup.py`, `setup.cfg`, and `package.xml` files are located.

Open `package.xml` with your text editor. Add the following dependencies corresponding to your launch files import statements:

```

1 <exec_depend>launch</exec_depend>
2 <exec_depend>launch_ros</exec_depend>

```

C.R. 78
xml

This declares the additional required launch and `launch_ros` dependencies when its code is executed.

Make sure to save the file.

Updating Setup File Reopen setup.py and add the line so that the launch files from the `launch` folder will be installed. The `data_files` field should now look like this:

```
1 data_files=[  
2     ...  
3     (os.path.join('share', package_name, 'launch'), glob('launch/*')),  
4 ],
```

C.R. 79
python

Also add the appropriate imports at the top of the file:

```
1 import os  
2 from glob import glob
```

C.R. 80
python

Build Run rosdep in the root of your workspace to check for missing dependencies.

```
1 rosdep install -i --from-path src --rosdistro humble -y
```

C.R. 81
bash

Still in the root of your workspace, build your package:

```
1 colcon build --packages-select learning_tf2_py
```

C.R. 82
bash

Open a new terminal, navigate to the root of your workspace, and source the setup files:

```
1 . install/setup.bash
```

C.R. 83
bash

Run Now run the launch file that will start the turtlesim simulation node and `turtle_tf2_broadcaster` node:

```
1 ros2 launch learning_tf2_py turtle_tf2_demo.launch.py
```

C.R. 84
bash

In the second terminal window type the following command:

```
1 ros2 run turtlesim turtle_teleop_key
```

C.R. 85
bash

You will now see that the turtlesim simulation has started with one turtle that you can control. Now, use the `tf2_echo` tool to check if the turtle pose is actually getting broadcast to tf2:

```
1 ros2 run tf2_ros tf2_echo world turtle1
```

C.R. 86
bash

This should show you the pose of the first turtle. Drive around the turtle using the arrow keys (make sure your `turtle_teleop_key` terminal window is active, not your simulator window). In your console output you will see something similar to this:

```
1 At time 1714913843.708748879                                         text
2 - Translation: [4.541, 3.889, 0.000]
3 - Rotation: in Quaternion [0.000, 0.000, 0.999, -0.035]
4 - Rotation: in RPY (radian) [0.000, -0.000, -3.072]
5 - Rotation: in RPY (degree) [0.000, -0.000, -176.013]
6 - Matrix:
7 -0.998  0.070  0.000  4.541
8 -0.070 -0.998  0.000  3.889
9  0.000  0.000  1.000  0.000
10 0.000  0.000  0.000  1.000
```

If you run `tf2_echo` for the transform between the world and turtle2, you should not see a transform, because the second turtle is not there yet. However, as soon as we add the second turtle in the next tutorial, the pose of turtle2 will be broadcast to tf2.

