Topics on Robotics & Vision

D. T. McGuiness, PhD

# B.Sc Mobile Robotics

## Lecture Book

2025.SS

# Contents

# List of Figures

# List of Tables

# Part I

# Mechanics of Mobile Robotics

# Chapter 1

# Locomotion

## Table of Contents



## 1.1 Introduction

An AMR needs locomotion mechanisms which enable it to move unbounded throughout its environment. However, as with everything in engineering our solution comes with options, and so the selection of a robot's approach to locomotion is an important aspect of mobile robot design. In laboratory settings, there are robots that can walk, jump, run, slide, swim, fly and of course roll.

> Most locomotion mechanisms have been inspired by biological counterparts, shown in **Fig.** 1.1.

There is, however, one (1) exception where there is, practically, **NO** natural equivalent:

> Actively powered wheel is a human invention achieving high efficiency on flat ground.

This mechanism is **NOT** completely foreign to biological systems[1]. Our bi-pedal walking system can be approximated by a rolling polygon, with sides equal in length to the span of the step. As the step size decreases, the polygon approaches a circle or wheel. But nature did not develop a fully rotating, actively powered joint, which is the technology necessary for wheeled locomotion.

[1] While this statement is practically true, single cell organism use a similar locomotion to what we call wheel [2].

Biological systems succeed in moving through a wide variety of harsh environments. Therefore it can be desirable to copy their selection of locomotion mechanisms[2]. Replicating nature in this regard, however, is extremely difficult for several reasons.

[2] Scientifically, this is called **Biomimetics** [3]

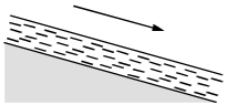
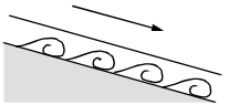
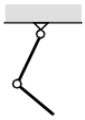
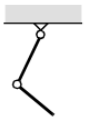
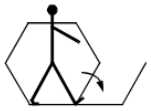
| Type of motion | Resistance to motion | Basic kinematics of motion |
|---|---|---|
| Flow in a Channel  | Hydrodynamic forces | Eddies  |
| Crawl  | Friction forces | Longitudinal vibration  |
| Sliding  | Friction forces | Transverse vibration  |
| Running  | Loss of kinetic energy | Oscillatory movement of a multi-link pendulum  |
| Jumping  | Loss of kinetic energy | Oscillatory movement of a multi-link pendulum  |
| Walking  | Gravitational forces | Rolling of a polygon (see figure 2.2)  |

**Figure 1.1:** Types of locomotion mechanisms used in biological systems [1].

- Mechanical complexity is easily achieved in biological systems through structural replication.

  Cell division, in combination with specialisation, can readily produce a millipede with several hundred legs and several tens of thousands of individually sensed cilia[3]. In man-made structures, each part must be fabricated individually, and therefore, no such economies of scale exist.

[3] short eyelash-like filament that is numerous on tissue cells of most animals and provides the means for locomotion of protozoans of the *phylum Ciliophora*

- Cell is a microscopic building block that enables extreme miniaturisation. With very small size and weight, insects achieve a level of robustness that we have not been able to match with human fabrication techniques.

- The biological energy storage system and the muscular and hydraulic activation systems used in animals and insects achieve torque, response time and conversion efficiencies that far exceed similarly scaled man-made systems.

Based on these aforementioned limitations, mobile robots generally generate motion, either using wheeled mechanisms, a well-known human technology for vehicles, or using a small number of articulated legs, the simplest of the biological approaches to locomotion (shown in **Fig.** 1.2).

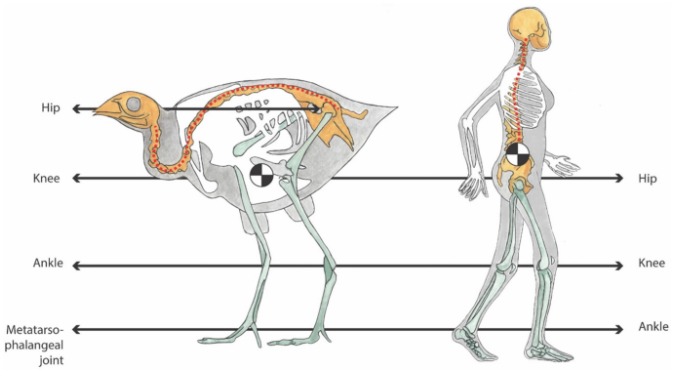



**Figure 1.2:** Bipedal motion is not unique to only humans as a wide variety of animals show bipedal motion [4].

In general, legged locomotion requires higher DoF and therefore greater mechanical complexity than wheeled locomotion [5]. Wheels, in addition to being simple, are extremely well suited to flat ground. As **Fig.** 1.3 depicts, on flat surfaces wheeled locomotion is one to two orders of magnitude more efficient than legged locomotion.

> The railway is ideally engineered for wheeled locomotion because rolling friction is minimised using a hard and flat steel surface.

But as the surface becomes soft, wheeled locomotion accumulates inefficiencies due to **rolling friction** while legged locomotion suffers much less because it consists only of **point contacts** with the ground. This is demonstrated in figure 2.3 by the dramatic loss of efficiency in the case of a tire on soft ground.

> the efficiency of wheeled locomotion depends on environmental qualities, such as the flatness and hardness of the ground, while the efficiency of legged locomotion depends on the leg mass and body mass, both of which the robot must support at various points in a legged gait.

It is understandable therefore nature favours legged locomotion, as locomotion systems in nature

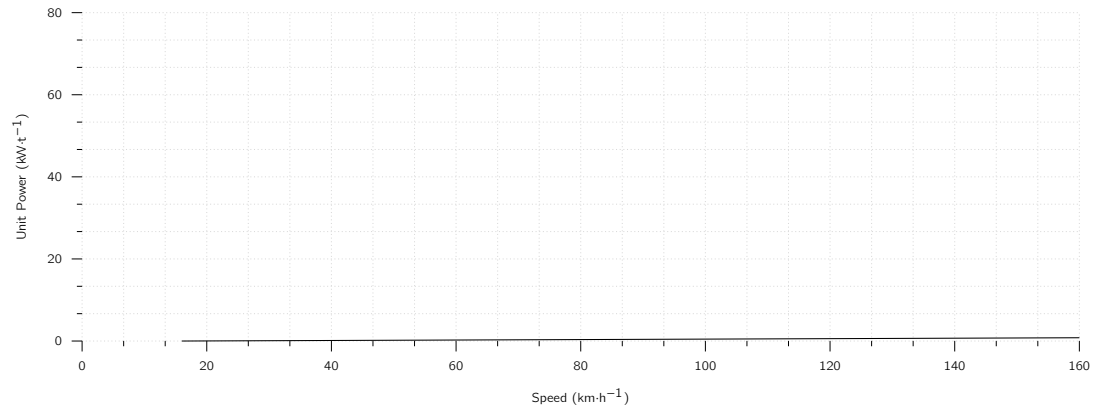



**Figure 1.3:** Specific power versus attainable speed of various locomotion mechanisms (Adapted from [1]).

must operate on rough and unstructured terrain. For example, in the case of insects in a forest the vertical variation in ground height is often an order of magnitude greater than the total height of the insect.

By the same token, the human environment frequently consists of engineered, smooth surfaces both indoors and outdoors. Therefore, it is also understandable that virtually all industrial applications of mobile robotics utilise some form of wheeled locomotion. Recently, for more natural outdoor environments, there has been some progress toward hybrid and legged industrial robots such as the forestry robot [6] shown in **Fig.** 1.4.

In the next section, we present general considerations that concern all forms of mobile robot locomotion. Following this will be overviews of legged locomotion and wheeled locomotion techniques for mobile robots.

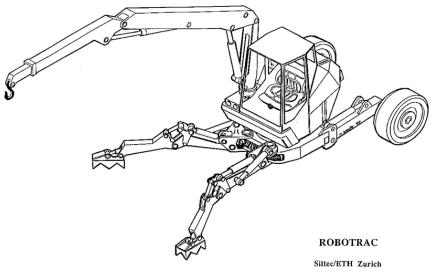



**Figure 1.4:** RoboTrac, a hybrid wheel-leg vehicle for rough terrain.

### 1.1.1 Key Issues for Locomotion

Locomotion is the complement of manipulation:

- In manipulation, the robot arm is fixed but moves objects in the workspace by imparting force to them.
- In locomotion, the environment is fixed and the robot moves by imparting force to the environment.

For both cases, the scientific basis is the study of actuators which generate interaction forces, and mechanisms that implement desired kinematic and dynamic properties. Locomotion and manipulation therefore share the same core issues of stability, contact characteristics and environmental type:

- Stability
  - number and geometry of contact points
  - centre of gravity
  - static/dynamic stability
  - inclination of terrain
- characteristics of contact
  - contact point/path size and shape
  - angle of contact
  - friction
- type of environment
  - structure
  - medium (e.g., water, air, soft or hard ground)

A theoretical analysis of locomotion begins with mechanics and physics. From this starting point,

we can formally define and analyse all manner of mobile robot locomotion systems. However, this Lecture Book puts more emphasis on the mobile robot navigation problem, particularly on the topics of perception, localisation and cognition. Therefore, we will not delve deeply into the physical basis of locomotion. Nevertheless, two remaining sections in this chapter present overviews of issues in legged locomotion and wheeled locomotion.

## 1.2 Legged Mobile Robots

Legged locomotion is characterised by a series of point contacts between the robot and the ground. The primary advantages include adaptability and manoeuvrability in rough terrain. Because only a set of point contacts is required, the quality of the ground between those points does not matter, so long as the robot can maintain adequate ground clearance. In addition, a walking robot is capable of crossing a hole or chasm so long as its reach exceeds the width of the hole. A final advantage of legged locomotion is the potential to manipulate objects in the environment with great skill.

**Figure 1.5:** Dug-beetle are a great example for legged mobile robotics [7] as not only they can manoeuvre in their environment using legged motion, they can also manipulate their environment and generate rotational motion [8].

> The dung beetle, is capable of rolling a ball while locomotion as a result of its dexterous front legs shown in **Fig.** 1.5.

The main disadvantages of legged locomotion include power and mechanical complexity. The leg, which may include several DoF, must be capable of sustaining part of the robot's total weight, and in many robots must be capable of lifting and lowering the robot. Additionally, high manoeuvrability will only be achieved if the legs have a sufficient number of DoF to impart forces in a number of different directions.

### Leg Configurations and Stability

Because legged robots are biologically inspired, it is instructive to examine biologically successful legged systems. A number of different leg configurations have been successful in a variety or organisms seen in **Fig.** 1.6.

Large animals such as mammals and reptiles have four (4)legs whereas insects have six (6)or more legs. In some mammals, the ability to walk on only two (2)legs has been perfected. Especially in the case of humans, balance has progressed to the point that we can even jump with one leg[4]. This

[4] In child development, one of the tests used to determine if the child is acquiring advanced locomotion skills is the ability to jump on one leg [12].

**(a)** Bipedal motion [9].

**(b)** Quadpedal motion [10].

**(c)** Hexapedal motion [11]

**Figure 1.6:** Types of motions used by different animals.

exceptional manoeuvrability comes at a price:

> Bipedal motion is much more complex active control to maintain balance.

In contrast, a creature with three (3) legs can exhibit a static, stable pose provided that it can ensure that its centre of gravity is within the tripod of ground contact. Static stability, demonstrated by a three-legged stool, means that balance is maintained with no need for motion. A small deviation from stability[5] is passively corrected towards the stable pose when the upsetting force stops. But a robot must be able to lift its legs in order to walk. To achieve static walking, a robot **must** have at least six (6)legs [13]. In such a configuration, it is possible to design a gait[6] in which a statically stable tripod of legs is in contact with the ground at all times.

[5] e.g., such as gently pushing the stool

[6] the pattern of steps of an animal at a particular speed.

Insects[7] are immediately able to walk when born. For them, the problem of balance during walking is relatively simple. Mammals, with four (4) legs, cannot achieve static walking, but are able to stand easily on four (4) legs. Fauns[8], for example, spend several minutes attempting to stand before they are able to do so, then spend several more minutes learning to walk without falling [14]. Humans, with two (2) legs, cannot even stand in one place with static stability. Infants require months to stand and walk, and even longer to learn to jump, run and stand on one leg.

[7] such as spiders, ant, beetles, . . .

[8] a baby deer

There is also the potential for great variety in the complexity of each individual leg. Once again, the biological world provides ample examples at both extremes. For instance, in the case of the caterpillar, each leg is extended using hydraulic pressure by constricting the body cavity and forcing an increase in pressure, and each leg is retracted longitudinally by relaxing the hydraulic pressure, then activating a single tensile muscle that pulls the leg in towards the body, seen in **Fig.** 1.7. Each leg has only a single DoF, which is oriented longitudinally along the leg. Forward locomotion depends on the hydraulic pressure in the body, which extends the distance between pairs of legs. The caterpillar leg is therefore mechanically very simple, using a minimal number of extrinsic muscles to achieve complex overall locomotion.

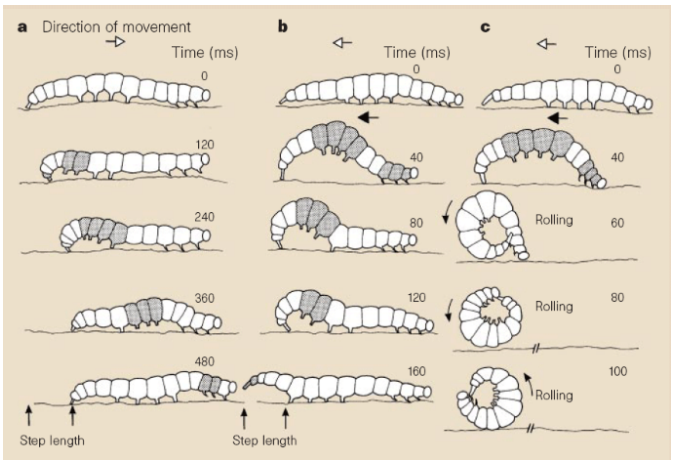



**Figure 1.7:** Main locomotory gaits in *Pleurotya* caterpillar [15].

At the other extreme, the human leg has more than six (6) major degrees of freedom, combined with further actuation at the toes, shown in **Fig.** 1.8. There are more than 50 muscles in each lower limb and at least half of them participate actively in the control of leg motion [16, 17].

In the case of legged mobile robots, a minimum of two (2) DoF is generally required to move a leg forward by lifting the left and swinging it forward. More common is the addition of a 3rd DoF for more complex manoeuvres, resulting in legs such as those shown in **Fig.** 1.9. Recent successes in the creation of bipedal walking robots have added a fourth DOF at the ankle joint [20]. The ankle enables more consistent ground contact by actuating the pose of the sole of the foot. In general, adding DoF to a robot leg increases the manoeuvrability of the robot, both augmenting the range of terrains on which it can travel and the ability of the robot to travel with a variety of gaits. The primary disadvantages of additional joints and actuators is, of course, energy, control and mass. Additional actuators require energy and control, and they also add to leg mass, further increasing power and load requirements on existing actuators.

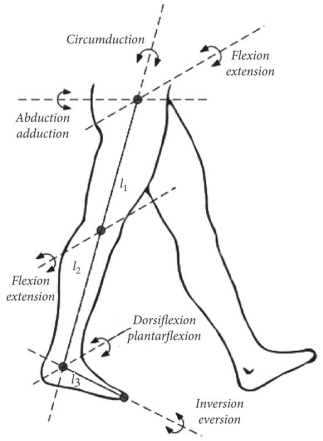



**Figure 1.8:** The DoF a human leg has [18].

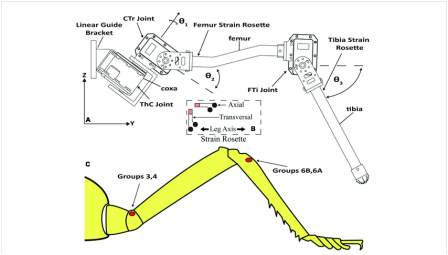



**Figure 1.9:** An example of a leg possessing three (3)DoF [19].

In the case of a multi-legged mobile robot, there is the issue of leg coordination for locomotion, or gait control.

> The number of possible gaits depends on the number of legs [21].

The gait is a sequence of lift and release events for the individual legs. For a mobile robot with $k$ legs, the total number of possible events $N$ for a walking machine is:

$$N = (2k - 1)! \tag{1.1}$$

For a bipedal walker ($k$=2) legs the number of possible events $N$ is:

$$N = (2k - 1)! = 3! = 3 \cdot 2 \cdot 1 = 6 \tag{1.2}$$

The six (6) different events are:

- lift right leg
- lift left leg
- release right leg
- release left leg
- lift both legs together
- release both legs together

As can we see, this list of possible events quickly grows quite large. For example, a robot with six (6) legs has far more gaits theoretically:

$$N = 11! = 39\,916\,800 \tag{1.3}$$

### 1.2.1 Examples of Legged Robot Locomotion

Although there are no high-volume industrial applications to date, legged locomotion is an important area of long-term research. Several interesting designs are presented below, beginning with the one-legged robot and finishing with six-legged robots.

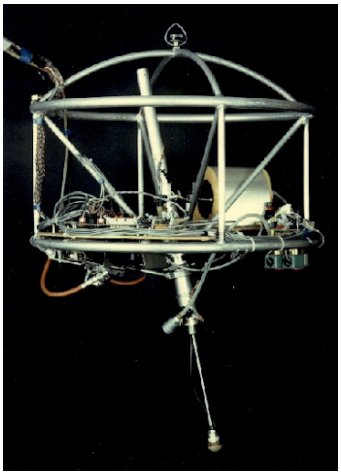


**Figure 1.10:** The Raibert hopper [22].

#### Single Leg

The minimum number of legs a legged robot can have is, of course, one. Minimising the number of legs is beneficial for several reasons.

- Body mass is particularly important to walking machines, and the single leg minimises cumulative leg mass.
- Leg coordination is required when a robot has several legs, but with one leg no such coordination is needed.
- The one-legged robot maximises the basic advantage of legged locomotion: legs have single points of contact with the ground in lieu of an entire track as with wheels.

  A single legged robot requires only a sequence of single contacts, making it useful in rough terrain.

Perhaps most importantly, a hopping robot can dynamically cross a gap that is larger than its stride by taking a running start, whereas a multi-legged walking robot that cannot run is limited to crossing gaps that are as large as its reach.

The major challenge of creating a single-leg robot is balance. For a robot with one leg, static walking is not only impossible, but static stability when stationary is also impossible. The robot must actively balance itself by either changing its centre of gravity or by imparting corrective forces. Thus, the successful single-leg robot **must be dynamically stable**.

**Fig.** 1.10 shows the Raibert Hopper [23, 24], one of the most well-known single-leg hopping robots created.

This robot makes continuous corrections to body attitude and to robot velocity by adjusting the leg angle with respect to the body. The actuation is hydraulic, including high-power longitudinal extension of the leg during stance to hop back into the air. Although powerful, these actuators require a large, off-board hydraulic pump to be connected to the robot at all times. **Fig.** 1.11 shows a more energy efficient design developed [26]. Instead of supplying power by means of an off-board hydraulic pump, the Bow Leg Hopper is designed to capture the kinetic energy of the robot as it lands using an efficient bow spring leg. This spring returns approximately 85% of the energy, meaning that stable hopping requires only the addition of 15% of the required energy on each hop. This robot, which is constrained along one axis by a boom, has demonstrated continuous hopping for 20 minutes using a single set of batteries carried on board the robot. As with the Raibert Hopper, the Bow Leg Hopper controls velocity by changing the angle of the leg to the body at the hip joint. The paper of Ringrose [27] demonstrates the very important duality of mechanics and controls as applied to a single leg hopping machine. Often clever mechanical design can perform the same operations as complex active control circuitry. In this robot, the physical shape of the foot is exactly the right curve so that when the robot lands without being perfectly vertical, the proper corrective force is provided from the impact, making the robot vertical by the next landing. This robot is dynamically stable, and is furthermore passive.

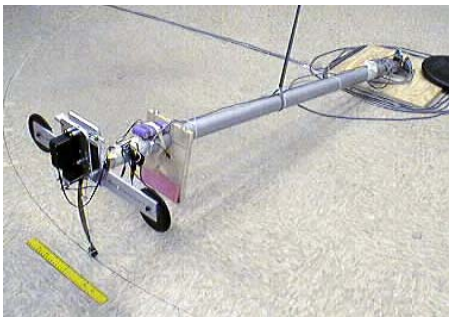


**Figure 1.11:** The 2D single Bow Leg Hopper.


**Figure 1.12:** The New ASIMO introduced in 2005 [25].

> The correction is provided by physical interactions between the robot and its environment, with no computer nor any active control in the loop.

### Two Legs (Bipedal)

A variety of successful bipedal robots have been demonstrated. Two-legged robots have been shown to run, jump, travel up and down stairs and even do aerial tricks such as somersaults. **Fig.** 1.12 shows the Honda P2 bipedal robot, which is the product of tens of millions of research dollars and more than a decade of work. This biped can walk on slopes, climb and descend stairs, and push shopping carts. The crucial technology that enables this robot is Honda's research into the fabrication of extremely high torque, low mass motors that serve as the robot's joints. In the case of P2, the most

significant obstacle that remains is energy capacity, efficiency and autonomous navigation. This robot can operate for only about 20 minutes with on-board power. An important feature of bipedal robots is their anthropomorphic shape. They can be built to have the same approximate dimensions as humans, and this makes them excellent vehicles for research in human-robot interaction.

Bipedal robots can only be statically stable within some limits, and so robots such as P2 and Wabian generally must perform continuous balance-correcting servoing even when standing still. Furthermore, each leg must have sufficient capacity to support the full weight of the robot. In the case of four-legged robots, the balance problem is facilitated along with the load requirements of each leg. An elegant design of a biped robot is the Spring Flamingo of MIT seen in **Fig.** 1.13. This robot inserts springs in series with the leg actuators to achieve a more elastic gait. Combined with "kneecaps" that limit knee joint angles, the Flamingo achieves surprisingly biomimetic motion.

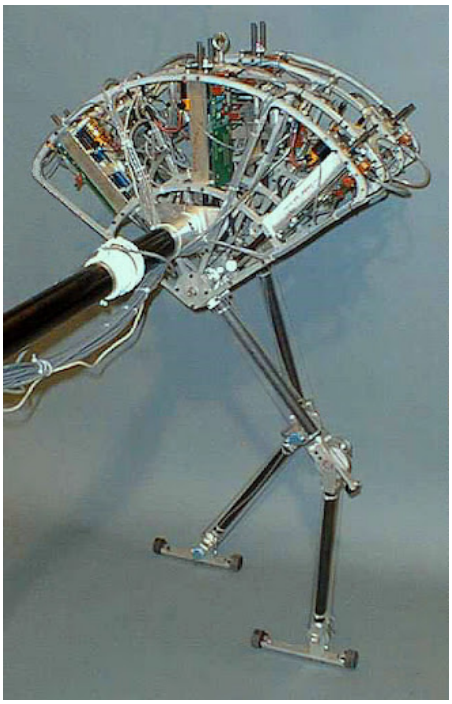


**Figure 1.13:** Spring Flamingo is a planar bipedal walking robot [28].

### Four Legs (Quadruped)

Although standing still on four legs is passively stable, walking remains challenging because to remain stable the robot's center of gravity must be actively shifted during the gait. Sony recently invested several million dollars to develop a four-legged robot (figure 2.14). To cre- ate this robot, Sony created both a new robot operating system that is near real-time and invented new geared servomotors that are sufficiently high torque to support the robot, yet backdriveable for safety. In addition to developing custom motors and software, Sony incorporated a color vision system that enables Aibo to chase a brightly colored ball. The robot is able to function for at most one hour before requiring recharging. Early sales of the robot have been very strong, with more than 60,000 units sold in the first year. Nevertheless, the number of motors and the technology investment behind this robot dog have resulted in a very high price of approximately 1500. Four legged robots have the potential to serve as effective artifacts for research in human- robot interaction (fig. 2.15). Humans can treat the Sony robot, for example, as a pet and might develop an emotional relationship similar to that between man and dog. Furthermore, Sony has designed Aibo's walking style and general behavior to emulate learning and maturation, resulting in dynamic behavior over time that is more interesting for the owner who can track the changing behavior. As the challenges of high energy storage and motor technology are solved, it is likely that quadruped robots much more capable than Aibo will become common throughout the human environment.

### Six Legs (Hexapod)

Six legged configurations have been extremely popular in mobile robotics because of their static stability during walking, thus reducing the control complexity (figure 2.16 and 2.17). In most cases, each leg has 3 DOF, including hip flexion, knee flexion and hip abduction (figure 2.6).

Genghis is a commercially available hobby robot that has six legs, each of which has 2 DOF provided by hobby servos (figure 2.18). Such a robot, which consists only of hip flexion and hip abduction, has less maneuverability in rough terrain but performs quite well on flat ground. Because it consists of a straightforward arrangement of servo motors and straight legs, such robots can be readily built by a robot hobbyist. Insects, which are arguably the most successful locomoting creatures on earth, excel at traversing all forms of terrain with six legs, even upside down. Currently, the gap between the capabilities of six-legged insects and artificial six-legged robots is still quite large. Interestingly, this is not due to a lack of sufficient numbers of degrees of freedom on the robots. Rather, insects combine a small number of active degrees of freedom with passive structures, such as microscopic barbs and textured pads, that increase the gripping strength of each leg significantly. Robotic research into such passive tip structures has only recently begun. For example, a research group is attempting to recreate the complete mechanical function of the cockroach leg (Roland, reference in notes (Espenschied et al.)). It is clear from all of the above examples that legged robots have much progress to make before they are competitive with thei24 Autonomous Mobile Robots have been realised recently, primarily due to advances in motor design. Creating actuation systems that approach the efficiency of animal muscle remains far from the reach of robotics, as does energy storage with the energy densities found in organic life forms

**Figure 1.14:** Genghis, one of the most famous walking robots from MIT uses hobby servomotors as its actuators.

## 1.3 Wheeled Mobile Robots

The wheel has been by far the most popular locomotion mechanism in mobile robotics and in man-made vehicles in general.[9] It can achieve high efficiencies, as demonstrated in figure 2.3, and does so with a relatively simple mechanical implementation. In addition, balance is not usually a research problem in wheeled robot designs, because wheeled robots are almost always designed so that all wheels are in ground contact at all times.

[9] This should be clear as wheel motion is one of the most efficient method of converting energy to motion.

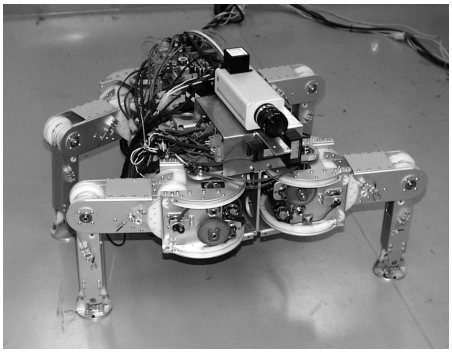


**Figure 1.15:** Genghis, one of the most famous walking robots from MIT uses hobby ser- vomotors as its actuators.

Therefore, three wheels are sufficient to guarantee stable balance, although as we will see below, two-wheeled robots can also be stable. When more than three wheels are used, a suspension system is required in order to allow all wheels to maintain ground contact when the robot encounters uneven terrain. Instead of worrying about balance, researchers in wheeled robots tends to focus on the problems of traction and stability, maneuverability and control:

> can the robot wheels provide sufficient traction and stability for the robot to cover all of the desired terrain, and does the robot's wheeled configuration enable sufficient control over the velocity of the robot?

### 1.3.1 Design

As we will see, there is a very large space of possible wheel configurations when we considers possible techniques for mobile robot locomotion. We will begin by discussing the wheel in detail, as there are a number of different wheel types with specific strengths and weaknesses. Then, we will examine complete wheel configurations that deliver particular forms of locomotion for a mobile robot.

#### Wheel Design

There are four (4) major wheel classes, as shown in **Fig.** 1.16. They differ widely in their kinematics, and therefore the choice of wheel type has a large effect on the overall kinematics of the mobile robot.

The standard wheel and the castor wheel have a primary axis of rotation and therefore are highly directional. To move in a different direction, the wheel must be steered first along a vertical axis. The key difference between these two (2) wheels is that the standard wheel can accomplish this steering motion with no side effects, as the centre of rotation passes through the contact patch

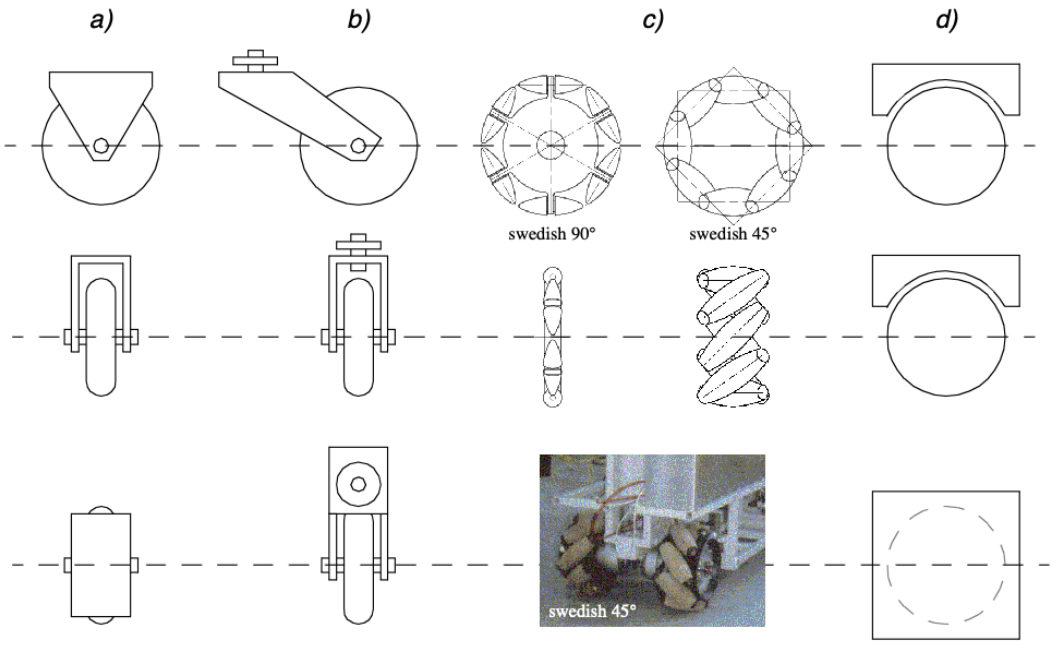



**Figure 1.16:** The four basic wheel types a)Standard wheel: Two degrees of freedom; rotation around the (motorized) wheel axle and the contact point b)castor wheel: Two degrees of freedom; rotation around an offset steering joint c)Swedish wheel: Three degrees of freedom; rotation around the (motorized) wheel axle, around the rollers and around the contact point

with the ground, while the castor wheel rotates around an offset axis, causing a force to be imparted to the robot chassis during steering.

[10]Sometimes known as Swedish wheel or Ilon wheel after its inventor Bengt Erland Ilon [29].

The mecanum wheel[10] and the spherical wheel are both designs that are less constrained by directionality than the conventional standard wheel. The swedish wheel functions as a normal wheel, but provides low resistance in another direction as well, sometimes perpendicular to the conventional direction as in the swedish 90 and sometimes at an intermediate angle as in the swedish 45. The small rollers attached around the circumference of the wheel are passive and the wheel's primary axis serves as the only actively powered joint. The key advantage of this design is that, although the wheel rotation is powered only along the one principal axis (through the axle), the wheel can kinematically move with very little friction along many possible trajectories, not just forward and backward.

The spherical wheel is a truly omnidirectional wheel, often designed so that it may be actively powered to spin along any direction. One mechanism for implementing this spherical design imitates the computer mouse, providing actively powered rollers that rest against the top surface of the sphere and impart rotational force.

Regardless of what wheel is used, in robots designed for all-terrain environments and in robots with more than three (3) wheels, a suspension system is normally required to maintain wheel contact with the ground. One of the simplest approaches to suspension is to design flexibility into the wheel itself. For instance, in the case of some four-wheeled indoor robots that use castor wheels, manufacturers have applied a deformeable tire of soft rubber to the wheel in order to create a

primitive suspension. Of course, this limited solution cannot compete with a sophisticated suspension system in applications where the robot needs a more dynamic suspension for significantly non-flat terrain.

### Wheel Geometry

The choice of wheel types for a mobile robot is strongly linked to the choice of wheel arrangement, or wheel geometry. When designing a mobile robot locomotion we must consider these two (2) issues simultaneously. Why does wheel type and wheel geometry matter? Three fundamental characteristics of a robot are governed by these choices:

- maneuverability,
- controllability
- stability.

Unlike automobiles, which are largely designed for a highly standardised environment[11], mobile robots are designed for applications in a wide variety of situations. Automobiles all share similar wheel configurations as there is one region in the design space that maximises maneuverability, controllability and stability for their standard environment:

the paved road.

[11] such as the road network

However, there is no single wheel configuration that maximises these qualities for the variety of environments faced by different mobile robots. So, we will see great variety in the wheel configurations of mobile robots. In fact, few robots use the Ackerman wheel configuration of the automobile because of its poor maneuverability, with the exception of mobile robots designed for the road system (figure 2.20).

Table 2.1 gives an overview of wheel configurations ordered by the number of wheels. This table shows both the selection of particular wheel types and their geometric configuration on the robot chassis. Note that some of the configurations shown are of little use in mobile robot applications. For instance, the 2-wheeled bicycle arrangement has moderate maneu- verability and poor controllability. Like a single-leg hopping machine, it can never stand still. Nevertheless, this table provides an indication of the large variety of wheel configura- tions that are possible in mobile robot design.

## 1.3.2 Stability

Surprisingly, the minimum number of wheels required for static stability is two (2). As shown above, a two-wheel differential drive robot can achieve static stability if the center of mass is below the wheel axle. Cye is a commercial mobile robot that uses this wheel configuration

**Figure 1.17:** NAVLAB I, the first autonomous highway vehicle that steers and controls the throttle using vision and radar sensors [30].

However, under ordinary circumstances such a solution requires wheel diameters that are impractically large. Dynamics can also cause a two-wheeled robot to strike the floor with a third point of contact, for instance with sufficiently high motor torques from standstill. Conventionally, static stability requires a minimum of three (3)wheels, with the additional caveat that the center of gravity must be contained within the triangle formed by the ground contact points of the wheels. Stability can be further improved by adding more wheels, although once the number of contact points exceeds three, the hyperstatic nature of the geometry will require some form of flexible suspension on uneven terrain.

### 1.3.3 Manoeuvrability

Some robots are omnidirectional, meaning that they can move at any time in any direction along the ground plane $(X, Y)$ regardless of the orientation of the robot around its vertical axis. This level of maneuverability requires wheels that can move in more than just a single direction, and so omnidirectional robots usually employ swedish or spherical wheels that are powered. A good example is Uranus, shown in figure 2.24. This robot uses four swedish wheels to rotate and translate independently and without constraints. In general, the ground clearance of robots with swedish and spherical wheels is somewhat limited, due to the mechanical constraints of constructing omnidirectional wheels. An inter- esting recent solution to the problem of omnidirectional navigation while solving this ground clearance problem is the four castor-wheeled configuration in which each castor wheel is actively steered and actively translated. In this configuration, the robot is truly omnidirectional because, even if the castor wheels are facing a direction perpendicular to the desired direction of travel, the robot can still move in the desired direction by steering these wheels. Because the vertical axis is offset from the ground contact path, the result of this steering motion is robot motion.

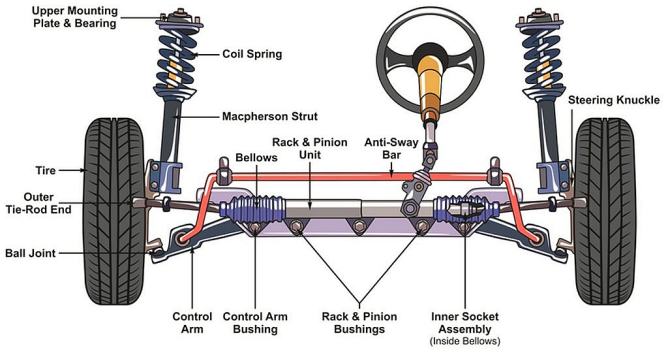



**Figure 1.18:** Example of an Ackerman drive used mostly in automotive industry [31].

In the research community, another classes of mobile robots are popular which achieve high maneuverability, only slightly inferior to that of the omnidirectional configurations. In such robots, motion in a particular direction may initially require a rotational motion. With a cir- cular chassis and an axis of rotation at the center of the robot, such a robot can spin without changing its ground footprint. The most popular such robot is the two-wheel differential drive robot where the two wheels rotate around the center point of the robot. One or two ad- ditional ground contact points may be used for stability, based on the application specifics.

In contrast to the above configurations, consider the Ackerman steering configuration com- mon in automobiles. Such a vehicle typically has a turning diameter that is larger than the car. Furthermore, for such a vehicle to move sideways requires a parking maneuver consist- ing of repeated changes in direction forward and backward. Nevertheless, Ackerman steer- ing geometries have been especially popular in the hobby robotics market, where a robot can be built by starting with a remote-control race car kit and adding sensing and autonomy to the existing mechanism. In addition, the limited maneuverability of Ackerman steering has an important advantage: its directionality and steering geometry provide it with very good lateral stability in high speed turns.

### 1.3.4 Controllability

There is generally an inverse correlation between controllability and maneuverability. For example, the omni-directional designs such as the four castor-wheeled configuration require significant processing to convert desired rotational and translational velocities to individual wheel commands. Furthermore, such omni-directional designs often have greater degrees of freedom at the wheel. For instance, the swedish wheel has a set of free rollers along the wheel perimeter. These degrees of freedom cause an accumulation of slippage, tend to reduce dead-reckoning accuracy and increase the design complexity.

Controlling an omnidirectional robot for a specific direction of travel is also more difficult and often less accurate when compared to less manoeuvrable designs.

For example, an Ackerman steering vehicle can go straight simply by locking the steerable wheels and driving the drive wheels, which can be seen in **Fig.** 1.18.

In a differential drive vehicle, the two (2) motors attached to the two wheels must be driven along exactly the same velocity profile, which can be challenging considering variations between wheels, motors and environmental differences. With four-wheel om- nidrive, such as the Uranus robot which has four swedish wheels, the problem is even harder because all four wheels must be driven at exactly the same speed for the robot to travel in a perfectly straight line.

In summary, there is **NO** "ideal" drive configuration that simultaneously maximises stability, manoeuvrability and controllability. Each mobile robot application places unique constraints on the robot design problem, and the designer's task is to choose the most appropriate drive configuration possible from among this space of compromises.

### 1.3.5 Case Studies for Wheeled Motion

Now let's describe four (4) specific wheel configurations, in order to demonstrate concrete applications of the concepts above to mobile robots built for real-world activities.

#### Synchro Drive

The synchro drive configuration (figure 2.22) is a popular arrangement of wheels in indoor mobile robot applications. It is an interesting configuration because, although there are three driven and steered wheels, only two motors are used in total. The one translation motor sets the speed of all three wheels together, and the one steering motor spins all the wheels together about each of their individual vertical steering axes. But note that the wheels are being steered with respect to the robot chassis, and therefore there is no direct way of re-orienting the robot chassis. In fact, the chassis orientation does drift over time due to uneven tire slippage, causing rotational dead-reckoning error.

Synchro drive is particularly advantageous in cases where omnidirectionality is needed. So long as each vertical steering axis is aligned with the contact path of each tire, the robot can always re-orient its wheels and move along a new trajectory without changing its footprint. Of course, if the robot chassis has directionality and the designers intend to re-orient the chassis purposefully, then synchro drive is only appropriate when combined with an independently rotating turret that attaches to the wheel chassis. Commercial research robots such as the Nomadics 150 or the RWI B21r have been sold with this configuration (figure 1.12). In terms of dead-reckoning, synchro drive systems are generally superior to true omni-directional configurations but inferior to differential drive and Ackerman steering systems. There are two main reasons for this. First and foremost, the translation motor generally drives the three wheels using a single belt. Due to slop and backlash in the drivetrain, whenever the drive motor engages, the closest wheel begins spinning before the furthest wheel, causing a small change in the orientation of the chassis. With additional changes in motor speed, these small angular shifts accumulate to create a large error in orientation during dead-reckoning.

Second, the mobile robot has no direct control over the orientation of the chassis. Depending on the orientation of the chassis, the wheel thrust can be highly asymmetric, with two wheels on one side and the third wheel alone, or symmetric, with one wheel on each side and one wheel straight ahead or behind, as shown in (2.22). The asymmetric cases results in a variety of errors when tire-ground slippage can occur, again causing errors in dead-reckoning of robot orientation.

### Omnidirectional Drive

As we will see later in chapter 3.4.2, omnidirectional movement is be of great interest for complete maneuverability. Omnidirectional robots that are able to move in any direction ( ) at any time are also holonomic (see chapter 3.4.2). They can be realized by either using spheric, castor or swedish wheels. Three examples of such holonomic robots are pre- sented below.

### Omnidirectional locomotion with three spheric wheels

The omnidirectional robot depicted in figure 2.23 is based on three spheric wheels, each ac- tuated by one motor. In this design, the spheric wheels are suspended by three contact points, two given by spherical bearings and one by the a wheel connected to the motor axle. This concept provides excellent maneuverability and is simple in design. However, it is limited to flat surfaces and small loads, and it is quite difficult to find round wheels with high fric- tion coefficients

### Omnidirectional locomotion with four swedish wheels

The omnidirectional arrangement depicted in figure 2.24 has been used successfully on sev- eral research robots, including the CMU Uranus. This configuration consists of four swedish 45 degree wheels, each driven by a separate motor. By varying the direction of rotation and relative speeds of the four wheels, the robot can be moved along any trajectory in the plane

and, even more impressively, can simultaneously spin around its vertical axis. For example, when all four wheels spin "forward" or "backward", the robot as a whole moves in a straight line forward and backward, respectively. However, when one diagonal pair of wheels is spun in the same direction and the other diagonal pair is spun in the opposite direction, the robot moves laterally. This four-wheel arrangement of swedish wheels is not minimal in terms of control motors. Because there are only 3 degrees of freedom in the plane, one can build a three-wheeled om- nidirectional robot chassis using three swedish 90 degree wheels as shown in Table 2.1. However, existing examples such as Uranus have been designed with four wheels due to ca- pacity and stability considerations. One application for which such omnidirectional designs are particular amenable is mobile manipulation. In this case, it is desirable to reduce the degrees of freedom of the manipulator arm to save arm mass by using the mobile robot chassis motion for gross motion. As with humans, it would be ideal if the base could move omnidirectionally without greatly impact-

#### Omnidirectional locomotion with four castor wheels and eight motors

Another solution for omnidirectionality is to use castor wheels. This is done for the Nomad XR4000 form Nomadics (fig. 2.25) giving it an excellent maneuverability. Unfortunately Nomadics Technology has ceased the production of mobile robots. The above two examples are drawn from Table 2.1, but this is not an exhaustive list of all wheeled locomotion techniques. Hybrid approaches that combine legged and wheeled loco- motion, or tracked and wheeled locomotion, can also offer particular advantages. Below are two unique designs created for specialized applications.

#### Tracked Slip/Skid Locomotion

In the wheel configurations discussed above, we have made the assumption that wheels are not allowed to skid against the surface. An alternative form of steering, termed slip/skid, may be used to re-orient the robot by spinning wheels that are facing the same direction at different speeds or in opposite directions. The army tank operates this way, and Nanokhod, pictured below (figure 2.26) is an example of a mobile robot based on the same concept. Robots that make use of tread have much larger ground contact patches, and this can signif- icantly improve their maneuverability in loose terrain compared to conventional wheeled designs. However, due to this large ground contact patch, changing the orientation of the ro- bot usually requires a skidding turn, wherein a large portion of the track must slide against the terrain. The disadvantage of such configurations is coupled to the slip/skid steering. Because of the large amount of skidding during a turn, the exact center of rotation of the robot is hard to predict and the exact change in position and orientation is also subject to variations depend- ing on the ground friction. Therefore, dead-reckoning on such robots is highly inaccurate. This is the trade-off that is made in return for extremely good maneuverability and traction over rough and loose terrain. Furthermore, a slip/skid approach on a high-friction surface can quickly overcome the torque capabilities of the motors being used. In terms of power efficiency, this approach is reasonably efficient on loose terrain but extremely inefficient otherwise.

### 1.3.6 Walking Wheels

Walking robots might offer the best maneuverability in rough terrain. However, they are in- efficient on flat ground and need sophisticated control. Hybrid solutions, combining the adaptability of legs with the efficiency of wheels offer an interesting compromise. Solutions that passively adapt to the terrain are of particular interest for field and space robotics. The Sojourner robot of NASA/JPL (fig. 1.2) represents such a hybrid solution, able to overcome objects up to the size of the wheels. A more advanced mobile robot design for similar appli- cations has recently been produced by EPFL (fig. 2.27). This robot, called Shrimp2, has 6 motorized wheels and is capable of climbing objects up to two times its wheel diameter [84,85]. This enables it to climb regular stairs though the robot is even smaller than the So- journer. Using a rhombus configuration, the Shrimp has a steering wheel in the front and the rear, and two wheels arranged on a bogy on each side. The front wheel has a spring suspen- sion to guarantee optimal ground contact of all wheels at any time. The

steering of the rover is realized by synchronizing the steering of the front and rear wheels and the speed differ- ence of the bogy wheels. This allows for high precision maneuvers and turning on the spot with minimum slip/skid of the four center wheels. The use of parallel articulations for the front wheel and the bogies creates a virtual center of rotation at the level of the wheel axis. This ensures maximum stability and climbing abilities even for very low friction coeffi- cients between the wheel and the ground. As mobile robotics research matures we find ourselves able to design more intricate me- chanical systems. At the same time, the control problems of inverse kinematics and dynam2 Locomotion 37 R. Siegwart, EPFL, Illah Nourbakhsh, CMU ics are now so readily conquered that these complex mechanics can in general be controlled. So, in the near future, you should expect to see a great number of unique, hybrid mobile ro- bots that draw together advantages from several of the underlying locomotion mechanisms that we have discussed in this chapter. They will each be technologically impressive, and each will be designed as the expert robot for its particular environmental niche.

# Chapter 2

# Perception

## Table of Contents



## 2.1 Introduction

One of the most important tasks of an AMR is to acquire knowledge about its environment.[1] This is achieved by taking measurements using various sensors and then extracting meaningful information from those measurements.

[1] One could even argue it is the definition of life, if you ask a biologist as the ability to feel and act on its environment is the bare necessity.

In this chapter we present the most common sensors used in AMR and then discuss strategies for extracting information from the sensors.

### 2.1.1 Sensors for Mobile Robotics

There is a wide variety of sensors used in AMRs (Fig. 4.1). Some are used to measure simple values like the internal temperature of a robot's electronics or the rotational speed of the motors in its wheels or actuators. Other, more sophisticated sensors can be used to acquire information about the robot's environment or even to directly measure a robot's global position. Here, we focus primarily on sensors used to extract information about the robot's environment. Because a AMR moves around, it will frequently encounter unforeseen environmental characteristics, and therefore such sensing is particularly critical. We begin with a functional classification of sensors. Then, after

presenting basic tools for describing a sensor's performance, we proceed to describe selected sensors in detail.

### 2.1.2 Sensor Classification

We classify sensors using two (2) important functional axes. Let's define these terms for clarity;

**Proprioceptive** sensors which measure values internal to the robot.

e.g., motor speed, wheel load, robot arm joint angles, battery voltage.

**Exteroceptive** sensors which measure information from the robot's environment;

e.g., distance measurements, light intensity, sound amplitude.

> exteroceptive sensor measurements are interpreted by the robot to extract meaningful environmental features.

**Passive** sensors measure ambient environmental energy entering the sensor.

e.g., temperature probes, microphones and CCD or CMOS cameras.

**Active** sensors emit energy into the environment, then measure the environmental reaction. Because active sensors can manage more controlled interactions with the environment, they often achieve superior performance. However, active sensing introduces several risks: the outbound energy may affect the very characteristics that the sensor is attempting to measure. Furthermore, an active sensor may suffer from interference between its signal and those beyond its control. For example, signals emitted by other nearby robots, or similar sensors on the same robot my influence the resulting measurements. Examples of active sensors include wheel quadrature encoders, ultrasonic sensors and laser rangefinders.

The sensor classes in Table (4.1) are arranged in ascending order of complexity and descending order of technological maturity. Tactile sensors and proprioceptive sensors are critical to virtually all mobile robots, and are well understood and easily implemented. Commercial quadrature encoders, for example, may be purchased as part of a gear-motor assembly used in a AMR. At the other extreme, visual interpretation by means of one or more CCD/CMOS cameras provides a broad array of potential functionalities, from obstacle avoidance and localisation to human face recognition. However, commercially available sensor units that provide visual functionalities are only now beginning to emerge

### 2.1.3 Characterising Sensor Performance

The sensors we describe in this chapter vary greatly in their performance characteristics. Some sensors provide extreme accuracy in well-controlled laboratory settings, but are overcome with error when subjected to real-world environmental variations. Other sensors provide narrow, high precision data in a wide variety settings. To quantify such performance characteristics, first we formally define the sensor performance terminology that will be valuable throughout the rest of this chapter.

#### Basic Sensor Response Ratings

A number of sensor characteristics can be rated quantitatively in a laboratory setting. Such performance ratings will necessarily be best-case scenarios when the sensor is placed on a real-world robot, but are nevertheless useful.

**Dynamic Range** Used to measure the spread between the lower and upper limits of inputs values to the sensor while maintaining normal sensor operation. Formally, the dynamic range is the ratio of the maximum input value to the minimum measurable input value. Because this raw ratio can be unwieldy, it is usually measured in Decibels, which is computed as ten times the common logarithm of the dynamic range. However, there is potential confusion in the calculation of Decibels, which are meant to measure the ratio between powers, such as Watts or Horsepower.

Suppose your sensor measures motor current and can register values from a minimum of 1 mA to 20 A. The dynamic range of this current sensor is defined as:

$$10 \cdot \log\left[\frac{20}{0.001}\right] = 43 \quad \text{dB} \tag{2.1}$$

Now suppose you have a voltage sensor that measures the voltage of your robot's battery, measuring any value from 1 mV to 20 V. Voltage is **NOT** a unit of power, but the square of voltage is proportional to power. Therefore, we use 20 instead of 10:

$$20 \cdot \log\left[\frac{20}{0.001}\right] = 86 \quad \text{dB} \tag{2.2}$$

**Range** An important rating in AMR because often robot sensors operate in environments where they are frequently exposed to input values beyond their working range. In such cases, it is critical to understand how the sensor will respond. For example, an optical rangefinder will have a minimum operating range and can thus provide spurious data when measurements are taken with object closer than that minimum.

**Resolution** The minimum difference between two (2) values that can be detected by a sensor. Usually, the lower limit of the dynamic range of a sensor is equal to its resolution. However, in the case of digital sensors, this is not necessarily so. For example, suppose that you have a sensor that measures voltage, performs an analogue-to-digital conversion and outputs the

converted value as an 8-bit number linearly corresponding to between 0 and 5 Volts. If this sensor is truly linear, then it has $2^8 - 1$ total output values or a resolution of:

$$\frac{5}{255} = 20 \quad \text{mV}$$

**Linearity** is an important measure governing the behaviour of the sensor's output signal as the input signal varies. A linear response indicates that if two (2) inputs, say $x$ and $y$ result in the two outputs $f(x)$ and $f(x)$, then for any values $a$ and $b$, the following relation can be derived:

$$f(x+y) = f(x) + f(y).$$

This means that a plot of the sensor's input/output response is simply a straight line.

**Bandwidth or Frequency** is used to measure the speed with which a sensor can provide a stream of readings. Formally, the number of measurements per second is defined as the sensor's frequency in Hz. Because of the dynamics of moving through their environment, mobile robots often are limited in maximum speed by the bandwidth of their obstacle detection sensors. Thus increasing the bandwidth of ranging and vision-based sensors has been a high-priority goal in the robotics community.

### In Situ Sensor Performance

The above sensor characteristics can be reasonably measured in a laboratory environment, with confident extrapolation to performance in real-world deployment. However, a number of important measures cannot be reliably acquired without deep understanding of the complex interaction between all environmental characteristics and the sensors in question. This is most relevant to the most sophisticated sensors, including active ranging sensors and visual interpretation sensors.

**Sensitivity** A measure of the degree to which an incremental change in the target input signal changes the output signal. Formally, sensitivity is the ratio of output change to input change. Unfortunately, however, the sensitivity of exteroceptive sensors is often confounded by undesirable sensitivity and performance coupling to other environmental parameters.

**Cross-Sensitivity** is the technical term for sensitivity to environmental parameters that are orthogonal to the target parameters for the sensor. For example, a flux-gate compass can demonstrate high sensitivity to magnetic north and is therefore of use for AMR navigation. However, the compass will also demonstrate high sensitivity to ferrous building materials, so much so that its cross-sensitivity often makes the sensor useless in some indoor environments. High cross-sensitivity of a sensor is generally undesirable, especially so when it cannot be modelled.

**Error** of a sensor is defined as the difference between the sensor's output measurements and the true values being measured, within some specific operating context.

As an example, given a true value $v$ and a measured value $m$, we can define error as:

$$\text{Error} = m - v.$$

**Accuracy** defined as the degree of conformity between the sensor's measurement and the true value, and is often expressed as a proportion of the true value (e.g. 97.5% accuracy):

$$\text{Accuracy} = 1 - \frac{|m - v|}{v}.$$

Of course, obtaining the ground truth ($v$), can be difficult or impossible, and so establishing a confident characterisation of sensor accuracy can be problematic. Further, it is important to distinguish between two different sources of error:

- Systematic errors are caused by factors or processes that can in theory be modelled. These errors are, therefore, deterministic.[2]

  Poor calibration of a laser rangefinder, un-modelled slope of a hallway floor and a bent stereo camera head due to an earlier collision are all possible causes of systematic sensor errors

- Random errors cannot be predicted using a sophisticated model nor can they be mitigated with more precise sensor machinery. These errors can only be described in probabilistic terms (i.e. stochastic). Hue instability in a colour camera, spurious range-finding errors and black level noise in a camera are all examples of random errors.

[2] Meaning, it's value is not determined by a random process and therefore should, in theory, be predictable.

**Precision** is often confused with accuracy, and now we have the tools to clearly distinguish these two terms. Intuitively, high precision relates to reproducibility of the sensor results. For example, one sensor taking multiple readings of the same environmental state has high precision if it produces the same output. In another example, multiple copies of this sensors taking readings of the same environmental state have high precision if their outputs agree. Precision does not, however, have any bearing on the accuracy of the sensor's output with respect to the true value being measured. Suppose that the random error of a sensor is characterised by some mean value ($\mu$) and a standard deviation ($\sigma$). The formal definition of precision is the ratio of the sensor's output range to the standard deviation:

$$\text{Precision} = \frac{\text{Range}}{\sigma}.$$

> Only $\sigma$ and **NOT** $\mu$ has impact on precision. In contrast mean error is directly proportional to overall sensor error and inversely proportional to sensor accuracy.

### Characterising Error

Mobile robots depend heavily on exteroceptive sensors. Many of these sensors concentrate on a central task for the robot:

> *acquiring information on objects in the robot's immediate vicinity so that it may interpret the state of its surroundings.*

Of course, these "objects" surrounding the robot are all detected from the viewpoint of its local reference frame.[3] Since the systems we study are mobile, their ever-changing position and their motion has a significant impact on overall sensor behaviour.

[3] In this case we are referring to the robot reference frame.

Now that we have the necessary knowledge on the fundamental concepts and terminology, we can now describe how dramatically the sensor error of an AMR disagrees with the ideal picture drawn in the previous section.

### Blurring of Systematical and Random Errors

Active ranging sensors tend to have failure modes which are triggered largely by specific relative positions of the sensor and environment targets.

> For example, a sonar sensor will product specular reflections,[4] producing grossly inaccurate measurements of range, at specific angles to a smooth sheet-rock wall.

[4] The incident light is reflected into a single outgoing direction.

During motion of the robot, such relative angles occur at stochastic intervals. This is especially true in a AMR outfitted with a ring of multiple sonars. The chances of one sonar entering this error mode during robot motion is high. From the perspective of the moving robot, the sonar measurement error is a random error in this case. However, if the robot were to stop, becoming motionless, then a very different error modality is possible.

If the robot's static position causes a particular sonar to fail in this manner, the sonar will fail consistently and will tend to return precisely the same (and incorrect!) reading time after time. Once the robot is motionless, the error appears to be systematic and high precision.

> The fundamental mechanism at work here is the cross-sensitivity of AMR sensors to robot pose and robot-environment dynamics.

The models for such cross-sensitivity are **NOT** , in an underlying sense, truly random. However, these physical interrelationships are rarely modelled and therefore, from the point of view of an incomplete model, the errors appear random during motion and systematic when the robot is at rest. Sonar is not the only sensor subject to this blurring of systematic and random error modality. Visual interpretation through the use of a CCD camera is also highly susceptible to robot motion and position because of camera dependency on lighting.[5]

[5] such as glare and reflections.

> The important point is to realise that, while systematic error and random error are well-defined in a controlled setting, the AMR can exhibit error characteristics that bridge the gap between deterministic and stochastic error mechanisms.

### Multi-Modal Error Distributions

It is common to characterise the behaviour of a sensor's random error in terms of a probability distribution over various output values. In general, one knows very little about the causes of random error and therefore several simplifying assumptions are commonly used. For example, we can assume that the error is zero-mean ($\mu = 0$), in that it symmetrically generates both positive and negative measurement error. We can go even further and assume that the probability density curve is Gaussian. Although we discuss the mathematics of this in detail later, it is important for now to recognise the fact that one frequently assumes symmetry as well as unimodal distribution. This means that measuring the correct value is most probable, and any measurement that is further away from the correct value is less likely than any measurement that is closer to the correct value. These are strong assumptions that enable powerful mathematical principles to be applied to AMR problems, but it is important to realise how wrong these assumptions usually are.

Consider, for example, the sonar sensor once again. When ranging an object that reflects the sound signal well, the sonar will exhibit high accuracy, and will induce random error based on noise, for example, in the timing circuitry. This portion of its sensor behaviour will exhibit error characteristics that are fairly symmetric and unimodal. However, when the sonar sensor is moving through an environment and is sometimes faced with materials that cause coherent reflection rather than returning the sound signal to the sonar sensor, then the sonar will grossly overestimate distance to the object. In such cases, the error will be biased toward positive measurement error and will be far from the correct value. The error is not strictly systematic, and so we are left modelling it as a probability distribution of random error. So the sonar sensor has two (2) separate types of operational modes, one in which the signal does return and some random error is possible, and the second in which the signal returns after a multi-path reflection, and gross overestimation error occurs. The probability distribution could easily be at least bimodal in this case, and since overestimation is more common than underestimation it will also be asymmetric.

As a second example, consider ranging via stereo vision. Once again, we can identify two (2) modes of operation. If the stereo vision system correctly correlates two images, then the resulting random error will be caused by camera noise and will limit the measurement accuracy. But the stereo vision system can also correlate two images incorrectly, matching two fence posts for example that are not the same post in the real world. In such a case stereo vision will exhibit gross measurement error, and one can easily imagine such behaviour violating both the unimodal and the symmetric assumptions. The thesis of this section is that sensors in a AMR may be subject to multiple modes of operation and, when the sensor error is characterised, uni modality and symmetry may be grossly violated. Nonetheless, as you will see, many successful AMR systems make use of these simplifying assumptions and the resulting mathematical techniques with great empirical success. The above sections have presented a terminology with which we can characterise the advantages and disadvantages of various mobile robot sensors. In the following sections, we do the same for a sampling of the most commonly used AMR sensors today.

### 2.1.4 Wheel and Motor Sensors

Wheel/motor sensors are devices use to measure the internal state and dynamics of a mobile robot. These sensors have vast applications outside of AMR and, as a result, AMR has enjoyed the benefits of high-quality, low-cost wheel and motor sensors which offer excellent resolution.

In the next part, we sample just one such sensor, the optical incremental encoder.

#### Optical Encoders

Optical incremental encoders have become the most popular device for measuring angular speed and position within a motor drive or at the shaft of a wheel or steering mechanism. In mobile robotics, encoders are used to control the position or speed of wheels and other mo- tor-driven joints. Because these sensors are proprioceptive, their estimate of position is best in the reference frame of the robot and, when applied to the problem of robot localisation, significant corrections are required as discussed in Chapter 5.

An optical encoder is basically a mechanical light chopper that produces a certain number of sine or square wave pulses for each shaft revolution. It consists of an illumination source, a fixed grating that masks the light, a rotor disc with a fine optical grid that rotates with the shaft, and fixed optical detectors. As the rotor moves, the amount of light striking the optical detectors varies based on the alignment of the fixed and moving gratings. In robotics, the resulting sine wave is transformed into a discrete square wave using a threshold to choose between light and dark states. Resolution is measured in Cycles Per Revolution (CPR).

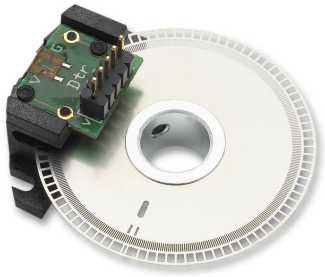


**Figure 2.1:** An example of a rotary encoder. [32]

The minimum angular resolution can be readily computed from an encoder's CPR rating. A typical encoder in AMR may have 2,000 CPR while the optical encoder industry can readily manufacture encoders with 10,000 CPR. In terms of required bandwidth, it is of course critical that the encoder be sufficiently fast to count at the shaft spin speeds that are expected. Industrial optical encoders present no bandwidth limitation to AMR applications. Usually in AMR the quadrature encoder is used. In this case, a second illumination and detector pair is placed 90ř shifted with respect to the original in terms of the rotor disc. The resulting twin square waves, shown in Fig. 4.2, provide significantly more information. The ordering of which square wave produces a rising edge first identifies the direction of rotation. Furthermore, the four detectability different states improve the resolution by a factor of four with no change to the rotor disc. Thus, a 2,000 CPR encoder in quadrature yields 8,000 counts. Further improvement is possible by retaining the sinusoidal wave measured by the optical detectors and performing sophisticated interpolation. Such methods, although rare in AMR, can yield 1000-fold improvements in resolution. As with most proprioceptive sensors, encoders are generally in the controlled environment of a AMR's internal structure, and so systematic error and cross-sensitivity can be engineered away. The accuracy of optical encoders is often assumed to be

100% and, although this may not entirely correct, any errors at the level of an optical encoder are dwarfed by errors downstream of the motor shaft.

### Heading Sensors

Heading sensors can be proprioceptive (gyroscope, inclinometer) or exteroceptive (com- pass). They are used to determine the robots orientation and inclination. They allow us, to- gether with appropriate velocity information, to integrate the movement to a position estimate. This procedure, which has its roots in vessel and ship navigation, is called dead reckoning.

### Compasses

The two most common modern sensors for measuring the direction of a magnetic field are the Hall Effect and Flux Gate compasses. Each has advantages and disadvantages, as described below. The Hall Effect describes the behaviour of electric potential in a semiconductor when in the presence of a magnetic field. When a constant current is applied across the length of a semi- conductor, there will be a voltage difference in the perpendicular direction, across the semi- conductor's width, based on the relative orientation of the semiconductor to magnetic flux

lines. In addition, the sign of the voltage potential identifies the direction of the magnetic field. Thus, a single semiconductor provides a measurement of flux and direction along one dimension. Hall Effect digital compasses are popular in AMR, and contain two such semiconductors at right angles, providing two axes of magnetic field (thresholded) direction, thereby yielding one of 8 possible compass directions. The instruments are inexpensive but also suffer from a range of disadvantages. Resolution of a digital hall effect compass is poor. Internal sources of error include the nonlinearity of the basic sensor and systematic bias errors at the semiconductor level. The resulting circuitry must perform significant filtering, and this lowers the bandwidth of hall effect compasses to values that are slow in AMR terms. For example the hall effect compasses pictured in figure 4.3 needs 2.5 seconds to settle after a 90ř spin. The Flux Gate compass operates on a different principle. Two small coils are wound on fer- rite cores and are fixed perpendicular to one-another. When alternating current is activated in both coils, the magnetic field causes shifts in the phase depending upon its relative alignment with each coil. By measuring both phase shifts, the direction of the magnetic field in two dimensions can be computed. The flux-gate compass can accurately measure the strength of a magnetic field and has improved resolution and accuracy; however it is both larger and more expensive than a Hall Effect compass. Regardless of the type of compass used, a major drawback concerning the use of the Earth's magnetic field for AMR applications involves disturbance of that magnetic field by other magnetic objects and man-made structures, as well as the bandwidth limitations of electronic compasses and their susceptibility

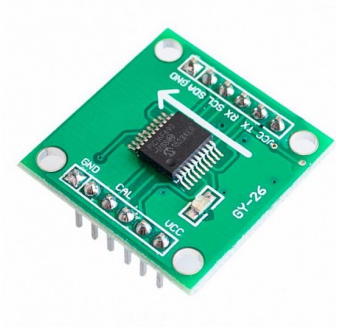



**Figure 2.2:** An example of an electronic compass [33].

to vibration. Particularly in indoor environments AMR applications have often avoided the use of compasses, although a compass can conceivably provide useful local orientation information indoors, even in the precense of steel structures.

### Gyroscope

Gyroscopes are heading sensors which preserve their orientation in relation to a fixed refer- ence frame. Thus they provide an absolute measure for the heading of a mobile system. Gy- roscopes can be classified in two categories, mechanical gyroscopes and optical gyroscopes.

### Mechanical Gyroscopes

The concept of a mechanical gyroscope relies on the inertial properties of a fast spinning rotor. The property of interest is known as the gyroscopic precession. If you try to rotate a fast spinning wheel around its vertical axis, you will feel a harsh reaction in the horizontal axis. This is due to the angular momentum associated with a spinning wheel and will keep the axis of the gyroscope inertially stable. The reactive torque $\tau$ and thus the tracking stability with the inertial frame are proportional to the spinning speed $\omega$, the precession speed $\Omega$ and the wheel's inertia I.

$$\tau = I\omega\Omega$$

By arranging a spinning wheel as seen in Figure 4.4, no torque can be transmitted from the outer pivot to the wheel axis. The spinning axis will therefore be space-stable (i.e. fixed in an inertial reference frame). Nevertheless, the remaining friction in the bearings of the gyro- axis introduce small torques, thus limiting the long term space stability and introducing small errors over time. A high quality mechanical gyroscope can cost up to $100,000 and has an angular drift of about 0.1ř in 6 hours. For navigation, the spinning axis has to be initially selected. If the spinning axis is aligned with the north-south meridian, the earth's rotation has no effect on the gyro's horizontal axis. If it points east-west, the horizontal axis reads the earth rotation. Rate gyros have the same basic arrangement as shown in Figure 4.4 but with a slight modification. The gimbals are restrained by a torsional spring with additional viscous damping. This enables the sensor to measure angular speeds instead of absolute orientation.

### Optical Gyroscopes

Optical gyroscopes are a relatively new innovation. Commercial use began in the early 1980's when they were first installed in aircraft. Optical gyroscopes are angular speed sen- sors that use two monochromatic light beams, or lasers, emitted from the same source in- stead of moving, mechanical parts. They work on the principle that the speed of light remains unchanged and, therefore, geometric change can cause light to take a varying amount of time to reach its destination. One laser beam is sent traveling clockwise through a fiber while the other travels counterclockwise. Because the laser

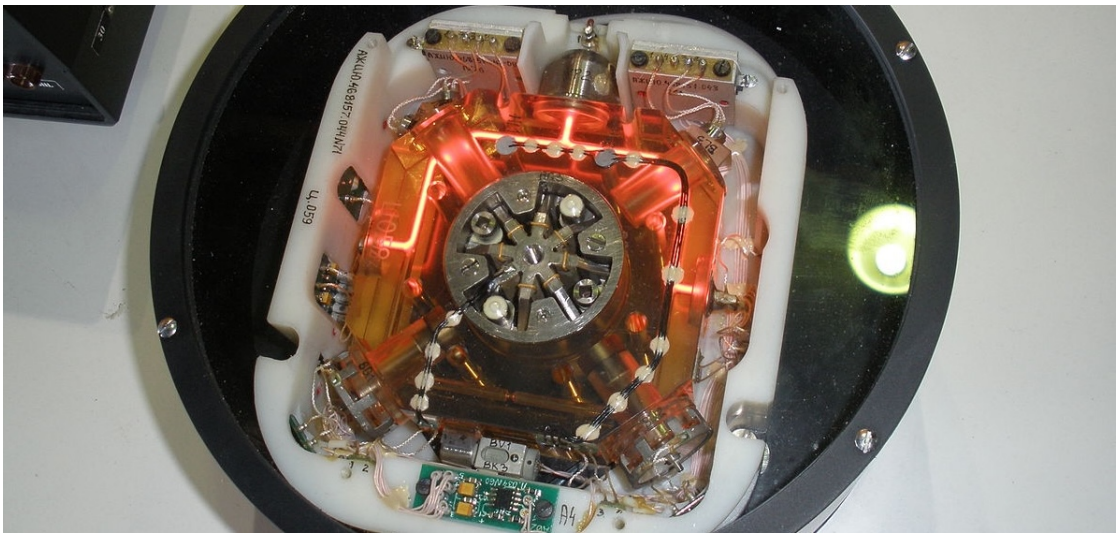


**Figure 2.3:** Optical Gyroscopes have no moving parts, (unlike mechanical gyroscopes) making them extremely reliable [34].

traveling in the direction of rotation has a slightly shorter path, it will have a higher frequency. The difference in fre- quency of the two beams is a proportional to the angular velocity of the cylinder. New solid-state optical gyroscopes based on the same principle are build using microfabrication technology, thereby providing heading information with resolution and bandwidth far beyond the needs of mobile robotic applications. Bandwidth, for instance, can easily exceed 100KHz while resolution can be smaller than 0.0001ř/hr.

### Ground Based Beacons

**Figure 2.4**

One elegant approach to solving the localization problem in AMR is to use active or passive beacons. Using the interaction of on-board sensors and the environmental bea- cons, the robot can identify its position precisely. Although the general intuition is identical to that of early human navigation beacons, such as stars, mountains and lighthouses, modern technology has enabled sensors to localize

an outdoor robot with accuracies of better than 5 cm within areas that are kilometres in size.

In the following subsection, we describe one such beacon system, the Global Positioning System (GPS), which is extremely effective for outdoor ground-based and flying robots. In- door beacon systems have been generally less successful for a number of reasons. The ex- pense of environmental modification in an indoor setting is not amortized over an extremely large useful area, as it is for example in the case of GPS. Furthermore, indoor environments offer significant challenges not seen outdoors, including multipath and environment dynam- ics. A laser-based indoor beacon system, for example, must disambiguate the one true laser signal from possibly tens of other powerful signals that have reflected off of walls, smooth floors and doors. Confounding this, humans and other obstacles may be constantly changing the environment, for example occluding the one true path from the beacon to the robot. In commercial applications such as manufacturing plants, the environment can be carefully controlled to ensure success. In less structured indoor settings, beacons have nonetheless been used, and the problems are mitigated by careful beacon placement and the useful of passive sensing modalities.

### Global Positioning System

The Global Positioning System (GPS) was initially developed for military use but is now freely available for civilian navigation. There are at least 24 operational GPS satellites at all times. The satellites orbit every 12 hours at a height of 20.190km. There are four (4) satellites which located in each of six planes inclined 55° with respect to the plane of the earth's equator (figure 4.5).

Each satellite continuously transmits data which indicates its location and the current time. Therefore, GPS receivers are completely passive but exteroceptive sensors. The GPS satellites synchronise their transmissions to allow their signals to be sent at the same time. When a GPS receiver reads the transmission of two (2) or more satellites, the arrival time differences inform the receiver as to its relative distance to each satellite.

By combining information regarding the arrival time and instantaneous location of four (4) satellites, the receiver can infer its own position.

> In theory, such triangulation requires only three (3) data points. However, timing is extremely critical in the GPS application because the time intervals being measured are in ns.

[6] An example of a csium clock for use in GPS.

It is, of course, mandatory the satellites to be well synchronised. To this end, they are updated by ground stations regularly and each satellite carries on-board atomic clocks[6] for timing. The GPS receiver clock is also important so that the travel time of each satellite's transmission can be accurately measured. But GPS receivers have a simple quartz clock. So, although 3 satellites would ideally provide position in three axes, the GPS receiver requires 4 satellites, using the additional information to solve for 4 variables: three position axes plus a time correction. The fact that the GPS receiver must read the transmission of 4 satellites simultaneously is a significant limitation. GPS satellite transmissions are extremely low-power, and reading them successfully requires direct

line-of-sight communication with the satellite. Thus, in confined spaces such as city blocks with tall buildings or dense forests, one is unlikely to receive 4 satellites reliably. Of course, most indoor spaces will also fail to provide sufficient visibility of the sky for a GPS receiver to function. For these reasons, GPS has been a pop- ular sensor in AMR, but has been relegated to projects involving AMR traversal of wide-open spaces and autonomous flying machines. A number of factors affect the performance of a localization sensor that makes use of GPS. First, it is important to understand that, because of the specific orbital paths of the GPS sat- ellites, coverage is not geometrically identical in different portions of the Earth and therefore resolution is not uniform. Specifically, at the North and South poles, the satellites are very close to the horizon and, thus, while resolution in the latitude and longitude directions is good, resolution of altitude is relatively poor as compared to more equatorial locations.

The second point is that GPS satellites are merely an information source. They can be employed with various strategies in order to achieve dramatically different levels of localisation resolution. The basic strategy for GPS use, called pseudorange and described above, generally performs at a resolution of 15m. An extension of this method is differential GPS, which makes use of a second receiver that is static and at a known exact position. A number of errors can be corrected using this reference, and so resolution improves to the order of 1m or less. A disadvantage of this technique is that the stationary receiver must be installed, its location must be measured very carefully and of course the moving robot must be within ki- lometers of this static unit in order to benefit from the DGPS technique. A further improved strategy is to take into account the phase of the carrier signals of each received satellite transmission. There are two carriers, at 19cm and 24cm, therefore signif- icant improvements in precision are possible when the phase difference between multiple satellites is measured successfully. Such receivers can achieve 1cm resolution for point po- sitions and, with the use of multiple receivers as in DGPS, sub-1cm resolution. A final consideration for AMR applications is bandwidth. GPS will generally offer no better than 200 - 300ms latency, and so one can expect no better than 5Hz GPS updates. On a fast-moving AMR or flying robot, this can mean that local motion integration will be required for proper control due to GPS latency limitations.

## 2.2 Active Ranging

Active range sensors continue to be the most popular sensors used in AMR. Many ranging sensors have a low price point, and most importantly all ranging sensors provide easily interpreted outputs:

> Direct measurements of distance from the robot to objects in its vicinity.

For obstacle detection and avoidance, most AMR rely heavily on active ranging sensors. But the local free-space information provided by range sensors can also be accumulated into representations beyond the robot's current local reference frame. Therefore, active range sensors are also commonly found as part of the localisation and environmental modelling processes of AMRs.

It is only with the slow advent of successful visual interpretation competency that we can expect the class of active ranging sensors to gradually lose their primacy as the sensor class of choice among AMR engineers.

Below, we present two (2) Time-of-Flight (ToF) active range sensors:

- the ultrasonic sensor,
- the laser rangefinder.

Continuing onwards, we then present two (2) geometric active range sensors:

- the optical triangulation sensor,
- the structured light sensor.

### Time-of-FLight Active Ranging

ToF ranging makes use of the propagation speed of sound or an electromagnetic wave. In general, the travel distance of a sound of electromagnetic wave is given by:

$$d = ct,$$

where $d$ is the distance travelled usually round-trip (m), $c$ the speed of wave propagation ($\mathrm{m\,s^{-1}}$), and $t$ is the time it takes to travel (s).

It is important to point out the propagation speed $v$ of sound is approximately 0.3 $\mathrm{m\,ms^{-1}}$ whereas the speed of an electromagnetic signal is 0.3 $\mathrm{m\,ns^{-1}}$, which is one million times faster. The ToF for a typical distance, say 3 m, is 10 ms for an ultrasonic system but only 10 ns for a laser rangefinder. It is therefore obvious that measuring the time of flight $t$ with electromagnetic signals is more technologically challenging.[7]

[7] This explains why laser range sensors have only recently become affordable and robust for use on mobile robots.

The quality of ToF range sensors depends mainly on the following:

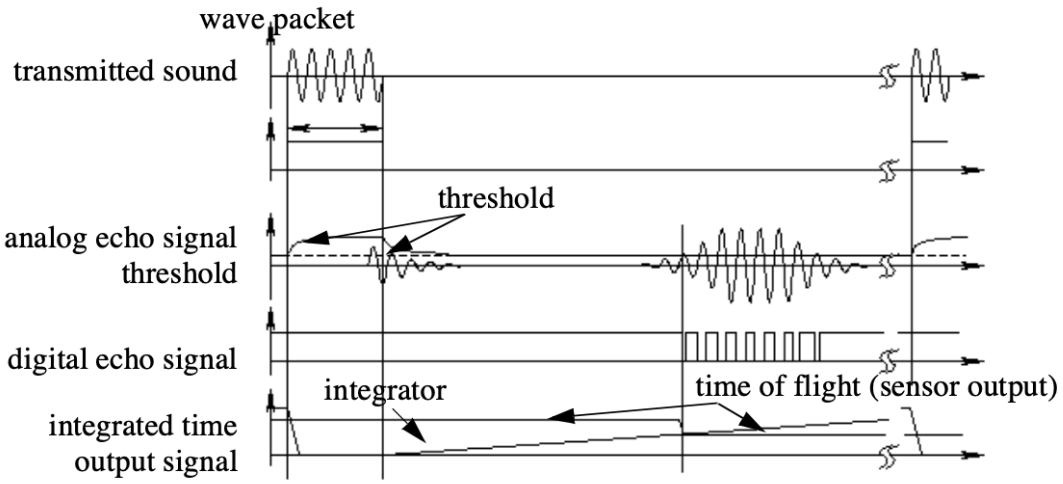



**Figure 2.5:** Signals of an ultrasonic sensor.

- Uncertainties in determining the exact time of arrival of the reflected signal,
- Inaccuracies in the time of flight measurement, particularly with laser range sensors,
- The dispersal cone of the transmitted beam mainly with ultrasonic range sensors
- Interaction with the target (e.g., surface absorption, specular reflections)
- Variation of propagation speed, and
- The speed of the AMR and target (in the case of a dynamic target).

As discussed below, each type of ToF sensor is sensitive to a particular subset of the above list of factors.

### 2.2.1 The Ultrasonic Sensor

The main ethos of an ultrasonic[8] sensor is to transmit a packet of ultrasonic pressure waves and to measure the time it takes for this wave to reflect and return to the receiver. The distance $d$ of the object causing the reflection can be calculated based on the propagation speed of sound[9] $c$ and the time of flight $t$.

$$d = \frac{c \times t}{2}$$

The speed of sound ($v$) in air is given by the following relation:

$$v = \sqrt{\gamma R T}$$

where $\gamma$ is the ratio of specific heat, $R$ is the gas constant ($\mathrm{J\,mol^{-1}\,K^{-1}}$), and $T$ is the temperature

[8] Ultrasound is sound with frequencies greater than 20 kHz.

[9] Of course in this regard careful consideration needs to be made if the medium is significantly different than that of air (i.e., water).

in Kelvin (K). In air, at standard pressure, and 20 °C the speed of sound is approximately:

$$v = 343 \quad \mathrm{m\,s^{-1}}.$$

We can see the different signal output and input of an ultrasonic sensor in **Fig.** 2.5.

First, a series of sound pulses are emitted, which creates the wave packet. An integrator also begins to linearly climb in value, measuring the time from the transmission of these sound waves to detection of an echo. A threshold value is set for triggering an incoming sound wave as a valid echo.

> This threshold is often decreasing in time, because the amplitude of the expected echo decreases over time based on dispersal as it travels longer.

But during transmission of the initial sound pulses and just afterwards, the threshold is set very high to suppress triggering the echo detector with the outgoing sound pulses. A transducer will continue to ring for up to several ms after the initial transmission, and this governs the blanking time of the sensor.

> If, during the blanking time, the transmitted sound were to reflect off of an extremely close object and return to the ultrasonic sensor, it may fail to be detected.

However, once the blanking interval has passed, the system will detect any above-threshold reflected sound, triggering a digital signal and producing the distance measurement using the integrator value.

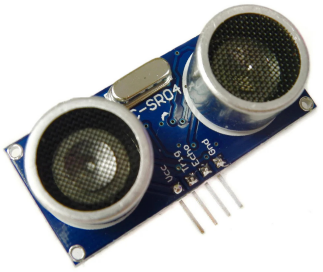
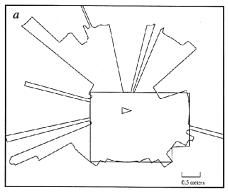



**Figure 2.6:** An example of an ultrasonic sensor used in Raspberry Pi applications [35].

The ultrasonic wave typically has a frequency between 40 and 180 kHz and is usually generated by a piezo or electrostatic transducer. Often the same unit is used to measure the reflected signal, although the required blanking interval can be reduced through the use of separate output and input devices. Frequency can be used to select a useful range when choosing the appropriate ultrasonic sensor for a AMR. Lower frequencies correspond to a longer range, but with the disadvantage of longer post-transmission ringing and, therefore, the need for longer blanking intervals.

Most ultrasonic sensors used by AMRs have an effective range of roughly 12 cm to 5 metres. The published accuracy of commercial ultrasonic sensors varies between 98% and 99.1%. In AMR applications, specific implementations generally achieve a resolution of approximately 2 cm.

In most cases one may want a narrow opening angle for the sound beam in order to also obtain precise directional information about objects that are encountered. This is a major limitation since sound propagates in a cone-like manner with opening angles around 20° and 40°. Consequently, when using ultrasonic ranging one does not acquire depth data points but, rather, entire regions of constant depth. This means that the sensor tells us only that there is an object at a certain distance in within the area of the measurement cone. The sensor readings must be plotted as segments of an arc (sphere for 3D) and not as point measurements.[10] However, recent research developments




[10] The results of a 360° scan of a room.

show significant improvement of the measurement quality in using sophisticated echo processing. Ultrasonic sensors suffer from several additional drawbacks, namely in the areas of error, bandwidth and cross-sensitivity. The published accuracy values for ultrasonic sensors are nominal values based on successful, perpendicular reflections of the sound wave off an acoustically reflective material.

This does not capture the effective error modality seen on a AMR moving through its environment. As the ultrasonic transducer's angle to the object being ranged varies away from perpendicular, the chances become good that the sound waves will coherently reflect away from the sensor, just as light at a shallow angle reflects off of a mirror. Therefore, the true error behavior of ultrasonic sensors is compound, with a well-understood error distribution near the true value in the case of a successful retro-reflection, and a more poorly-understood set of range values that are grossly larger than the true value in the case of coherent reflection.

Of course the acoustic properties of the material being ranged have direct impact on the sensor's performance. Again, the impact is discrete, with one material possibly failing to produce a reflection that is sufficiently strong to be sensed by the unit. For example, foam, fur and cloth can, in various circumstances, acoustically absorb the sound waves. A final limitation for ultrasonic ranging relates to bandwidth. Particularly in moderately open spaces, a single ultrasonic sensor has a relatively slow cycle time.

For example, measuring the distance to an object that is 3 m away will take such a sensor 20ms, limiting its operating speed to 50 Hz. But if the robot has a ring of 20 ultrasonic sensors, each firing sequentially and measuring to minimize interference between the sensors, then the ring's cycle time becomes 0.4s and the overall update frequency of any one sensor is just 2.5 Hz. For a robot conducting moderate speed motion while avoiding obstacles using ultrasonic sensor, this update rate can have a measurable impact on the maximum speed possible while still sensing and avoiding obstacles safely.

> Ultrasonic measurements may be limited through barrier layers with large salinity, temperature or vortex differentials.

### Laser Rangefinder

The laser rangefinder is a ToF sensor which achieves significant improvements over the ultrasonic range sensor due to the use of laser light instead of sound. This type of sensor consists of a transmitter which illuminates a target with a collimated[11] beam (e.g. laser), and a receiver capable of detecting the component of light which is essentially coaxial with the transmitted beam. Often referred to as optical radar or Light Detection and Ranging (LIDAR), these devices produce a range estimate based on the time needed for the light to reach the target and return.

[11] meaning all the rays in questions are made accurately parallel.

A mechanical mechanism with a mirror sweeps the light beam to cover the required scene in a plane or even in 3 dimensions, using a rotating mirror. One way to measure the ToF for the light beam is to use a pulsed laser and then measured the elapsed time directly, just as in the ultrasonic solution

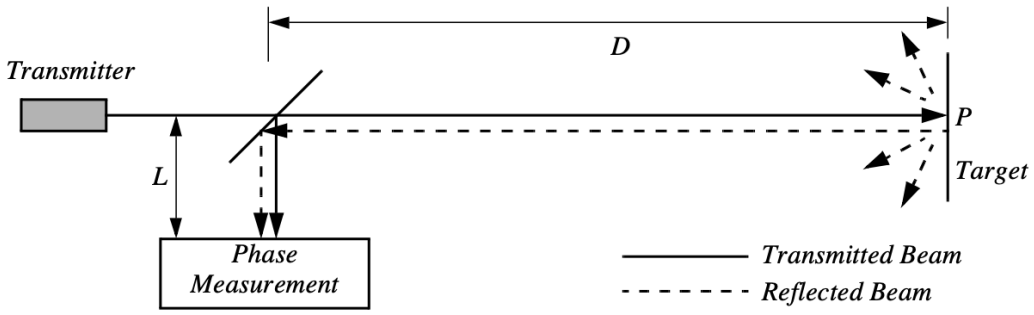



**Figure 2.8:** Schematic of laser rangefinding by phase-shift measurement.

described in just a little bit. Electronics capable of resolving ps are required in such devices and they are therefore very expensive. A second method is to measure the beat frequency between a frequency modulated continuous wave and its received reflection. Another, even easier method is to measure the phase shift of the reflected light.

**Continuous Wave Radar** It is a type of radar system where a known stable frequency continuous wave radio energy is transmitted and then received from any reflecting objects. Individual objects can be detected using the Doppler effect, which causes the received signal to have a different frequency from the transmitted signal, allowing it to be detected by filtering out the transmitted frequency.

Doppler-analysis of radar returns can allow the filtering out of slow or non-moving objects, thus offering immunity to interference from large stationary objects and slow-moving clutter. This makes it particularly useful for looking for objects against a background reflector, for instance, allowing a high-flying aircraft to look for aircraft flying at low altitudes against the background of the surface. Because the very strong reflection off the surface can be filtered out, the much smaller reflection from a target can still be seen.

**Figure 2.7:** A laser range finder used in robotics applications

**Phase Shift Measurement** Near infrared light, which could be from an Light-Emitting Diode (LED) or a laser, is collimated and transmitted from the transmitter $T$ in **Fig.** 2.8 and hits a point $P$ in the environment.

For surfaces having a roughness greater than the wavelength of the incident light, diffuse reflection will occur, meaning that the light is reflected almost isotropically[12]. The wavelength of the infrared light emitted is 824 nm and so most surfaces with the exception of only highly polished reflecting objects, will be diffuse reflectors. The component of the infrared light which falls within the receiving aperture of the sensor will return almost parallel to the transmitted beam, for distant objects. The sensor transmits 100% amplitude modulated light at a known frequency and measures the phase

[12] Something that is isotropic has the same size or physical properties when it is measured in different directions

shift between the transmitted and reflected signals.

**Fig.** 2.9 shows how this technique can be used to measure range. The wavelength of the modulating signal obeys the equation $c = f\lambda$ where $c$ is the speed of light and $f$ the modulating frequency.

For example, $f$ = 5 MHz, the wavelength is $\lambda$ = 60 m.

The total distance $D'$ covered by the emitted light is:

$$D' = L + 2D = L\frac{\theta}{2\pi}\lambda$$

where $D$ and $L$ are the distances defined in **Fig.** 2.8. The required distance $D$, between the beam splitter and the target, is therefore given by:

$$D = \frac{\lambda}{4\pi}\theta$$

where $\theta$ is the electronically measured phase difference between the transmitted and reflected light beams, and $\lambda$ the known modulating wavelength. It can be seen that the transmission of a single frequency modulated wave can theoretically result in ambiguous range estimates since

For example if $\lambda$ = 60m, a target at a range of 5 m would give an indistinguishable phase measurement from a target at 65 m, since each phase angle would be 360° apart.

We therefore define an ambiguity interval of $\lambda$, but in practice we note that the range of the sensor is much lower than $\lambda$ due to the attenuation of the signal in air. It can be shown that the confidence in the range (phase estimate) is inversely proportional to the square of the received signal amplitude, directly affecting the sensor's accuracy. Hence dark, distant objects will not produce as good range estimates as close, bright objects.

As with ultrasonic ranging sensors, an important error mode involves coherent reflection of the energy. With light, this will only occur when striking a highly polishes surface. Practically, a AMR may encounter such surfaces in the form of a polished desktop, file cabinet or of course a mirror. Unlike ultrasonic sensors, laser rangefinders cannot detect the presence of optically transparent materials such as glass, and this can be a significant obstacle in environments, for example museums, where glass is commonly used.

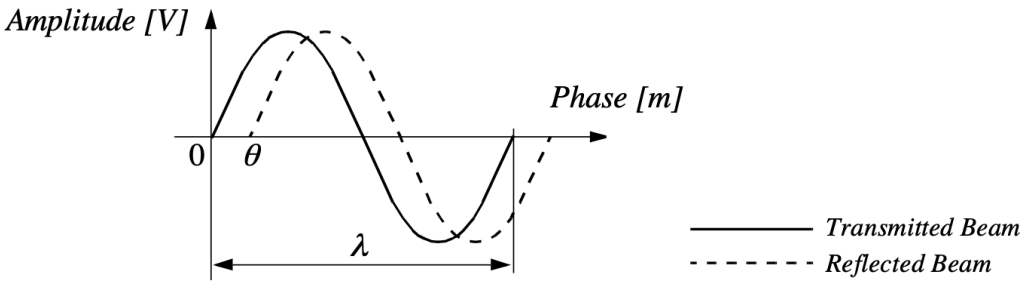



**Figure 2.9:** Range estimation by measuring the phase shift between transmitted and received signals.

### Triangulation-based Active Ranging

Triangulation-based ranging sensors use geometrical properties in their measuring strategy to establish distance readings to objects. The simplest class of triangulation-based rangers are active because they project a known light pattern (e.g., a point, a line or a texture) onto the environment. The reflection of the known pattern is captured by a receiver and, together with known geometric values, the system can use simple triangulation to establish range measurements. If the receiver measures the position of the reflection along a single axis, we call the sensor an optical triangulation sensor in 1D. If the receiver measures the position of the reflection along two orthogonal axes, we call the sensor a structured light sensor.

### Optical Triangulation (1D Sensor)

The principle of optical triangulation in 1D is straightforward, as depicted in **Fig.** 2.10. A collimated

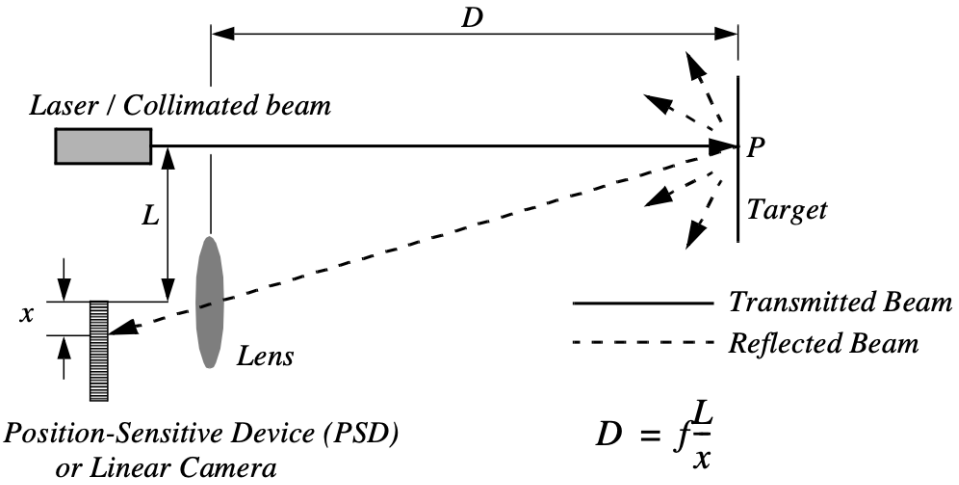



**Figure 2.10:** Principle of 1D laser triangulation.

beam is transmitted toward the target. The reflected light is collected by a lens and projected onto a position sensitive device[13] or linear camera. Given the geometry of **Fig.** 2.10 the distance $D$ is given by:

$$D = f\frac{L}{x}$$

The distance is proportional to $\frac{1}{x}$, therefore the sensor resolution is best for close objects and becomes worse as distance increases. Sensors based on this principle are used in range sensing up to one or two m, but also in high precision industrial measurements with resolutions far below one ţm. Optical triangulation devices can provide relatively high accuracy with very good resolution for close objects. However, the operating range of such a device is normally fairly limited by geometry. For

[13] A position sensitive device and/or position sensitive detector is an optical position sensor which can measure a position of a light spot in one or two-dimensions on a sensor surface.

example, an off-the-shelf optical triangulation sensor can operate over a distance range of between 8 cm and 80 cm.

> It is inexpensive compared to ultrasonic and laser rangefinder sensors.

Although more limited in range than sonar, the optical triangulation sensor has high bandwidth and does not suffer from cross-sensitivities that are more common in the sound domain.

### Structured Light (2D Sensor)

If one replaced the linear camera or Position Sensing Device (PSD) of an optical triangulation sensor with a two-dimensional receiver such as a CCD or CMOS camera, then one can recover distance to a large set of points instead of to only one point. The emitter must project a known pattern, or structured light, onto the environment. Many systems exist which either project light textures, which can be seen in **Fig.** 2.12, or emit collimated light by means of a rotating mirror. Yet another popular alternative is to project a laser stripe by turning a laser beam into a plane using a prism. Regardless of how it is created, the projected light has a known structure, and therefore the image taken by the CCD or CMOS receiver can be filtered to identify the pattern's reflection.

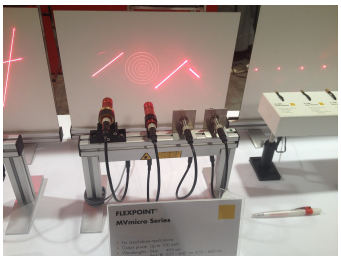


**Figure 2.11:** Structured light sources on display at the 2014 Machine Vision Show in Boston [36].

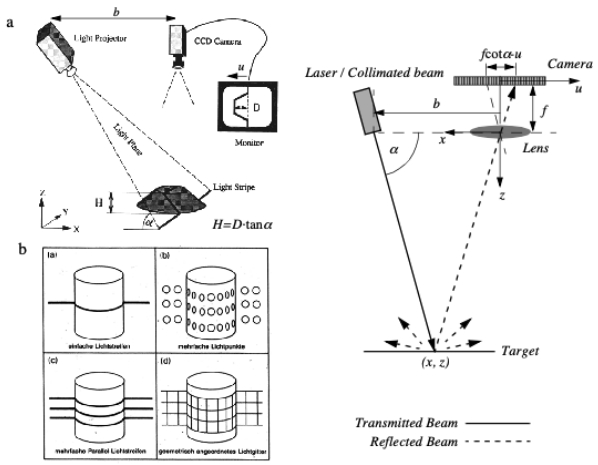



**Figure 2.12:** a) Principle of active two dimensional triangulation b) Other possible light structures c) One-dimensional schematic of the principle

> The problem of recovering depth here far simpler than the problem of passive image analysis.

In passive image analysis, as we discuss later, existing features in the environment must be used to perform correlation, while the present method projects a known pattern upon the environment and thereby avoids the standard correlation problem altogether. Furthermore, the structured light sensor

is an active device; so, it will continue to work in dark environments as well as environments in which the objects are featureless[14]. In contrast, stereo vision would fail in such texture-free circumstances. Figure 4.15c shows a one-dimensional active triangulation geometry. We can examine the trade-off in the design of triangulation systems by examining the geometry in figure 4.15c. The measured values in the system are $\alpha$ and $u$, the distance of the illuminated point from the origin in the imaging sensor.[15] From figure 4.15c, simple geometry shows that:

[14] e.g., uniformly coloured and possess no visible edge.

[15] The imaging sensor here can be a camera or an array of photo diodes of a position sensitive device (e.g. a 2D PSD

$$x = \frac{bu}{f\cot\alpha - u} \qquad \text{and} \qquad z = \frac{bf}{f\cot\alpha - u}.$$

where $f$ is the distance of the lens to the imaging plane. In the limit, the ratio of image resolution to range resolution is defined as the triangulation gain $G_p$ and from equation 4.12 is given by:

$$\frac{\partial u}{\partial z} = G_p = \frac{bf}{z^2}$$

This shows that the ranging accuracy, for a given image resolution, is proportional to source/ detector separation $b$ and focal length $f$, and decreases with the square of the range $z$. In a scanning ranging system, there is an additional effect on the ranging accuracy, caused by the measurement of the projection angle $\alpha$. From equation 4.12 we see that:

$$\frac{\partial \alpha}{\partial z} = G_{ff} = \frac{b\sin\alpha^2}{z^2}$$

We can summarise the effects of the parameters on the sensor accuracy as follows:

**Baseline Length (b)** the smaller $b$ is the more compact the sensor can be.The larger $b$ is the better the range resolution will be. Note also that although these sensors do not suffer from the correspondence problem, the disparity problem still occurs. As the baseline length b is increased, one introduces the chance that, for close objects, the illuminated point(s) may not be in the receiver's field of view.

**Detector length and focal length f** A larger detector length can provide either a larger field of view or an improved range resolution or partial benefits for both. Increasing the detector length however means a larger sensor head and worse electrical characteristics (increase in random error and reduction of bandwidth). Also, a short focal length gives a large field of view at the expense of accuracy and vice versa.

At one time, laser stripe-based structured light sensors were common on several mobile robot bases as an inexpensive alternative to laser range-finding devices. However, with the in- creasing quality of laser range-finding sensors in the 1990's the structured light system has become relegated largely to vision research rather than applied mobile robotics.

### 2.2.2 Motion and Speed Sensors

Some sensors directly measure the relative motion between the robot and its environment. Since such motion sensors detect relative motion, so long as an object is moving relative to the robot's

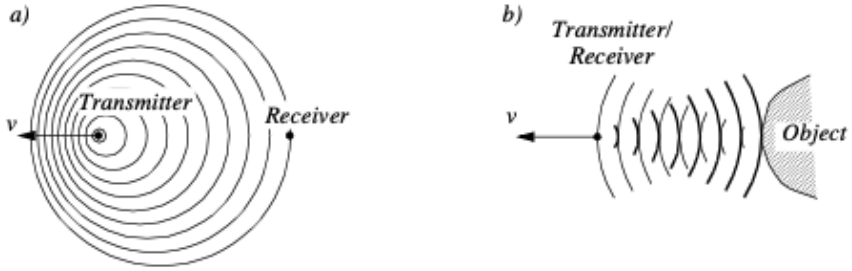



**Figure 2.13:** Doppler effect between two moving objects (a) or a moving and a stationary object(b)

reference frame, it will be detected and its speed can be estimated. There are a number of sensors that inherently measure some aspect of motion or change.

For example, a pyroelectric[16] sensor detects change in heat.

[16] An example of a pyroelectric sensor.

When someone walks across the sensor's field of view, his motion triggers a change in heat in the sensor's reference frame. In the next subsection, we describe an important type of motion detector based on the Doppler effect. These sensors represent a well-known technology with decades of general applications behind them.

> For fast-moving AMRs such as autonomous highway vehicles and unmanned flying vehicles, Doppler-based motion detectors are the obstacle detection sensor of choice.

### Doppler Effect

Anyone who has noticed the change in siren pitch when an ambulance approaches and then passes by is familiar with the Doppler effect.[17]

[17] For anyone who needs a bit more information, it is the change in the frequency of a wave in relation to an observer who is moving relative to the source of the wave. The Doppler effect is named after the physicist Christian Doppler, who described the phenomenon in 1842. A common example of Doppler shift is the change of pitch heard when a vehicle sounding a horn approaches and recedes from an observer. Compared to the emitted frequency, the received frequency is higher during the approach, identical at the instant of passing by, and lower during the recession.

A transmitter emits an electromagnetic or sound wave with a frequency f t. It is either received by a receiver **Fig.** 2.13(a) or reflected from an object **Fig.** 2.13 (b). The measured frequency $f_r$ at the receiver is a function of the relative speed $v$ between transmitter and receiver according to

$$f_r = f_t \frac{1}{1 + \frac{v}{c}}$$

if the transmitter is moving and

$$f_r = f_t \left(1 + \frac{v}{c}\right)$$

if the receiver is moving. In the case of a reflected wave **Fig.** 2.13 (b) there is a factor of two introduced, since any change x in relative separation affects the round-trip path length by $2x$.

> In such situations it is generally more convenient to consider the change in frequency $\Delta f$, known as the Doppler shift, as opposed to the Doppler frequency notation above.

$$\Delta f = f_t - f_r = \frac{2 f_t \upsilon \cos\theta}{c} \qquad \text{and} \qquad \upsilon = \frac{\Delta f c}{2 f_t \cos\theta}$$

A current application area is both autonomous and manned highway vehicles. Both micro- wave and laser radar systems have been designed for this environment. Both systems have equivalent range, but laser can suffer when visual signals are deteriorated by environmental conditions such as rain, fog, etc. Commercial microwave radar systems are already avail- able for installation on highway trucks. These systems are called VORAD (vehicle on-board radar) and have a total range of approximately 150m. With an accuracy of approximately 97%, these systems report range rate from 0 to 160 km/hr with a resolution of 1 km/ hr. The beam is approximately 4ř wide and 5ř in elevation. One of the key limitations of radar technology is its bandwidth. Existing systems can provide information on multiple targets at approximately 2 Hz.

## 2.3 Vision Based Sensors

Vision is our most powerful sense. It provides us with an enormous amount of information about the environment and enables rich, intelligent interaction in dynamic environments. It is therefore not at all surprising that a great deal of effort has been devoted to providing machines with sensors which can at least try to mimic the capabilities of the human vision system.

The first step in this process is the creation of sensing devices that capture the same raw information which is the light the human vision system uses. The main topics which will be described are the two (2) current technologies for creating vision sensors:

1. CCD,
2. CMOS.

Of course, these sensors have specific limitations in performance compared to the human eye, and it is important to understand these limitations. Later sections describe vision-based sensors which are commercially available, similar to the sensors discussed previously, along with their disadvantages and most popular applications.

**Figure 2.14:** Sony ICX493AQA 10.14-megapixel APS-C (23.4 Œ 15.6 mm) CCD from digital camera Sony DSLR-A200 or DSLR-A300, sensor side [37].

### CCD and CMOS Sensors

When it comes to the marketplace, CCD is the most popular fundamental ingredient for robotic vision systems.[18] The CCD chip, which you can see in **Fig.** 2.14 is an array of light-sensitive picture elements, or pixels, usually with between 20 000 and 2 million pixels total.

Each pixel can be thought of as a light-sensitive, discharging capacitor that is 5 to 25 ţm in size. First, the capacitors of all pixels are fully charged, then the integration period begins. As photons of light strike each pixel, the electrons are liberated, which are captured by electric fields and retained at the pixel. Over time, each pixel accumulates a varying level of charge based on the total number of photons that have struck it. After the integration period is complete, the relative charges of all pixels need to be frozen and read.

In a CCD, the reading process is performed at one corner of the CCD chip.[19] The bottom row of pixel charges are transported to this corner and read, then the rows above shift down and the process repeats. This means that each charge must be transported across the chip, and it is critical the value be preserved.

> This requires specialised control circuitry and custom fabrication techniques to ensure the stability of transported charges.

[18] Willard Boyle and George E. Smith invented the CCD in 1969 at AT&T Bell Labs. Their original idea was to create a memory device. However, with its publication in 1970, other scientists began experimenting with the technology on a range of applications. Astronomers discovered that they could produce high-resolution images of distant objects, because CCDs offered a photo-sensitivity one hundred times greater than film [38].

[19] Because the entire array is read through a single amplifier the output can be highly optimised to give very low noise and extremely high dynamic range. CCDs can have over 100 dB dynamic range with less than 2e of noise [38].

The photo-diodes used in CCD chips[20] are **NOT** equally sensitive to all frequencies of light. They are sensitive to light between 400 nm and 1000 nm wavelength.[21]

[20] This also includes CMOS as well.

[21] This number range is usually given for easier numbers as both CCD and CMOS have sensitivity values at approximately 350 - 1050 nm.

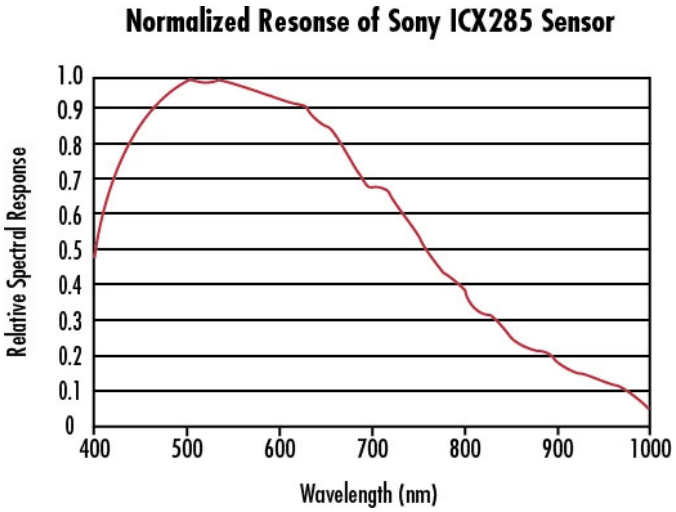



**Figure 2.15:** Normalized Spectral Response of a Typical Monochrome CCD.

It is important to remember that photodiodes are **less sensitive to the ultraviolet** part of the spectrum and are overly **sensitive to the infrared** portion (e.g. heat) which you can see in **Fig.** 2.15. You can see that the basic light-measuring process is colourless.[22]

[22] It is just measuring the total number of photons that strike each pixel in the integration period regardless whether the light hitting it is blue, red or green.

There are two (2) common approaches for creating color images. If the pixels on the CCD chip are grouped into 2-by-2 sets of four (4), then red, green and blue dyes can be applied to a colour filter so each individual pixel receives only light of just one color.

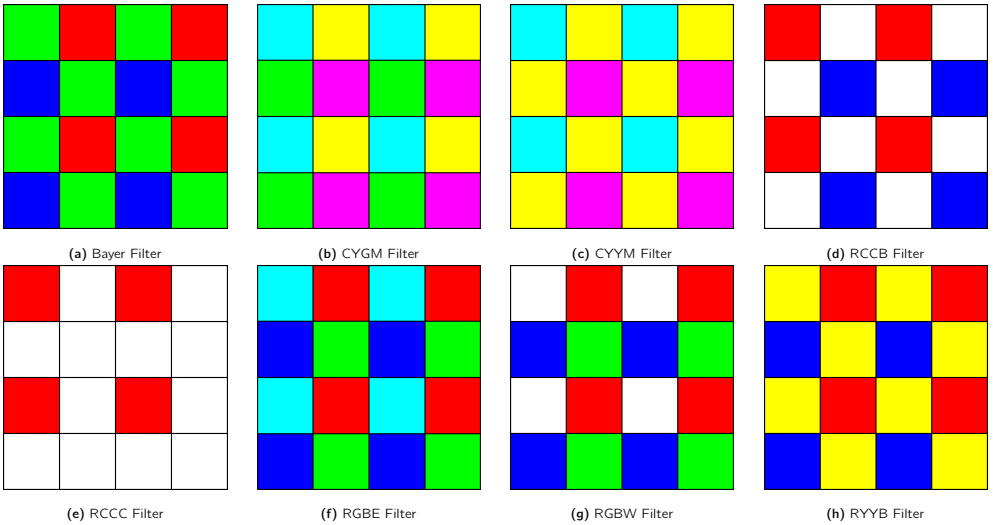



**Figure 2.16:** Types of colour filter used in commercial and industrial applications

Normally, two (2)pixels measure green while one pixel each measures red and blue light intensity. Of course, this 1-chip color CCD has a geometric resolution disadvantage.

The number of pixels in the system has been effectively cut by a factor of 4, and therefore the image resolution output by the CCD camera will be sacrificed.

The 3-chip color camera avoids these problems by splitting the incoming light into three (3) complete[23] copies. Three separate CCD chips receive the light, with one red, green or blue filter over each entire chip. Thus, in parallel, each chip measures light intensity for just one color, and the camera must combine the CCD chips' outputs to create a joint color image.

[23] Albeit, with lower resolution.

Resolution is preserved in this solution, although the 3-chip color cameras are, as one would expect, significantly more expensive and therefore more rarely used in mobile robotics.

Both 3-chip and single chip color CCD cameras suffer from the fact that photo-diodes are much more sensitive to the near-infrared end of the spectrum. This means that the overall system detects blue light much more poorly than red and green. To compensate, the gain must be increased on the blue channel, and this introduces greater absolute noise on blue[24] than on red and green. It is not uncommon to assume at least 1 - 2 bits of additional noise on the blue channel.

[24] This is generally defined as the amplifier noise.

The CCD camera has several camera parameters that affect its behavior. In some cameras, these parameter values are fixed. In others, the values are constantly changing based on built-in feedback loops. In higher-end cameras, the user can modify the values of these parameters via software embedded into the device. The iris position and shutter speed[25] regulate the amount of light being measured by the camera. The iris is simply a mechanical aperture that constricts incoming light, just as in standard 35mm cameras. Shutter speed regulates the integration period of the chip. In higher-end cameras, the effective shutter speed can be as brief at 1/30,000s and as long as 2s. Camera gain controls the overall amplification of the analog signal, prior to A/D conversion. However, it is very important to understand that, even though the image may appear brighter after setting high gain, the shutter speed and iris may not have changed at all. Thus gain merely amplifies the signal, and amplifies along with the signal all of the associated noise and error. Although useful in applications where imaging is done for human consumption (e.g. photography, television), gain is of little value to a mobile roboticist.

[25] It's the speed at which the shutter of the camera closes. A fast shutter speed creates a shorter exposure - the amount of light the camera takes in - and a slow shutter speed gives a longer exposure.

In colour cameras, an additional control exists for white balance. Depending on the source of illumination in a scene[26] the relative measurements of red, green and blue light which combine to define pure white light will change dramatically which can be seen in **Fig.** 2.17 which can also be adjusted with algorithms [39]. The human eyes compensate for all such effects in ways that are not fully understood, however, the camera can demonstrate glaring inconsistencies in which the same table looks blue in one image, taken during the night, and yellow in another image, taken during the day. White balance controls enable the user to change the relative gain for red, green and blue in order to maintain more consistent color definitions in varying contexts.

[26] For example this could be fluorescent lamps, incandescent lamps, sunlight, underwater filtered light, etc.

The key disadvantages of CCD cameras are primarily in the areas of inconstancy and **dynamic range**.

**Information** — **Dynamic Range**

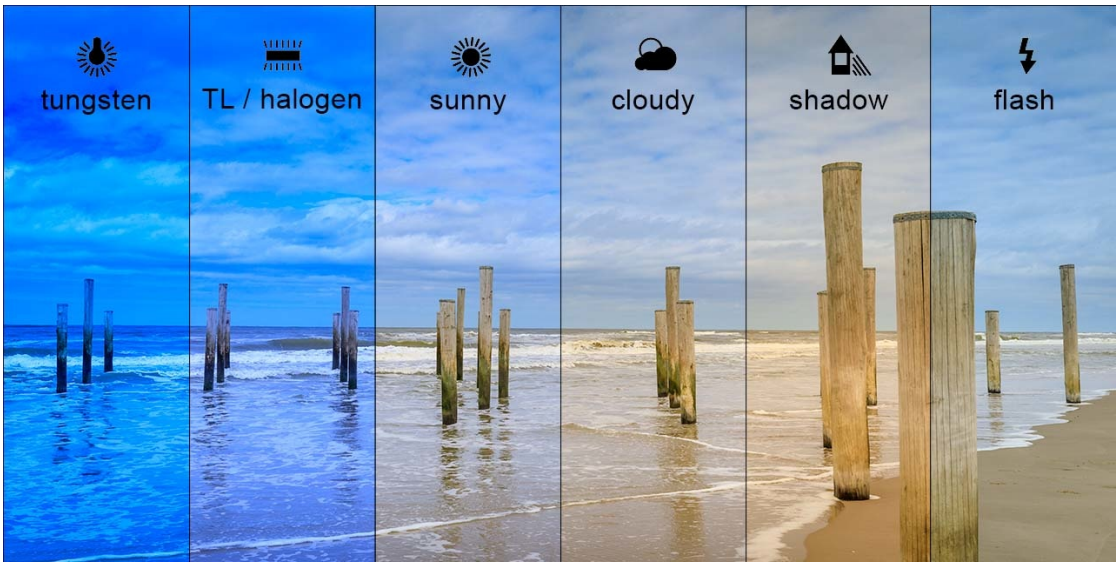



**Figure 2.17:** Example of white balance. Here the same scene is emulated to be shot under different light conditions [40].

Dynamic range in photography describes the ratio between the maximum and minimum measurable light intensities (white and black, respectively). In the real world, one never encounters true white or black - only varying degrees of light source intensity and subject reflectivity. Therefore the concept of dynamic range becomes more complicated, and depends on whether you are describing a capture device (such as a camera or scanner), a display device (such as a print or computer display), or the subject itself.

---

As mentioned above, a number of parameters can change the brightness and colours with which a camera creates its image.

> Manipulating these parameters in a way to provide consistency over time and over environments, for example ensuring a green shirt always looks green, and something dark grey is always dark grey, remains an open problem [41].

The second type of disadvantages relates to the behavior of a CCD chip in environments with extreme illumination. In cases of very low illumination, each pixel will receive only a small number of photons. The longest possible shutter speed and camera optics (i.e. pixel size, chip size, lens focal length and diameter) will determine the minimum level of light for which the signal is stronger than random error noise. In cases of very high illumination, a pixel fills its well with free electrons and, as the well reaches its limit, the probability of trapping additional electrons falls and therefore the linearity between incoming light and electrons in the well degrades. This is termed saturation[27] and can indicate the existence of a further problem related to cross-sensitivity [43]. When a well has reached its limit, then additional light within the remainder of the integration period may cause further charge to leak into neighbouring pixels, causing them to report incorrect values or even reach secondary saturation. This effect, called blooming, means that individual pixel values are **NOT** truly independent. The camera parameters may be adjusted for an environment with a particular light level, but the problem remains that the dynamic range of a camera is limited by the well capacity of

[27] Example of blooming caused by saturation of a sensor pixel. The sun is so bright in the image that there is blooming on the sun itself, leaking into the surrounding pixels, and a vertical smear across the whole image [42].

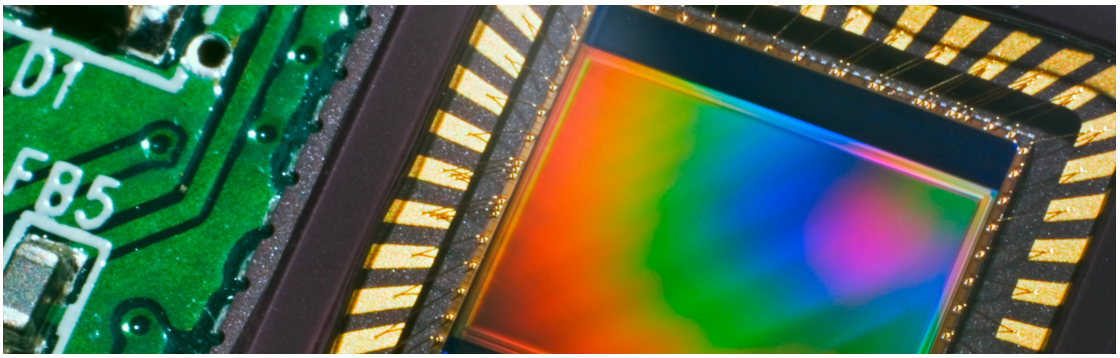


**Figure 2.18:** A close-up view of a CMOS sensor and its circuitry [44].

the individual pixels.

> For example, a high quality CCD may have pixels that can hold 40 000 electrons. The noise level for reading the well may be 11 electrons, and therefore the dynamic range will be 40,000:11, or 3,600:1, which is 35 dB.

### 2.3.1 CMOS Technology

The Complementary Metal Oxide Semiconductor (CMOS) chip is a significant departure from the CCD. Similar to CCD, it too has an array of pixels, but located alongside each pixel are several transistors specific to that pixel. Just as in CCD chips, all of the pixels accumulate charge during the integration period. During the data collection step, the CMOS takes a new approach:

> The pixel-specific circuitry next to every pixel measures and amplifies the pixel's signal, all in parallel for every pixel in the array.

Using more traditional traces from general semiconductor chips, the resulting pixel values are all carried to their destinations. CMOS has a number of advantages over CCD technologies. First and foremost, there is no need for the specialized clock drivers and circuitry required in the CCD to transfer each pixel's clock down all of the array columns and across all of its rows.[28]

> This also means that specialized semiconductor manufacturing processes are not required to create CMOS chips.

Therefore, the same production lines that create microchips can create inexpensive CMOS chips as well. The CMOS chip is so much simpler that it consumes significantly less power, it operates with a power consumption a tenth the power consumption of a CCD chip [46].

> In a AMR, power is a scarce resource and therefore this is an important advantage.

On the other hand, the CMOS chip also faces several disadvantages.

[28]-CAM80CUNX is an 8MP Ultra-lowlight MIPI CSI-2 camera capable of streaming 4K @ 44 fps. This 8MP camera is based on SONY STARVIS IMX415 CMOS image sensor [45]

- Most importantly, the circuitry next to each pixel consumes valuable real estate on the face of the light-detecting array. Many photons hit the transistors rather than the photodiode, making the CMOS chip significantly less sensitive than an equivalent CCD chip.
- CMOS, compared to CCD is still finding ground in the marketplace, and as a result, the best resolution that one can purchase in CMOS format continues to be far inferior to the best CCD chips available.
- CMOS sensors have a lower dynamic range,
- CMOS sensors have higher levels of noise.

Compared to the human eye, these chips all have worse performance, cross-sensitivity and a limited dynamic range. As a result, vision sensors today continue to be fragile. Only over time, as the underlying performance of imaging chips improves, will significantly more robust vision-based sensors for AMRs be available.

**Information** **Shot Noise**

Shot noise or Poisson noise is a type of noise which can be modeled by a Poisson process.

In electronics shot noise originates from the discrete nature of electric charge. Shot noise also occurs in photon counting in optical devices, where shot noise is associated with the particle nature of light.

**Figure 2.19:** Photon noise simulation. Number of photons per pixel increases from left to right and from upper row to bottom row [47].

### 2.3.2 Visual Ranging Sensors

Range sensing is extremely important in AMR as it is a basic input for successful obstacle avoidance. As we have seen earlier, a number of sensors are popular in robotics specifically for their ability to

recover depth estimates:

ultrasonic, laser rangefinder, optical rangefinder, etc.

It is natural to attempt to implement ranging functionality using vision chips as well. However, a fundamental problem with visual images makes rangefinding relatively difficult.

Any vision chip collapses the three-dimensional world into a two-dimensional image plane, thereby losing depth information. If one can make strong assumptions regarding the size of objects in the world, or their particular colour and reflectance, then one can directly interpret the appearance of the two-dimensional image to recover depth. But such assumptions are rarely possible in real-world AMR applications.

> Without such assumptions, a single picture does not provide enough information to recover spatial information.

The general solution is to recover depth by looking at several images of the scene to gain more information, which will be hopefully enough to at least partially recover depth. The images used must be different, so that taken together they provide additional information. They could differ in viewpoint, which would allow the use of stereo or motion algorithms.

An alternative is to create different images, not by changing the viewpoint, but by changing the camera geometry, such as the focus position or lens iris. This is the fundamental idea behind depth from focus and depth from defocus techniques. We will now look into the general approach to the depth from focus techniques as it presents a straightforward and efficient way to create a vision-based range sensor.

### 2.3.3 Depth from Focus

The depth from focus class of techniques relies on the fact that image properties not only change as a function of the scene, but also as a function of the camera parameters. The relationship between camera parameters and image properties is depicted in **Fig.** 2.20. The fundamental formula governing image formation relates the distance of the object from the lens, $d$ in **Fig.** 2.20, to the distance $e$ from the lens to the focal point, based on the focal length $f$ of the lens:

$$\frac{1}{f} = \frac{1}{d} + \frac{1}{e}$$

If the image plane is located at distance $e$ from the lens, then for the specific object voxel[29] depicted, all light will be focused at a single point on the image plane and the object voxel will be focused. However, when the image plane is **NOT** at $e$, as is seen in **Fig.** 2.20, then the light from the object voxel will be cast on the image plane as a blur circle. To a first approximation, the light is homogeneously distributed throughout this blur circle, and the radius $R$ of the circle can be characterized according to the equation:

$$R = \frac{L\delta}{2e}$$

[29] A three-dimensional counterpart to a pixel.

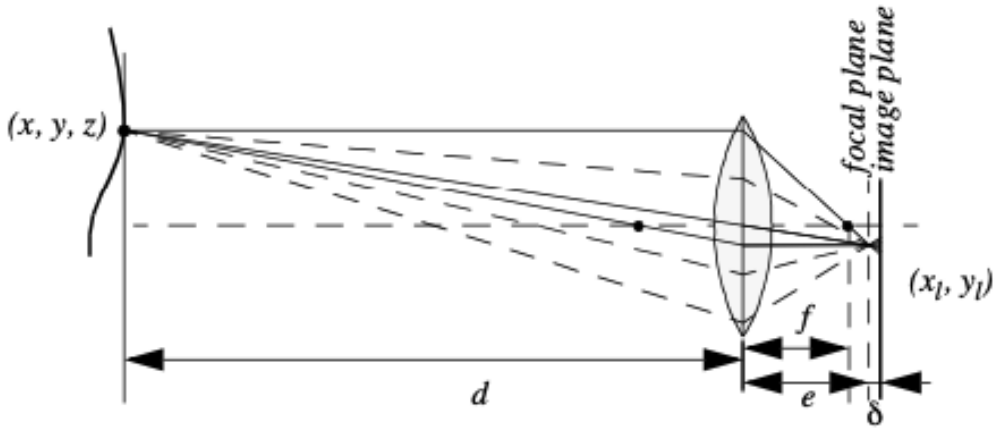



**Figure 2.20:** Depiction of the camera optics and its impact on the image. To get a sharp image, the image plane must coincide with the focal plane. Otherwise the image of the point $(x, y, z)$ will be blurred in the image as can be seen in the drawing above.

where $L$ is the diameter of the lens or aperture and $\delta$ is the displacement of the image plan from the focal point.

Given these formulae, several basic optical effects are clear.

[30] The aperture is the opening in the lens that allows light to enter the camera and onto the sensor or film.

> For example, if the aperture[30] or lens is reduced to a point, as in a pin-hole camera, then the radius of the blur circle approaches zero.

This is consistent with the fact that decreasing the iris aperture opening causes the depth of field to increase until all objects are in focus. Of course, the disadvantage of doing so is that we are allowing less light to form the image on the image plane and so this is practical only in bright circumstances The second property to be deduced from these optics equations relates to the sensitivity of blurring as a function of the distance from the lens to the object.

Suppose the image plane is at a fixed distance 1.2 from a lens with diameter L = 0.2 and focal length f = 0.5. We can see from Equation (4.20) that the size of the blur circle R changes proportionally with the image plane displacement . If the object is at distance d = 1, then from Equation (4.19) we can compute e=1 and therefore = 0.2. Increase the object distance to d = 2 and as a result =

**Figure 2.21:** Three images of the same scene taken with a camera at three different focusing positions. Note the significant change in texture sharpness between the near surface and far surface [48].

0.533. Using Equation (4.20) in each case we can compute R = 0.02 R = 0.08 respectively. This demonstrates high sensitivity for defocusing when the object is close to the lens. In contrast suppose the object is at d = 10. In this case we compute e = 0.526. But if the object is again moved one unit, to d = 11, then we compute e = 0.524. Then resulting blur circles are R = 0.117 and R = 0.129, far less than the quadrupling in R when the obstacle is 1/10 the distance from the lens. This analysis demonstrates the fundamental limitation of depth from focus techniques: they lose sensitivity as objects move further away (given a fixed focal length). Interestingly, this limitation will turn out to apply to virtually all visual ranging techniques, including depth from stereo and depth from motion. Nevertheless, camera optics can be customised for the depth range of the intended application. For example, a "zoom" lens with a very large focal length f will enable range resolution at significant distances, of course at the expense of field of view. Similarly, a large lens diameter, coupled with a very fast shutter speed, will lead to larger, more detectable blur circles. Given the physical effects summarised by the above equations, one can imagine a visual ranging sensor that makes use of multiple images in which camera optics are varied (e.g. image plane displacement ) and the same scene is captured (see Fig. 4.20). In fact this approach is not a new invention. The human visual system uses an abundance of cues and techniques, and one system demonstrated in humans is depth from focus. Humans vary the focal length of their lens continuously at a rate of about 2 Hz. Such approaches, in which the lens optics are actively searched in order to maximise focus, are technically called depth from focus. In contrast, depth from defocus means that depth is recovered using a series of images that have been taken with different camera geometries. Depth from focus methods are one of the simplest visual ranging techniques. To determine the range to an object, the sensor simply moves the image plane (via focusing) until maximizing the sharpness of the object. When the sharpness is maximised, the corresponding position of the image plane directly reports range. Some autofocus cameras and virtually all autofocus video cameras use this technique. Of course, a method is required for measuring the sharpness of an image or an object within the image. The most common techniques are approximate measurements of the sub-image gradient:

$$\text{sharpness}_1 = \sum_{x,y} \left| I(x, y) - I(x-1, y) \right| \tag{2.3}$$

$$\text{sharpness}_2 = \sum_{x,y} \left( I(x, y) - I(x-2, y-2) \right)^2 \tag{2.4}$$

A significant advantage of the horizontal sum of differences technique (Equation (4.21)) is that the calculation can be implemented in analog circuitry using just a rectifier, a low-pass filter and a high-pass filter. This is a common approach in commercial cameras and video recorders. Such systems will be sensitive to contrast along one particular axis, although in practical terms this is rarely an issue. However depth from focus is an active search method and will be slow because it takes time to change the focusing parameters of the camera, using for example a servo-controlled focusing ring. For this reason this method has not been applied to AMRs. A variation of the depth from focus technique has been applied to a AMR, demonstrating obstacle avoidance in a variety of environments as well as avoidance of concave obstacles such as steps and ledges [95]. This robot uses three monochrome cameras placed as close together as possible with different, fixed lens focus positions (Fig. 4.21).

Several times each second, all three frame-synchronised cameras simultaneously capture three images

of the same scene. The images are each divided into five columns and three rows, or 15 subregions. The approximate sharpness of each region is computed using a variation of Equation (4.22), leading to a total of 45 sharpness values. Note that Equation 22 calculates sharpness along diagonals but skips one row. This is due to a subtle but important issue. Many cameras produce images in interlaced mode. This means that the odd rows are captured first, then afterwards the even rows are captured. When such a camera is used in dynamic environments, for example on a moving robot, then adjacent rows show the dynamic scene at two different time points, differing by up to 1/30 seconds. The result is an artificial blurring due to motion and not optical defocus. By comparing only even-number rows we avoid this interlacing side effect.

Recall that the three images are each taken with a camera using a different focus position. Based on the focusing position, we call each image close, medium or far. A 5x3 coarse depth map of the scene is constructed quickly by simply comparing the sharpness values of each three corresponding regions. Thus, the depth map assigns only two bits of depth information to each region using the values close, medium and far. The critical step is to adjust the focus positions of all three cameras so that flat ground in front of the obstacle results in medium readings in one row of the depth map. Then, unexpected readings of either close or far will indicate convex and concave obstacles respectively, enabling basic obstacle avoidance in the vicinity of objects on the ground as well as drop-offs into the ground. Although sufficient for obstacle avoidance, the above depth from focus algorithm presents unsatisfyingly coarse range information. The alternative is depth from defocus, the most desirable of the focus-based vision techniques. Depth from defocus methods take as input two or more images of the same scene, taken with different, known camera geometry. Given the images and the camera geometry settings, the goal is to recover the depth information of the three-dimensional scene represented by the images. We begin by deriving the relationship between the actual scene properties (irradiance and depth), camera geometry settings and the image g that is formed at the image plane. The focused image f(x,y) of a scene is defined as follows. Consider a pinhole aperture (L=0) in lieu of the lens. For every point p at position (x,y) on the image plane, draw a line through the pinhole aperture to the corresponding, visible point P in the actual scene. We define f(x,y) as the irradiance (or light intensity) at p due to the light from P. Intuitively, f(x,y) rep- resents the intensity image of the scene perfectly in focus

## 2.4 Feature Extraction

An AMR must be able to determine its relationship to the environment by making measurements with its sensors and then using those measured signals. A wide variety of sensing technologies are available, as we discussed previously. But every sensor we have presented is imperfect:

> measurements always have error and, therefore, un- certainty associated with them.

Therefore, sensor inputs must be used in a way that enables the robot to interact with its environment successfully in spite of measurement uncertainty. There are two (2) strategies for using uncertain sensor input to guide the robot's behavior. One strategy is to use each sensor measurement as a raw and individual value. Such raw sensor values could for example be tied directly to robot behavior, whereby the robot's actions are a function of its sensor inputs. Alternatively, the raw sensors values could be used to update an intermediate model, with the robot's actions being triggered as a function of this model rather than the individual sensor measurements.

The second strategy is to extract information from one or more sensor readings first, gener- ating a higher-level percept that can then be used to inform the robot's model and perhaps the robot's actions directly. We call this process feature extraction, and it is this next, op- tional step in the perceptual interpretation pipeline (Fig. 4.34) that we will now discuss.

In practical terms, mobile robots do not necessarily use feature extraction and scene inter- pretation for every activity. Instead, robots will interpret sensors to varying degrees depend- ing on each specific functionality. For example, in order to guarantee emergency stops in the face of immediate obstacles, the robot may make direct use of raw forward-facing range readings to stop its drive motors. For local obstacle avoidance, raw ranging sensor strikes may be combined in an occupancy grid model, enabling smooth avoidance of obstacles meters away. For map-building and precise navigation, the range sensor values and even vision sensor measurements may pass through the complete perceptual pipeline, being sub- jected to feature extraction followed by scene interpretation to minimize the impact of indi- vidual sensor uncertainty on the robustness of the robot's map-making and navigation skills. The pattern that thus emerges is that, as one moves into more sophisticated, long-term per- ceptual tasks, the feature extraction and scene interpretation aspects of the perceptual pipe- line become essential.

### 2.4.1 Defining Feature

Features are recognizable structures of elements in the environment. They usually can be extracted from measurements and mathematically described. Good features are always perceivable and easily detectable from the environment. We distinguish between low-level fea- tures (geometric primitives) like lines, circles or polygons and high-level features (objects) such as edges, doors, tables or a trash can. At one extreme, raw sensor data provides a large volume of data, but with low distinctiveness of each individual quantum of data. Making use of raw data has the potential advantage that every bit of information is fully used, and thus there is a high conservation of information. Low level

features are abstractions of raw data, and as such provide a lower volume of data while increasing the distinctiveness of each fea- ture. The hope, when one incorporates low level features, is that the features are filtering out poor or useless data, but of course it is also likely that some valid information will be lost as a result of the feature extraction process. High level features provide maximum ab- straction from the raw data, thereby reducing the volume of data as much as possible while providing highly distinctive resulting features. Once again, the abstraction process has the risk of filtering away important information, potentially lowering data utilization.

Although features must have some spatial locality, their geometric extent can range widely. For example, a corner feature inhabits a specific coordinate location in the geometric world. In contract, a visual "fingerprint" identifying a specific room in an office building applies to the entire room, but has a location that is spatially limited to the one, particular room. In mobile robotics, features play an especially important role in the creation of environmen- tal models. They enable more compact and robust descriptions of the environment, helping a mobile robot during both map-building and localization. When designing a mobile robot, a critical decision revolves around choosing the appropriate features for the robot to use. A number of factors are essential to this decision:

**Target Environment** For geometric features to be useful, the target geometries must be readily detected in the actual environment. For example, line features are extremely useful in office building environments due to the abundance of straight walls segments while the same feature is virtually useless when navigating Mars.

**Available Sensors** Obviously the specific sensors and sensor uncertainty of the robot im- pacts the appropriateness of various features. Armed with a laser rangefinder, a robot is well qualified to use geometrically detailed features such as corner features due to the high qual- ity angular and depth resolution of the laser scanner. In contrast, a sonar-equipped robot may not have the appropriate tools for corner feature extraction.

**Computational Power** Vision-based feature extraction can effect a significant computa- tional cost, particularly in robots where the vision sensor processing is performed by one of the robot's main processors.

**Environment representation** Feature extraction is an important step toward scene inter- pretation, and by this token the features extracted must provide information that is consonant with the representation used for the environment model. For example, non-geometric vi- sion-based features are of little value in purely geometric environment models but can be of great value in topological models of the environment. Figure 4.35 shows the application of two different representations to the task of modeling an office building hallway. Each ap- proach has advantages and disadvantages, but extraction of line and corner features has much more relevance to the representation on the left. Refer to Chapter 5, Section 5.5 for a close look at map representations and their relative tradeoffs. In the following two sections, we present specific feature extraction techniques based on the two most popular sensing modalitites of mobile robotics: range sensing and visual appear- ance-based sensing.

### 2.4.2 Using Range Data

Most of today's features extracted from ranging sensors are geometric primitives such as line segments or circles. The main reason for this is that for most other geometric primitives the parametric description of the features becomes too complex and no closed form solution exists. Here we will describe line extraction in detail, demonstrating how the uncertainty models presented above can be applied to the problem of combining multiple sensor mea- surements. Afterwards, we briefly present another very successful feature for indoor mobile robots, the corner feature, and demonstrate how these features can be combined in a single representation.

#### Line Extraction

Geometric feature extraction is usually the process of comparing and matching measured sensor data against a predefined description, or template, of the expect feature. Usually, the system is overdetermined in that the number of sensor measurements exceeds the number of feature parameters to be estimated. Since the sensor measurements all have some error, there is no perfectly consistent solution and, instead, the problem is one of optimization. One can, for example, extract the feature that minimizes the discrepancy with all sensor measurements used (e.g. least squares estimation). In this section we present an optimization-based solution to the problem of extracting a line feature from a set of uncertain sensor measurements. For greater detail than is presented below, refer to [19], pp. 15 and 221.

#### Probabilistic Line Extraction

4.36. There is uncertainty associated with each of the noisy range sensor measurements, and so there is no single line that passes through the set. Instead, we wish to select the best pos- sible match, given some optimization criterion. More formally, suppose n ranging measurement points in polar coordinates x = ( , ) iii are produced by the robot's sensors. We know that there is uncertainty associated with each measurement, and so we can model each measurement using two random variables X = ( P , Q ) . In this analysis we assume that uncertainty with respect to the actual value iii of P and Q are independent. Based on Equation (4.56) we can state this formally: Furthermore, we will assume that each random variable is subject to a Gaussian probability density curve, with a mean at the true value and with some specified variance: Given some measurement point (, ) , we can calculate the corresponding Euclidean co- ordinatesasx = cos andy = sin. Iftherewerenoerror,wewouldwanttofinda line for which all measurements lie on that line: Of course there is measurement error, and so this quantity will not be zero. When it is non-zero, this is a measure of the error between the measurement point (, ) and the line, spe- cifically in terms of the minimum orthogonal distance between the point and the line. It is always important to understand how the error that shall be minimized is being measured. For example a number of line extraction techniques do not minimize this orthogonal point- line distance, but instead the distance parallel to the y-axis between the point and the line. A good illustration of the variety of

optimization criteria is available in [18] where several algorithms for fitting circles and ellipses are presented which minimize algebraic and geo- metric distances. For each specific ( , ) , we can write the orthogonal distance d between ( , ) and iiiii the line as:

# Part II

# Localisation and Mapping

# Chapter 3

# Mobile Robot Localisation

## Table of Contents



## 3.1 Introduction

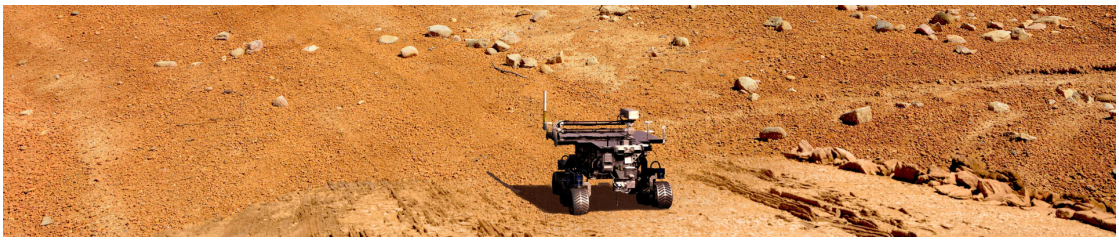


**Figure 3.1:** Navigation is one if not the most demanding and complicated task in AMR. However a successful implementation will result in a versatile AMR which can find its way in unknown environments such as exploring other planets [49].

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,

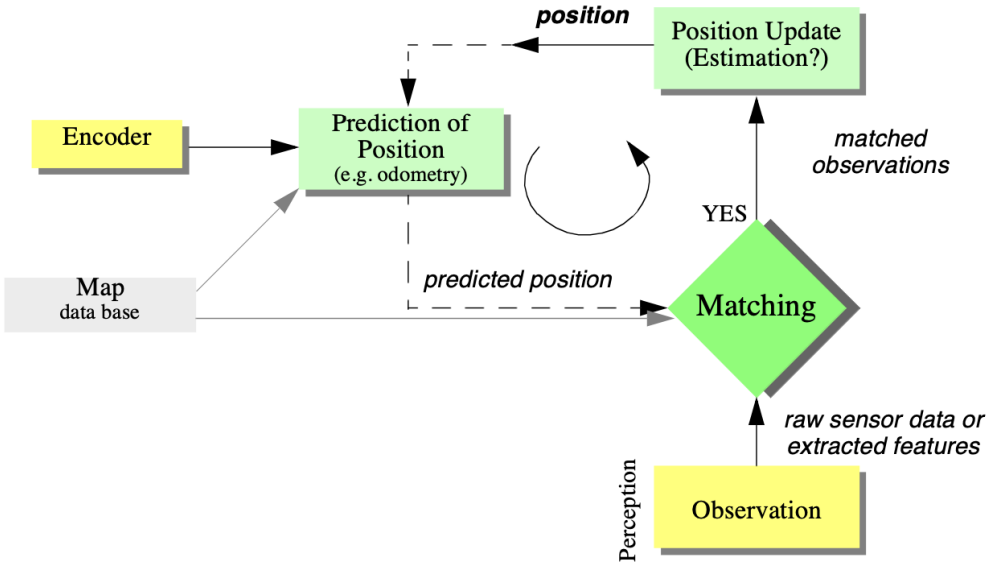



**Figure 3.2:** General schematic for mobile robot localisation.

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

**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 [50], [51], and [52]. In this chapter, we will explore the successful localisation methodologies and techniques used in academic research and industrial application [53].

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 3.2,
- Then, in Section 3.3, we will have a look at the two (2) extreme approaches to dealing with the challenge of robot localisation [54]:
  - 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.

## 3.2 The problems of Noise and Aliasing

If one could attach an accurate 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?* [55]

would always be immediately available.

Unfortunately, such a sensor is **NOT** currently practical.[1] The existing GPS network provides accuracy to within several m [56], 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.

[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 [52].

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 [57],
- 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.

[2] i.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.

> 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.

> 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.

### 3.2.1 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.

For example, a vision system used for indoor navigation[3] may use the colour values detected by its colour 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, hue[4] 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.

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

[4] One 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.

> Illumination dependency is only one example of the apparent noise in a vision-based sensor system [58]. 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.

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 interference[5] 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.

[5] The 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.

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 fusion[6] to increase the overall information content of the robot's inputs.

[6] Sensor fusion is the process of using information from several different sensors to estimate the state of a dynamic system. The resulting estimate is, in some senses, better than it would be if the sensors were used individually [59].

### 3.2.2 Sensor Aliasing

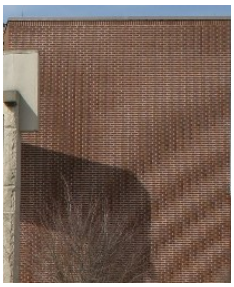
Aliasing is the second major shortcoming of AMR sensors which cause them to give little information content, further amplifying the problem of perception and localisation.

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 [60]. 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 ascolor, 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.



[7] Sensor 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 [61].

> A classical problem with sonar-based robots involves distinguishing between humans and inanimate objects in an indoor setting [62, 63].
>
> > 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.

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.

### 3.2.3 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.

[8] An 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.

[9] The 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 reckoning[9] 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 [64].[10]

[10] Using additional heading sensors (e.g. gyroscope) can help to reduce the cumulative errors, but the main problems remain the same.

[11] time increments, measurement resolution

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

- Limited resolution during integration[11]
- Misalignment of wheels causing deterministic error,
- Unequal wheel diameter, which again, causing deterministic error,
- Unequal floor contact, which can cause slipping during operation.

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

Some of the errors might be deterministic[12] (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:

**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.

## 3.3 Localisation v. Hard-Coded Navigation

**Fig.** 3.3 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 3.3:** 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.** 3.3, 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.

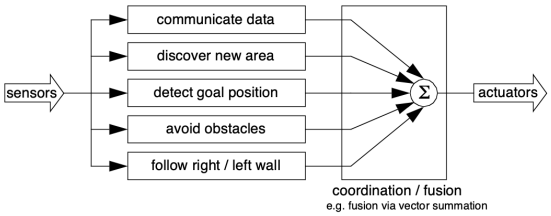



**Figure 3.4:** An Architecture for Behavior-based Navigation

The architecture of this solution to a specific navigation problem is shown in **Fig.** 3.4. 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.** 3.5.

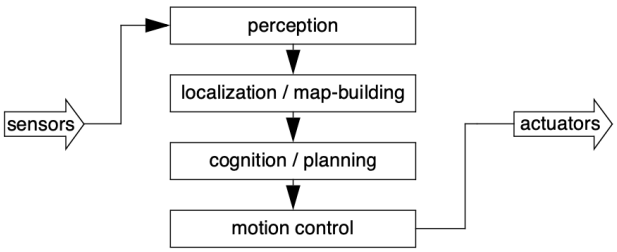



**Figure 3.5:** An Architecture for Map-based (or model-based) Navigation

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 transpar-

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,[13] 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.

[13] As in if the robot gets the wrong idea about its environment and draws the wrong map.

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.

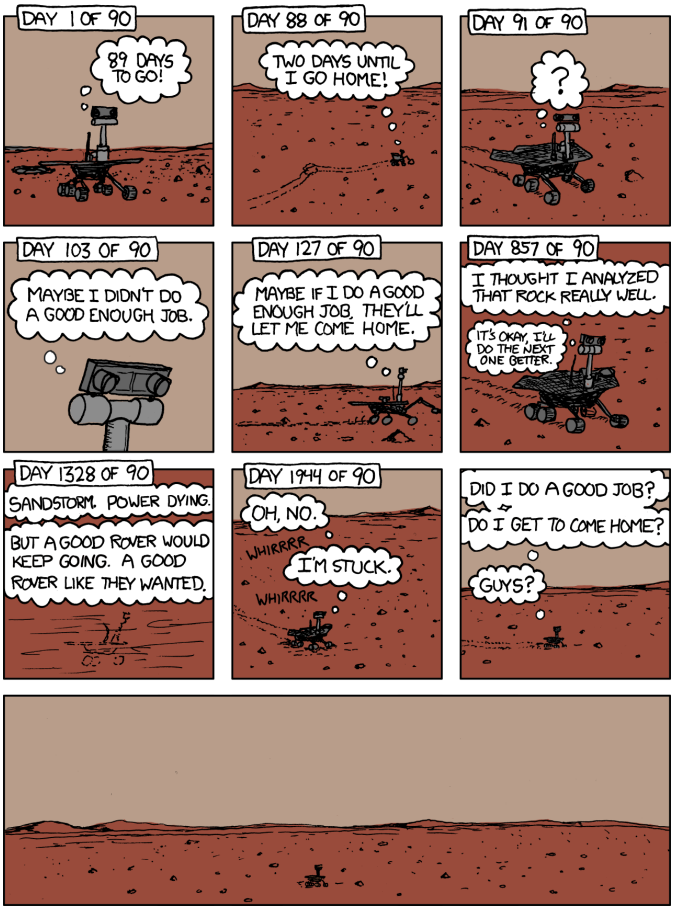



**Figure 3.6:** On January 26th, 2274 Mars days into the mission, NASA declared Spirit a 'stationary research station', expected to stay operational for several more months until the dust buildup on its solar panels forces a final shutdown [65].

## 3.4 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.

### 3.4.1 Single Hypothesis Belief

The single hypothesis belief representation is the most direct possible postulation of an AMR's position [66].

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

In **Fig.** 3.7, three (3) examples of a single hypothesis belief are shown using three different map representations of the same actual environment shown in **Fig.** 3.7a. In **Fig.** 3.7b, a single point is geometrically annotated as the robot's position in a continuous two-dimensional geometric map.

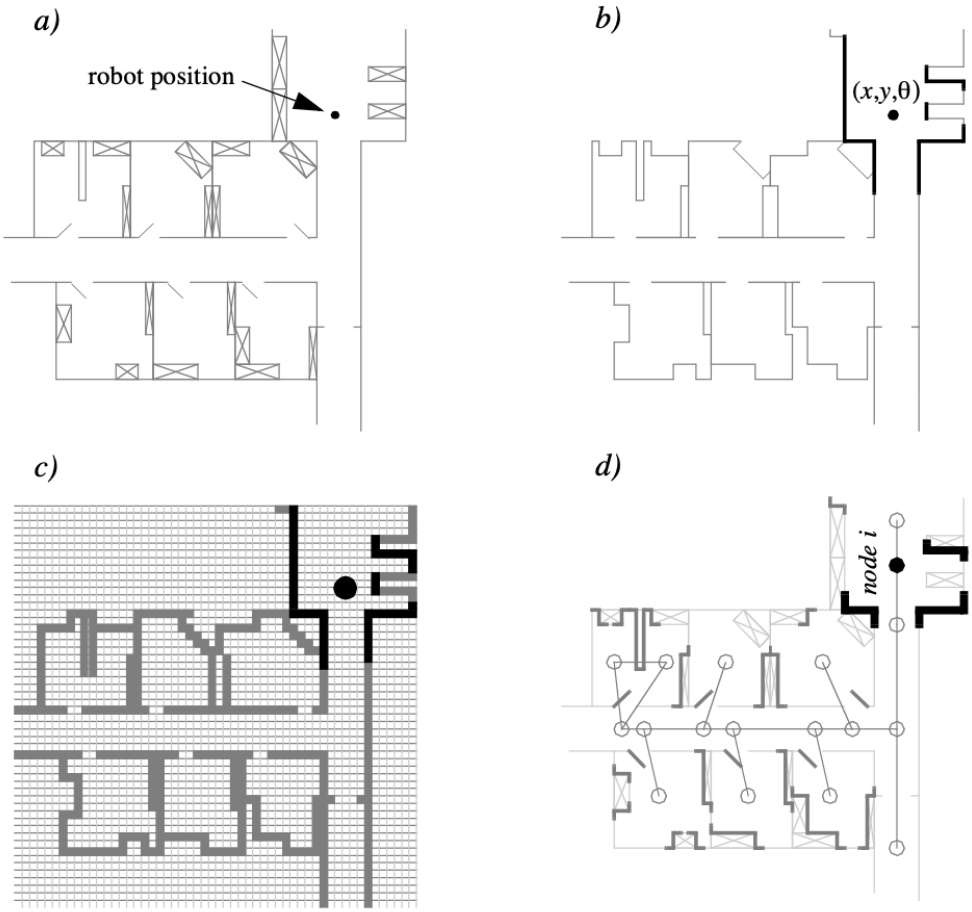



**Figure 3.7:** The three (3) examples of single hypotheses of position using different map representation. **a)** real map with walls, doors and furniture **b)** line-based map -> around 100 lines with two parameters **c)** occupancy grid based map -> around 3000 gird cells sizing 50x50 cm **d)** topological map using line features (Z/S-lines) and doors -> around 50 features and 18 nodes

In **Fig.** 3.7c, 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.** 3.7d, 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.

### 3.4.2 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] notate the robot's position belief using an X,Y point in the two-dimensional environment as the mean $\mu$ plus a standard deviation parameter $\sigma$, 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 posN sible 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.

## 3.5 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.

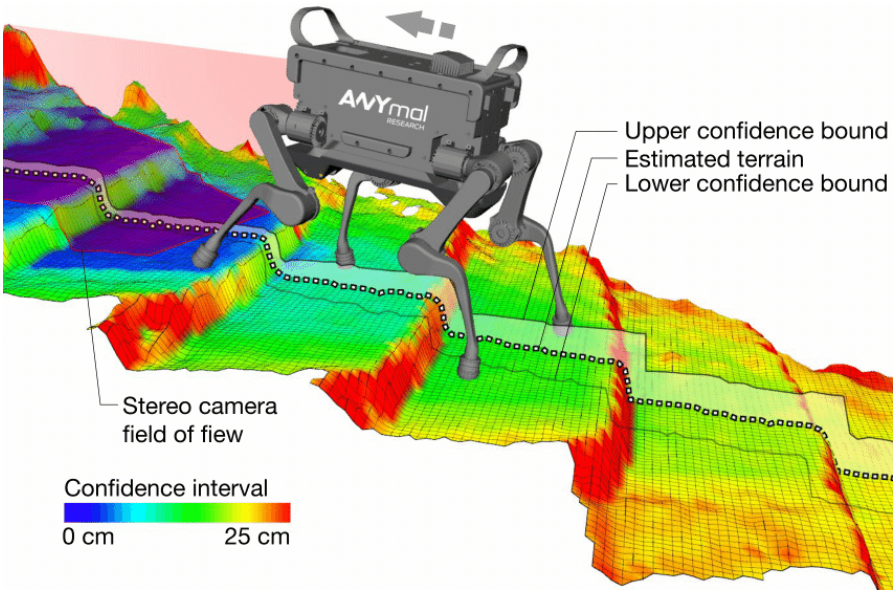



**Figure 3.8:** The presented robot-centric mapping framework enables mobile robots to create consistent elevation maps of the terrain. Mapping does not necessarily need to be done only in 2D as robots which will be used in outdoor environment would need the height of the map as well [67].

### 3.5.1 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.** 3.9, is a 2D representation in which polygons represent all obstacles in a continuos-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.

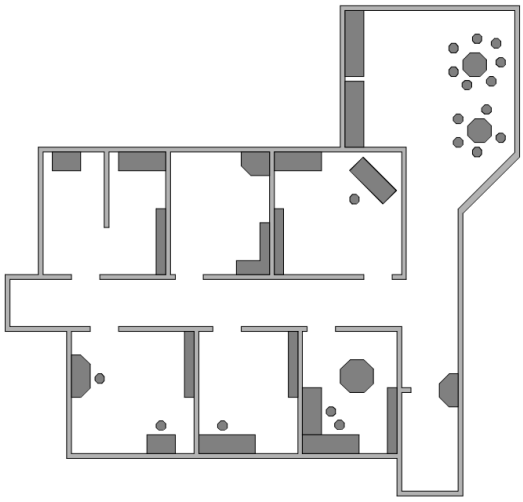


**Figure 3.9:** A continous representation using polygons as environmental obstacles.

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 related 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,[14] sacrificing map felicity for the sake of computational speed.

[14] A 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.

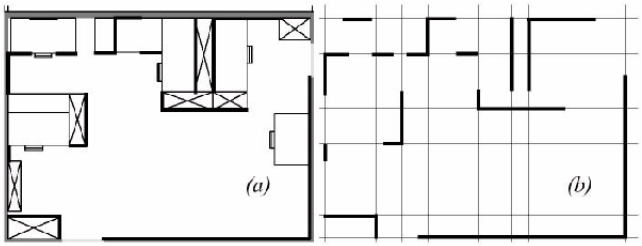



**Figure 3.10:** Example of a continuous-valued line representation of EPFL. left: real map right: representation with a set of infinite lines.

**Fig.** 3.10 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 continuos 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.

### 3.5.2 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 rep- resentation 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 continuos real environment into a discrete approximation for the map. Such a transforma- tion 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 in- exact 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 cer- tain 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 poten- tially very small memory requirements in large, sparse environments. In contrast, the occu- pancy 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, regard- less 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 environ- ment 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 de- composition 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 us- ing 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 connec- tivity. 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 to- pological graph. Second, it must have a means for traveling between nodes using robot mo- tion. 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. How- ever, 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 rep- resentations. Yet, a chief motivation of the topological approach is that the environment may contain im- portant 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. Ex- amples of working systems operating with this landmark-based strategy have also demon- strated 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 repro- duce 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 function- al requirements (e.g. required goal precision and accuracy).

### 3.5.3 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 map- ping 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 robots[15] 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.

[15] An Example of a museum tour guide robot used in the National Museum of Korea [68].

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.

> 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.[16]

[16] Of course with the use of a GNSS, the localisation problem may completely be solved, however in cost saving measures one would wish to avoid the use of them as they can be expensive.

We end this section with one final open challenge that represents one of the fundamental ac- ademic 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.

## 3.6 Probabilistic Map-Based Localisation

### 3.6.1 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 t t-1 state 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:

$$s_t' = \text{Act}\left(o_t, s_{t-1}\right) \tag{3.1}$$

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 tt state representing the

robot's current position:

$$s_t = \text{See}\left(i_t, s'_{t-1}\right) \tag{3.2}$$

The perception model See and sometimes the action model Act are abstract functions of both the map and the robot's physical configuration.[17]

[17] such 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 singe, well-defined Gaussian probability density function, and therefore retains just a $\mu$ and $\sigma$ 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 [69]:

- 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 un- imodal, 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 [70] or by combining both methods to create a hybrid localization system [69].

We will now look at them in great detail.

### 3.6.2 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 subse- quent subsections provide case studies, one robot implementing a simple feature-driven to- pological representation of the environment [71], and the other using a geometric grid-based map [70].

#### 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 $t$ 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:

$$P(A \wedge B) = P(A|B)\, P(B) \tag{3.3}$$

The equation given in Eq. (3.3) is relatively straightforward, as the probability of both $A$ and[18] $B$ 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:

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

$$P(A \wedge B) = P(B|A)\, P(A) \tag{3.4}$$

Using both Eq. (3.3) and Eq. (3.4) together, we can derive Bayes formula for computing $P(A|B)$:

$$P(A|B) = \frac{P(B|A)\, P(A)}{P(B)} \tag{3.5}$$

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. (3.2) 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 $P(r = l)$:

$$P(l|i) = \frac{P(i|l)\, P(l)}{P(i)} \tag{3.6}$$

The value of $P(l|i)$ is key to Eq. (3.6), 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. (3.6) 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. (3.1). 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:

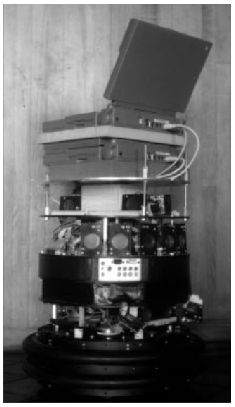
$$P\left(l_t|o_t\right) = \int P\left(l_t|l'_{t-1}, o_t\right) P\left(l'_{t-1}\right) \, dl'_{t-1} \tag{3.7}$$

Thus, the total probability for a specific position l is built up from the individual contribu- tions 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 as- sumption 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 environ- ment 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 environ- ment. 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, contain- ing 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 sen- sors to navigate from a chosen starting position to a target room.



[19].

Dervish[19], 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 ar- ranged radially around the robot in a ring. Robots with such sensor configurations are sub- ject to both tripping over short objects below the ring and to decapitation by tall objects (such as ledges, shelves and tables) that

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 trig- gers.

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 sup- pressing sensor events when its angle to the hallway parallel exceeded 9 degrees. Interest- ingly, 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 ob- stacles 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 abstrac- tion 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, re- sulting 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 represen- tation 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) . 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 prob- abilities.

The perception update process for Dervish functions precisely as in Equation (5.21). Per- ceptual events are generated asynchronously, each time the feature extractor is able to rec- ognize 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 3.1:** The certainty matrix for the robot [72].

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

$$P\left(n|i\right) = P\left(i|\right) \tag{3.8}$$

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 re- alization that, because the feature extraction system only extracts 4 total features and be- cause 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 x 5 table. Dervish's certainty matrix (show 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 fea- tures 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 sys- tem 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 com- putation of p(i|n) during the perception update process.

For Dervish's particular sensory suite and for any specific environment it intends to navi- gate, humans generate a specific certainty matrix that loosely represents its perceptual con- fidence, 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 up- date 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 combi- nation of the general form of action update and perception update. The likelihood of posi- tion n given perceptual event i is calculated as in Equation (5.22):

$$P\left(l_t|o_t\right) = \int P\left(l_t|l'_{t-1},\, o_t\right) P\left(l'_{t-1}\right)\, \mathrm{d}l'_{t-1} \tag{3.9}$$

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 n' , i ) is performed by multi- tt-it plying the probability of generating perceptual event i at position n by the probability of hav- ing 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 probabil- ities, and so computational effort is minimized in Dervish by avoiding normalization alto- gether. 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 elimi- nated 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 like- lihood 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 mis- takes 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 like- lihood 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 suffi- cient for navigation of four indoor office environments: the artificial office environment cre- ated explicitly for the 1994 National Conference on Artificial Intelligence; the psychology department, the history department and the computer science department at Stanford Uni- versity. 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 infor- mation, 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 ro- bot 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 short- comings 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. There- fore, 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, Der- vish'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 explic- itly taking into account not only robot position but also the measured likelihood of each po- sition. 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 lo- calization confidence enroute.

### 3.6.3 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 densi- ties. 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 produc- ing 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 un- certainty 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 per- formance 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 appli- cation of that theory to the problem of AMR localization. Finally, the third subsec- tion will present a case study of a AMR that navigates indoor spaces by virtue of Kalman filter localization.

### 3.6.4 An Implementation of Kalman Filter

Let's make a toy example: You've built a little robot that can wander around in the woods, and the robot needs to know exactly where it is so that it can navigate.[20]

[20] The toybot is currently observing its surrounding.

We'll say our robot has a state $\boldsymbol{x}_k$, which is just a position and a velocity:

$$\boldsymbol{x}_k = (\boldsymbol{p}, \boldsymbol{v}) \tag{3.10}$$

Note that the state is just a list of numbers about the underlying configuration of your system; it could be anything. In our example it's position and velocity, but it could be data about the amount of fluid in a tank, the temperature of a car engine, the position of a user's finger on a touchpad, or any number of things you need to keep track of.

Our robot also has a GPS sensor, which is accurate to about 10 meters, which is good, but it needs to know its location more precisely than 10 meters. There are lots of gullies and cliffs in these woods, and if the robot is wrong by more than a few feet, it could fall off a cliff. So GPS by itself is not good enough.[21]

[21] The toybot has miscalculated its position.

#### 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.

In Markov localisation, the entire perception[22] is used to update each possible robot position in the belief state individually using Bayes formula.

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

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.

[23] as in Dervish.

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.** 3.11 depicts the particular schematic for Kalman filter localisation.

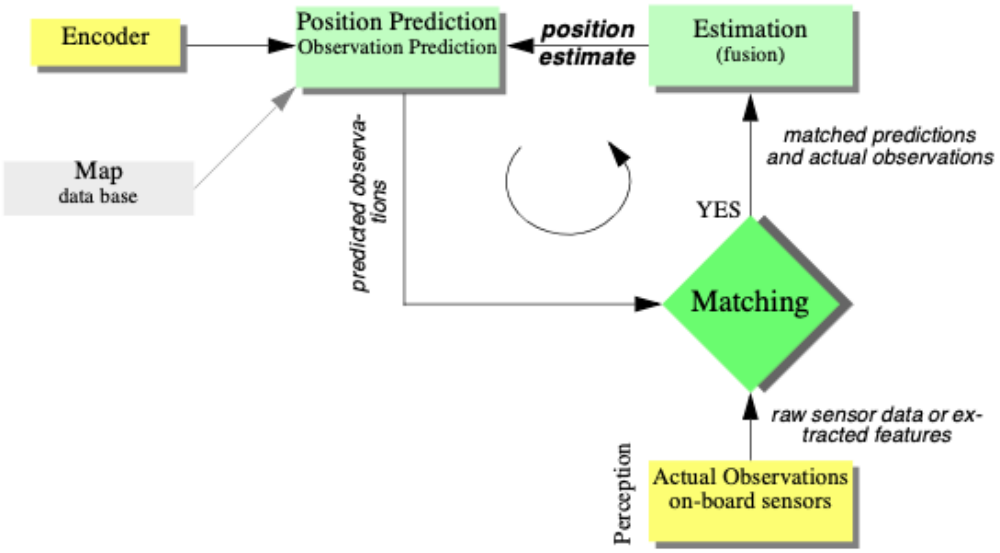



**Figure 3.11:** The schematic for the Kalman filter localisation

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 features[24] 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

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

robot identifies the best pairings between the features actually extracted during observation and the 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.

## 3.7 Other Examples of Localisation Methods

Markov localisation and Kalman filter localisation have been two extremely popular strat- egies 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 lo- calisation 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 sys- tems 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 Kal- man filter localisation communities eschew. In the following sections, we briefly identify the general strategy incorporated by each category and reference example systems, includ- ing as appropriate those that modify the environment and those that function without envi- ronmental modification.

### 3.7.1 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 successful upon reaching the next landmark.

**Fig.** 3.12 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

(a) (b)

**Figure 3.12:** 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 [73].

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.

### 3.7.2 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 re cording 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 localisa- tion using only current sensor information is unlikely to succeed. Humans often have excel- lent 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 pro- totype 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.

### 3.7.3 Positioning Beacon systems

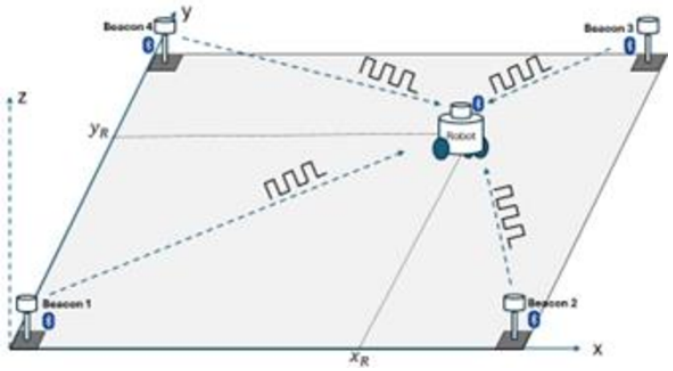
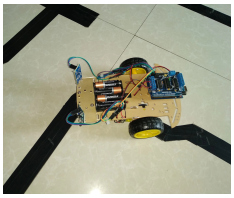



**Figure 3.13**

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.

### 3.7.4 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.



[25] A 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 [74].

> 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 give such localisation means, and therefore any change to the robot's behavior requires significant engineering and time.

## 3.8 Building Maps

Humans are excellent navigators due to their remarkable ability to build cognitive maps [75] which form the basis of spatial memory [76], [77]. 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 [78]. 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 [79], [80]. 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 3.14:** 2005 DARPA Grand Challenge winner Stanley performed SLAM as part of its autonomous driving system [81].

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 of an imprecisely known feature, the resulting position estimate becomes correlated with the feature

[26] Computational 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.

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 between position estimation and feature location estimation. Such cross-correlated maps are called stochastic maps [82]. Unfortunately, implementing such an optimal solution is computationally prohibitive.

### 3.8.1 Stochastic Map Technique

**Fig.** 3.15 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.

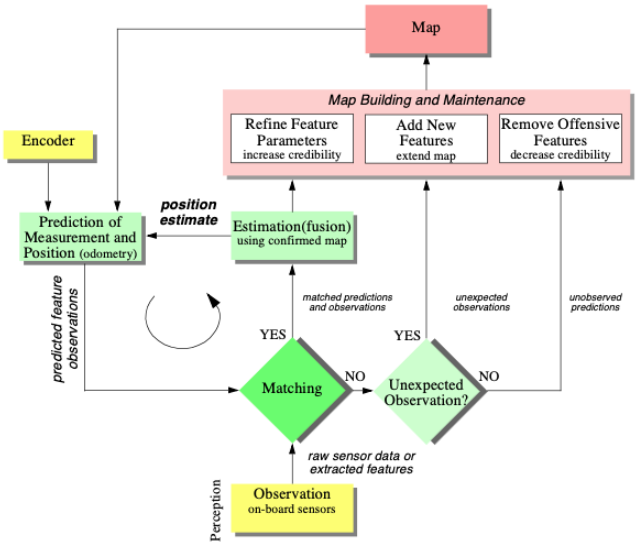



**Figure 3.15:** General schematic for concurrent localization and map building.

> 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 $\Sigma_t$ 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)):

$$M = \left\{ z_t, \Sigma_t, c_t \middle| \left(1 \leqq t \leqq n\right) \right\} \tag{3.11}$$

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 $\Sigma_t$.

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 correllated 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 [83] the following function is proposed for calculating credibility:

$$c_t(k) = 1 - \exp\left( - \left( \frac{n_s}{a} - \frac{n_u}{b} \right) \right) \tag{3.12}$$

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 $\Sigma_t$ 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:

$$\boldsymbol{X} = \begin{bmatrix} x_r(k) & x_1(k) & x_2(k) & \dots & x_n(k) \end{bmatrix}^T \tag{3.13}$$

and a system state covariance matrix:

$$\Sigma = \begin{bmatrix} C_{rr} & C_{r1} & \cdots & C_{rn} \\ C_{1r} & C_{11} & \cdots & C_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{nr} & C_{n1} & \cdots & C_{nn} \end{bmatrix} \tag{3.14}$$

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.

### 3.8.2 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:

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

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

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.** 3.16.

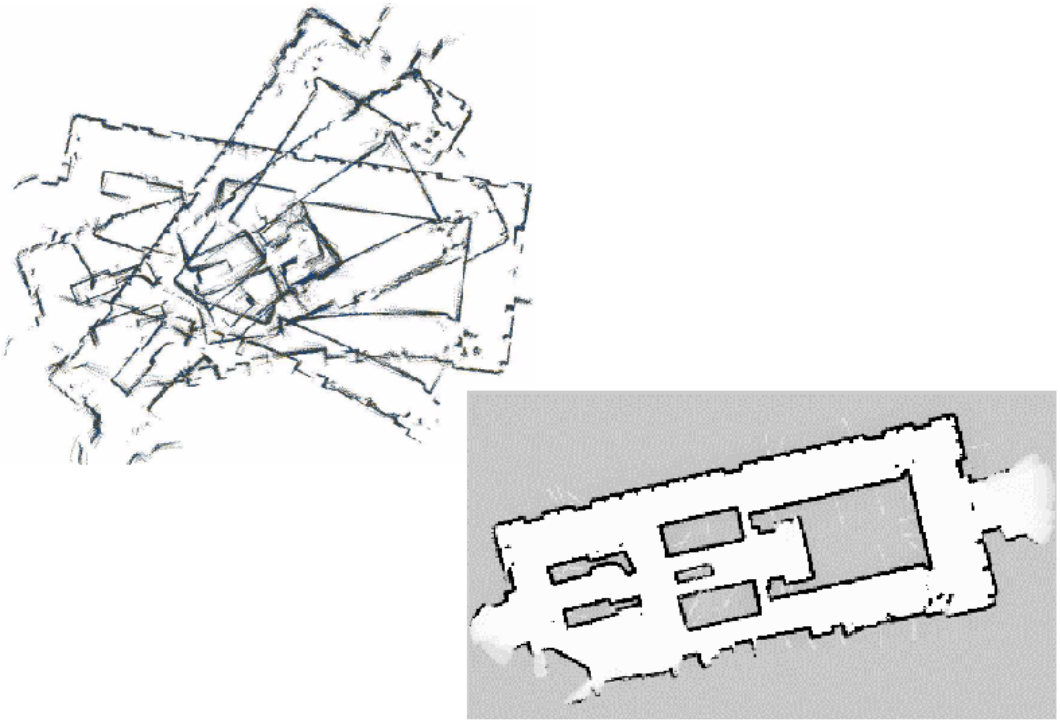


**Figure 3.16:** A naive, local mapping strategy with small local error leads to global maps that have a significant error, as demonstrated by this real-world run on the left. By applying topological correction, the grid map on the right is extracted [84].

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 [85]. 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.

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

In all such dynamic environments, an automatic mapping system should capture the salient[28] 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 too 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 3.17:** Stanford Racing and Victor Tango together at an intersection in the DARPA Urban Challenge Finals.

# Part III

# GNU/Linux Operating System

# Chapter 4

# Welcome to Linux

## Table of Contents



## 4.1 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.

C.R. 1

```bash
echo "Hello, Linux!"
```

```text
Hello, Linux!
```

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.

### 4.1.1 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.

**Figure 4.1:** Hughes telegraph, an early (1855) teleprinter built by Siemens and Halske. The centrifugal governor to achieve synchronicity with the other end can be seen [86].

> 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 4.2:** 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

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.

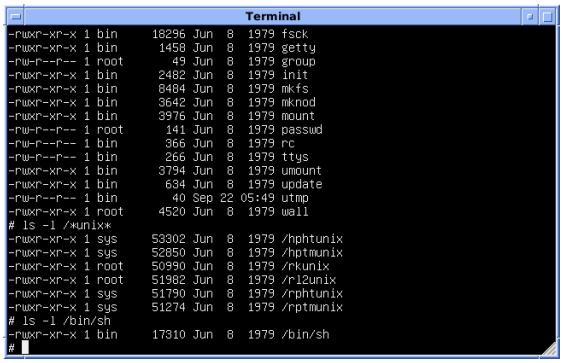



**Figure 4.3:** Bourne shell interaction on Version 7 Unix (Original).

### 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 macOS,[1] with `bash`, `tcsh`, and the KornShell also provided.

[1] This was implemented as of macOS Catalina.

> Before macOS Catalina, bash was the default shell.

### 4.1.2 Linux is a Nutshell

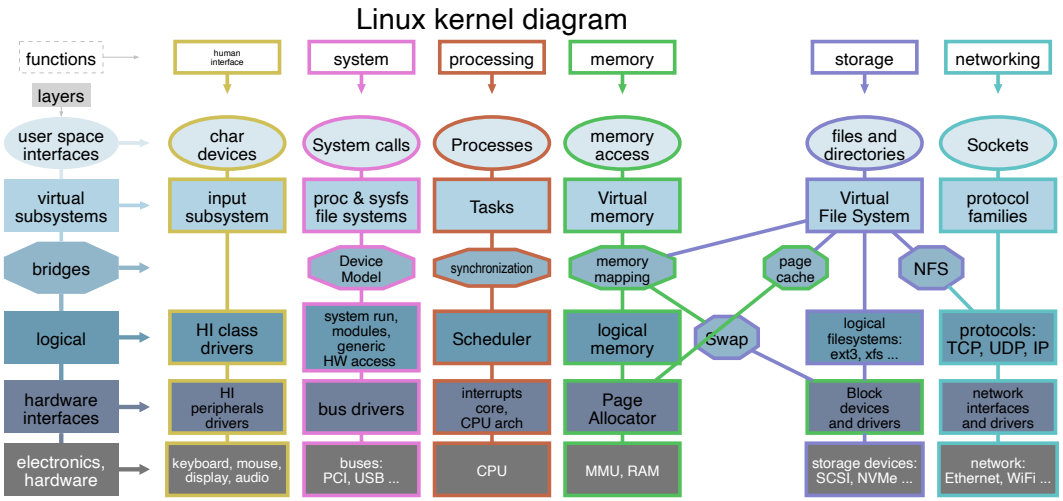



**Figure 4.4:** The kernel mapping of the Linux operating system.

#### 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 [87]. It was to create a new free operating system kernel which the resulting kernel has been marked by constant growth throughout its history.[2]

[2] The 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.

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.** 4.4.

> Imagine the kernel as the middle-man between your software and the hardware. This allows you to write a program without worrying to 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 [88] as it was mostly written with the C programming language [89].

#### 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.

[3] are 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.

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.

### 4.1.3 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]

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

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.

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 |
| **Fedora** | 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 | Fedora'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 4.1:** Most popular distributions used according to distrowatch.

## 4.2 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 4.5:** The docker logo

## 4.3 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.


[5] The logo of the podman software.

> Docker is not the only containerisation software available as there is also podman[5] 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 objects[6] 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.

[6] such as images and containers

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.

---

### 4.3.1 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:

C.R. 2
dockerfile
```dockerfile
# Declare the ubuntu version
FROM ubuntu:jammy-20250404
```

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).

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

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.

C.R. 4
dockerfile
```dockerfile
# Execute the following command as string
SHELL ["/bin/bash", "-c"]
```

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

C.R. 5 dockerfile

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

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.

C.R. 6 dockerfile

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

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, Xcfe but for our application we shall use mate which is the default environment for Linux Mint.

C.R. 7 dockerfile

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

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.

C.R. 8 dockerfile

```
# Install noVNC and Websockify
RUN git clone \
    https://github.com/AtsushiSaito/noVNC.git \
    -b add_clipboard_support /usr/lib/novnc

RUN pip install git+https://github.com/novnc/websockify.git@v0.10.0
RUN ln -s /usr/lib/novnc/vnc.html /usr/lib/novnc/index.html

# Set remote resize function enabled by default
```

C.R. 9 dockerfile

```
RUN sed -i \
    "s/UI.initSetting('resize', 'off');/UI.initSetting('resize', 'remote');/g" \
    /usr/lib/novnc/app/ui.js

# Disable auto update and crash report
RUN sed -i 's/Prompt=.*/Prompt=never/' /etc/update-manager/release-upgrades
RUN sed -i 's/enabled=1/enabled=0/g' /etc/default/apport
```

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.

C.R. 10 dockerfile

```
# Install Firefox and its configuration
RUN DEBIAN_FRONTEND=noninteractive add-apt-repository ppa:mozillateam/ppa -y && \
    echo 'Package: *' > /etc/apt/preferences.d/mozilla-firefox && \
    echo 'Pin: release o=LP-PPA-mozillateam' \
    » /etc/apt/preferences.d/mozilla-firefox && \
    echo 'Pin-Priority: 1001' » /etc/apt/preferences.d/mozilla-firefox && \
    apt-get update -q && \
    apt-get install -y \
    firefox && \
    apt-get autoclean && \
    apt-get autoremove && \
    rm -rf /var/lib/apt/lists/*

```

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

C.R. 11 dockerfile

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

```

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

C.R. 12 dockerfile

```
# Install ROS Humble version
ENV ROS_DISTRO=humble

# Install Desktop version
ARG INSTALL_PACKAGE=desktop

RUN apt-get update -q && \
    apt-get install -y curl gnupg2 lsb-release && \
    curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
    -o /usr/share/keyrings/ros-archive-keyring.gpg && \
    echo "deb [arch=$(dpkg --print-architecture)
    ↪ signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
    ↪ http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | tee
    ↪ /etc/apt/sources.list.d/ros2.list > /dev/null && \
    apt-get update -q && \
    apt-get install -y ros-${ROS_DISTRO}-${INSTALL_PACKAGE} \
    python3-argcomplete \
    python3-colcon-common-extensions \
    python3-rosdep python3-vcstool && \
    rosdep init && \
    rm -rf /var/lib/apt/lists/*

RUN rosdep update

# Install simulation package only on amd64
RUN if [ "$TARGETPLATFORM" = "linux/amd64" ]; then \
    apt-get update -q && \
    apt-get install -y \
    ros-${ROS_DISTRO}-gazebo-ros-pkgs \
    ros-${ROS_DISTRO}-ros-ign && \
    rm -rf /var/lib/apt/lists/*; \
    fi
```

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.

C.R. 13 dockerfile

```
# Download the Linux Tutorial file from repo
ARG ZIPFILE="https://github.com/dTmC0945/L-MCI-BSc-Mobile-Robotics/raw/refs/heads/main/
↪ datasets/linux-tutorials.zip"

# Create some user directories to simulate a desktop environment
RUN mkdir -p \
    /home/ubuntu/Downloads \
    /home/ubuntu/Downloads \
    /home/ubuntu/Desktop

# Download the tutorial files to their correct place
RUN cd "/home/ubuntu/Desktop" && \
    wget -O "linux.zip" "$ZIPFILE" && \
    unzip "linux.zip" && \
    rm "linux.zip"

```

C.R. 14

```dockerfile
# Enable apt-get completion after running `apt-get update` in the container
RUN rm /etc/apt/apt.conf.d/docker-clean

# Copy the entrypoint.sh into the image
COPY ./entrypoint.sh /

# Convert file for linux compatability
RUN dos2unix /entrypoint.sh
ENTRYPOINT [ "/bin/bash", "-c", "/entrypoint.sh" ]

# Define user and password
ENV USER=ubuntu
ENV PASSWD=ubuntu
```

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 of 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.

C.R. 15

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

### 4.3.2 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.

C.R. 16

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

Let's have a look at all the options given here and understand whats 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.

C.R. 17
```bash
docker container stop "ros2linux"
```

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

C.R. 18
```bash
docker container start "ros2linux"
```

# Chapter 5

# Command Line Fundamentals

## Table of Contents



## 5.1 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 servers[1], 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.

[1] A 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 5.1:** 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** 5.1.

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 UNIX[2]. 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 macOS[3] and in Windows with the Windows subsystem for Linux.

[2] A 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.

[3] newer 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.

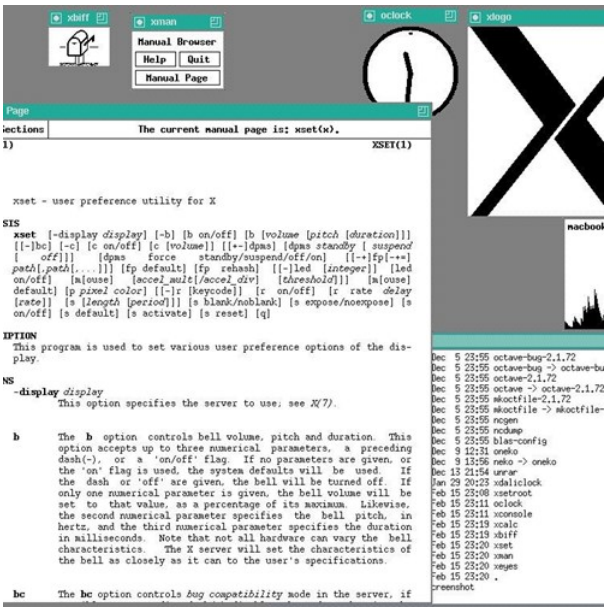



**Figure 5.1:** A graphical interface from the late 1980s, which features a TUI window for a man page, a shaped window (oclock) as well as several iconified windows. In the lower right we can see a terminal emulator running a Unix shell, in which the user can type commands as if they were sitting at a terminal. - *From Wikipedia*

## 5.2 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:

C.R. 1

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

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.

C.R. 2

```bash
ls -l /tmp
cd /usr/local
cat /etc/passwd
```

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]

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

### 5.2.1 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,now,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.

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

## 5.4 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**.

```
MAN(1)                        Manual pager utils                        MAN(1)

NAME
       man - an interface to the system reference manuals

SYNOPSIS
       man [man options] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [man options] [section] term ...
       man -f [whatis options] page ...
```

text

A `man` page[5] 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.

[5] Stands for short for manual page.

Think of the man pages as a technical reference book for your Linux distribution[6]. 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.

[6] i.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

```
man ls | head -10
```

bash

```
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

```

text

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

Here, you can see some information about the `ls` command[7]. 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.

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

The terms in square brackets[8] 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.

[8] for example `[OPTION]` and `[FILE]`

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --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.

C.R. 4

```
ls --help | head -10
```
bash

```
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print C-style escapes for nongraphic characters
      --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.

C.R. 5

```bash
help | head -10
```

```text
GNU bash, version 5.2.21(1)-release (aarch64-unknown-linux-gnu)
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.

C.R. 6

```bash
apropos list | head -10
```

```text
port-contents(1)         - List the files installed by a given port
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
```

```
port-installed(1)        - List installed versions of a given port, or all installed ports
port-list(1)             - List available ports
port-outdated(1)         - List outdated ports
port-variants(1)         - Print a list of variants with descriptions provided by a port
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
```
text

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

```
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.

# 5.5 Additional Information

## 5.5.1 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.

C.R. 7

```bash
~$ ls
```

```text
macptopbm                   make-ssl-cert
mag                         mako-render
mailmail3                   man
make                        mandb
make4ht                     manpath
makeconv                    man-recode
makedtx                     mapfile
make-first-existing-target  mapscrn
makeglossaries              match_parens
makeglossaries-lite         mathspic
makeindex                   mattrib
makejvf                     mawk
```

## 5.5.2 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.

C.R. 8

```bash
info ls | head -10
```

```text
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directory listing

10.1 ls: List directory contents
================================

The ls program lists information about files (of any type, including
directories).  Options and file arguments can be intermixed arbitrarily,
as usual.  Later options override earlier options that are incompatible.

```

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

The `info` command reads documentation in the info format[9]. 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.

[9] A 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.

**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.

[10] 1 for commands, 2 for system calls, etc...

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 troff[11] 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.

[11] A major component of a document processing system developed by Bell Labs for the Unix operating system. It is mostly outdated.

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.

---

### 5.5.3 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.

C.R. 9

```
1  whatis ls | head -10                                                    bash
```

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

# Chapter 6

# Working with Files and Folders

## Table of Contents



## 6.1 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.

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

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.

- 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:

C.R. 1

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

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:

C.R. 2

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

```text
sample.txt: ASCII 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:

[2] an acronym for American Standard Code for Information Interchange, is a character encoding standard for representing a particular set of 95 (English language focused) printable and 33 control characters - a total of 128 code points. The set of available punctuation had significant impact on the 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.

C.R. 3

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

```text
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

available there. These commands can be helpful to know about if you come across an unknown file. In the graphical environment[3], 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 CLI[4].

[3] i.e., GUI

[4] In 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 you 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.

C.R. 4

```bash
ls
```

```
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.

C.R. 5
```
ls
```
bash

```
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.

C.R. 6
```
ls -R Books/
```
bash

```
Books/:
Classics  Fantasy  Literature  Music  Poetry  Sci-Fi  Science

Books/Classics:
'Charles Dickens'  'Herman Mellvile'  'Jane Austen'

'Books/Classics/Charles Dickens':

'Books/Classics/Herman Mellvile':

'Books/Classics/Jane Austen':

Books/Fantasy:
'Brandon Sanderson'  'G. R. R. Martin'    'J.R.R. Tolkien'  'Robert Jordan'

'Books/Fantasy/Brandon Sanderson':

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

'Books/Fantasy/J.R.R. Tolkien':
Unfinished_LotR_Sequel.txt

'Books/Fantasy/Robert Jordan':

Books/Literature:
American  English  Greek  Turkish

Books/Literature/American:

Books/Literature/English:

Books/Literature/Greek:

Books/Literature/Turkish:

Books/Music:
```
text

```text
Books/Poetry:

Books/Sci-Fi:
'Arthur C. Clarke'  'Frank Herbert'  'Isaac Asimov'

'Books/Sci-Fi/Arthur C. Clarke':

'Books/Sci-Fi/Frank Herbert':

'Books/Sci-Fi/Isaac Asimov':

Books/Science:
Biology  Chemistry  Physics

Books/Science/Biology:

Books/Science/Chemistry:

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.

## 6.2 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.

C.R. 7
```bash
cd "home/ubuntu/Desktop/Linux Tutorials" && ls
```

```text
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:

C.R. 8
```bash
ls -l
```

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

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`.

C.R. 9

```bash
ls -lh
```

```text
total 20K
drwxrwxr-x 1 ubuntu ubuntu 4.0K Nov 22  2020 Books
-rw-rw-r-- 1 ubuntu ubuntu  102 Nov 20  2020 Data.txt
drwxrwxr-x 1 ubuntu ubuntu 4.0K Nov 20  2020 Folder
-rw-rw-r-- 1 ubuntu ubuntu  592 Nov 18  2020 Poem.txt
```

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 is 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.

## 6.3 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.

C.R. 10
```bash
mkdir Folder
```

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.

C.R. 11
```bash
ls -l
```

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

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`.

C.R. 12
```bash
mkdir Books/Lyrics
```

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.

C.R. 13
```bash
mkdir -p Books/Music/Beethoven
```

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 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.

## 6.4 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.

C.R. 14
```bash
ls -l
```

```text
total 20
drwxrwxr-x 1 ubuntu ubuntu 4096 May 19 06:07 Books
-rw-rw-r-- 1 ubuntu ubuntu  102 Nov 20  2020 Data.txt
drwxrwxr-x 1 ubuntu ubuntu 4096 Nov 20  2020 Folder
-rw-rw-r-- 1 ubuntu ubuntu  592 Nov 18  2020 Poem.txt
```

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

Let's write to the terminal[5], `cp Poem.txt poem_Copy.txt`. The first file name is the file you want to copy (`Poem.txt`), and the second file name is where you want to copy it to (`Poem_Copy.txt`).

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

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`.

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

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

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

```
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.

C.R. 18
```
cd - && mv Poem_Copy.txt Books/
```
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.

C.R. 19
```
cd Books && mv Poem_Copy.txt Poem_Duplicate.txt
```
bash

C.R. 20
```
rm -rf ./
```
bash

C.R. 21
```
ls
```
bash

## 6.5 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.


**Figure 6.1:** Beware of the sudo ghost.

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.

C.R. 22

```bash
ls /root
```

```text
ls: cannot open directory '/root': Permission denied
```

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

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 be. You start first use the `passwd` command to delete the password of the user (which is you)

```
sudo passwd -d username
```

After entering you 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.

## 6.6 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 6.1:** 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** 6.1.

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** 6.2.

| **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 6.2:** 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:

C.R. 23
```bash
chmod u+rwx
```

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

C.R. 24
```bash
chmod g=r
```

C.R. 25
```bash
chmod u-rwx
```

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.

## 6.7 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.

[6] It 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.

> 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.

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.

### 6.7.1 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 pointer[7] to the original one.

[7] In 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.

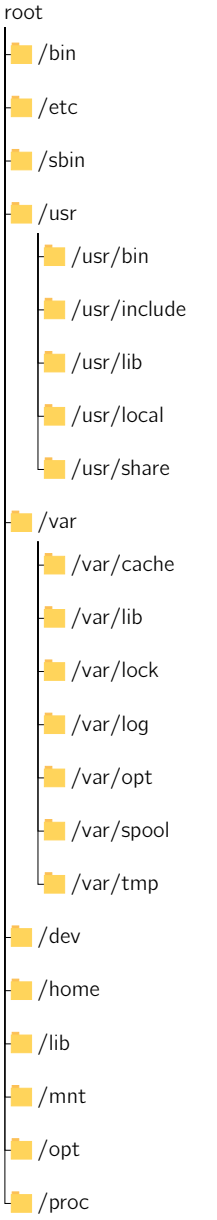
Below is a quick guide to the options the ln command has.

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

## 6.8 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]




[8] A Visual description of the linux file system

**/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`

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.

# 6.9 Common Command-Line Tools and Tasks

## 6.9.1 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.

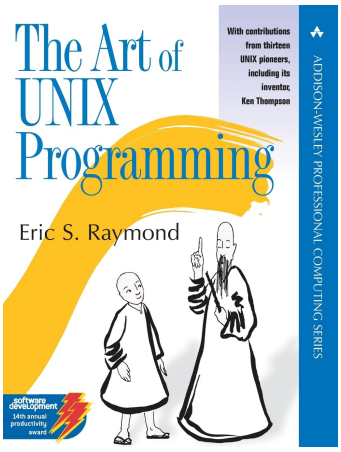



**Figure 6.2:** For anyone who is interested in the UNIX philosopy, I would suggest reading this book as it has parts written by numerous people who were the original developers of the UNIX.

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.

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:

C.R. 26

```bash
apt help
```

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

```text
apt 2.7.14 (arm64)
Usage: apt [options] command

apt is a commandline package manager and provides commands for
searching and managing as well as querying information about packages.
It provides the same functionality as the specialized APT tools,
like apt-get and apt-cache, but enables options more suitable for
interactive use by default.

Most used commands:
  list - list packages based on package names
  search - search in package descriptions
  show - show package details
  install - install packages
  reinstall - reinstall packages
  remove - remove packages
  autoremove - automatically remove all unused packages
  update - update list of available packages
  upgrade - upgrade the system by installing/upgrading packages
  full-upgrade - upgrade the system by removing/installing/upgrading packages
  edit-sources - edit the source information file
  satisfy - satisfy dependency strings

See apt(8) for more information about the available commands.
Configuration options and syntax is detailed in apt.conf(5).
Information about how to configure sources can be found in sources.list(5).
Package and version choices can be expressed via apt_preferences(5).
Security details are available in apt-secure(8).
                                        This APT has Super Cow Powers.
```

### 6.9.2 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.

C.R. 27

```bash
echo "hello"
```

```text
hello
```

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.

C.R. 28

```bash
echo "hello" | wc
```

```text
      1       1       6
```

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.

C.R. 29

```bash
echo "hello, world! from the command-line interface" | wc
```

```text
      1       6      46
```

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.

### 6.9.3 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).

# 6.10 Advanced Topics

## 6.10.1 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

C.R. 30

```bash
ls -lah /etc/*release
```

```text
-rw-r--r-- 1 root root 104 Feb  5 16:08 /etc/lsb-release
lrwxrwxrwx 1 root root  21 Feb  5 16:08 /etc/os-release -> ../usr/lib/os-release
```

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.

C.R. 31

```bash
cat /etc/*release
```

```text
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=24.04
DISTRIB_CODENAME=noble
DISTRIB_DESCRIPTION="Ubuntu 24.04.2 LTS"
PRETTY_NAME="Ubuntu 24.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.2 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
```

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

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.

C.R. 32

```
uname -a
```
bash

```
Linux 807fe6353460 6.10.14-linuxkit #1 SMP Fri Nov 29 17:22:03 UTC 2024 aarch64 aarch64
↪ aarch64 GNU/Linux
```
text

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.

### 6.10.2 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.

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

```text
               total        used        free      shared  buff/cache   available
Mem:           7.7Gi       479Mi       7.1Gi       1.5Mi       294Mi       7.2Gi
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`.

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

```text
processor    : 0
BogoMIPS    : 48.00
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 dcpodp flagm2 frint
CPU implementer    : 0x61
CPU architecture: 8
CPU variant    : 0x0
CPU part    : 0x000
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.

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

```text
Filesystem            Size  Used Avail Use% Mounted on
overlay                59G   32G   24G  58% /
tmpfs                  64M     0   64M   0% /dev
shm                   512M     0  512M   0% /dev/shm
/dev/vda1              59G   32G   24G  58% /etc/hosts
/run/host_mark/Users  461G  338G  124G  74% /home/ubuntu/Desktop/Host
tmpfs                 3.9G     0  3.9G   0% /proc/scsi
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

# Part IV

# Robot Operating System

# Chapter 7

# Installation

## Table of Contents



## 7.1 ROS 2 Humble Hawksbill

### 7.1.1 Introduction

> This section is written and edited for students who wish to install ROS on their own computers which has Linux[1] 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.

[1] Of 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."

### 7.1.2 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

[2] A 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).

C.R. 1

```bash
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-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 OS to provide access to POSIX-compliant applications. The standard also defines a command interpreter[3] 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.

[3] such as the shell.

### 7.1.3 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 Universe[4] repository is enabled.

[4] A 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

C.R. 2 bash
```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

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.

---

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

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.

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

**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.

---

### 7.1.4 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.

C.R. 5
```bash
sudo apt update && sudo apt upgrade
```

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.

### 7.1.5 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).

C.R. 6
```bash
source /opt/ros/humble/setup.bash
```

Alternatively, if you prefer to automate this action, simply type the following code in your terminal:

C.R. 7
```bash
echo "source /opt/ros/humble/setup.bash" » "~/.bashrc"
```

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.

## 7.2 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.

C.R. 8

bash

```bash
#!/bin/bash

# First check your Ubuntu Version
# For maximum compatability with ROS it needs to be 22.04 LTS

# Creating log for troubleshooting
echo "###### BEGIN ATTEMPT #######" »install_log.txt

echo "Welcome to ROS 2 Automated Installation"
echo ""
echo ""

# Accessing the Ubuntu version using AWK and piping it to grep for Regex
version=$(
    awk "/VERSION_ID/" IGNORECASE=1 /etc/*release |
        grep -Eo "[[:digit:]]+([.][[:digit:]]+)?"
       )

# Checks version for ROS Compliance
if [[ "${version}" == *"22.04"* ]]; then
    echo "Version is supported."
    sleep 1
    echo "Continuing installation..."
    sleep 1
else
    echo "Your version: "$version", What is needed: 22.04"
    echo "I am sorry but your version is not supported."
    echo "This install script will terminate"
    exit

fi

echo ""
echo "Installing UTF-8 Compliance ..."

{
    locale # check for UTF-8

    sudo apt update
    sudo apt install locales
    sudo locale-gen en_US en_US.UTF-8
    sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
    export LANG=en_US.UTF-8

    locale # verify settings
} &»install_log.txt

```

C.R. 9

bash

```
echo ""
echo "Enabling Ubuntu Universe Repositories..."

{
    sudo apt install software-properties-common
    echo | sudo add-apt-repository universe
} &»install_log.txt

echo ""
echo "Adding ROS 2 GPG Keys ..."

{
    sudo apt update
    sudo apt install curl -y
    sudo curl -sSL \
         https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
         -o /usr/share/keyrings/ros-archive-keyring.gpg
} &»install_log.txt

echo ""
echo "Adding ROS 2 to repository ..."

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

echo ""
echo "Getting Updates ..."

{
    sudo apt update
    echo yes | sudo apt upgrade
} &»install_log.txt

echo ""
echo "Installing ROS ..."

{
    echo yes | sudo apt install ros-humble-desktop
    yes | sudo apt install ros-dev-tools
} &»install_log.txt

{
    sudo apt install dbus-x11
} &»install_log.txt

echo ""
echo "Sourcing ROS file ..."
sleep 1
```

C.R. 10

bash

```
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
```

# Chapter 8

# ROS Concepts

## Table of Contents



## 8.1 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-ware[1] software based on a **strongly-typed**, **anonymous publish/-subscribe** mechanism which allows for message passing between different processes.

[1] a software that facilitates communication and data management between applications, systems, and databases.

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.

## 8.2 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.

## 8.3 Nodes - The Building Blocks

A node is an actor in the overall ROS graph, which uses a client library[2] to communicate with other nodes.

[2] This could either be `rospy` or `roscpp` 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.

## 8.4 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.

[3] This is set with the `ROS_DOMAIN_ID` environment variable which is used to isolate multiple ROS systems from each other on the same network.

> Nodes will only establish connections with others having compatible Quality of Service.[4]

[4] In 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.

As an example to see whats 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.

C.R. 1

```bash
. ~/ros2_humble/install/local_setup.bash
ros2 run demo_nodes_cpp talker
```

```text
[INFO] [1748930700.789182596] [talker]: Publishing: 'Hello World: 1'
[INFO] [1748930701.785043679] [talker]: Publishing: 'Hello World: 2'
[INFO] [1748930702.785231597] [talker]: Publishing: 'Hello World: 3'
[INFO] [1748930703.785897764] [talker]: Publishing: 'Hello World: 4'
```

Now it is running in one, open another terminal window, and type:

C.R. 2

```bash
. ~/ros2_humble/install/local_setup.bash
ros2 run demo_nodes_py listener
```

```text
[INFO] [1748930759.815469762] [listener]: I heard: [Hello World: 60]
[INFO] [1748930760.787999512] [listener]: I heard: [Hello World: 61]
[INFO] [1748930761.787663179] [listener]: I heard: [Hello World: 62]
[INFO] [1748930762.787127847] [listener]: I heard: [Hello World: 63]
```

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.

## 8.5 Communication Between Nodes

### 8.5.1 Description

ROS applications generally communicate through interfaces of one of three (3) types:

1. topics,
2. services,
3. actions.

ROS uses a simplified description language[5] 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:

[5] interface definition language.

[6] i.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.

### 8.5.2 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:

C.R. 3
```text
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

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:

C.R. 4
```text
int32 my_int
string my_string
```

**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 8.1**: Current types supported by ROS Humble.

| Type name | C++ | Python | DDS type |
|---|---|---|---|
| `bool` | `bool` | `builtins.bool` | `boolean` |
| `byte` | `uint8_t` | `builtins.bytes*` | `octet` |
| `char` | `char` | `builtins.int*` | `char` |
| `float32` | `float` | `builtins.float*` | `float` |

Continued on next page

**Table 8.1**: 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:

C.R. 5

text

```
int32[] unbounded_integer_array
int32[5] five_integers_array
int32[<=5] up_to_five_integers_array

string string_of_unbounded_size
string<=10 up_to_ten_characters_string

string[<=5] up_to_five_unbounded_strings
string<=10[] unbounded_array_of_strings_up_to_ten_characters_each
string<=10[<=5] up_to_five_strings_up_to_ten_characters_each
```

**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.

**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]

[7] types **NOT** present in the built-in-types which applies to all nested messages.

Defining a default value is done by adding a third element to the field definition line. For example:

C.R. 6
```text
fieldtype fieldname fielddefaultvalue
```

An example implementation would be:

C.R. 7
```text
uint8 x 42
int16 y -2000
string full_name "John Doe"
int32[] samples [-200, -100, 0, 100, 200]
```

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:

C.R. 8
```text
constanttype CONSTANTNAME=constantvalue
```

For example:

C.R. 9
```text
int32 X=123
int32 Y=-123
string FOO="foo"
string EXAMPLE='bar'
```

> 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:

C.R. 10
text
```
uint32 request
---
uint32 response
```

We can of course get much more complicated:[8]

[8] if we want to refer to a message from the same package we must **NOT** mention the package name.

C.R. 11
text
```
uint32 a
uint32 b
---
uint32 sum
```

> 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.

[9] This could be many seconds up to minutes.

Action definitions have the following form:

C.R. 12
text
```
<request_type> <request_fieldname>
---
<response_type> <response_fieldname>
---
<feedback_type> <feedback_fieldname>
```

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:

C.R. 13

text

```
int32 order
---
int32[] sequence
---
int32[] sequence
```

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.

## 8.6 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.

### 8.6.1 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.

### 8.6.2 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.

[10] If needed though, it can find out.

### 8.6.3 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:

C.R. 14

```
uint32 field1
string field2
```

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.

## 8.7 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:

C.R. 15

```
uint32 request
---
uint32 response
```

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:

[11] but is in a different namespace.

- the service server,
- the service client.

### 8.7.1 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:

C.R. 16

```
uint32 a
uint32 b
---
uint32 sum
```

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.

## 8.8 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:

C.R. 17

```text
<request_type> <request_fieldname>
---
<response_type> <response_fieldname>
---
<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.

### 8.8.1 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:

C.R. 18

```text
int32 order
---
int32[] sequence
---
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.

### 8.8.2 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.

## 8.9 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**.

[12] though 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.

### 8.9.1 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.

[13] A way for two or more computer programs or components to communicate with each other. It is a type of software interface, offering a service to other pieces of software.

### 8.9.2 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.

## 8.10 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:

C.R. 19

```bash
ros2 --help
```

```text
usage: ros2 [-h] [--use-python-default-buffering] Call `ros2 <command> -h` for more 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.

## 8.11 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 9

# Command Line Tools

## Table of Contents



## 9.1 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.

[1] In 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 distributions[2] 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.

[2] or "distros", e.g. Dashing and Eloquent.

In other words, we won't be able to use ROS. To source, simply type the following:

C.R. 1
```bash
# Replace ".bash" with your shell if you're not using bash
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/humble/setup.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]

[3] For our case it is `.bashrc`

C.R. 2
```bash
# Replace ".bash" with your shell if you're not using bash
echo "source /opt/ros/humble/setup.bash" » ~/.bashrc
```

**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:

C.R. 3
```bash
printenv | grep -i ROS
```

**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.

```text
ROS_VERSION=2
ROS_PYTHON_VERSION=3
AMENT_PREFIX_PATH=/opt/ros/humble
PYTHONPATH=/opt/ros/humble/lib/python3.10/site-packages:/opt/ros/humble/local/lib/
↪ python3.10/dist-packages
LD_LIBRARY_PATH=/opt/ros/humble/opt/rviz_ogre_vendor/lib:/opt/ros/humble/lib/aarch64-
↪ linux-gnu:/opt/ros/humble/lib
ROS_LOCALHOST_ONLY=0
PATH=/opt/ros/humble/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
ROS_DISTRO=humble
```

> 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:

C.R. 4
```bash
export ROS_DOMAIN_ID=<your_domain_id>
```

To maintain this setting between shell sessions, we can also add the command to our shell startup script `.bashrc`:

C.R. 5
```bash
echo "export ROS_DOMAIN_ID=<your_domain_id>" >> ~/.bashrc
```

**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:

C.R. 6
```bash
export ROS_LOCALHOST_ONLY=1
```

Of course, to maintain it across different shell session we can write this command to our `.bashrc`:

C.R. 7
```bash
echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

[4] Localhost 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.

## 9.2 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.

C.R. 8

```bash
sudo apt update
sudo apt install ros-humble-turtlesim
```

```text
0 upgraded, 0 newly installed, 0 to remove and 6 not upgraded.
```

To check if the package is installed, run the following command in the terminal:

C.R. 9

```bash
ros2 pkg executables turtlesim
```

which should return a list of turtlesim's executable options:

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

**Installing and Starting Turtlesim** Let's begin the tutorial by starting turtlesim:

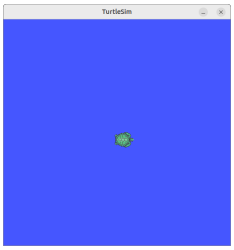
C.R. 10

```bash
ros2 run turtlesim turtlesim_node
```

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.

```text
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-ubuntu'
[INFO] [1748967100.900464593] [turtlesim]: Starting turtlesim with node name /turtlesim
[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]




[5] Here 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.

**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:

C.R. 11

```bash
ros2 run turtlesim turtle_teleop_key
```

```text
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
'Q' to quit.
```

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:

C.R. 12

```bash
echo "~~ Node information ~~" &&
    ros2 node list &&
    echo "~~ Topic information ~~" &&
    ros2 topic list &&
    echo "~~ Service information ~~" &&
    ros2 service list &&
    echo "~~ Action information ~~" &&
    ros2 action list
```

```text
~~ Node information ~~
/teleop_turtle
/turtlesim
~~ Topic information ~~
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
~~ Service information ~~
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
~~ Action information ~~
/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:

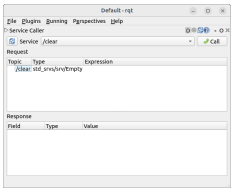
C.R. 13

```bash
sudo apt update
sudo apt install '~nros-humble-rqt*'
```

Once installed, running it is pretty straightforward:

```bash
rqt
```
C.R. 14

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]



[6] The view of `rqt` with only turtlesim and teleop running.

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.

**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`.

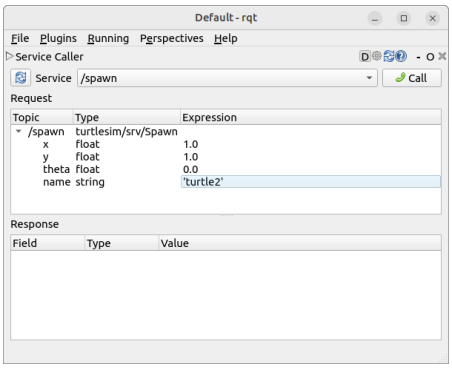


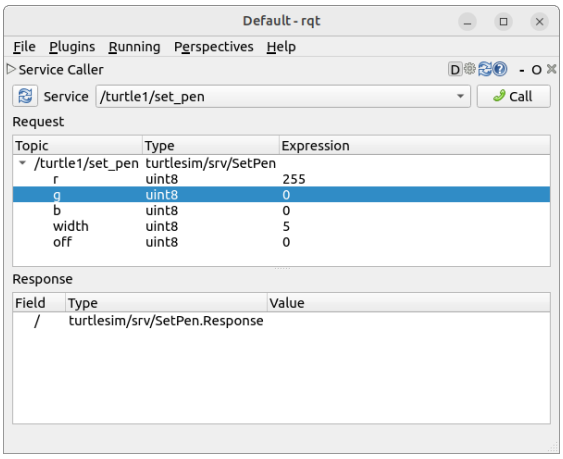

**Figure 9.1:** Generating a new turtlebot to be spawned.

**Changing the Trail Parameters** Now let's give `turtle1` a unique pen using the `/set_pen` service:

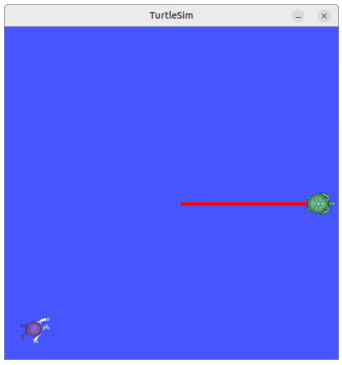
The values for `r`, `g`, and `b`, which are between 0 and 255, set the colour of the pen `turtle1` draws with, and width sets the thickness of the line.

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.




(a) The parameters of the `/set_pen` service.




(b) The trail line of `turtle1` has changed

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:

C.R. 15

```bash
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Now, we can move `turtle2` when this terminal is active, and `turtle1` when the other terminal running `turtle_teleop_key` is active.

## 9.3 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.

[7] e.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 executable[8] can contain one or more nodes.

[8] C++ 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.

C.R. 16

```bash
ros2 run <package_name> <executable_name>
```

To run turtlesim, open a new terminal (Ctrl + Alt + T), and enter the following command:

C.R. 17

```bash
ros2 run turtlesim turtlesim_node
```

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:

C.R. 18

```bash
ros2 node list
```

```text
/turtlesim
```

Open another new terminal and start the teleop node with the command:

C.R. 19

```bash
ros2 run turtlesim turtle_teleop_key
```

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:

C.R. 20
```bash
ros2 node list
```

```text
/turtlesim
/teleop_turtle
```

**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:

C.R. 21
```bash
ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

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:

```text
/my_turtle
/turtlesim
/teleop_turtle
```

**ros2 node info** Now that we know the names of our nodes, we can access more information about them with:

C.R. 22
```bash
ros2 node info <node_name>
```

To examine our latest node, `my_turtle`, run the following command:

C.R. 23
```bash
ros2 node info /my_turtle
```

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

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

`ros2 node info` returns a list of subscribers, publishers, services, and actions.[9]

[9] i.e., the ROS graph connections that interact with that node.

Now try running the same command on the `/teleop_turtle` node, and see how its connections differ from `my_turtle`.

## 9.4 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:

C.R. 24
```bash
ros2 run turtlesim turtlesim_node
```

And of course we would like to control this turtle so we need to add the `teleop` node as well.

C.R. 25
```bash
ros2 run turtlesim turtle_teleop_key
```

**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:

C.R. 26
```bash
rqt_graph
```

Wou 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 data[10] to the `/turtle1/cmd_vel` topic, and the `/turtlesim` node is subscribed to that topic to receive the data.

[10] the keystrokes you enter to move the turtle around

> The highlighting feature of `rqt_graph` is very helpful when examining more complex systems with many nodes and topics connected in many different ways.

**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:

C.R. 27
```bash
ros2 topic list
```

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

Alternatively, running `ros2 topic list -t` will return the same list of topics, this time with the topic type appended in brackets:

C.R. 28
```bash
ros2 topic list -t
```

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

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:

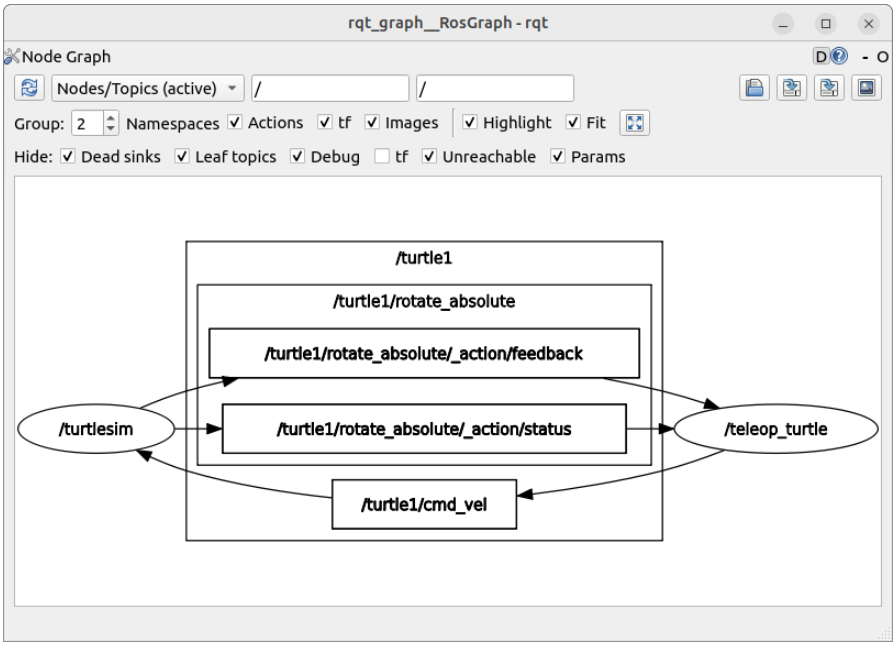



**Figure 9.3:** A visual representation of the communication happening between the nodes `/turtlesim` and `/teleop`.

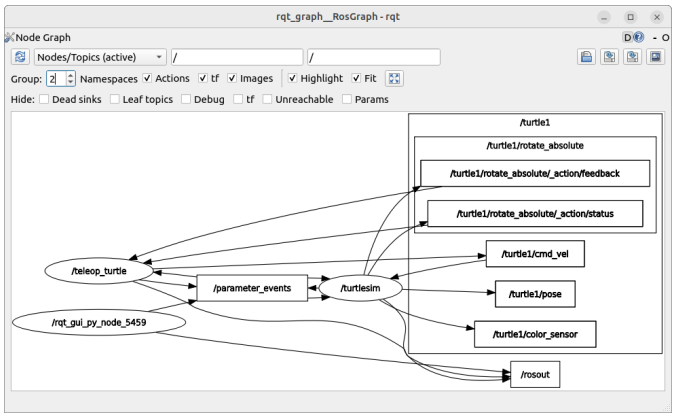



**Figure 9.4**

**ros2 topic echo** To see the data being published on a topic, use:

C.R. 29

```bash
ros2 topic echo <topic_name>
```

Since we know that `/teleop_turtle` publishes data to `/turtlesim` over the `/turtle1/cmd_vel` topic, let's use echo to introspect that topic:

C.R. 30

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

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:

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

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.

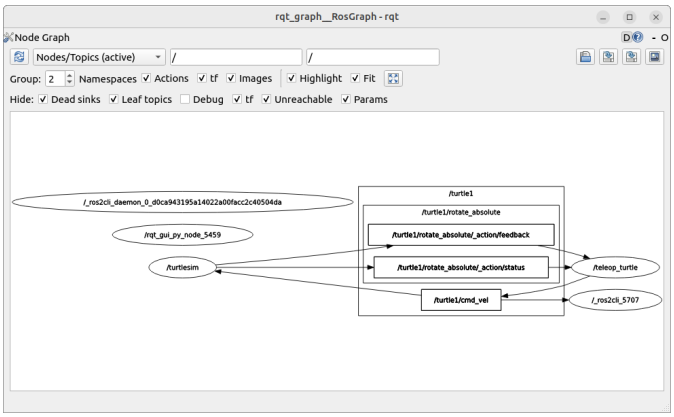



**Figure 9.5**

**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:

C.R. 31

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

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

C.R. 32

```bash
geometry_msgs/msg/Twist
```

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.

C.R. 33

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

Which will return

```text
# This expresses velocity in free space broken into its linear and angular parts.
    Vector3  linear
            float64 x
            float64 y
            float64 z
    Vector3  angular
            float64 x
            float64 y
            float64 z
```

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:

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

## 9.5 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:

C.R. 34
```bash
ros2 run turtlesim turtlesim_node
```

C.R. 35
```bash
ros2 run turtlesim turtle_teleop_key
```

**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:

C.R. 36
```bash
ros2 service list
```

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

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:

C.R. 37 bash
```
ros2 service type <service_name>
```

Let's take a look at turtlesim's `/clear` service. In a brand new terminal, let's enter the command:

C.R. 38 bash
```
ros2 service type /clear
```

text
```
std_srvs/srv/Empty
```

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:

C.R. 39 bash
```
ros2 service list -t
```

text
```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
```

```
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
...
```
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:

C.R. 40
```
ros2 service find <type_name>
```
bash

For example, we can find all the `Empty` typed services like this:

C.R. 41
```
ros2 service find std_srvs/srv/Empty
```
bash

```
/clear
/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:

C.R. 42
```
ros2 interface show <type_name>
```
bash

Try this on the `/clear` service's type, `Empty`:

C.R. 43
```
ros2 interface show std_srvs/srv/Empty
```
bash

```
---
```
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:

C.R. 44
```
ros2 interface show turtlesim/srv/Spawn
```
bash

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

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:

C.R. 45
```
ros2 service call <service_name> <service_type> <arguments>
```
bash

The <arguments> part is optional. For example, we know that Empty typed services don't have any arguments:

C.R. 46
```
ros2 service call /clear std_srvs/srv/Empty
```
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:

C.R. 47
```
ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
```
bash

```
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='')

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

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:

## 9.6 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:

C.R. 48
```bash
ros2 run turtlesim turtlesim_node
```

And of course we would like to control this turtle so we need to add the `teleop` node as well.

C.R. 49
```bash
ros2 run turtlesim turtle_teleop_key
```

**ros2 param list** Now that both of them are running at the same time, lets look at all the parameters in this system

C.R. 50
```bash
ros2 run turtlesim turtle_teleop_key
```

```text
/teleop_turtle:
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  qos_overrides./parameter_events.publisher.depth
  qos_overrides./parameter_events.publisher.durability
  qos_overrides./parameter_events.publisher.history
  qos_overrides./parameter_events.publisher.reliability
  use_sim_time
```

> 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

C.R. 51
```bash
ros2 param get <node_name> <parameter_name>
```

Now, let's use this command to determine they type and the value of the parameter:

```text
Integer value is: 86
```

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.

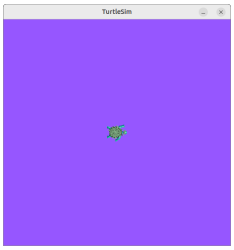
**ros2 param set** Time to change these parameters and play around. To change a parameter's value at runtime, use the command:

C.R. 52
```bash
ros2 param set <node_name> <parameter_name> <value>
```

Let's do something simple and try to change the `/turtlesim` node's background:

C.R. 53
```bash
ros2 param set /turtlesim background_r 150
```

```text
Set parameter successful
```




[11] While the original value was 86 by using `ros param set` we were able to change the value of the parameter and change the background to a nice purple.

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:

C.R. 54
```bash
ros2 param dump <node_name>
```

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:

C.R. 55
```bash
ros2 param dump /turtlesim > turtlesim.yaml
```

In the current working directory we will find a new file. If we were to open this file, we'll see the following content:

C.R. 56

```yaml
/turtlesim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
    qos_overrides:
      /parameter_events:
        publisher:
          depth: 1000
          durability: volatile
          history: keep_last
          reliability: reliable
    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:

C.R. 57

```bash
ros2 param load <node_name> <parameter_file>
```

To load the `turtlesim.yaml` file we just generated with `ros2 param dump` into `/turtlesim` node parameters, we just have to enter the command:

C.R. 58

```bash
ros2 param load /turtlesim turtlesim.yaml
```

```text
Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter qos_overrides./parameter_events.publisher.depth failed: parameter
↪ 'qos_overrides./parameter_events.publisher.depth' cannot be set because it is read-only
Set parameter qos_overrides./parameter_events.publisher.durability failed: parameter
↪ 'qos_overrides./parameter_events.publisher.durability' cannot be set because it is
↪ read-only
Set parameter qos_overrides./parameter_events.publisher.history failed: parameter
↪ 'qos_overrides./parameter_events.publisher.history' cannot be set because it is
↪ read-only
Set parameter qos_overrides./parameter_events.publisher.reliability failed: parameter
↪ 'qos_overrides./parameter_events.publisher.reliability' cannot be set because it is
↪ read-only
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:

C.R. 59
```bash
ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>
```

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:

C.R. 60
```bash
ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml
```

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.

## 9.7 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:

C.R. 61

```bash
ros2 run turtlesim turtlesim_node
```

And of course we would like to control this turtle so we need to add the `teleop` node as well.

C.R. 62

```bash
ros2 run turtlesim turtle_teleop_key
```

When we launch the `/teleop_turtle` node, we will see the following message in our terminal:

```text
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
```

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:

```text
[INFO] [turtlesim]: Rotation goal completed successfully
```

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:

```text
[INFO] [turtlesim]: Rotation goal canceled
```

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:

```text
[WARN] [turtlesim]: Rotation goal received before a previous goal finished.
                    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:

C.R. 63

```bash
ros2 node info /turtlesim
```

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

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

The 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:

C.R. 64

```bash
ros2 node info /teleop_turtle
```

```text
/teleop_turtle
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Service Servers:
    /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
    /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
    /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
    /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

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

**ros2 action list** To identify all the actions in the ROS graph, run the command:

C.R. 65

```bash
ros2 action list
```

```text
/turtle1/rotate_absolute
```

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:

C.R. 66

```bash
ros2 action list -t
```

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

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.

## 9.8 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 we 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 system[12] at once.

[12] which includes all nodes and their configurations

C.R. 67
```bash
ros2 launch turtlesim multisim.launch.py
```

This command will run the following launch file:

C.R. 68
```python
from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            namespace='turtlesim1', package='turtlesim',
            executable='turtlesim_node', output='screen'),
        launch_ros.actions.Node(
            namespace='turtlesim2', package='turtlesim',
            executable='turtlesim_node', output='screen'),
    ])
```

> 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:

C.R. 69
```bash
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}}"
```

In a third terminal:

C.R. 70

```bash
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}}"
```

After running these commands, we should see the two turtles in their respective envrionment spinning like a record.

# Chapter 10

# Client Libraries

## Table of Contents



## 10.1 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 [90].

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`.

We begin by installing colcon:[1]

[1] Of 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.

C.R. 1
```bash
sudo apt install python3-colcon-common-extensions
```

```text
Reading package lists... 0%
Reading package lists... 0%
Reading package lists... 7%
Reading package lists... Done
Building dependency tree... 0%
Building dependency tree... 0%
Building dependency tree... 50%
Building dependency tree... 50%
Building dependency tree... Done
Reading state information... 0%
Reading state information... 0%
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.
```

**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.

[2] This 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 directories as same hierarchy of the `src` directory:

**build** Where intermediate files are stored. For each package a subfolder will be created in which e.g., CMake[3] 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.

[3] a free, cross-platform, software development tool for building applications via compiler-independent instructions and allows automate testing, packaging and installation. It runs on a variety of platforms and supports many programming languages.

> For any student who worked with ROS 1 and `catkin`, there is no `devel` directory.

To get started we first create a directory (`ros2_ws`) to contain our workspace:[4]

[4] The name of the directory is up to the user.

C.R. 2
```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
```

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.

C.R. 3
```bash
ls
```

```text
src
```

Now let's populate our newly created environment with some tutorial files from the official ROS repo:

C.R. 4
```bash
git clone https://github.com/ros2/examples src/examples -b humble
```

```text
Cloning into 'src/examples'...
remote: Enumerating objects: 9987, done.
remote: Counting objects: 100% (2546/2546), done.
remote: Compressing objects: 100% (358/358), done.
remote: Total 9987 (delta 2376), reused 2195 (delta 2188), pack-reused 7441 (from 3)
Receiving objects: 100% (9987/9987), 1.56 MiB | 4.28 MiB/s, done.
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]

[5] i.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 space[6] for faster iteration.

[6] e.g., Python files or other non-compiled resources

C.R. 5
```bash
colcon build --symlink-install --executor sequential
```

> 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:

C.R. 6
```bash
ls -l
```

```text
drwxr-xr-x 24 ubuntu ubuntu 4096 Jun  6 13:58 build
drwxr-xr-x 24 ubuntu ubuntu 4096 Jun  6 13:58 install
drwxr-xr-x  4 ubuntu ubuntu 4096 Jun  6 13:57 log
drwxr-xr-x  3 ubuntu ubuntu 4096 Jun  6 13:54 src
```

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.

C.R. 7
```bash
source install/setup.bash
```

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:

C.R. 8
```bash
ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function
```

C.R. 9
```bash
ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

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.

## 10.2 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]

[7] Keep in mind that an underlay does **NOT** necessarily have to be the main ROS installation.

Depending on how we installed ROS, either from source or binaries, and which platform we're on, our exact source command will vary:

C.R. 10

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

**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":

C.R. 11

```bash
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

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:

C.R. 12
```bash
git clone https://github.com/ros/ros_tutorials.git -b humble
```

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`).

C.R. 13
```bash
cd ..
rosdep install -i --from-path src --rosdistro humble -y
```

If we already have all our dependencies, the console will return:

```text
#All required rosdeps installed successfully
```

[8] We 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:

C.R. 14
```bash
colcon build
```

```text
Starting >> turtlesim
Finished << turtlesim [5.49s]

Summary: 1 package finished [5.58s]
```

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:

C.R. 15
```bash
ls
```

```text
build  install  log  src
```

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:

C.R. 16
```bash
source /opt/ros/humble/setup.bash
```

Time to go into the root of our workspace:

C.R. 17
```bash
cd ~/ros2_ws
```

In the root, source our overlay:

C.R. 18
```bash
source install/local_setup.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:

C.R. 19
```bash
ros2 run turtlesim turtlesim_node
```

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 `setWindowTitle("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.

C.R. 20
```bash
ros2 run turtlesim turtlesim_node
```

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:

C.R. 21
```bash
ros2 run turtlesim turtlesim_node
```

You can see that modifications in the overlay did not actually affect anything in the underlay.

## 10.3 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:

## 10.4 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:

C.R. 22 bash
```
ros2 pkg create --build-type ament_python --license Apache-2.0 py_pubsub
```

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.

### 10.4.1 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:

C.R. 23 bash
```
wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/minimal_publisher/examples_rclpy_minimal_publisher/publisher_member_function.py
```

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:

C.R. 24 python
```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalPublisher(Node):

```

C.R. 25
python
```
    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1


def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

**Deconstructing the Code** The first lines of code after the comments import `rclpy` so its Node class can be used.[9]

[9] As a reminder, the `rclpy` is the ROS library written for Python.

C.R. 26
python
```
import rclpy
from rclpy.node import Node
```

The next statement imports the built-in string message type which the node uses to structure the data it passes on the topic.

C.R. 27
python
```
from std_msgs.msg import String
```

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 inherits[10] from Node.

[10] or is a subclass of

C.R. 28
```python
class MinimalPublisher(Node):
```

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`.

[11] which 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.

C.R. 29
```python
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5  # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.i = 0
```

`timer_callback` creates a message with the counter value appended, and publishes it to the console with `get_logger().info`.

C.R. 30
```python
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

Lastly, the main function is defined.

C.R. 31
```python
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

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]

[12] This could of course be emacs, or something which is not emacs so we can see why emacs is better.

As mentioned previously, make sure to fill in the `<description>`, `<maintainer>` and `<license>` tags:

C.R. 32

```xml
<description>Examples of minimal publisher/subscriber using rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to our node's import statements:[13]

[13] Remember, we need to let ROS know the dependencies required by the python script.

C.R. 33

```xml
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

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`:

C.R. 34

```python
maintainer='YourName',
maintainer_email='you@email.com',
description='Examples of minimal publisher/subscriber using rclpy',
license='Apache License 2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

C.R. 35

```python
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
        ],
},
```

**Checking the Configuration File** The contents of the `setup.cfg` file should be correctly populated automatically, like so:

C.R. 36

```cfg
[develop]
script_dir=$base/lib/py_pubsub
[install]
install_scripts=$base/lib/py_pubsub
```

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.

### 10.4.2 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:

C.R. 37 bash
```
wget https://raw.githubusercontent.com/ros2/examples/humble/rclpy/topics/minimal_subscriber/examples_rclpy_minimal_subscriber/subscriber_member_function.py
```

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.

C.R. 38 python
```
import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

```

C.R. 39
```python
    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

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.

C.R. 40
```python
self.subscription = self.create_subscription(
    String,
    'topic',
    self.listener_callback,
    10)
```

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

C.R. 41
```python
def listener_callback(self, msg):
    self.get_logger().info('I heard: "%s"' % msg.data)
```

The main definition is almost exactly the same, replacing the creation and spinning of the publisher with the subscriber.

C.R. 42
```python
minimal_subscriber = MinimalSubscriber()

rclpy.spin(minimal_subscriber)
```

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:

C.R. 43

```python
entry_points={
        'console_scripts': [
                'talker = py_pubsub.publisher_member_function:main',
                'listener = py_pubsub.subscriber_member_function:main',
        ],
},
```

### 10.4.3 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:

## 10.5 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:

C.R. 44

```bash
ros2 pkg create \
     --build-type ament_python \
     --license Apache-2.0 py_srvcli \
     --dependencies rclpy example_interfaces
```

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:

C.R. 45

```xml
int64 a
int64 b
---
int64 sum
```

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.

C.R. 46 xml
```
<description>Python client server tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

**Updating the Configuration** Add the same information to the setup.py file for the maintainer, maintainer_email, description and license fields:

C.R. 47 python
```
maintainer='Your Name',
maintainer_email='you@email.com',
description='Python client server tutorial',
license='Apache License 2.0',
```

### 10.5.1 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:

C.R. 48 python
```
from example_interfaces.srv import AddTwoInts

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddTwoInts, 'add_two_ints',
        ↪  self.add_two_ints_callback)

    def add_two_ints_callback(self, request, response):
        response.sum = request.a + request.b
        self.get_logger().info('Incoming request\na: %d b: %d' % (request.a, request.b))

        return response


def main():
    rclpy.init()

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()

```

C.R. 49
python
```
29
30 if __name__ == '__main__':
31     main()
```

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.

C.R. 50
python
```
1 from example_interfaces.srv import AddTwoInts
2
3 import rclpy
4 from rclpy.node import Node
```

The `MinimalService` class constructor initializes the node with the name `minimal_service`. Then, it creates a service and defines the type, name, and callback.

C.R. 51
python
```
1     def __init__(self):
2         super().__init__('minimal_service')
3         self.srv = self.create_service(AddTwoInts, 'add_two_ints',
          ↪  self.add_two_ints_callback)
```

The definition of the service callback receives the request data, sums it, and returns the sum as a response.

C.R. 52
python
```
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
```

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:

[14] This is located in the `ros2_ws` ▸ `src` ▸ `py_srvcli` directory.

C.R. 53
python
```
1 'service = py_srvcli.service_member_function:main',
```

### 10.5.2 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:

C.R. 54
python
```
import sys

from example_interfaces.srv import AddTwoInts
import rclpy
from rclpy.node import Node


class MinimalClientAsync(Node):

    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddTwoInts, 'add_two_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddTwoInts.Request()

    def send_request(self, a, b):
        self.req.a = a
        self.req.b = b
        return self.cli.call_async(self.req)


def main():
    rclpy.init()

    minimal_client = MinimalClientAsync()
    future = minimal_client.send_request(int(sys.argv[1]), int(sys.argv[2]))
    rclpy.spin_until_future_complete(minimal_client, future)
    response = future.result()
    minimal_client.get_logger().info(
        'Result of add_two_ints: for %d + %d = %d' %
        (int(sys.argv[1]), int(sys.argv[2]), response.sum))

    minimal_client.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

**Examining the Code** As with the service code, we first `import` the necessary libraries.

C.R. 55
python
```
import sys

from example_interfaces.srv import AddTwoInts
```

C.R. 56
```python
import rclpy
from rclpy.node import Node
```

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.

C.R. 57
```python
def __init__(self):
    super().__init__('minimal_client_async')
    self.cli = self.create_client(AddTwoInts, 'add_two_ints')
    while not self.cli.wait_for_service(timeout_sec=1.0):
        self.get_logger().info('service not available, waiting again...')
    self.req = AddTwoInts.Request()
```

Below the constructor is the `send_request` method, which will send the request and spin until it receives the response or fails.

C.R. 58
```python
def send_request(self, a, b):
    self.req.a = a
    self.req.b = b
    return self.cli.call_async(self.req)
```

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:

C.R. 59
```python
entry_points={
    'console_scripts': [
        'service = py_srvcli.service_member_function:main',
        'client = py_srvcli.client_member_function:main',
    ],
},
```

> It's good practice to run `rosdep` in the root of our workspace (`ros2_ws`) to check for missing dependencies before building:

C.R. 60
```python
rosdep install -i --from-path src --rosdistro humble -y
```

Navigate back to the root of our workspace, 📁 ros2_ws, and build our new package:

C.R. 61
```python
colcon build --packages-select py_srvcli
```

Open a new terminal, navigate to 📁 ros2_ws, and source the setup files:

C.R. 62
```python
source install/setup.bash
```

Now run the service node:

C.R. 63
```python
ros2 run py_srvcli service
```

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:

C.R. 64
```python
ros2 run py_srvcli client 2 3
```

```text
[INFO] [minimal_client_async]: Result of add_two_ints: for 2 + 3 = 5
```

Return to the terminal where our service node is running. We will see that it published log messages when it received the request:

```text
[INFO] [minimal_service]: Incoming request
a: 2 b: 3
```

## 10.6 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:

C.R. 65

```bash
ros2 pkg create --build-type ament_cmake --license Apache-2.0 tutorial_interfaces
```

`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.

[15] It 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:

C.R. 66

```bash
mkdir msg srv
```

### 10.6.1 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:

C.R. 67
```text
int64 num
```

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:

C.R. 68
```bash
geometry_msgs/Point center
float64 radius
```

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 moment ago and make a new file called `AddThreeInts.srv` with the following request and response structure:

C.R. 69
```bash
int64 a
int64 b
int64 c
---
int64 sum
```

This is our custom service which requests three (3) integers aptly named `a`, `b`, and `c`, and responds with an integer called `sum`.

[16] This can be either C++ or Python.

**CMakeLists** To convert the interfaces we defined into language-specific code[16] so that they can be used in those languages, let's add the following lines to our `CMakeLists.txt`:

C.R. 70
```cmake
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs # Add packages that above messages
  #depend on, in this case geometry_msgs for Sphere.msg
)
```

> 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`:

C.R. 71

```xml
<depend>geometry_msgs</depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

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:

C.R. 72

```bash
colcon build --packages-select tutorial_interfaces
```

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:

C.R. 73

```bash
source install/setup.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:

C.R. 74

```bash
ros2 interface show tutorial_interfaces/msg/Num
```

```text
int64 num
```

C.R. 75

```bash
ros2 interface show tutorial_interfaces/msg/Sphere
```

```
geometry_msgs/Point center
        float64 x
        float64 y
        float64 z
float64 radius
```
text

C.R. 76

```
ros2 interface show tutorial_interfaces/srv/AddThreeInts
```
bash

```
int64 a
int64 b
int64 c
---
int64 sum
```
text

### 10.6.2 Testing the Newly Built Interfaces

Time to see our new interface in actiony. 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

C.R. 77

```python
import rclpy
from rclpy.node import Node

from tutorial_interfaces.msg import Num                          # CHANGE


class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(Num, 'topic', 10)  # CHANGE
        timer_period = 0.5
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = Num()                                              # CHANGE
        msg.num = self.i                                         # CHANGE
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%d"' % msg.num)       # CHANGE
        self.i += 1


def main(args=None):
```
python

C.R. 78

```python
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    minimal_publisher.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

### Publisher and Subscriber System: Subscriber Code

C.R. 79

```python
import rclpy
from rclpy.node import Node

from tutorial_interfaces.msg import Num                         # CHANGE


class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            Num,                                                # CHANGE
            'topic',
            self.listener_callback,
            10)
        self.subscription

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%d"' % msg.num)  # CHANGE


def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    minimal_subscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

We also need to edit our 🗁 package.xml file to make all the previous python code to work.

C.R. 80
```xml
<exec_depend>tutorial_interfaces</exec_depend>
```

Once we have done the necessary changes let's build our package and execute it.

C.R. 81
```bash
colcon build --packages-select py_pubsub
```

Then open two new terminals, source 🗁 ros2_ws in each, and run:

C.R. 82
```bash
ros2 run py_pubsub talker
```

C.R. 83
```bash
ros2 run py_pubsub talker
```

Since 🗁 Num.msg relays only an integer, the talker should only be publishing integer values, as opposed to the string it published previously:

```text
[INFO] [minimal_publisher]: Publishing: '0'
[INFO] [minimal_publisher]: Publishing: '1'
[INFO] [minimal_publisher]: Publishing: '2'
```

### 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

C.R. 84
```python
from tutorial_interfaces.srv import AddThreeInts  # CHANGE

import rclpy
from rclpy.node import Node


class MinimalService(Node):

    def __init__(self):
        super().__init__('minimal_service')
        self.srv = self.create_service(AddThreeInts,
                                       'add_three_ints',
                                       self.add_three_ints_callback)  # CHANGE

```

C.R. 85

```python
    def add_three_ints_callback(self, request, response):
        response.sum = request.a + request.b + request.c  # CHANGE
        self.get_logger()\
            .info('Incoming request\na: %d b: %d c: %d' % (request.a, request.b,
            ↪  request.c))  # CHANGE

        return response

def main(args=None):
    rclpy.init(args=args)

    minimal_service = MinimalService()

    rclpy.spin(minimal_service)

    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

## Client

C.R. 86

```python
from tutorial_interfaces.srv import AddThreeInts                    # CHANGE
import sys
import rclpy
from rclpy.node import Node


class MinimalClientAsync(Node):

    def __init__(self):
        super().__init__('minimal_client_async')
        self.cli = self.create_client(AddThreeInts, 'add_three_ints')       # CHANGE
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddThreeInts.Request()                                   # CHANGE

    def send_request(self):
        self.req.a = int(sys.argv[1])
        self.req.b = int(sys.argv[2])
        self.req.c = int(sys.argv[3])                                       # CHANGE
        self.future = self.cli.call_async(self.req)


def main(args=None):
    rclpy.init(args=args)

    minimal_client = MinimalClientAsync()
    minimal_client.send_request()

```

C.R. 87 python
```python
    while rclpy.ok():
        rclpy.spin_once(minimal_client)
        if minimal_client.future.done():
            try:
                response = minimal_client.future.result()
            except Exception as e:
                minimal_client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                minimal_client.get_logger().info(
                    'Result of add_three_ints: for %d + %d + %d = %d' %
                    # CHANGE
                    (minimal_client.req.a, minimal_client.req.b, minimal_client.req.c,
                    response.sum))  # CHANGE
            break

    minimal_client.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

To add these features to our package we need to let `package.xml` know, which for that we need to write:

C.R. 88 xml
```xml
<exec_depend>tutorial_interfaces</exec_depend>
```

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:

C.R. 89 bash
```
ros2 run py_srvcli service
```

C.R. 90 bash
```
ros2 run py_srvcli client 2 3 1
```

## 10.7 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:

C.R. 91
bash
```
ros2 pkg create --build-type ament_python \
     --license Apache-2.0 python_parameters \
     --dependencies rclpy
```

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.

C.R. 92
xml
```
<description>Python parameter tutorial</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

**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:

C.R. 93
python
```
import rclpy
import rclpy.node

```

C.R. 94 python

```python
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)

    def timer_callback(self):
        my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

        self.get_logger().info('Hello %s!' % my_param)

        my_new_param = rclpy.parameter.Parameter(
            'my_parameter',
            rclpy.Parameter.Type.STRING,
            'world'
        )
        all_new_parameters = [my_new_param]
        self.set_parameters(all_new_parameters)

def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

if __name__ == '__main__':
    main()
```

**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.

C.R. 95 python

```python
class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        self.declare_parameter('my_parameter', 'world')

        self.timer = self.create_timer(1, self.timer_callback)
```

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
def timer_callback(self):
    my_param = self.get_parameter('my_parameter').get_parameter_value().string_value

    self.get_logger().info('Hello %s!' % my_param)

    my_new_param = rclpy.parameter.Parameter(
        'my_parameter',
        rclpy.Parameter.Type.STRING,
        'world'
    )
    all_new_parameters = [my_new_param]
    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
def main():
    rclpy.init()
    node = MinimalParam()
    rclpy.spin(node)

if __name__ == '__main__':
    main()
```

**(Optional) Adding a Parameter Descriptor** Optionally, we can set a descriptor for the parameter. Descriptors allow we 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
# ...

class MinimalParam(rclpy.node.Node):
    def __init__(self):
        super().__init__('minimal_param_node')

        from rcl_interfaces.msg import ParameterDescriptor
        my_parameter_descriptor = ParameterDescriptor(description='This parameter is
        ↪  mine!')

        self.declare_parameter('my_parameter', 'world', my_parameter_descriptor)

        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:

C.R. 99
```python
maintainer='YourName',
maintainer_email='you@email.com',
description='Python parameter tutorial',
license='Apache License 2.0',
```

Add the following line within the `console_scripts` brackets of the `entry_points` field:

C.R. 100
```python
entry_points={
    'console_scripts': [
        'minimal_param_node = python_parameters.python_parameters_node:main',
    ],
},
```

**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:

C.R. 101
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Navigate back to the root of our workspace, `ros2_ws`, and build our new package:

C.R. 102
```bash
colcon build --packages-select python_parameters
```

Open a new terminal, navigate to `ros2_ws`, and source the setup files:

C.R. 103
```bash
source install/setup.bash
```

Now run the node. The terminal should return Hello world! every second:

C.R. 104
```bash
ros2 run python_parameters minimal_param_node
```

```text
[INFO] [parameter_node]: Hello world!
```

Now we can see the default value of our parameter, but we want to be able to set it ourself. There are two ways to accomplish this.

**Changing Output using the Console** Make sure the node is running:

C.R. 105
```bash
ros2 run python_parameters minimal_param_node
```

Open another terminal, source the setup files from inside `ros2_ws` again, and enter the following line:

C.R. 106
```bash
ros2 param list
```

There we will see the custom parameter `my_parameter`. To change it, simply run the following line in the console:

C.R. 107
```bash
ros2 param set /minimal_param_node my_parameter earth
```

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`

C.R. 108
```python
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='python_parameters',
            executable='minimal_param_node',
            name='custom_minimal_param_node',
            output='screen',
            emulate_tty=True,
            parameters=[
                {'my_parameter': 'earth'}
            ]
        )
    ])
```

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.

C.R. 109
```bash
output="screen",
emulate_tty=True,
```

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:

C.R. 110

```bash
import os
from glob import glob
# ...

setup(
  # ...
  data_files=[
      # ...
      (os.path.join('share', package_name, 'launch'), glob('launch/*')),
    ]
  )
```

Open a console and navigate to the root of our workspace, `ros2_ws`, and build our new package:

C.R. 111

```bash
colcon build --packages-select python_parameters
```

Then source the setup files in a new terminal:

C.R. 112

```bash
source install/setup.bash
```

Now run the node using the launch file we have just created. The terminal should return the following message the first time:

C.R. 113

```bash
ros2 launch python_parameters python_parameters_launch.py
```

```text
[INFO] [custom_minimal_param_node]: Hello earth!
```

Further outputs should show `[INFO] [minimal_param_node]: Hello world!` every second.

## 10.8 Managing Dependencies

### 10.8.1 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.

### 10.8.2 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>`.

## 10.9 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:

C.R. 114
bash
```
mkdir -p ros2_ws/src # you can reuse an existing workspace
cd ros2_ws/src
ros2 pkg create action_tutorials_interfaces
```

**Defining an Action** Actions are defined in `.action` files of the form:

C.R. 115
text
```
# Request
---
# Result
---
# Feedback
```

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`:

C.R. 116
bash
```
cd action_tutorials_interfaces
mkdir action
```

Within the action directory, create a file called Fibonacci.action with the following contents:

C.R. 117
text
```
int32 order
---
int32[] sequence
---
```

C.R. 118
```text
int32[] partial_sequence
```

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`:

C.R. 119
```cmake
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Fibonacci.action"
)
```

We should also add the required dependencies to our package.xml:

C.R. 120
```xml
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

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:

C.R. 121
```bash
cd ~/ros2_ws # Change to the root of the workspace
colcon build # Build
```

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:

C.R. 122
```bash
. install/setup.bash  # Source our workspace.
# Check that our action definition exists
ros2 interface show action_tutorials_interfaces/action/Fibonacci
```

We should see the Fibonacci action definition printed to the screen.

## 10.10 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:

C.R. 123

python

```python
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node

from action_tutorials_interfaces.action import Fibonacci


class FibonacciActionServer(Node):

    def __init__(self):
        super().__init__('fibonacci_action_server')
        self._action_server = ActionServer(
            self,
            Fibonacci,
            'fibonacci',
            self.execute_callback)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        result = Fibonacci.Result()
        return result


def main(args=None):
    rclpy.init(args=args)

    fibonacci_action_server = FibonacciActionServer()

    rclpy.spin(fibonacci_action_server)


if __name__ == '__main__':
    main()
```

## 10.11 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:

C.R. 124

```bash
mkdir launch
```

**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:

C.R. 125

```python
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
```

C.R. 126

```python
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

**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.

C.R. 127 python
```python
from launch import LaunchDescription
from launch_ros.actions import Node
```

Next, the launch description itself begins:

C.R. 128 python
```python
def generate_launch_description():
    return LaunchDescription([
        Node(
    ])
```

The first two actions in the launch description launch the two turtlesim windows:

C.R. 129 python
```python
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
```

The final action launches the mimic node with the remaps:

C.R. 130 python
```python
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
```

**Launching the Package** To run the launch file created above, enter into the directory we created earlier and run the following command:

C.R. 131 bash
```bash
cd launch
ros2 launch turtlesim_mimic_launch.py
```

Two turtlesim windows will open, and we will see the following [INFO] messages telling we which nodes our launch file has started:

C.R. 132
```text
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [turtlesim_node-1]: process started with pid [11714]
[INFO] [turtlesim_node-2]: process started with pid [11715]
[INFO] [mimic-3]: process started with pid [11716]
```

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:

C.R. 133
```bash
ros2 topic pub -r 1 \
     /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}}"
```

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:

C.R. 134
```bash
rqt_graph
```

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 11

# Transform Library

## Table of Contents



## 11.1 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?

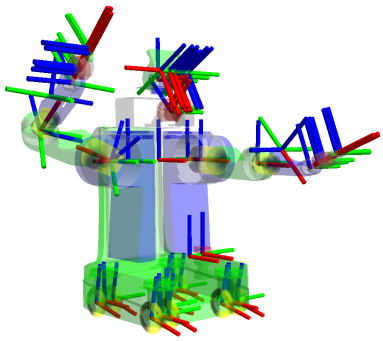


**Figure 11.1:** A robot is comprised of numerous coordinate system as can be seen from this robot and needs to be constantly be kept in check which `tf2` allows [91].

- 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:

C.R. 1
```bash
sudo apt-get install \
     ros-humble-rviz2 \
     ros-humble-turtle-tf2-py \
     ros-humble-tf2-ros \
     ros-humble-tf2-tools \
     ros-humble-turtlesim
```

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] While the docker container setup has the source predefined, it is nevertheless a good practice to do this if you are working with multiple versions.

C.R. 2
```bash
ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
```

You will see the turtlesim start with two (2) turtles. In the second terminal window type the following command:

C.R. 3
```bash
ros2 run turtlesim turtle_teleop_key
```

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.

**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.

C.R. 4

```bash
ros2 run tf2_tools view_frames
```

```text
Listening to tf data during 5 seconds...
Generating graph in frames.pdf file...
```

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:

C.R. 5
```bash
ros2 run tf2_ros tf2_echo [source_frame] [target_frame]
```

Let's look at the transform of the `turtle2` frame with respect to `turtle1` frame which is equivalent to:

C.R. 6
```bash
ros2 run tf2_ros tf2_echo turtle2 turtle1
```

```text
At time 1683385337.850619099
- Translation: [2.157, 0.901, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.172, 0.985]
- Rotation: in RPY (radian) [0.000, -0.000, 0.345]
- Rotation: in RPY (degree) [0.000, -0.000, 19.760]
- Matrix:
  0.941 -0.338  0.000  2.157
  0.338  0.941  0.000  0.901
  0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
At time 1683385338.841997774
- Translation: [1.256, 0.216, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.016, 1.000]
- Rotation: in RPY (radian) [0.000, 0.000, -0.032]
- Rotation: in RPY (degree) [0.000, 0.000, -1.839]
- Matrix:
  0.999  0.032  0.000  1.256
 -0.032  0.999 -0.000  0.216
 -0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
```

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:

C.R. 7
```bash
ros2 run rviz2 rviz2 -d \
     $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz
```

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`.

## 11.2 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:

C.R. 8

```bash
ros2 pkg create --build-type ament_python \
    --license Apache-2.0 \
    -- learning_tf2_py
```

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:

C.R. 9

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/
↪ turtle_tf2_py/static_turtle_tf2_broadcaster.py
```

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.

C.R. 10 python
```
from geometry_msgs.msg import TransformStamped
```

Afterwards, `rclpy` is imported so its Node class can be used.

C.R. 11 python
```
import rclpy
from rclpy.node import Node
```

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.

C.R. 12 python
```
from tf2_ros.static_transform_broadcaster import StaticTransformBroadcaster
```

The `StaticFramePublisher` class constructor initializes the node with the name `static_turtle_tf2_broadcaste` Then, `StaticTransformBroadcaster` is created, which will send one static transformation upon the startup.

C.R. 13 python
```
self.tf_static_broadcaster = StaticTransformBroadcaster(self)
self.make_transforms(transformation)
```

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

C.R. 14 python
```
t = TransformStamped()

t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'world'
t.child_frame_id = transformation[1]
```

Here we populate the 6D pose (translation and rotation) of the turtle.

C.R. 15
```python
t.transform.translation.x = float(transformation[2])
t.transform.translation.y = float(transformation[3])
t.transform.translation.z = float(transformation[4])
quat = quaternion_from_euler(
    float(transformation[5]), float(transformation[6]), float(transformation[7]))
t.transform.rotation.x = quat[0]
t.transform.rotation.y = quat[1]
t.transform.rotation.z = quat[2]
t.transform.rotation.w = quat[3]
```

Finally, we broadcast static transform using the `sendTransform()` function.

C.R. 16
```python
self.tf_static_broadcaster.sendTransform(t)
```

**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:

C.R. 17
```xml
<description>Learning tf2 with rclpy</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

After the lines above, add the following dependencies corresponding to your nodes import statements:

C.R. 18
```xml
<exec_depend>geometry_msgs</exec_depend>
<exec_depend>python3-numpy</exec_depend>
<exec_depend>rclpy</exec_depend>
<exec_depend>tf2_ros_py</exec_depend>
<exec_depend>turtlesim</exec_depend>
```

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:

C.R. 19
```python
'static_turtle_tf2_broadcaster = learning_tf2_py.static_turtle_tf2_broadcaster:main',
```

**Build** Its good practice to run rosdep in the root of your workspace to check for missing dependencies before building:

C.R. 20
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your new package:

C.R. 21
```bash
colcon build --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

C.R. 22
```bash
. install/setup.bash
```

**Run** Now run the `static_turtle_tf2_broadcaster` node:

C.R. 23
```bash
ros2 run \
     learning_tf2_py static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0
```

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:

C.R. 24
```bash
ros2 topic echo /tf_static
```

```text
transforms:
- header:
   stamp:
      sec: 1622908754
      nanosec: 208515730
   frame_id: world
child_frame_id: mystaticturtle
transform:
   translation:
      x: 0.0
      y: 0.0
      z: 1.0
   rotation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
```

### 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.

C.R. 25

```bash
ros2 run tf2_ros \
     static_transform_publisher \
     --x 0 --y 0 --z 1 \
     --yaw 0 --pitch 0 --roll 0 \
     --frame-id world --child-frame-id mystaticturtle
```

The following command publishes the same static coordinate transform to tf2, but using quaternion representation for the rotation.

C.R. 26

```bash
ros2 run tf2_ros \
     static_transform_publisher \
     --x 0 --y 0 --z 1 \
     --qx 0 --qy 0 --qz 0 --qw 1 \
     --frame-id world --child-frame-id mystaticturtle
```

`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:

C.R. 27

```python
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            arguments=[
                '--x', '0', '--y', '0', '--z', '1',
                '--yaw', '0', '--pitch', '0', '--roll',
                '0', '--frame-id', 'world', '--child-frame-id', 'mystaticturtle']
        ),
    ])
```

Note that all arguments except for –frame-id and –child-frame-id are optional; if a particular option isnt specified, then the identity will be assumed.

## 11.3 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:

C.R. 28

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/turtle_tf2_py/turtle_tf2_listener.py
```

Now open the file called `turtle_tf2_listener.py` using your preferred text editor.

C.R. 29

```python
import math

from geometry_msgs.msg import Twist

import rclpy
from rclpy.node import Node

from tf2_ros import TransformException
from tf2_ros.buffer import Buffer
from tf2_ros.transform_listener import TransformListener

from turtlesim.srv import Spawn


class FrameListener(Node):

    def __init__(self):
        super().__init__('turtle_tf2_frame_listener')

        # Declare and acquire `target_frame` parameter
        self.target_frame = self.declare_parameter(
          'target_frame', 'turtle1').get_parameter_value().string_value

        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer, self)

        # Create a client to spawn a turtle
        self.spawner = self.create_client(Spawn, 'spawn')
        # Boolean values to store the information
        # if the service for spawning turtle is available
        self.turtle_spawning_service_ready = False
        # if the turtle was successfully spawned
        self.turtle_spawned = False

        # Create turtle2 velocity publisher
        self.publisher = self.create_publisher(Twist, 'turtle2/cmd_vel', 1)

        # Call on_timer function every second
        self.timer = self.create_timer(1.0, self.on_timer)
```

C.R. 30

python

```python
    def on_timer(self):
        # Store frame names in variables that will be used to
        # compute transformations
        from_frame_rel = self.target_frame
        to_frame_rel = 'turtle2'

        if self.turtle_spawning_service_ready:
            if self.turtle_spawned:
                # Look up for the transformation between target_frame and turtle2 frames
                # and send velocity commands for turtle2 to reach target_frame
                try:
                    t = self.tf_buffer.lookup_transform(
                        to_frame_rel,
                        from_frame_rel,
                        rclpy.time.Time())
                except TransformException as ex:
                    self.get_logger().info(
                        f'Could not transform {to_frame_rel} to {from_frame_rel}: {ex}')
                    return

                msg = Twist()
                scale_rotation_rate = 1.0
                msg.angular.z = scale_rotation_rate * math.atan2(
                    t.transform.translation.y,
                    t.transform.translation.x)

                scale_forward_speed = 0.5
                msg.linear.x = scale_forward_speed * math.sqrt(
                    t.transform.translation.x ** 2 +
                    t.transform.translation.y ** 2)

                self.publisher.publish(msg)
            else:
                if self.result.done():
                    self.get_logger().info(
                        f'Successfully spawned {self.result.result().name}')
                    self.turtle_spawned = True
                else:
                    self.get_logger().info('Spawn is not finished')
        else:
            if self.spawner.service_is_ready():
                # Initialize request with turtle name and coordinates
                # Note that x, y and theta are defined as floats in turtlesim/srv/Spawn
                request = Spawn.Request()
                request.name = 'turtle2'
                request.x = float(4)
                request.y = float(2)
                request.theta = float(0)
                # Call request
                self.result = self.spawner.call_async(request)
                self.turtle_spawning_service_ready = True
            else:
```

C.R. 31
python
```
                # Check if the service is ready
                self.get_logger().info('Service is not ready')


def main():
    rclpy.init()
    node = FrameListener()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    rclpy.shutdown()
```

**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.

C.R. 32
python
```
from tf2_ros.transform_listener import TransformListener
```

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.

C.R. 33
python
```
self.tf_listener = TransformListener(self.tf_buffer, self)
```

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:

C.R. 34
python
```
'turtle_tf2_listener = learning_tf2_py.turtle_tf2_listener:main',
```

**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

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='learning_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='learning_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle2'}
            ]
        ),
        Node(
            package='learning_tf2_py',
            executable='turtle_tf2_listener',
            name='listener',
            parameters=[
                {'target_frame': LaunchConfiguration('target_frame')}
            ]
        ),
    ])
```

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.

C.R. 36
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your package:

C.R. 37
```bash
colcon build --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

C.R. 38
```bash
. install/setup.bash
```

**Run** Now youre ready to start your full turtle demo:

C.R. 39
```bash
ros2 launch learning_tf2_py turtle_tf2_demo.launch.py
```

You should see the turtle sim with two turtles. In the second terminal window type the following command:

C.R. 40
```bash
ros2 run turtlesim turtle_teleop_key
```

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 youll see the second turtle following the first one!

## 11.4 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:

C.R. 41

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/turtle_tf2_py/fixed_frame_tf2_broadcaster.py
```

Now open the file called `fixed_frame_tf2_broadcaster.py`.

C.R. 42

```python
from geometry_msgs.msg import TransformStamped

import rclpy
from rclpy.node import Node

from tf2_ros import TransformBroadcaster


class FixedFrameBroadcaster(Node):

   def __init__(self):
       super().__init__('fixed_frame_tf2_broadcaster')
       self.tf_broadcaster = TransformBroadcaster(self)
```

C.R. 43 python
```
        self.timer = self.create_timer(0.1, self.broadcast_timer_callback)

    def broadcast_timer_callback(self):
        t = TransformStamped()

        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'turtle1'
        t.child_frame_id = 'carrot1'
        t.transform.translation.x = 0.0
        t.transform.translation.y = 2.0
        t.transform.translation.z = 0.0
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self.tf_broadcaster.sendTransform(t)


def main():
    rclpy.init()
    node = FixedFrameBroadcaster()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

    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.

C.R. 44 python
```
t = TransformStamped()

t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'turtle1'
t.child_frame_id = 'carrot1'
t.transform.translation.x = 0.0
t.transform.translation.y = 2.0
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:

C.R. 45
```python
'fixed_frame_tf2_broadcaster = learning_tf2_py.fixed_frame_tf2_broadcaster:main',
```

**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:

C.R. 46
```python
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.substitutions import PathJoinSubstitution
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    return LaunchDescription([
        IncludeLaunchDescription(
            PathJoinSubstitution([
                FindPackageShare('learning_tf2_py'), 'launch',
                ↪  'turtle_tf2_demo.launch.py'])
        ),
        Node(
            package='learning_tf2_py',
            executable='fixed_frame_tf2_broadcaster',
            name='fixed_broadcaster',
        ),
    ])
```

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.

C.R. 47
```python
        Node(
            package='learning_tf2_py',
            executable='fixed_frame_tf2_broadcaster',
            name='fixed_broadcaster',
        ),
```

**Build** Run rosdep in the root of your workspace to check for missing dependencies.

C.R. 48
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your package:

C.R. 49
```bash
colcon build --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

C.R. 50
```bash
. install/setup.bash
```

**Run** Now you can start the turtle broadcaster demo:

C.R. 51
```bash
ros2 launch learning_tf2_py turtle_tf2_fixed_frame_demo.launch.py
```

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:

C.R. 52
```bash
ros2 launch learning_tf2_py \
     turtle_tf2_fixed_frame_demo.launch.py \
     target_frame:=carrot1
```

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.

C.R. 53
```python
def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        ...,
        launch_arguments={'target_frame': 'carrot1'}.items(),
        )
```

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:

C.R. 54

```bash
wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/turtle_tf2_py/dynamic_frame_tf2_broadcaster.py
```

Now open the file called dynamic_frame_tf2_broadcaster.py:

C.R. 55

```python
import math

from geometry_msgs.msg import TransformStamped

import rclpy
from rclpy.node import Node

from tf2_ros import TransformBroadcaster


class DynamicFrameBroadcaster(Node):

    def __init__(self):
        super().__init__('dynamic_frame_tf2_broadcaster')
        self.tf_broadcaster = TransformBroadcaster(self)
        self.timer = self.create_timer(0.1, self.broadcast_timer_callback)

    def broadcast_timer_callback(self):
        seconds, _ = self.get_clock().now().seconds_nanoseconds()
        x = seconds * math.pi

        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'turtle1'
        t.child_frame_id = 'carrot1'
        t.transform.translation.x = 10 * math.sin(x)
        t.transform.translation.y = 10 * math.cos(x)
        t.transform.translation.z = 0.0
        t.transform.rotation.x = 0.0
        t.transform.rotation.y = 0.0
        t.transform.rotation.z = 0.0
        t.transform.rotation.w = 1.0

        self.tf_broadcaster.sendTransform(t)


def main():
    rclpy.init()
    node = DynamicFrameBroadcaster()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass

```

C.R. 56
```python
    rclpy.shutdown()
```

**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.

C.R. 57
```bash
seconds, _ = self.get_clock().now().seconds_nanoseconds()
x = seconds * math.pi
...
t.transform.translation.x = 10 * math.sin(x)
t.transform.translation.y = 10 * math.cos(x)
```

**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:

C.R. 58
```python
'dynamic_frame_tf2_broadcaster = learning_tf2_py.dynamic_frame_tf2_broadcaster:main',
```

**Writing the Launch File** To test this code, create a new launch file `turtle_tf2_dynamic_frame_demo.launch.p` in the `src▸learning_tf2_py▸launch` directory and paste the following code:

C.R. 59
```python
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.substitutions import PathJoinSubstitution
from launch_ros.actions import Node
from launch_ros.substitutions import FindPackageShare


def generate_launch_description():
    return LaunchDescription([
        IncludeLaunchDescription(
            PathJoinSubstitution([
                FindPackageShare('learning_tf2_py'), 'launch',
                ↪  'turtle_tf2_demo.launch.py']),
            launch_arguments={'target_frame': 'carrot1'}.items(),
        ),
        Node(
            package='learning_tf2_py',
            executable='dynamic_frame_tf2_broadcaster',
            name='dynamic_broadcaster',
        ),
    ])
```

**Buld** Run rosdep in the root of your workspace to check for missing dependencies.

C.R. 60
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your package:

C.R. 61
```bash
colcon build --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

C.R. 62
```bash
. install/setup.bash
```

**Run** Now you can start the dynamic frame demo:

C.R. 63
```bash
ros2 launch learning_tf2_py turtle_tf2_dynamic_frame_demo.launch.py
```

You should see that the second turtle is following the carrots position that is constantly changing.

## 11.5 Writing a Broadcaster

**Writing the Broadaster 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:

C.R. 64 bash
```
wget https://raw.githubusercontent.com/ros/geometry_tutorials/humble/turtle_tf2_py/turtle_tf2_py/turtle_tf2_broadcaster.py
```

Now open the file called `turtle_tf2_broadcaster.py` using your preferred text editor.

C.R. 65 python
```
import math

from geometry_msgs.msg import TransformStamped

import numpy as np

import rclpy
from rclpy.node import Node

from tf2_ros import TransformBroadcaster

from turtlesim.msg import Pose


def quaternion_from_euler(ai, aj, ak):
    ai /= 2.0
    aj /= 2.0
    ak /= 2.0
    ci = math.cos(ai)
    si = math.sin(ai)
    cj = math.cos(aj)
    sj = math.sin(aj)
    ck = math.cos(ak)
    sk = math.sin(ak)
    cc = ci*ck
    cs = ci*sk
    sc = si*ck
    ss = si*sk

    q = np.empty((4, ))
    q[0] = cj*sc - sj*cs
    q[1] = cj*ss + sj*cc
    q[2] = cj*cs - sj*sc
    q[3] = cj*cc + sj*ss

    return q


class FramePublisher(Node):
```

C.R. 66

python

```
    def __init__(self):
        super().__init__('turtle_tf2_frame_publisher')

        # Declare and acquire `turtlename` parameter
        self.turtlename = self.declare_parameter(
          'turtlename', 'turtle').get_parameter_value().string_value

        # Initialize the transform broadcaster
        self.tf_broadcaster = TransformBroadcaster(self)

        # Subscribe to a turtle{1}{2}/pose topic and call handle_turtle_pose
        # callback function on each message
        self.subscription = self.create_subscription(
            Pose,
            f'/{self.turtlename}/pose',
            self.handle_turtle_pose,
            1)
        self.subscription  # prevent unused variable warning

    def handle_turtle_pose(self, msg):
        t = TransformStamped()

        # Read message content and assign it to
        # corresponding tf variables
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'world'
        t.child_frame_id = self.turtlename

        # Turtle only exists in 2D, thus we get x and y translation
        # coordinates from the message and set the z coordinate to 0
        t.transform.translation.x = msg.x
        t.transform.translation.y = msg.y
        t.transform.translation.z = 0.0

        # For the same reason, turtle can only rotate around one axis
        # and this why we set rotation in x and y to 0 and obtain
        # rotation in z axis from the message
        q = quaternion_from_euler(0, 0, msg.theta)
        t.transform.rotation.x = q[0]
        t.transform.rotation.y = q[1]
        t.transform.rotation.z = q[2]
        t.transform.rotation.w = q[3]

        # Send the transformation
        self.tf_broadcaster.sendTransform(t)


def main():
    rclpy.init()
    node = FramePublisher()
    try:
        rclpy.spin(node)
```

C.R. 67
```python
    except KeyboardInterrupt:
        pass

    rclpy.shutdown()
```

**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`.

C.R. 68
```python
self.turtlename = self.declare_parameter(
  'turtlename', 'turtle').get_parameter_value().string_value
```

Afterward, the node subscribes to topic `{self.turtlename}/pose` and runs function `handle_turtle_pose` on every incoming message.

C.R. 69
```python
self .subscription = self.create_subscription(
    Pose,
    f'/{self.turtlename}/pose',
    self.handle_turtle_pose,
    1)
```

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`.

C.R. 70
```python
t = TransformStamped()

# Read message content and assign it to
# corresponding tf variables
t.header.stamp = self.get_clock().now().to_msg()
t.header.frame_id = 'world'
t.child_frame_id = self.turtlename
```

Here we copy the information from the 3D turtle pose into the 3D transform.

C.R. 71
python
```
# Turtle only exists in 2D, thus we get x and y translation
# coordinates from the message and set the z coordinate to 0
t.transform.translation.x = msg.x
t.transform.translation.y = msg.y
t.transform.translation.z = 0.0

# For the same reason, turtle can only rotate around one axis
# and this why we set rotation in x and y to 0 and obtain
# rotation in z axis from the message
q = quaternion_from_euler(0, 0, msg.theta)
t.transform.rotation.x = q[0]
t.transform.rotation.y = q[1]
t.transform.rotation.z = q[2]
t.transform.rotation.w = q[3]
```

Finally we take the transform that we constructed and pass it to the sendTransform method of the TransformBroadcaster that will take care of broadcasting.

C.R. 72
python
```
# Send the transformation
self.tf_broadcaster.sendTransform(t)
```

**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:

C.R. 73
python
```
# Send the transformation
self.tf_broadcaster.sendTransform(t)
```

**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:

C.R. 74
python
```
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='learning_tf2_py',
```

C.R. 75
python
```
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
    ])
```

**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.

C.R. 76
python
```
from launch import LaunchDescription
from launch_ros.actions import Node
```

Now we run our nodes that start the turtlesim simulation and broadcast turtle1 state to the tf2 using our `turtle_tf2_broadcaster` node.

C.R. 77
python
```
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='learning_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
```

**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:

C.R. 78
xml
```
<exec_depend>launch</exec_depend>
<exec_depend>launch_ros</exec_depend>
```

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:

C.R. 79
```python
data_files=[
    ...
    (os.path.join('share', package_name, 'launch'), glob('launch/*')),
],
```

Also add the appropriate imports at the top of the file:

C.R. 80
```python
import os
from glob import glob
```

**Build** Run rosdep in the root of your workspace to check for missing dependencies.

C.R. 81
```bash
rosdep install -i --from-path src --rosdistro humble -y
```

Still in the root of your workspace, build your package:

C.R. 82
```bash
colcon build --packages-select learning_tf2_py
```

Open a new terminal, navigate to the root of your workspace, and source the setup files:

C.R. 83
```bash
. install/setup.bash
```

**Run** Now run the launch file that will start the turtlesim simulation node and `turtle_tf2_broadcaster` node:

C.R. 84
```bash
ros2 launch learning_tf2_py turtle_tf2_demo.launch.py
```

In the second terminal window type the following command:

C.R. 85
```bash
ros2 run turtlesim turtle_teleop_key
```

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:

C.R. 86
```bash
ros2 run tf2_ros tf2_echo world turtle1
```

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:

```text
At time 1714913843.708748879
- Translation: [4.541, 3.889, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.999, -0.035]
- Rotation: in RPY (radian) [0.000, -0.000, -3.072]
- Rotation: in RPY (degree) [0.000, -0.000, -176.013]
- Matrix:
 -0.998  0.070  0.000  4.541
 -0.070 -0.998  0.000  3.889
  0.000  0.000  1.000  0.000
  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.

# Glossary

**AMR** Autonomous Mobile Robotics. 5, 10, 27–30, 32–37, 39–43, 45, 49, 55–57, 59, 61, 67–74, 76, 77, 81–83, 85–90, 98, 99, 102–107, 110, 111

**API** Application Programming Interface. 206, 207

**CCD** Charge Coupled Device. 10, 28, 32, 47, 51–56, 70, 103, 104

**CLI** Command-Line Interface. 117–119, 130, 133, 134, 136, 139, 149, 154, 156, 165, 214

**CMOS** Complimentary MOS. 10, 47, 51, 52, 55, 56

**dddd** **First In First Out.** A method where the oldest items or elements in a system are processed or removed before newer ones. It's a common principle used in various areas, including inventory management, data processing, and queue management.. 190

**DoF** Degrees of Freedom. 7, 9–12

**GNSS** Global Navigation Satellite System. 69, 89

**GPS** Global Positioning System. 38, 69, 88

**GUI** Graphical User Interface. 117, 126, 133, 134, 149, 182, 214, 216

**LED** Light-Emitting Diode. 44

**LIDAR** Light Detection and Ranging. 43

**OS** Operating System. 123–126, 129, 182

**PC** Personal Computer. 118

**POSIX** Portable Operating System Interface. 181, 182

**PSD** Position Sensing Device. 47, 48

**RAM** Random Access Memory. 131

**ROS** Robot Operating System 2. 1, 123, 125, 128, 181–184, 189, 191–195, 197, 199–201, 203, 205, 206, 208, 211–215, 218, 219, 221, 222, 227, 231, 235, 239, 241–245, 247, 249, 251, 253, 256, 257, 259, 263, 265, 271, 273, 279, 281, 282, 288–291

**SLAM** Simultaneous Localisation and Mapping. 107

**SNR** Signal-to-Noise Ratio. 70

**ToF** Time-of-Flight. 40, 41, 43

**VNC** Virtual Network Computing. 127

# Bibliography


[1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] Michael LaBarbera. "Why the wheels won't go". In: *The American Naturalist* 121.3 (1983), pp. 395–408.

[3] Julian FV Vincent et al. "Biomimetics: its practice and theory". In: *Journal of the Royal Society Interface* 3.9 (2006), pp. 471–482.

[4] Fran *c*cois Druelle et al. "Convergence of bipedal locomotion: why walk or run on only two legs". In: *Convergent Evolution: Animal Form and Function*. Springer, 2023, pp. 431–476.

[5] Damian M Lyons and Kiran Pamnany. "Rotational legged locomotion". In: *ICAR'05. Proceedings., 12th International Conference on Advanced Robotics, 2005.* IEEE. 2005, pp. 223–228.

[6] G Schweitzer. "ROBOTRAC-a Mobile Manipulator Platform for Rough Terrain". In: *Proc. of Int. Symp. on Advanced Robot Technology*. 1991, pp. 411–416.

[7] Mathias Thor et al. "A dung beetle-inspired robotic model and its distributed sensor-driven control for walking and ball rolling". In: *Artificial Life and Robotics* 23 (2018), pp. 435–443.

[8] Z. P. Square R. Jones. *All Praise The Humble Dung Beetle*. 2018. URL: https://www.smithsonianmag.com/science-nature/the-humble-dung-beetle-180967781/.

[9] Sharp Photography. *Common ostrich (Struthio camelus australis) male running (composite image), Damaraland, Namibia*. 2018. URL: https://commons.wikimedia.org/wiki/File:Common_ostrich_(Struthio_camelus_australis)_male_running_composite.jpg.

[10] Porges. *Zebra in Wellington Zoo*. 2025. URL: https://commons.wikimedia.org/wiki/File:Zebra_sideview.jpg.

[11] Encyclopaedia Brittanica. *Ant*. 2025. URL: https://cdn.britannica.com/42/223142-050-7033F421/Red-ant-on-a-green-branch.jpg.

[12] Zhanbing Song et al. "The Impact of Exercise Play on the Biomechanical Characteristics of Single-Leg Jumping in 5-to 6-Year-Old Preschool Children". In: *Sensors* 25.2 (2025), p. 422.

[13] Sven Böttcher. "Principles of robot locomotion". In: *Proceedings of human robot interaction seminar*. 2006.

[14] Andrzej Krzywinski, Anna Niedbalska, and L Twardowski. "Growth and development of hand reared fallow deer fawns." In: *Acta theriologica* 29.29 (1984), pp. 349–356.

[15] John Brackenbury. "Caterpillar kinematics". In: *Nature* 390.6659 (1997), pp. 453–453.

[16] David A Winter. *Biomechanics and motor control of human gait: normal, elderly and pathological*. 1991.

[17] Francesco Lacquaniti, Yuri P Ivanenko, and Myrka Zago. "Patterned control of human locomotion". In: *The Journal of physiology* 590.10 (2012), pp. 2189–2199.

[18] José L Pons. *Wearable robots: biomechatronic exoskeletons*. John Wiley & Sons, 2008.

[19] William P Zyhowski, Sasha N Zill, and Nicholas S Szczecinski. "Adaptive load feedback robustly signals force dynamics in robotic model of Carausius morosus stepping". In: *Frontiers in Neurorobotics* 17 (2023), p. 1125171.

[20] Hyunglae Lee and Neville Hogan. "Investigation of human ankle mechanical impedance during locomotion using a wearable ankle robot". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE. 2013, pp. 2651–2656.

[21] R McN Alexander. "Optimization and gaits in the locomotion of vertebrates". In: *Physiological reviews* 69.4 (1989), pp. 1199–1227.

[22] MIT. *The Raibert Hopper*. 1984. URL: http://www.ai.mit.edu/projects/leglab/robots/3D_hopper/3D_hopper.html.

[23] Seshashayee S Murthy and Marc H Raibert. "3D balance in legged locomotion: modeling and simulation for the one-legged case". In: *ACM SIGGRAPH Computer Graphics* 18.1 (1984), pp. 27–27.

[24] Marc H Raibert, H Benjamin Brown Jr, and Michael Chepponis. "Experiments in balance with a 3D one-legged hopping machine". In: *The International Journal of Robotics Research* 3.2 (1984), pp. 75–92.

[25] Citizendium. *Asimo*. 2005. URL: https://citizendium.org/wiki/ASIMO.

[26] Ben Brown and Garth Zeglin. "The bow leg hopping robot". In: *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No. 98CH36146)*. Vol. 1. IEEE. 1998, pp. 781–786.

[27] Robert Ringrose. "Self-stabilizing running". In: *Proceedings of International Conference on Robotics and Automation*. Vol. 1. IEEE. 1997, pp. 487–493.

[28] Flamingo. *Spring Flamingo*. 2000. URL: http://www.ai.mit.edu/projects/leglab/robots/Spring_Flamingo/Spring_Flamingo.html.

[29] Ilon Bengt Erland. "Rad fuer ein laufstabiles, selbstfahrendes fahrzeug". In: *German Patent No. DE2354404A1* (1974).

[30] Kevin Dowling et al. "NAVLAB An Autonomous Navigation Testbed". In: *Vision and Navigation: The Carnegie Mellon Navlab*. Springer, 1990, pp. 259–282.

[31] ackerman. *Ackerman Steering Mechanism*. 2025. URL: https://www.dubizzle.com/blog/cars/ackerman-steering-mechanism/.

[32] Farnell. *Rotary Encoder, Module, Optical, Incremental, 500 PPR, 0 Detents, Vertical, Without Push Switch*. 2025. URL: https://at.farnell.com/en-AT/broadcom-limited/aedb-9140-a13/encoder-3channel-500cpr-8mm/dp/1161087.

[33] Flyrobo. *GY-26 Digital Electronic Compass Sensor Module*. 2025. URL: https://www.flyrobo.in/gy-26-digital-electronic-compass-sensor-module.

[34] FindLight. *Optical Gyroscopes: Measuring Rotational Changes With Sagnac Effect*. 2025. URL: https://www.findlight.net/blog/optical-gyroscopes-measuring-rotations/.

[35] PiHut. *HC-SR04 Ultrasonic Range Sensor on the Raspberry Pi*. 2025. URL: https://thepihut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi.

[36] Reinhold. *Structured light sources on display at the 2014 Machine Vision Show in Boston*. 2014. URL: https://commons.wikimedia.org/wiki/File:Structured_light_sources.agr.jpg.

[37] Andrzej. *CCD image sensor SONY ICX493AQA 10,14 (Gross 10,75) M pixels APS-C 1.8" 28.328mm (23.4 x 15.6 mm) from module IS-026 from digital camera SONY DSLR-A200 or DSLR-A300 sensor side*. 2014. URL: https://commons.wikimedia.org/wiki/File:CCD_SONY_ICX493AQA_sensor_side.jpg.

[38] TeledyneCCD. *How a Charge Coupled Device (CCD) Image Sensor Works*. 2020. URL: https://www.teledyneimaging.com/media/1300/2020-01-22_e2v_how-a-charge-coupled-device-works_web.pdf.

[39] K. Hirakawa and T.W. Parks. "Chromatic adaptation and white-balance problem". In: *IEEE International Conference on Image Processing 2005*. Vol. 3. 2005, pp. III–984. DOI: 10.1109/ICIP.2005.1530559.

[40] Fstoppers. *Is There a Difference Between Color Temperature and White Balance?* 2022. URL: https://fstoppers.com/natural-light/there-difference-between-color-temperature-and-white-balance-596031.

[41] Adrian Ilie and Greg Welch. "Ensuring color consistency across multiple cameras". In: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. Vol. 2. IEEE. 2005, pp. 1268–1275.

[42] Teledyne. *Saturation and Blooming*. 2025. URL: https://www.photometrics.com/learn/imaging-topics/saturation-and-blooming.

[43] Zhen Zhang et al. "Analysis and simulation to excessive saturation effect of CCD". In: *2nd International Symposium on Laser Interaction with Matter (LIMIS 2012)*. Vol. 8796. SPIE. 2013, pp. 96–102.

[44] Baumer. *Operating principle and features of CMOS sensors*. 2025. URL: https://www.baumer.com/int/en/service-support/function-principle/operating-principle-and-features-of-cmos-sensors/a/EMVA1288.

[45] econ Systems. *CMOS camera module*. https://www.directindustry.com/prod/e-con-systems/product-168594-2365044.html. URL: https://www.directindustry.com/prod/e-con-systems/product-168594-2365044.html.

[46] TS Holst and GC Lomheim. *CMOS/CCD Sensors*. JCD publishing, 2007.

[47] Mdf. *A photon noise simulation*. 2010. URL: https://commons.wikimedia.org/wiki/File:Photon-noise.jpg.

[48] Mark-j. *Open Camera Blog*. 2018. URL: https://sourceforge.net/p/opencamera/blog/2018/09/focus-bracketing-with-open-camera/.

[49] UoW. *How Mars rovers use artificial intelligence*. 2023. URL: https://online.wlv.ac.uk/how-mars-rovers-use-artificial-intelligence/.

[50] Tim Bailey. "Mobile robot localisation and mapping in extensive outdoor environments". PhD thesis. Citeseer, 2002.

[51] Sean Campbell et al. "Where am I? Localization techniques for mobile robots a review". In: *2020 6th International Conference on Mechatronics and Robotics Engineering (ICMRE)*. IEEE. 2020, pp. 43–47.

[52] Victoria J Hodge. "Sensors and data in mobile robotics for localisation". In: *Encyclopedia of Data Science and Machine Learning* (2023), pp. 2223–2238.

[53] Omar Jaradat et al. "Challenges of safety assurance for industry 4.0". In: *2017 13th European Dependable Computing Conference (EDCC)*. IEEE. 2017, pp. 103–106.

[54] David Filliat and Jean-Arcady Meyer. "Map-based navigation in mobile robots:: I. a review of localization strategies". In: *Cognitive systems research* 4.4 (2003), pp. 243–282.

[55] John J Leonard and Hugh F Durrant-Whyte. *Directed sonar sensing for mobile robot navigation*. Vol. 175. Springer Science & Business Media, 2012.

[56] Michael G Wing, Aaron Eklund, and Loren D Kellogg. "Consumer-grade global positioning system (GPS) accuracy and reliability". In: *Journal of forestry* 103.4 (2005), pp. 169–173.

[57] Nicola Bellotto and Huosheng Hu. "People tracking and identification with a mobile robot". In: *2007 International Conference on Mechatronics and Automation*. IEEE. 2007, pp. 3565–3570.

[58] Mohan Sridharan and Peter Stone. "Color learning and illumination invariance on mobile robots: A survey". In: *Robotics and Autonomous Systems* 57.6-7 (2009), pp. 629–644.

[59] Diego Galar and Uday Kumar. "Chapter 1 - Sensors and Data Acquisition". In: *eMaintenance*. Ed. by Diego Galar and Uday Kumar. Academic Press, 2017, pp. 1–72. ISBN: 978-0-12-811153-6. DOI: https://doi.org/10.1016/B978-0-12-811153-6.00001-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128111536000014.

[60] David R Williams. "Aliasing in human foveal vision". In: *Vision research* 25.2 (1985), pp. 195–205.

[61] maksim. *An example of a poorly sampled brick pattern*. 2006. URL: https://commons.wikimedia.org/wiki/File:Moire_pattern_of_bricks_small.jpg.

[62] Gaddi Blumrosen, Ben Fishman, and Yossi Yovel. "Noncontact wideband sonar for human activity detection and classification". In: *IEEE Sensors Journal* 14.11 (2014), pp. 4043–4054.

[63] Angelo M Sabatini and Valentina Colla. "A method for sonar based recognition of walking people". In: *Robotics and Autonomous Systems* 25.1-2 (1998), pp. 117–126.

[64] Parag H Batavia and Illah Nourbakhsh. "Path planning for the Cye personal robot". In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*. Vol. 1. IEEE. 2000, pp. 15–20.

[65] xkcd. *xkcd-Spirit*. 2025. URL: https://xkcd.com/695/.

[66] J Reuter. "Scan-and featurebased multiple hypothesis tracking for mobile robot localization: a data fusion approach". In: *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 99CH37028)*. Vol. 4. IEEE. 1999, pp. 714–719.

[67] Péter Fankhauser, Michael Bloesch, and Marco Hutter. "Probabilistic terrain mapping for mobile robots with uncertain localization". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3019–3026.

[68] Yonhap News Agency. *Museum guide robot*. 2018. URL: https://en.yna.co.kr/view/PYH20181221009400341.

[69] Oliver Brock and Oussama Khatib. "High-speed navigation using the global dynamic window approach". In: *Proceedings 1999 ieee international conference on robotics and automation (Cat. No. 99CH36288C)*. Vol. 1. IEEE. 1999, pp. 341–346.

[70] J Borenstein and Y Koren. "Fast obstacle avoidance for mobile robots". In: *IEEE Trans Rob Autom. v7* (), pp. 278–287.

[71] Rodney Brooks. "A robust layered control system for a mobile robot". In: *IEEE journal on robotics and automation* 2.1 (1986), pp. 14–23.

[72] Illah Nourbakhsh, Rob Powers, and Stan Birchfield. "DERVISH an office-navigating robot". In: *AI magazine* 16.2 (1995), pp. 53–53.

[73] Fan Wang et al. "Object-based reliable visual navigation for mobile robot". In: *Sensors* 22.6 (2022), p. 2387.

[74] AaaravG. *How to Make Line Follower Robot Using Arduino*. 2018. URL: https://www.instructables.com/Line-Follower-Robot-Using-Arduino-2/.

[75] Timothy P McNamara, James K Hardy, and Stephen C Hirtle. "Subjective hierarchies in spatial memory." In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 15.2 (1989), p. 211.

[76] Marvin M Chun and Yuhong Jiang. "Contextual cueing: Implicit learning and memory of visual context guides spatial attention". In: *Cognitive psychology* 36.1 (1998), pp. 28–71.

[77] Chenguang Huang et al. "Visual language maps for robot navigation". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 10608–10615.

[78] Daniel Meyer-Delius et al. "Using artificial landmarks to reduce the ambiguity in the environment of a mobile robot". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 5173–5178.

[79] Jakub Hazik et al. "Fleet management system for an industry environment". In: *Journal of Robotics and Control (JRC)* 3.6 (2022), pp. 779–789.

[80] Elias Xidias, Paraskevi Zacharia, and Andreas Nearchou. "Intelligent fleet management of autonomous vehicles for city logistics". In: *Applied Intelligence* 52.15 (2022), pp. 18030–18048.

[81] Kivaan. *An official DARPA photograph of Stanley at the 2005 DARPA Grand Challenge*. 2007. URL: https://commons.wikimedia.org/wiki/File:Stanley2.JPG.

[82] Shuji Hashimoto et al. "Humanoid robots in waseda university–hadaly-2 and wabian". In: *Autonomous Robots* 12 (2002), pp. 25–38.

[83] Raymond J Carroll and David Ruppert. *Transformation and weighting in regression*. Chapman and Hall/CRC, 2017.

[84] James Bruce, Tucker Balch, and Manuela Veloso. "Fast and inexpensive color image segmentation for interactive robots". In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*. Vol. 3. IEEE. 2000, pp. 2061–2066.

[85] Illah R Nourbakhsh et al. "Mobile robot obstacle avoidance via depth from focus". In: *Robotics and Autonomous Systems* 22.2 (1997), pp. 151–158.

[86] Manske. *Hughes Letter-Printing Telegraph Set built by Siemens and Halske in Saint Petersburg, Russia, ca.1900*. 2009. URL: https://commons.wikimedia.org/wiki/File:Printing_Telegraph.jpg.

[87] Oded Koren. "A study of the Linux kernel evolution". In: *ACM SIGOPS Operating Systems Review* 40.2 (2006), pp. 110–112.

[88] Maurice J Bach. "The Design of the UNIX". In: *RTM. Operating system Prentice Hall* (1986), pp. 312–329.

[89] Steven C Johnson and Dennis M Ritchie. "UNIX time-sharing system: Portability of C programs and the UNIX system". In: *The Bell System Technical Journal* 57.6 (1978), pp. 2021–2048.

[90] Debian. *python3-colcon-ros*. 2025. URL: https://packages.debian.org/sid/python3-colcon-ros.

[91] Open Robotics. *ROS 2 Iron Documentation*. 2025. URL: https://docs.ros.org/en/iron/Concepts/Intermediate/About-Tf2.html.