Robotics

FOR PROGRAMMERS

Andreas Bihlmaier

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Robotics for Programmers

Version 4

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Robotics for Programmers*.

I want to congratulate and at the same time warn you that this may very well be the beginning of a life-long passion. I can still vividly remember when I first got in touch with robotics more than 15 years ago, after having been programming for more than a decade already. Robot software is (just) software. However, only robots create this unique intimate connection between virtual worlds residing in computers and the real world around us. You will very soon learn why this is the case, what it entails and how you can participate in it.

You do not require any prior experience with robotics nor with related fields such as physics, mechanics, electronics or control. I do assume you know how to read and write Python code at an intermediate level including basic object-oriented programming (OOP) constructs such as classes and inheritance. You should also be familiar with the basics of Linux and its command line interface. The same goes for generic software tools such as git and ssh together with having a conceptual understanding of computer networking.

You will learn, step by step, how to conceptualize and design robot software systems, break them down into their building blocks, implement these core robot software components and combine them into a working overall solution.

Part 1 provides you with a systematic overview of robotics and robot software systems. In part 2 you will learn about the fundamental robot capabilities along with learning how to use the Robot Operating System (ROS2) and the Gazebo simulator. More advanced robot capabilities are the focus of part 3. In part 4 we put your new skills to use in four concrete robot applications. Part 5 shows you how to build robot software in a professional manner and you will learn about important practical considerations when working with robots.
Part 6 concludes the book by pointing out even more use cases for your newly acquired robotics skills.

I hope you enjoy *Robotics for Programmers* and I'm very much looking forward to your questions, comments or suggestions in the liveBook discussion forum as well as seeing your future robots!

- Andreas Bihlmaier

brief contents

1 Robotics or Bits Meet Atoms

This chapter covers

- A definition and overview of robotics
- Essential building blocks of robotics software
- The robotics software development lifecycle

Given its origins in the 1960s, modern robotics is not a young industry anymore, however it has never been more exciting than today. Why are current times so exciting for roboticists? Because the gradual technological progress in many of the areas that make up the foundation of robotics are enabling an entirely new generation of robots. Although we are far away from general purpose robots with human-level capabilities, the emerging new capabilities are not less impressive. Robots are becoming more adaptive to dynamic environments, are gaining entirely new skills and will be much easier to program and use. Within the next decades, robots that have been the subject of science fiction will become part of everyday life. This goes along with tremendous opportunities for roboticists, be it as employees, freelancers, entrepreneurs as well as those simply doing robotics in their spare time. What could be more exciting than being part of this journey?

In this introductory chapter you will learn what makes up a robotics system and what differentiates robots from other types of machines. This book is about robot *software* development. You will learn a lot about robots beyond programming, but the focus clearly is set on software. Robotics is the field where bits meet atoms, or, put differently, where the virtual and the physical world interact intimately. Therefore, hardware *is* an essential part of robots. Nevertheless, the majority of new robot capabilities are software implemented. All of them rely on sophisticated hardware, from better motors and gears to improved power electronics and high resolution sensors, such as cameras. Let me state it again: disregarding the hardware aspect of robotics is a recipe for failure. Everyone approaching robotics with a background in software needs to keep this in mind. On the other hand, no matter how capable the hardware, a robot without the right software will be a very expensive paperweight. Everyone with a background in mechanical engineering, electrical engineering or another hardware field must remember this other side of the same coin.

What does this mean for you as aspiring roboticist, or robot software engineer, be it as an industry professional or an ambitious maker? Learn how the physical world is acquired into data through sensors and how data is transformed into physical actions through actuators. Everything in between happens in the realm of code already familiar to you as a software engineer. Robot software is software, but it is as different from other kinds of software as a web application is different from an operating system. The specific skills needed to go from knowing how to write other kinds of software to being able to write software for robots is the essence of this book.

1.1 Robots and other machines

Before having a look at what makes up a robot system, let's get a better understanding what is a robot and what is not. To avoid getting into theoretical debates on the nature of robots or getting lost in encyclopedic definitions[1], we answer this question, for the purpose of this book, from a practical perspective, by looking at some examples.

The first boundary between robots and other things is simple: If there is no significant direct physical interaction with the real world, it is not a robot. Showing something on a screen, blinking LEDs or generating sounds is not sufficient to count as significant interaction. Thus, chat bots, voice assistants, robotic process automation (RPA), smartphones and regular computers in general are *not* considered as robots.

The second boundary we want to draw is between robots and machines in general. All machines satisfy the criteria of significant direct physical interaction with the real world. No matter whether you think of a microwave, your coffee maker, (non-self-driving) cars or industrial equipment, each of them clearly significantly alters something in its physical environment. However, what the majority of machines is lacking to qualify them as robots is programmability and a general purpose nature. Being able to use software to control the operation of a device is a quite straightforward criterion. Although more and more machines, even very simple ones, are controlled through software running on some sort of microcontrollers, the firmware, it is not intended for their program to be changed, apart from bug fixes. Yes, it is possible to hack your coffee maker and change details of its operation by modifying its firmware, but likely its purpose would still be to - well - make coffee.

This is where the second part comes in, the general purpose nature or open-endedness of robots. Robots are *intended* to change their behavior and tasks according to the program being executed on them. A computer that always runs the same program, is still general purpose in the sense here. One only has to deploy different software to it in order to (completely) change its purpose and behavior. The opposite end of the spectrum is a specialized chip, e.g. an application-specific integrated circuit (ASIC), which can only perform one highly specific function. Therefore, the latter would not be considered general purpose. Of course, a particular type of robot is limited in what it can do by hardware or physics. There are no universal robots in the sense of computers being universal in the computations they can perform.[2] Still, using some common sense, it is quite obvious how a robot that can be programmed to either weld car chassis, assemble a smartphone, draw calligraphy or serve coffee, is much more general purpose than a device that can make hot water, espresso and latte macchiato.

The reason for making sure you have a solid understanding of what is a robot and what is not can be found in the proverb if all you have is a hammer, everything looks like a nail. I hope after reading this book, you will not just have gained robot programming skills, but will also see your environment under a robotics perspective. In other words, you identify the robotic aspects of man-made and natural systems around you. Yet, as enlightening (and at times hilarious!) this new perspective can be, you should make sure that you apply robotics software to robots. Don't force robotics software principles onto things that are not robots.

1.2 What you will learn about robots

Now that we have clarified what a robot is, let's have a look at what you will learn in this book and what you are assumed to already know. Most importantly, you do not require any prior experience with robotics. However, the book makes some assumptions about other skills you have already acquired. First of all and most importantly, you are assumed to know how to read and write Python code at an intermediate level including basic object-oriented programming (OOP) constructs such as classes and inheritance.[3] Second, you should be familiar with the basics of Linux and its command line. In terms of specific tools, being able to check out code with `git` and login to remote hosts via `ssh` is important. Third, a conceptual understanding of computer networking such as DNS, IP addresses, ports and protocols such as TCP and UDP is assumed. Finally, some topics outside of programming are relevant to this robot software book. We will touch on some math and some topics from classical physics. You will not need to solve equations or remember the relation between a body's mass, acceleration and force from the top of your head, but you should have a general idea what these terms mean. In any case, don't worry if you do not feel very confident with these fields, appendix B contains practice-oriented minimal recaps on each. You are not expected to read them up front, rather they will be explicitly referred to when each topic comes up. In summary, if you know intermediate Python, some Linux, some networking and still remember some math and physics from school, you are ready for learning robot software development.

This book will teach you to

- conceptualize and design robot software systems,

- break them down into their building blocks and
- combine these software components into a working overall solution.

You will learn about different types of robots, such as

- Robot arms, also known as (industrial) robot manipulators that handle objects.
- Mobile robots, both those driving on the ground as well as those flying through the air.
- Mobile manipulators that combine capabilities of handling objects and moving around the environment.

The examples scenarios will cover a wide range of robot applications from A as in automated assembly to P as in Pizza delivery to Z as in Zebrafish handling[4]. Code and exercises utilize common open-source robotics software libraries and simulators such as the Robot Operating System (ROS) framework[5] and Gazebo[6]. Although you will be able to control real and simulated robots using ROS, the goal is not to teach you a particular robotics framework, but rather to enable you to quickly become productive with any commercial or open-source robotics software stack by understanding the common concepts implemented in them.[7]

1.3 Robotics the big picture

Every robot system can be conceptually split into six distinct parts: Environment, Sensors, Sensing, Planning, Acting and Actuators. Don't worry if these terms do not yet mean much - or anything - to you. Each part will be introduced and explained below. Figure 1.1 illustrates how the six parts belong together and how information flows through the robot system from left to right.

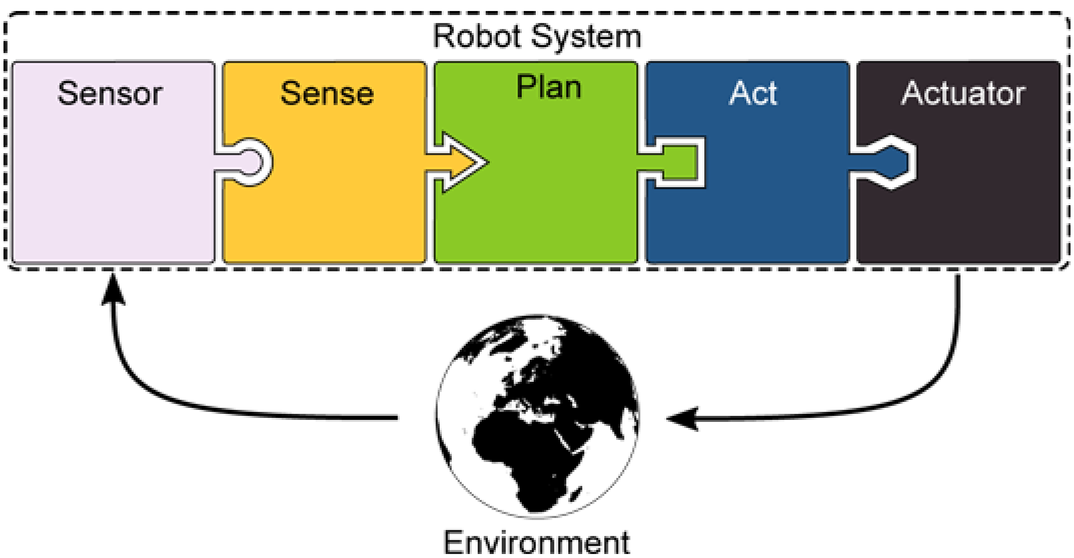


Figure 1.1 The conceptual structure and essential six parts of all robot systems.

The elements and shapes in figure 1.1 will accompany you throughout this book. Depending on what subsystem of the robot we focus on and what type of robot we are looking at, they will have different sizes, perform different functions and show more or less inner structure, but they will still be easy to recognize. Figure 1.2 shows three different robot types and two different emphasis on subsystems. You do not need to understand the details of this figure right now, everything will be introduced step by step in the next chapters. The important part is to realize that widely different types of robots can be seen and structured in a common way. Also, it is important that you understand how this commonality will be illustrated in the book.

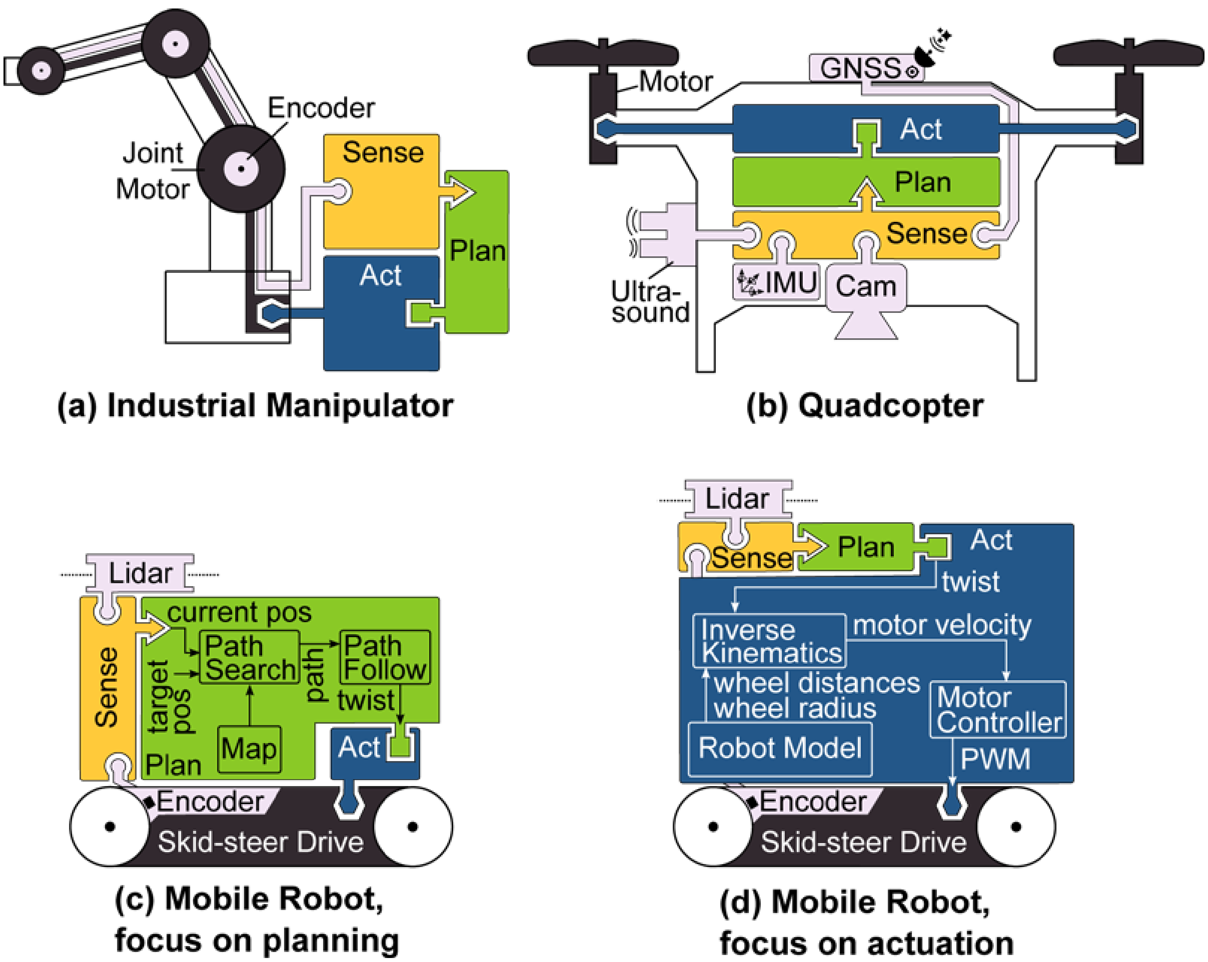


Figure 1.2 An example of different robot systems and how they can be structured in a common way. The diagrams reflect the relative importance of each part through its size and degree of detail.

At times we will zoom into one particular subsystem and leave the others out of the diagram. Still, either the surrounding robot system will have just been presented before, thus making the context of the shown subsystem clear, or the context will not be relevant for the discussion. An example of such a zoomed-in diagram is shown in figure 1.3.

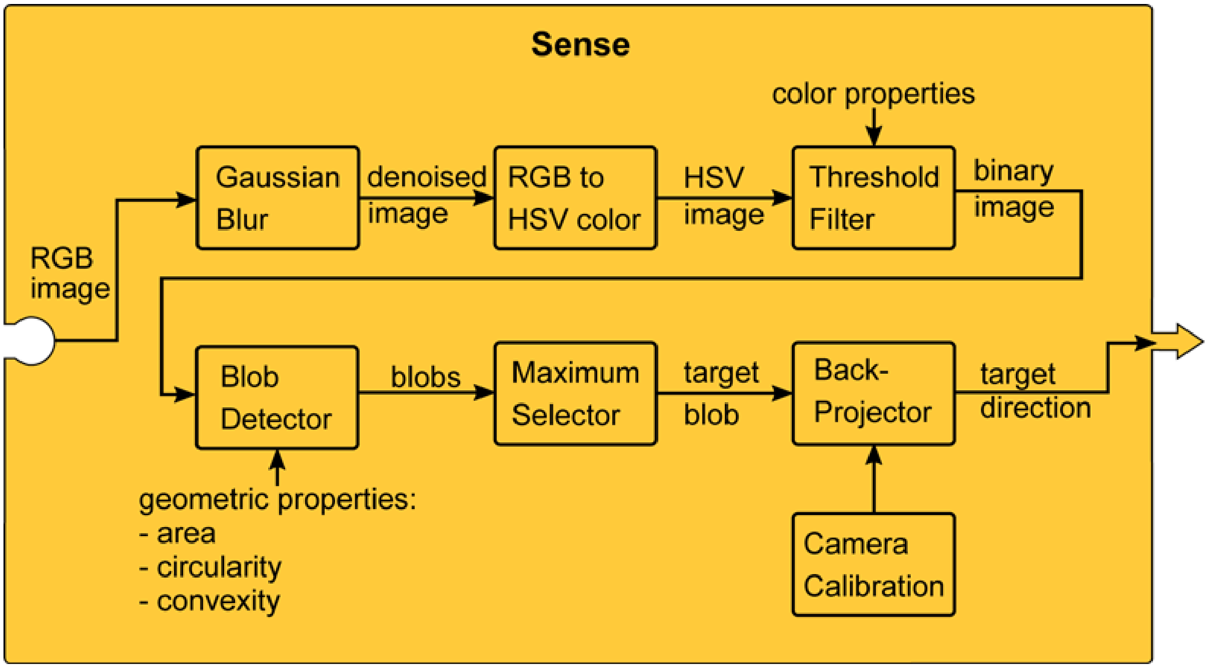


Figure 1.3 An example of a zoomed-in diagram. Only some parts of the overall robot system are shown (here: Sensing).

Again the details shown it the diagram do not matter, I just want to give you an impression of how these diagrams relate to the conceptual structure of figure 1.1.

After this brief detour about the book's robot diagrams, let's go back to the six essentials parts of a robot system from figure 1.1.

1.3.1 Environment

As explained above, robots are programmable general purpose machines that directly interact with the real world to perform tasks. The robot's environment, the specific segment of the real world the robot is currently interacting with, is both the first and last part in the robot system. The environment is the first part because the robot system needs to acquire information about the environment in order to compute the next action towards achieving the intended task. The environment is the last part because once information has been processed and the next action is executed the state of the environment is modified via this action. Once the sequence environment, sensor, sensing, planning, acting, actuators and environment is finished, it starts again from the beginning. A common way to talk about this circular structure is to state that the environment closes the loop of the robot system.

However, repeating this sequence over and over does not lead to the same results over and over again. One reason is that actions change the state of the environment and the state of the robot. For example, a mobile robot that executes the action to rotate its wheels in one iteration of the loop, will have moved until the next loop iteration. Another reason is that the environment might change by itself over time. This change can be caused by natural phenomena such as unsupported objects falling down and thus having a different position each time the loop is executed. Also other natural or artificial systems, e.g. humans and machines, might share the environment with the robot system and change this shared environment.

Since the environment and its role in the robot loop are so essential, it is worth looking at it from another perspective: code. A minimal Python snippet of the robot loop could look like this:

```
while True:
  state = sense()
  action = plan(state)
  act(action)
```

This loop does not lead to the same result over and over again because `sense()` does not return the same `state` each time and because `act()` does not lead to the same effect each time, even if the parameter `action` is identical. The first part is likely easy to understand for Software Engineers. The `sense()` function gets data from outside the function and can thus return different results each time it is called. Common examples in the software realm are reading from a database or a network connection. What about the second part, `act()` not leading to the same effect even when passing in the same argument? At first glance this might seem like a less familiar situation. It might even sound like a bug. Let me give you two examples where the same happens in pure software. When you run a program that creates a file for the first time the file will likely not exist. However, when you run the identical program again, it will probably behave differently as then the file already exists. The second example is executing `del` on a Python object. The first time the object will be deleted. The second time the program will crash as the object no longer exists. To make it general, actions can depend on the *state* of things. In robotics, this is not only the state within the robot software, but to a large extent also the state of the robot and its environment. Repeatably moving forward 1 cm is performing the same action over and over again, but the result will differ greatly between being more and being less than 1 cm away from a wall.

To summarize, the environment is the external state of every robot system that will determine not only the result of sensing but also the effect of actions and it closes the robot loop.

1.3.2 Sensors

Sensors are a robot's input devices. They measure various physical quantities in the environment and make these measurements available as data to the sensing part of the robot software system. In this book sensors are considered to be hardware devices.[8] We will basically treat all sensor types as black boxes with a known input-output behavior and an API that makes this data available for processing. The sensor input is some physical quantity in the environment and the output is data representing a measurement of this quantity.

Later chapters will explain specific types of sensors in detail. In order to give you an initial idea what a sensor is here are some examples:

- A temperature sensor has heat as input and outputs a temperature in units of Kelvin, Celsius or Fahrenheit.
- A camera sensor has light as input and outputs a (two dimensional) array of spatial light intensities as output.
- A joint position sensor has the relative rotation of two bodies as input and outputs their angular rotation in increments, degrees or radians.

We will not dive into the sensor hardware, that is, we will not discuss through which physical principles the input is converted to data, unless in cases where this is necessary to understand *common* challenges with sensor data in robotics. The challenges mostly fall into two categories: sensor noise and incorrect sensor measurements. Sensor noise means that the measured value fluctuates around the real value that an ideal sensor would measure. Unfortunately, there are no ideal sensors and therefore all sensor measurements are imperfect and have limited accuracy. Whether it is an issue in practice that a temperature sensor's measurement alternates between 23 °C and 24 °C, when the real temperature is 23,42 °C, depends on the particular application. In general increased accuracy goes along with increased cost and sensor size, thus trade-offs are necessary. Incorrect sensor measurements can be much more problematic than normal[9] sensor noise. Incorrect means that the measured value does not just somewhat fluctuate around the real value, but that it is *way* off. The value may not have any relation to the real value at all, although the sensor itself is working just fine. To give you an example, certain types of joint position sensors are heavily influenced by strong magnetic fields. The influence can be so strong that the sensor will actually only measure the magnetic field and no longer anything related to joint position. Another example that is more familiar today, when most people carry digital camera sensors around in their pockets, is underexposure and overexposure. Camera sensors can only measure a certain interval of light intensity. All intensities outside this interval, i.e. too dark or too bright, will be represented as the lowest or the highest data value. Thus all structure in an underexposed or overexposed area of an image is lost. In other words, the data is not just noisy, but incorrect. It has no relation to the real values anymore.

Sensors used in robot systems are as varied as the robot systems, their environments and tasks. While a radiation sensor is - hopefully - not at all relevant to a home vacuum robot, it is highly relevant for robots used for decontamination of nuclear power plants. Therefore, the lifecycle described in section 1.4 starts by looking at the environment, the tasks to be accomplished and then goes to select sensors that provide the necessary data as input to the robot software's sensing part.

1.3.3 Sensing

After looking at the hardware devices, sensors, that provide us with data about the state of the environment and the robot, let us now have a glance at the processing performed with this data. The purpose of the sensing part is to give meaning to the raw data received from the sensors. Reading the value `42` in units of degrees from a rotation sensor, of course tells us, well, that the rotation is 42 degrees. But this does not yet mean anything for the robot system. However, once we combine this data with further information, we can, for example, calculate that the robot is rotated by 42 degrees on its base. From this we could then infer where the tip of the robot arm is positioned at the moment. An example of such additional information is the robot model, which provides the location of the sensor in the robot and the robot's overall mechanical structure. Knowing the current robot pose most certainly has an important meaning in robot systems.[10]

As usual in software development, defining good abstractions and good interfaces between different subsystems is crucial, can be challenging and often does not have one best solution. In robot software, we face the same challenge between the sensing and the planning part. We could do more or less sensor data processing in the sensing part and leave more or less to the planning part. My advice to you is to use the same criterion used in this book: Everything related to getting information out of sensor data *without considering the robot's objectives* is part of sensing. Once objectives are taken into account, we consider it to belong to planning. This type of decision falls more in the category of software development as an art than as science, thus take it as what it is, a helpful heuristic to structure your robot software. To give an example: Locating a mobile robot in a known map based on data from a laser scanner is part of sensing. Finding a path to a target location from the current position, requires having the target location as an objective and is, thus, part of planning.

The input to sensing is raw sensor data and the output is meaningful information that we can use to plan towards achieving the robot's objectives.

1.3.4 Planning

Given an objective and information about the state of the robot and its environment, the planning part calculates the robot's next action(s) towards achieving the objective. Let's look at a few examples of objectives to get a feeling for the kind of planning done in different robot systems (taking those from figure 1.2).

- Industrial manipulator
  - Objective: Move gripper attached to manipulator from current pose into a specific target pose.
  - Planning: Calculate the positions for all manipulator joint motors that result in the gripper being in the target pose. Send next (intermediate) motor positions to the acting part to execute motion.
- Quadcopter

    - Objective: Find and land on a big yellow H marked on the floor inside a building.
    - Planning: Calculate a collision free path through the immediate surrounding area to explore the entire building. If a yellow H is detected, perform a landing procedure. Incrementally send next relative position to acting part for execution.
- Mobile robot fleet
    - Objective: Under the assumption of having a central coordinator, find the optimal assignment between a fleet of mobile robots to transport tasks within a logistics environment that minimizes total distance travelled and maximizes the distance between all robots while moving.
    - Planning: Discretize the map of the environment and the current position of all robots into a graph and utilize graph algorithms to find a solution. Transform this solution back into paths on the actual map and send each robot's path to the acting part. Monitor all robots following their intended path and calculate new paths in case of deviations.

Planning is the part of robot software that is most isolated from the real world, given its position between sensing and acting. The latter abstract away interfacing with hardware and performing physical interaction. Although planning is thus the closest part to pure software in a robotics system, the actual processing is still very much specific to robotics. You will recognize many basic algorithms and data structures used in robot planning from software in general. On the other hand, the way they are combined into solutions in the context of the robot system is quite unique. So is the relationship between information processing and events in the real world.

All data going into planning refers to something in the real world mediated by the sensing part. All results of planning translate to actions taken in the real world executed via the acting part.

1.3.5 Acting

Once an action has been decided in the planning part, it must be carried out. The acting part is taking care of transforming planned actions to performed actions. Similar to the boundary between sensing and planning discussed above, we also need to define a boundary between planning and acting. We draw this boundary along the line of global action and local action, which can also be stated as action with context and action without context. Global action or action with context belongs to the planning part, as it has all the information available about the system state and objectives. Local action or action without context is, hence, where the acting part begins.

Let's look at a simple example to make this boundary clear. Take a robot with one wheel on the left side and one wheel on the right side, each driven by its own motor, a so called differential drive system.[11] Assuming both wheels touch the ground, the resulting motion of the robot will depend on the motion of both wheels. Without going into too much detail here, if only one wheel rotates, the robot would rotate around the other one. If both wheels rotate at the same velocity, the robot would move in a straight line. Other combinations will result in a combination of forward motion and rotation, the robot makes a turn. A good interface from planning to acting in this example would be to command a linear and angular velocity, a so called twist to acting.

Acting is then responsible to calculate the velocity of each wheel and control each wheels motor with this velocity. Whether the resulting motion actually moves the robot into the direction intended by the planning part depends on many external factors such as the surface the robot is driving on, the payload carried and many others.

We always leave it to the planning part to make sure that executed actions work towards the robot's objective. The acting part only ensures that commanded actions are properly transformed to actuator control commands, which are executed via the actuator hardware.

1.3.6 Actuators

We have reached the final stage in the robot system before we are back to the loop-closing environment: actuators. In many ways, actuators are the inverse of sensors. While sensors convert physical quantities to data, actuators do the opposite and convert data into physical quantities. A temperature sensor measures heat and provides it as data. A heating element is an actuator that converts data into heat. To give a more robotics relevant example, a position sensor converts rotation to angular data and a (properly controlled) motor converts angular data to rotation.

In the same way we do with sensors, we treat actuators as a black box with a known API and input-output behavior in this book. Furthermore, same as sensors, actuators nowadays frequently come packaged together with (further) electronics that provide a higher level interface compared to the raw actuator. We will discuss this in some detail for important types of actuators in later chapters. The general principle, however, is the same as for software APIs. You can use the raw file API in Python to `open`, `read` and parse a JSON file or use Python's `json` module to do so. In the former case you need to deal with all the JSON parsing intricacies yourself, while in the latter case someone else has already taken care of this for you, allowing you to focus on working with the JSON data. The same goes for actuators, sometimes you need to implement your own motor position controller. Often you can rely, nonetheless, on already existing off-the-shelf solutions and focus on higher level aspects.

The diversity of actuators is at least as great as that of sensors and no less robot, environment and task dependent. Fortunately, understanding how to deal with some basic actuator types will enable you to quickly figure out how to work with the specific ones in your robot system.[12]

Now we have reached the end point of the robot system. The actuators purposefully alter the environment and robot state to bring our robot one step closer to achieving its objectives. This preliminary end point is not just the beginning of the next loop through the robot system, but also only the beginning of your journey into robotics.

1.4 Robot software development lifecycle

Before concluding this introductory chapter on bits meeting atoms through robots, I want to give you a brief overview of the robot software lifecycle and your role in it as robot software engineer. Let's go through it step by step in figure 1.4.

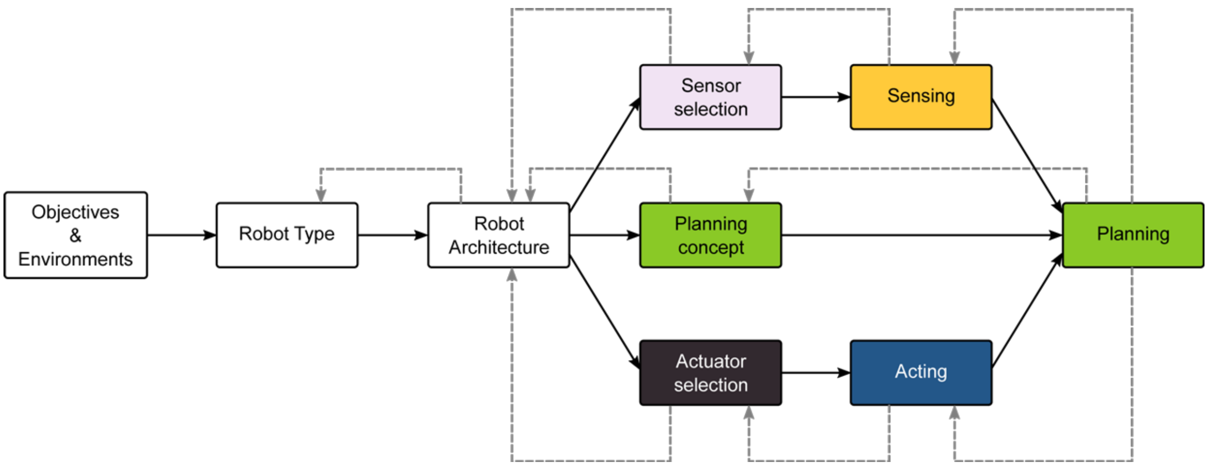


Figure 1.4 The robot (software) development lifecycle.

Robot development should start with a clear definition of the robot's purpose or the objective(s) that it should fulfill. Together with the purpose, we also need to define the environment(s) that the robot should operate in. The more specific and restrictive we can be in this stage, the simpler, more reliable and cost effective our system will turn out.

Afterwards, we need to make a decision what principal type of robot we are going to build. Will we build a stationary manipulator, a driving, flying, swimming or other kind of mobile robot or a mobile manipulator?[13] Sometimes this decision is obvious, as in a flying robot not being well suited for subsea exploration. In other cases, it is much more difficult. For example: Is a legged robot or a flying one more suitable to inspect construction sites? Do we actually need a team of both or a legged flying hybrid? In any case, we must decide what kind of robot we are going to build.

Next, we need to make some fundamental decisions what our robot will look like and how its hardware will be structured, that is decide on the robot architecture. In case of a driving mobile robot, likely questions would be: How big should the robot be? What payload must it be able to carry (including its own weight)? Should it have wheels or tracks? How many and what size? Does it have to be rugged? What about dust-proof and water-proof? This list could go on and on. You might wonder why you as a robot *software* engineer should care about all these hardware topics. The reason is that the robot architecture has a significant impact on what sensors and actuators can be used, what you will have to deal with and what you will be able to do from the software side.

Once the (initial) robot architecture is set, three development tasks can start in parallel:

- Sensor selection
- Planning concept definition
- Actuator selection

Selecting the right sensors and actuators is very much a collaborative effort between hardware and Software Engineers. The selection happens in anticipation of what information can be derived from the sensor raw data via sensing and what acting can be achieved through the actuators. Both taken together with the constraints posed by the robot architecture. Defining a planning concept means that we figure out the planning functions or building blocks that we will need based on the architecture of the robot. In case of a mobile robot on wheels, we are likely looking at a 2D navigation task. However, if we have a legged mobile robot, in addition to navigation, we need to look into such topics as gait patterns, planning foot placement and keeping the robot in balance. This work can be performed in parallel to selecting specific sensors and actuators.

Having selected sensors and actuators, we need to work on the actual sensing and acting parts of the robot software. During selecting we made assumptions about what type of relevant information we can extract from the specific sensors and what type of action we can perform via the selected actuator. Now we need to actually make this happen by implementing both parts and evaluating their performance.

Finally, we can put everything together by implementing the planning part. The planning part must not only work in theory or with ideal input data and perfect action execution, rather it must cope with the (imperfect) information provided by sensing and the (imperfect) execution of actions performed by acting.

Thus far we have only looked at the solid arrows going from left to right in the diagram, let's have a look at the dashed ones going in the opposite direction. What these arrows pointing back to earlier stages indicate is that you should not expect to create a robot system in a strict sequence, but that you will have to iterate. You will likely have to iterate *a lot*. The real world our robot is interacting with is not the clean and well defined virtual world within computers. Often, the real world is actually quite messy with lots of strange edge cases, unknown environments and entirely unforeseen events. In other words, the real world does not have a specification that we can program against. We have to use trial and error to basically reverse engineer it for each robot system. This is one of the contributing factors that sometimes makes robotics difficult, but this also what makes and keeps it exciting. The remainder of the book will teach you how to deal with all these interesting challenges and successfully combine the solutions into a working robot system.

1.5 Summary

- Robots are programmable general purpose machines that directly interact with the real world to perform tasks in it.
- The six conceptual building blocks of robot systems are: Environment, Sensors, Sensing, Planning, Acting and Actuators.
- Sensors and actuators of a robot system are its inputs from and outputs to the environment.
- Sensing transforms raw sensor data to meaningful information that is passed to planning.
- Planning derives actions that bring the robot one step closer to achieving its task from the current robot and environment state.
- Acting transforms commands from planning into physical actions that alter the actual environment and robot state.
- Robot software development is an iterative process starting with a clear definition of the robot's objective and the environment in which this objective is to be achieved.

2 Robots from a Software Point of View

This chapter covers

- **Interacting with the real world through robot APIs**
- **Three fundamental types of robots**
- **Using basic robot sensors**
- **Processing common sensor data**
- **Using basic robot actuators**
- **Controlling common actuators**

In the previous chapter we discussed what properties make a device a robot. Furthermore, we looked at the conceptual structure that can be used for all types of robot systems. In this chapter our focus is on the software interface, the API, which we use to interact with the different parts of a robot.

Along with introducing the robot hardware building blocks, especially sensors and actuators, we will look at a possible API for each of them. Why a *possible* API and not *the* API? Each robot software ecosystem, both open source and proprietary, has come up with their own *specific* APIs for each type of robot, sensor and actuator. However, each type of device has a natural *abstract* API due to its inner workings, the functionality it provides and its purpose in the robot system. Therefore, once you have understood the abstract API for different devices, it is easy to learn the specific APIs that you will encounter. When working with (real or simulated) actual robots in later chapters, we will see in detail how our abstract APIs map to the Robot Operating System (ROS2) API.[14]

After introducing robot APIs in general in the next section, we will look at different fundamental types of robots in section 2.2: manipulators, mobile robots and mobile manipulators. Section 2.3 is about the basics of robot sensors. Processing their data in the sensing part is covered in section

2.4. The actuator basics are then covered in section 2.5. The chapter concludes with the basics of controlling the actuators, the acting part, in section 2.6.

2.1 Interacting with the Real World

Besides algorithms & data structures and programming languages the third major ingredient in software development are interfaces, also known as APIs.[15] APIs come in many flavors. They range from close-to-the-metal bit manipulation in hardware registers that turn on an LED to highly abstract REST interfaces that create entire cloud services. The same goes for robotics software. Our focus in this chapter will be on APIs that enable us to interact with robotics hardware components, such as sensors and actuators.

The generic inner structure of a sensor is shown in figure 2.1. The actuator's inner structure is depicted in figure 2.2.

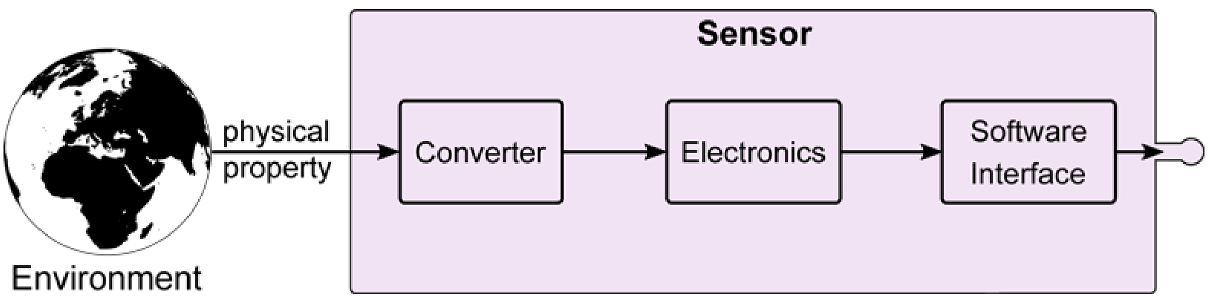


Figure 2.1 The inner structure of a generic sensor.

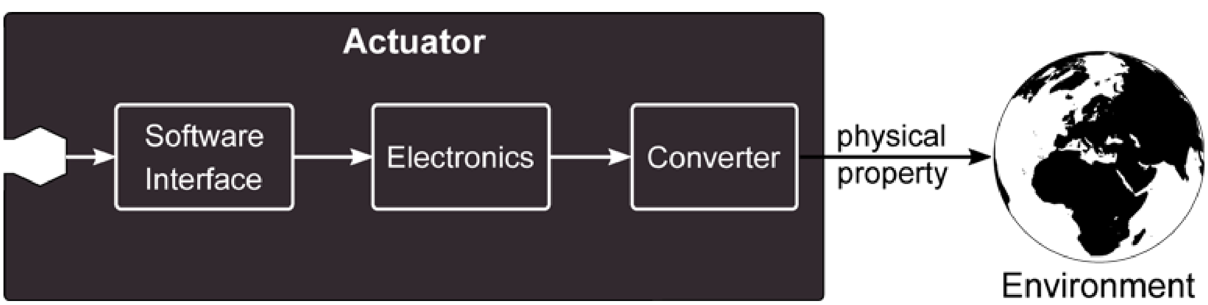


Figure 2.2 The inner structure of a generic actuator.

The chain within the sensor goes from physical property to the converter to electronics to software interface. In case of the actuator, the chain is basically reversed: software interface, electronics, converter, physical property. For complex sensors and actuators some or all of the stages can occur multiple times.

An actual hardware component can either be complete, meaning it contains all parts of the chain, or it can provide only some parts of it. Complete sensors and actuators are sometimes also called smart due to directly providing a software interface. If the sensor or actuator is incomplete, an additional hardware component that completes the sensor is required.

An example of an incomplete sensor is a thermistor, also known as NTC or PTC, which can be used to measure temperatures. In a suitable electric circuit the thermistor's output is a voltage that varies with the surrounding temperature, e.g. the lower the temperature the lower the voltage. For us this is an incomplete sensor as software cannot directly read voltages. Software can only read data. For this reason we need another component that converts voltages to data. An example of such a device is an analog-to-digital converter (ADC), which we will discuss in more detail in section 2.3. Taken together the thermistor and ADC make up a complete sensor because we have a software interface that provides us with data about physical properties in the environment. Figure 2.3 illustrates how the two hardware parts together constitute a complete sensor.

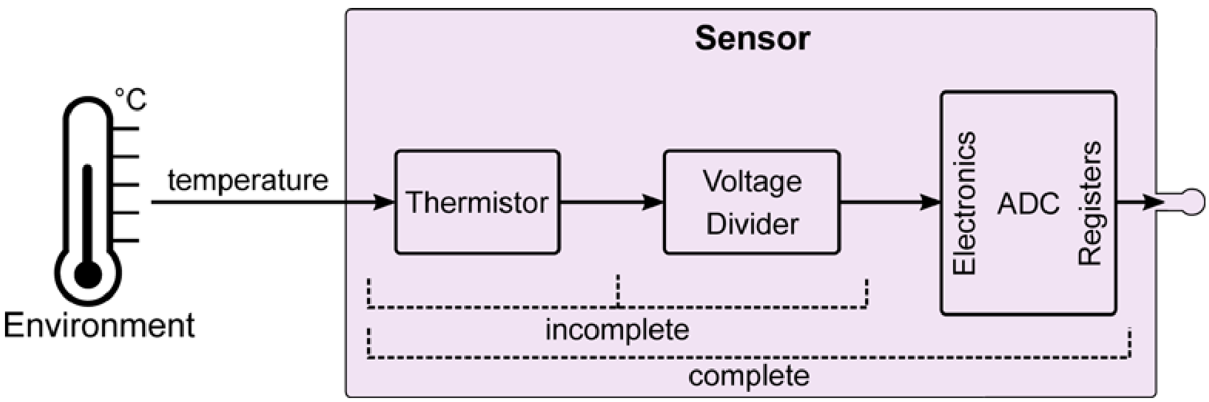


Figure 2.3 A thermistor and an ADC as example of an incomplete and a complete temperature sensor.

As we will see in section 2.5, the same distinction between incomplete and complete also applies to actuators.

Let's look at two possible APIs to a fictional temperature sensor.

API#1 (don't worry about any details here):

```
adc_to_volt = 3.3 * (1.0 / 1024)
volt_bias = 1.2
volt_to_temp = 180.0 / 2.0
temperature = ((get_adc_reg(0x42) * adc_to_volt) - volt_bias) * volt_to_temp
print(f'The temperature is {temperature:2.1f} °C')
```

API#2:

```
temperature = get_temperature()
print(f'The temperature is {temperature:2.1f} °C')
```

It is safe to assume that most developers will find API#2 easier to use than API#1. API#1 deals with reading ADC hardware registers and has to convert them to a temperature value. In contrast, API#2 directly provides the temperature value that we are interested in. The topic we are touching upon here is the API's level of abstraction or API layers.

For example, API#2 could be implemented in terms of API#1 by simply wrapping the conversion from ADC values to temperature into the `get_temperature()` function:

```
def get_temperature():
  ...
  temperature = (((get_adc_reg(0x42) * adc_to_volt) - volt_bias)
                 * volt_to_temp)
  return temperature
```

We will get back to the topic of API layers later in the next chapter when looking at the robotics software stack. The important point I want to get across here is that, just as with other types of software systems, each part of the robot software system can have different levels of abstractions. Writing software for robots does not necessarily mean working closely with the hardware interfaces. Robot software always utilizes hardware as we use sensor and actuator hardware to interact with the real world. Still, if you are not responsible for implementing the hardware drivers, the API you are using might be - it actually *should* be - on a much higher level. [16] Identifying and creating good layers of abstraction is key to an easy to develop and maintainable robot software system.

Having discussed robot APIs in general terms, we next look at different fundamental robot types and their (abstract) APIs.

2.2 Types of Robots

When getting into a new field, it is often useful to sort the large variety of things into a few broad categories. This is also true for robotics. The following three fundamental types of robots seem like clear and distinct categories to subdivide all robots at first. However, you will encounter edge cases and gray areas in your robotics journey. Nevertheless, distinguishing these types is very useful both now and later on.

2.2.1 Manipulators

The first fundamental type of robots are stationary manipulators. This category of robot includes the classic industrial robot arms that have shaped the commercial robotics industry since its beginning. Until recently they also made up the majority of robots in the world.[17] The two important properties required for a robot to fall into the manipulator category are:

- The robot's purpose is to manipulate objects, i.e. to purposefully move objects through space.
- The robot itself is stationary.

Figure 2.4 shows examples of different manipulators.

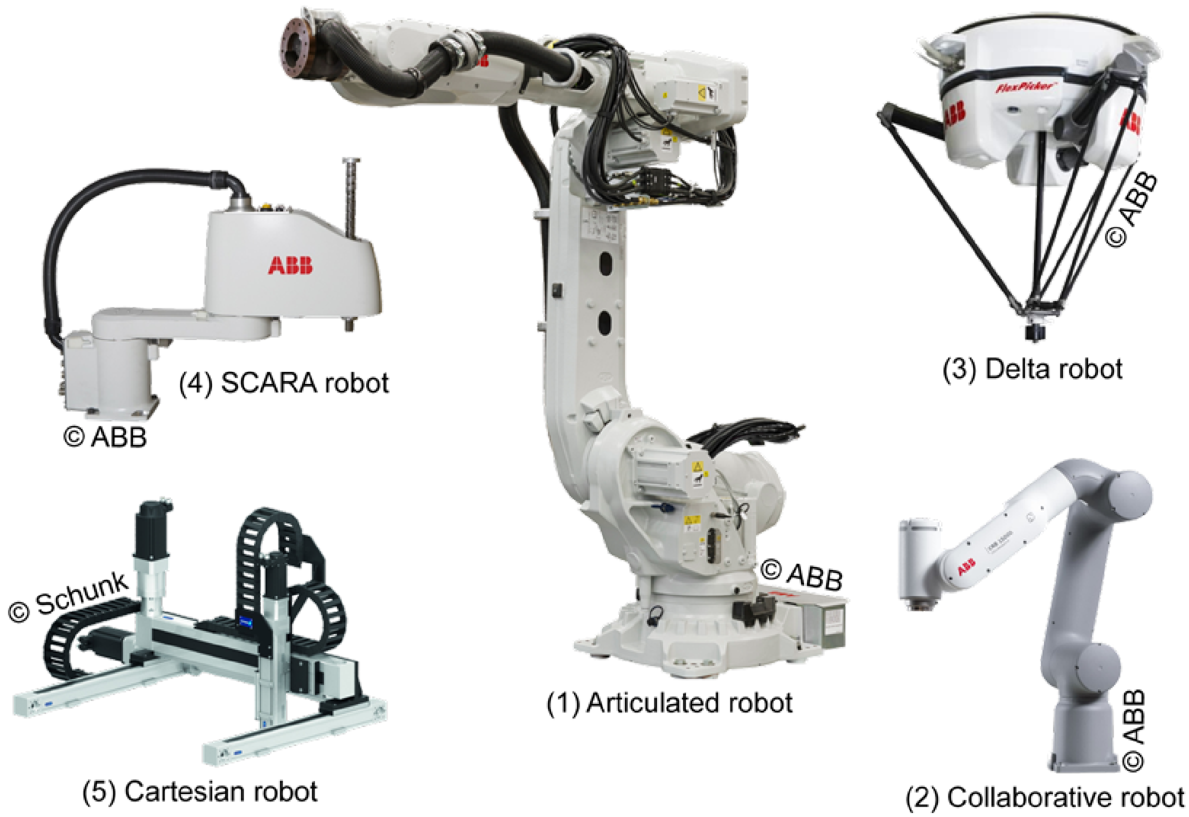


Figure 2.4 Examples of different robot manipulators.

Now that you know what manipulators are and what they look like, the important question is what we can do with them. As exemplified in figure 2.4, manipulators come in various shapes and sizes. Also the ways in which they can move varies widely, depending on their construction. [18] Nevertheless, the basic operation for each manipulator is to move its so called end effector to different positions. The end effector is either the last part of a robot manipulator, with the base of the robot being the first part, or the tool attached there. See figure 2.5 for an illustration.

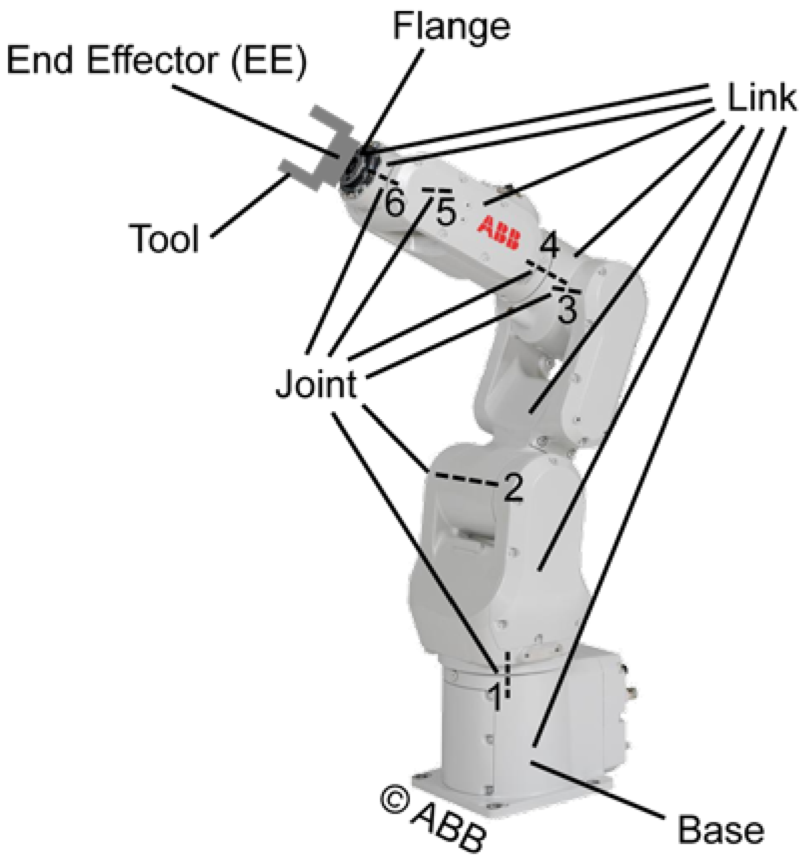


Figure 2.5 The different parts making up a manipulator.

Some additional robot terminology was introduced in this figure, let's look at each term in a bit more detail.

- Joint: The moveable parts in the robot hardware. The most common joint types are those that can rotate, like a hinge, and those that can extend linearly, like a drawer slide or telescopic pole. Joints contain the actuators that make the robot move.
- Link: A rigid mechanical part connecting the joints. The links give the manipulator its shape and make up the bulk of what is visible when looking at a robot.
- Robot Base: The first link of the robot. The base attaches the robot to the environment.
- End Effector (EE): The last link of the robot, i.e. the link farthest away when counting from the robot base.
- Tool Flange: A standardized mounting plate that allows to fasten tools to the last link of the robot. The tool flange is often simply referred to as flange.
- Tool: A device attached to the tool flange that enables the robot to perform the intended process. Another common term here is End Of Arm Tooling (EOAT). The most common robot tools are grippers. However, the variety of robot tools is as big as the applications performed by manipulators. Other examples of tools are welding equipment, tool changers, force-torque sensors, paint sprayers, material removal equipment and 3D printing nozzles.

The term end effector (EE) is unfortunately used ambiguously depending on whether we are talking about the bare manipulator or about one with tools attached. If no tool is attached, the EE is identical to the robot's last link that provides the tool flange. If a tool is attached, the tool or rather a specific location on the tool is *usually* considered the EE. Just remember to check what has been defined as the EE when working with robot systems. In this book, I'll follow the just stated definition and always additionally make the EE location explicit.

Now that we have learned some important robot terms, let's come back to the fact that the essential operation with a manipulator is moving the EE to different positions. Thus, the abstract API for manipulators is

```
robot.move(pose)
```

This one-liner deserves some explanation. Let's start with the `pose` argument. Why pose and not simply position? Because we do not only care about the position, or location, but also about the orientation. Looking at a 2D example, like the one shown in figure 2.6, will make this clear.

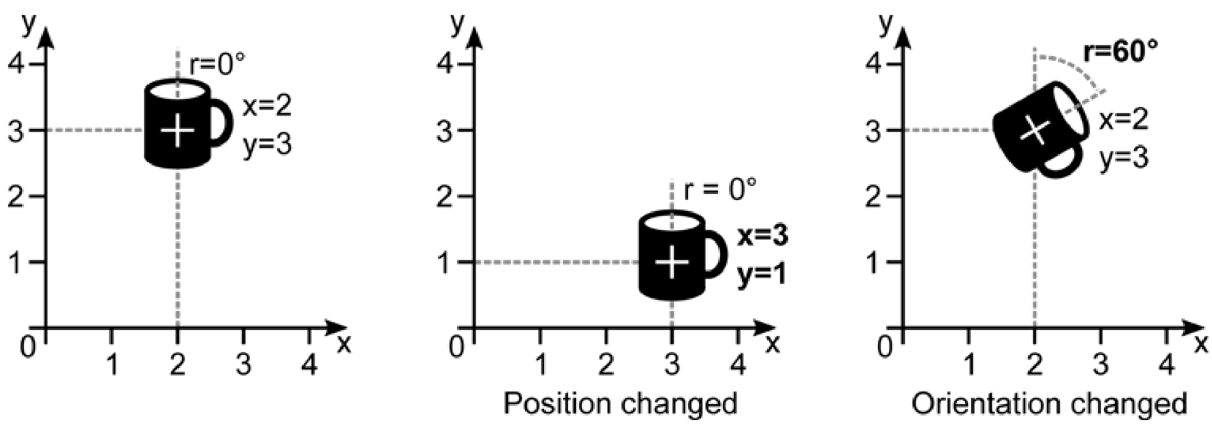


Figure 2.6 Example of position and orientation in 2D.

For objects that are not rotationally symmetric, such as a perfectly round object with no markings on it, not only the position but also the orientation is important. Take a cup of coffee for example. I, at least, care very much whether the open end of the cup is pointing upward or sideways when there is hot coffee inside. This holds true even if the position, e.g. next to my keyboard, remains unchanged. Therefore, in general, robotics works with position and orientation, i.e. poses. Since representing objects and their motion is a very important topic in robotics, we will cover this in much more detail in chapter 4.

After discussing the `pose` argument in `robot.move(pose)`, we can now shift our focus to the `move` method. While it sounds straightforward at first, moving the robot from its current pose to the target pose raises a number of questions. By that, I don't mean how we implement `move` to go from a specified pose to commanding the joint motors. This will be the question for chapter 5. The current questions are much more basic: Can the target pose be reached? Can it be reached without colliding with the environment and without the robot colliding with itself? How fast should the robot move? How quickly can it accelerate and slow down? What path should be followed in between the current pose and the target pose? All these questions are about specifying the intended motion behind move in a more detailed manner.

Disregarding reachability, obstacles, speed, acceleration and many other aspects, let's briefly discuss what happens in between current pose and target pose by the 2D example in figure 2.7.

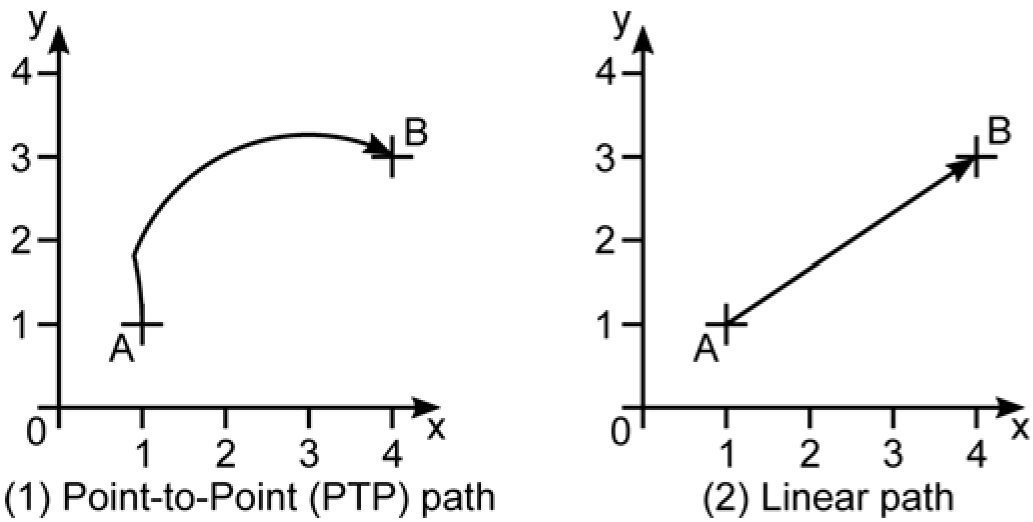


Figure 2.7 Example of moving from pose A to pose B along (1) a point-to-point (PTP) path and (2) a linear path.

Looking at path (1) you might ask yourself, why anyone would go from A to B in this manner. It seems obvious that not only is path (2) the more natural one, but also the shortest and thus best path from A to B. This is true in some sense, but actually wrong when taking a different perspective, the robot's perspective. From the robot actuators' point of view, path (1) is more natural, shorter and also easier to follow.

Let's assume we have a function that calculates the joint positions for a given pose. You will learn how to create such functions in chapter 5 and will also understand why we give this function the name `inverse_kinematics` in the following example. The `move_ptp` function takes a `pose` argument, calculates the joint positions for this pose and then sequentially[19] moves each joint from its current position to the target position.

```
def move_ptp(self, pose):
  joint_positions = inverse_kinematics(self.robot_model, pose)
  for i in range(len(self.joints)):
    self.joints[i].move(joint_positions[i])
```

The resulting motion is shown in figure 2.8.

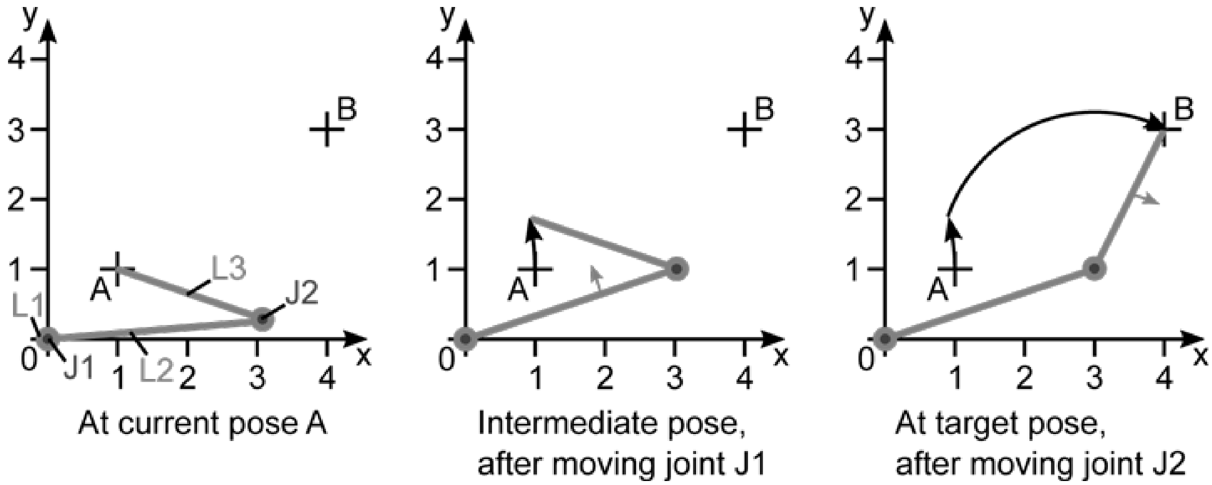


Figure 2.8 Motion along the point-to-point (PTP) path (1) from figure 2.7. The two dimensional robot consists of two joints (J1, J2) and three links (L1, L2, L3). The robot is fixed to the environment via its base link (L1) at position (0,0).

To a large extent, robots are motion generating machines. A large portion of this book will hence be dedicated to motion related topics. At this point, it is sufficient to know that `move` is as simple as it looks, but that there is a lot of decisions and parametrization to be done in the background to have it function as intended.

Let's wrap up this section with a function that moves the robot's end effector in a rectangle pattern. If we attach a pen as tool to our manipulator flange and set everything else up just right, the following snippet would draw a rectangle of size 2x3.

```
def rectangle_motion(width, height):
  for pose in [Pose(x=0, y=0), Pose(width,0), Pose(width,height),
               Pose(0,height), Pose(0,0)]:
    robot.move_linear(pose)

rectangle_motion(2, 3)
```

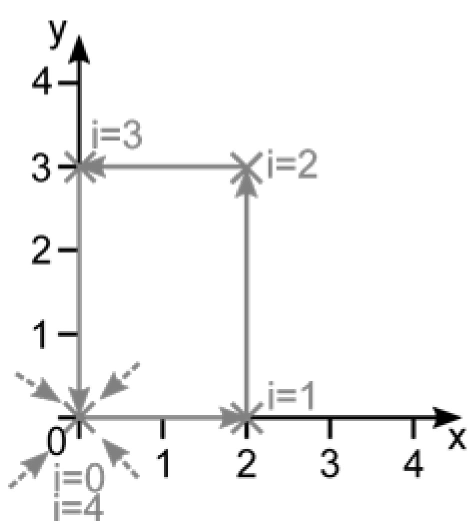


Figure 2.9 Moving along a rectangular pattern of size `width` times `height`. As the initial pose is not known, the `i=0` motion to pose (0,0) is unknown apart from the commanded endpoint.

I hope this small example gives you confidence that working with robots is not difficult for Software Engineers. Yes, there are still many details we need to look into before you can program a robot manipulator to make pancakes in your kitchen. But these details can be addressed one step at a time, as we are going to do. Before diving deeper into manipulators, let's have a look at mobile robots first.[20]

2.2.2 Mobile Robots

Mobile robots move *themselves* through the environment. As discussed in the previous section, this contrasts with robot manipulators that move *objects* around, while the manipulator itself is stationary as it fixed to the environment.

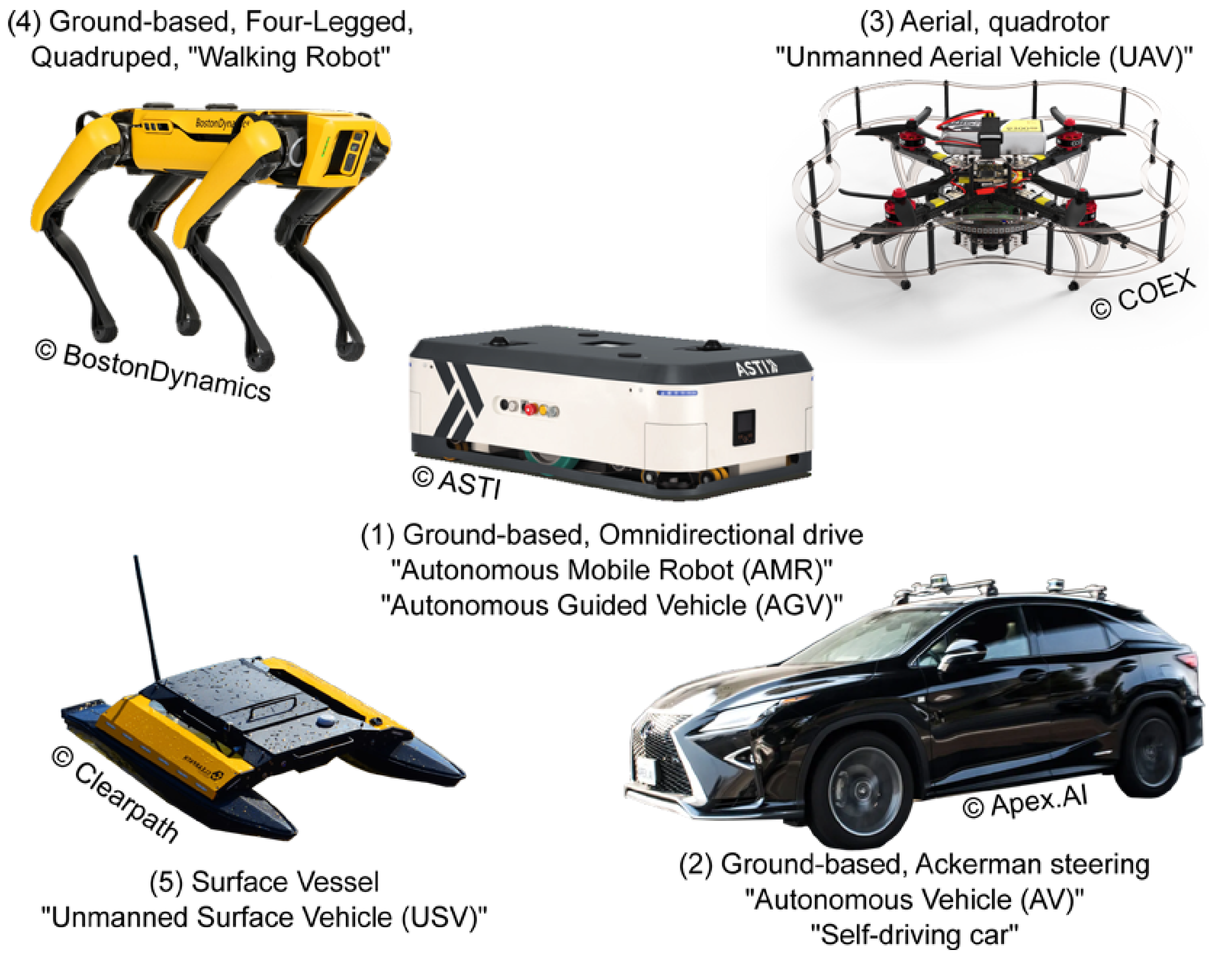


Figure 2.10 Examples of different mobile robots.

In some cases having the robot traverse the environment is already the robot's objective. In other cases the objective is to move something other than the robot along with it, i.e. the robot carries a payload. Let's start with this second case.

Moving objects from A to B using some kind of carrier device is a very common task. Once you start thinking about all the tasks that fall into this category, so many examples come to mind that it is hard to select a few instances from this abundance of cases. Nearly all types of traffic at land, sea, air and space is about transporting discrete objects[21] from A to B, including transporting humans. As it is still challenging to operate mobile robots fully autonomously in outdoor environments, the majority of object transporting mobile robots work in indoor environments today. Most of these indoor transportation robots can be found in logistics and manufacturing. The enormous gap between all transportation tasks and the tiny segment that mobile robots are actually deployed in shows the untapped potential of this rapidly growing industry.[22]

Let's move on to the other type of mobile robot mentioned, the ones not intended to carry payloads, but to simply move themselves around. The value of having a robot transport objects is quite obvious. After all, who of us wouldn't sometimes enjoy getting coffee delivered to our desks instead of having to walk to the coffee maker ourselves. The purpose of a robot just moving around without transporting any payload is less obvious outside entertainment purposes. [23] We need to look at all the components that are part of the robot system and what value they could bring *beyond* allowing the robot to move through the environment.

A component, such as a sensor or actuator, is considered part of the robot if it is connected to the sensing or acting part of the robot. Otherwise the component is considered to be a payload. Take the example of a camera drone. If the camera images are only streamed to a display for the person piloting the drone, then the camera sensor is a payload and not part of the robot system. On the other hand, if the camera images are (also) processed by the drone's sensing part, e.g. for automatic collision avoidance, then the sensor is part of the robot system. Have a look at the difference in connection between the camera sensor and the sensing part in figure 2.11.

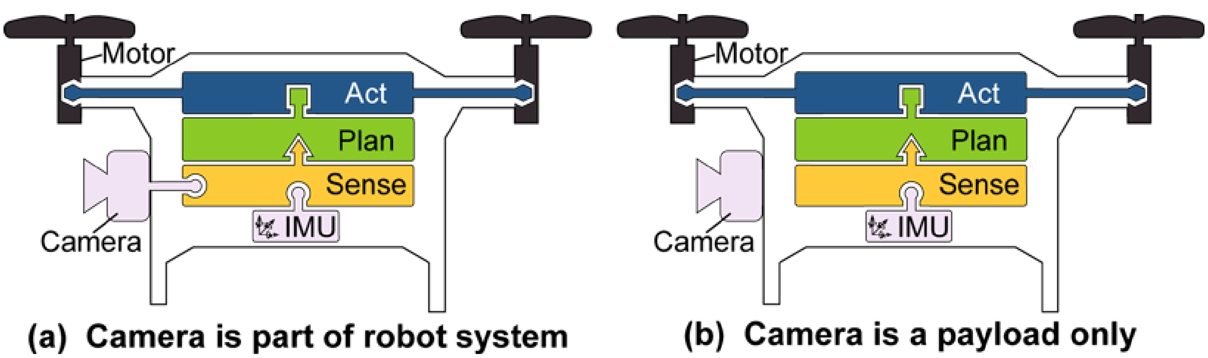


Figure 2.11 Distinction between a sensor that (a) is part of the robot system and (b) is only a payload.

The value of carrying around payloads has already been discussed above. Given the distinction between payloads and robot system parts that we just defined, it becomes easier to see the value that parts of the robot can provide, beyond allowing the robot to function. We continue with the example of an autonomous camera drone that is able to explore an environment without requiring a human pilot.[24] While the value of having the drone just fly around might be low, being able to look at a 3D model of the explored environment or even just the recorded camera images after the flight can be of high value. Think of surveying a construction side, inspecting an oil pipeline or supporting recovery management after a natural disaster. There are many examples where the data available from the sensors required for the robot's operation provides additional information value. But there are also examples for the actuator side.

One such example where a mobile robot moving through the environment without an additional payload already results in doing something useful is an autonomous steamroller. The objective of a steamroller is to flatten a surface by driving across it. Our steamroller robot uses its special wheels, the rolls, as actuators to move around. Thus, they are an integral part of the robot system and not an additional payload. Although these examples exist, as just demonstrated, it is interesting to note that most mobile robots that serve a practical purpose either carry some kind of additional payload or they are of the information gathering type.[25]

Now that we have a basic understanding of the purpose of mobile robots, let's have a look at the abstract API for them:

```
robot.navigate(pose)
```

This looks very similar to the manipulator abstract API `robot.move(pose)` from the previous section. Indeed, there is a lot of common ground, but also significant differences. Let's start with what is not just similar, but even identical, the `pose` argument. As we discussed in the previous section, a pose specifies both position and orientation.[26] The most important difference between the pose argument to manipulator `move` and to mobile robot `navigate` is the range of valid poses. A manipulator is stationary and can thus only reach poses in a limited volume around its fixed base, the so called robot workspace. All poses outside are invalid as they cannot be reached. A mobile robot on the other hand can in principle reach all poses in the accessible environment, no matter how far away from the current position.

Figure 2.12 shows a 2D navigation task example.

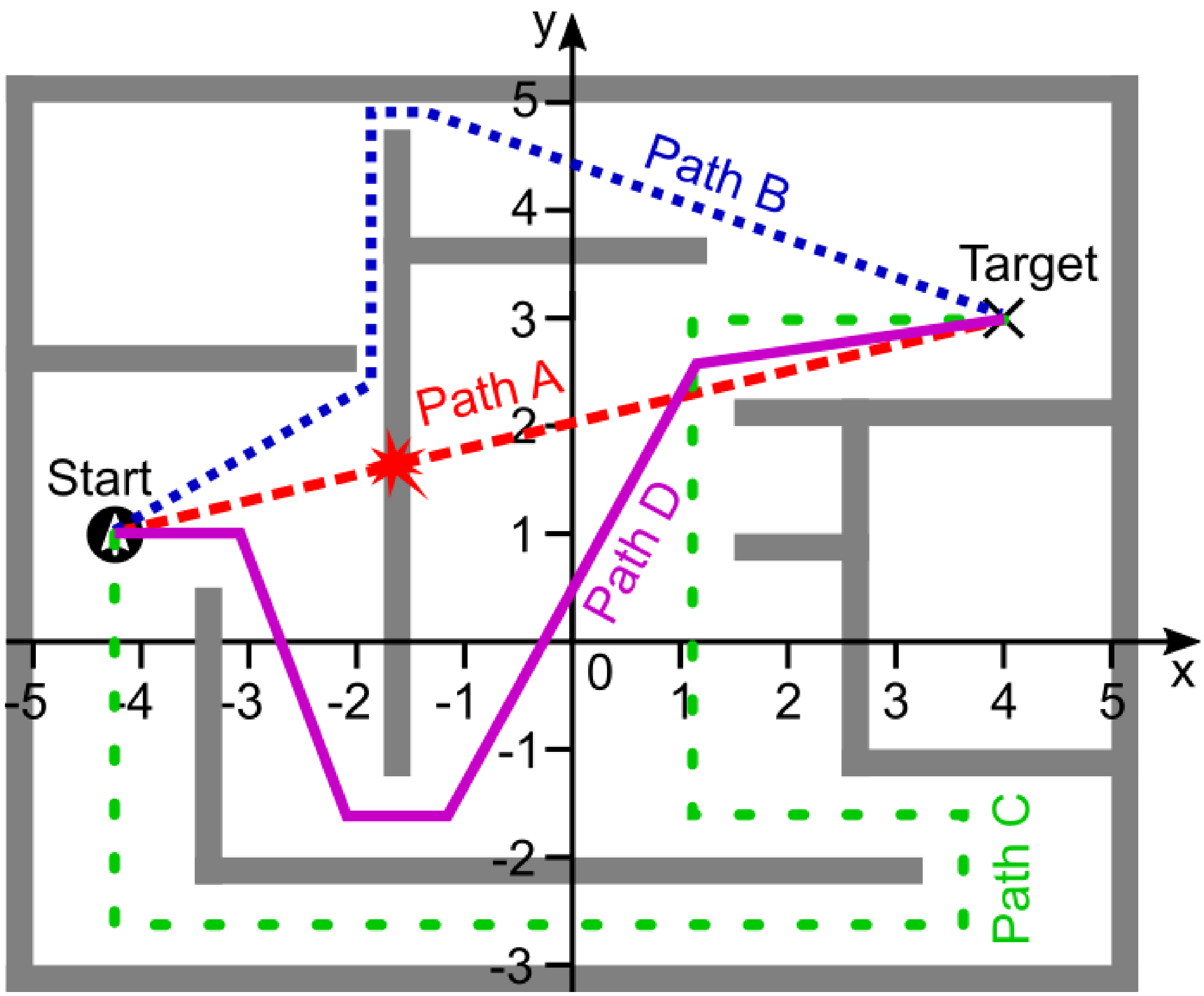


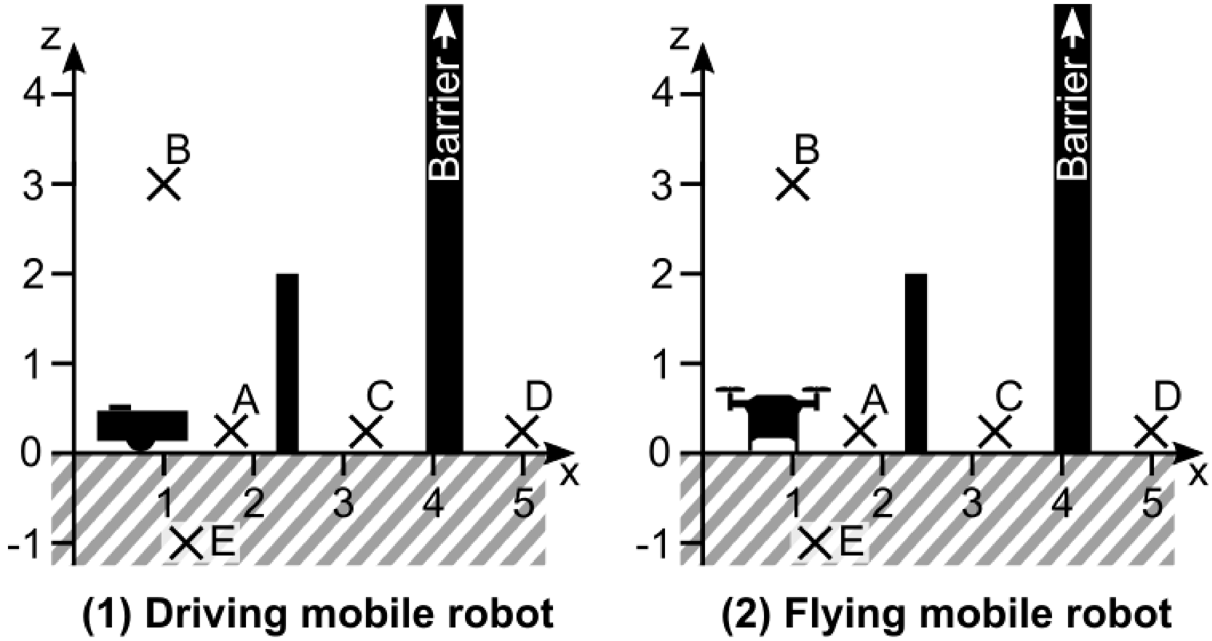
Figure 2.12 Mobile robot navigation example. The round mobile robot is located at the Start position and is commanded to navigate to Target position. Different impossible paths (A and B) and feasible ones (C and D) are shown.

In the example, path A is the shortest possible path, but it would result in a collision with the obstacle at x -1.75. Although path B does not go through an obstacle, it would still result in a collision at the chokepoints due to the size of the mobile robot. Path C is a valid path, but it is far from optimal considering the path length. Finally, path D is both collision free and (close to) the optimal path.

After this navigation example, we should lay out some important terminology. Let us disregard practical considerations for now, such as limited battery runtime and traffic regulations, and consider an abstract notion of reachability. It helps to first distinguish between a target pose being (in)valid and a pose being (un)reachable.

- A target pose is *reachable* if there is a collision-free path between the robot's current pose and the target pose.
- A target pose is *valid* if it would be reachable in an environment without any obstacles.

As these definitions are a bit abstract, let's look at a concrete example in figure 2.13.



pose	valid	reachable
A	yes	yes
B	no	n/a
C	yes	no
D	yes	no
E	no	n/a

pose	valid	reachable
A	yes	yes
B	yes	yes
C	yes	yes
D	yes	no
E	no	n/a

Figure 2.13 An illustration of (in)valid and (un)reachable poses for mobile robots.

For a ground-based driving mobile robot valid poses require the robot’s wheels to touch a supporting surface. For a flying mobile robot all poses above ground (and up to a maximum height) are valid. In figure 2.13(a) our driving robot is unable to reach the valid pose C due to an obstacle between its current pose and pose C. The same holds for pose D. Our flying robot (b) can reach pose C, but it can not reach pose D either.

It is important to consider another difference between manipulators' `move(pose)` and mobile robots' `navigate(pose)`: What moves to the target pose? In case of a manipulator, the robot end effector is moved to the target pose, while the manipulator itself remains at a fixed position in the environment. In case of a mobile robot, the entire robot moves to the target pose.

In the small 2D world of figure 2.13 it is easy to see which poses are reachable. It is also relatively easy to have our planning part find a path in such a simple environment as we will learn later on in chapter 10. In real world applications, the task of finding a path and navigating the mobile robot along it is the core robot capability provided by `navigate`. Next we take a look at the result of combining a mobile robot with a manipulator, namely mobile manipulators.

2.2.3 Mobile Manipulators

Since we have already discussed robot manipulators and mobile robots in the previous sections, it is straight forward to define mobile manipulators: A mobile manipulator is a manipulator mounted on a mobile robot. The assumption here of course is that the combined robot is still mobile and that the manipulator can still function.

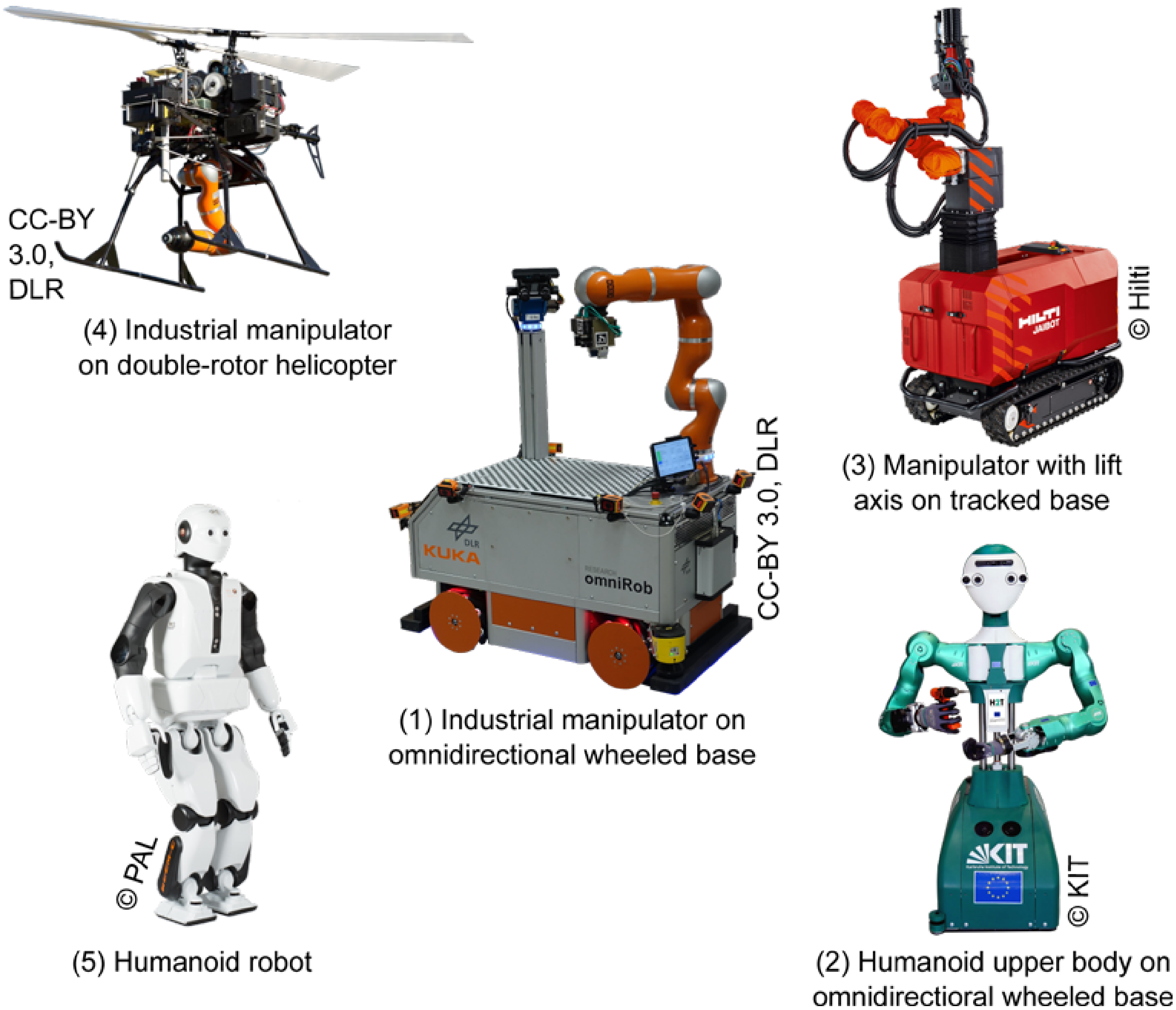


Figure 2.14 Examples of mobile manipulators. TODO copyright (Hilti)

There are different motivations behind having mobile manipulators in terms of new capabilities.

1. Using a manipulator in different places. The mobile aspect removes the stationary property from the manipulator by transporting it. The focus is on manipulation.
2. Enhancing the mobile robots payload transportation flexibility. The manipulator aspects allows loading and unloading of payloads. The focus is on transportation.
3. Extending the manipulator workspace to infinity. The mobile and manipulation aspect are intimately linked by simultaneously using the motion capabilities of the manipulator with those of the mobile base. This is also known as whole-body control.

The equivalent functionality to the case 1 could be achieved by having a separate mobile robot and manipulator. The mobile robot lifts up the manipulator as a payload, transports it to another location and sets it down at a new location. During transportation the manipulator is a pure payload - it does not even have to be powered.

For case 2, we can also think of a solution that does not involve mobile manipulation, but separate mobile robots and manipulators. We would put a stationary manipulator at the loading and the unloading site instead of mounting one to the mobile robot. Thereby achieving the same capability. Given there are only a few pre-defined loading and unloading sites, this is a practical alternative. However, if there are either many sites or their exact location is unknown or changes over time, this would not be a real alternative.

Finally, in case 3 we cannot separate the mobile robot and manipulator and still achieve the same functionality from a practical point of view. The mobile base basically acts as a set of additional manipulator joints. What makes it impractical to replicate these joints with a fixed base is that they would have to be of infinite size - or at least unreasonable large size.

Closely related to the three cases just discussed are the following two basic approaches for integrating the mobile and manipulation functionality.

- Independent successive
- Coupled simultaneous

Independent successive means that we perform a sequence of manipulator `move()` and mobile robot `navigate()` commands. In other words, at any point in time we *either* move the mobile base *or* the manipulator. For this approach we do not need any new abstract API. We only need to use the manipulator and mobile robot API already known from the previous two sections. The various aspects that need to be taken into account to make this combination work in real applications will be covered in the next chapters. These include coordinate systems, calibration routines and essential sensing functions. We will come back to mobile manipulation afterwards.

For the coupled simultaneous approach, we need a way to control the mobile base *and* the manipulator *together* in a unified manner. As this is quite an advanced topic and also an active area of robotics research, we postpone further details until after we have discussed simpler forms of mobile manipulation. Although I can imagine that you might be interested to dive right into the most advanced and also potentially most capable type of mobile manipulation, my advice is to be patient and follow along with the book in first learning all the great things you can do with manipulators, mobile robots and less demanding ways of integrating them. If it does not comfort you to know that advanced mobile manipulators are a tiny niche application today, then perhaps the following does: Just imagine all the things humans can do when moving their feet and arms successively instead of simultaneously. Although playing tennis or baseball does not fall in that category, countless other activities actually are included. Hence we will first focus on these before returning to advanced mobile manipulation.

After we have now encountered the three basic types of robots, our focus shifts to the parts making up all robot types. In the following section we will start with the basics of sensors and sensing functions.

2.3 Robot Sensor Basics

Let's briefly repeat what we learned in the previous chapter: Sensors are a robot's input devices. They measure various physical quantities in the environment and make these measurements available as data to the sensing part of the robot software system. The purpose of the sensing part is to give meaning to the raw data received from the sensors.

We will do a deep dive into different types of sensors and sensing in chapter 12. The goal of this section is to gain an initial understanding of these parts from a software point of view.

2.3.1 Digital Electric Inputs

The simplest sensor is a binary electric input, also known as digital input. This type of sensor only distinguishes between two states that correspond either to high and low voltages or current flowing and no current flowing. Independently of the specific electrical implementation, e.g. voltage and current thresholds, the API always looks the same:

```
input.status()
```

with `status()` returning a Boolean value, i.e. `0` and `1` or `False` and `True`. This is enough information for surprisingly many needs in real world robot applications.

The meaning of this Boolean raw data varies widely with the mechanism that creates the binary input. In the most simple case it could be connected to a physical switch or button. The switch could be an input device for giving commands to the robot. Frequently the switch is used to indicate the presence or absence of an object. For example, simple mobile robots often use bumper switches to detect that they have hit an obstacle.

In industrial automation light barriers and proximity sensors are frequently used to detect presence/absence of objects without physical contact. Another common use in industrial automation is to coordinate the actions of different equipment, e.g. machines and robots. A digital input of one machine is connected to a digital output[27] of another machine in order to signal states such as ready for next cycle. In a machine tending application, a robot might insert raw material into a machine, then signal the machine that material is available by setting the digital electric signal to high. The machine will then perform its operation. Once finished, the machine in turn signals the robot, that the next part should be inserted. This coordination can be achieved with merely one digital input and one digital output on each device.

Let's turn our attention to figure 2.15, which visualizes different types of sensors that all provide a binary signal to the robot system.

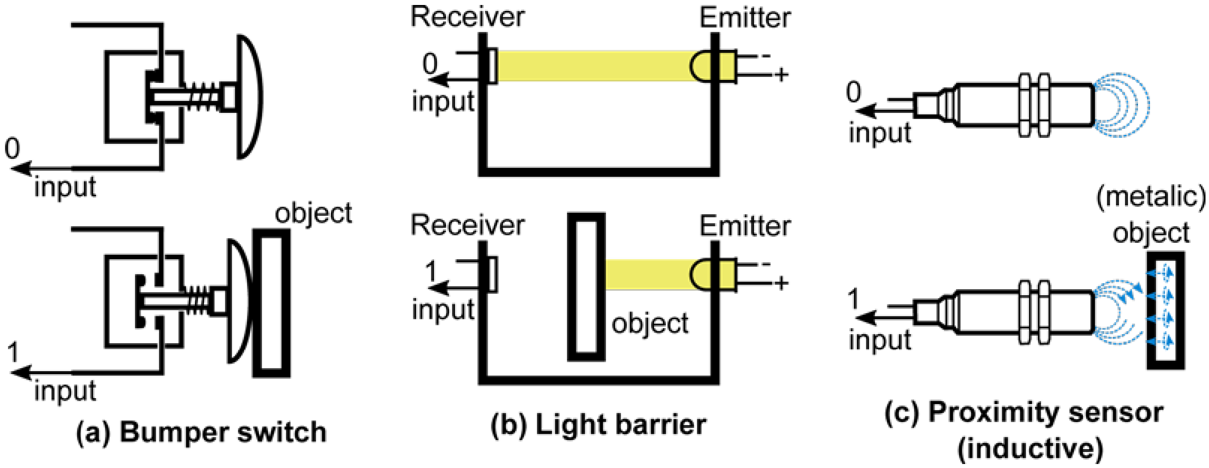


Figure 2.15 Three examples of sensors with binary interfaces.

Assuming we have a mobile robot with a bumper switch attached to input number 3, the following code would drive the robot forward until an obstacle is hit and then stop it.

```
no_collision = False
bumper_input = 3
while robot.inputs[bumper_input].status() == no_collision:
  robot.drive.forward()
robot.drive.stop()
```

As you can see, the meaning of the input status requires knowing what kind of sensor is attached to which input and how states map to Boolean values. We will come back to this point in the sensing basics section below.

2.3.2 Analog Electric Inputs / ADCs

Another type of electrical input are digitized analog signals. While a binary input can only differentiate between two states and provides a Boolean value, an analog input, also known as analog-to-digital converter (ADC), provides a floating point or integral value. This numeric ADC value represents the magnitude of the electric signal, usually its voltage, with a certain finite resolution. For example, a 8-bit ADC with a working range of 0 V to 5 V, can distinguish between 2 = 256 voltage levels. If the output type is an integer, 0 V would result in a numeric value of 0 and 5 V would result in the number 255. For a floating output between 0.0 and 1.0, 0 V would be represented as 0.0 and 5 V as 1.0. Thus, depending on the interface, 2 V would be presented as 2 V / 5 V * 256 = 102 or respectively 2 V / 5 V = 0.398. It is worth highlighting that due to the limited resolution of 256 possible values, all values between 1.99 V and 2.01 V are mapped to the same value of 102 or 0.398 respectively.[28]

The abstract API for such an analog input, ADC input, is

```
input.value()
```

In this book the return value is assumed to be a floating point number that is normalized to the range 0.0 to 1.0. If not specified explicitly, we assume the ADC to have a resolution of somewhere between 8 bit (256 different values) and 16 bit (65536 different values).

Apart from robot applications where measuring voltage or current are directly relevant, there are a number of common sensors that provide an analog electric output signal and can thus be connected to an analog input. Examples are temperature sensors, distance sensors and light sensors. Figure 2.16 illustrates the relation between environment properties and the analog input value.

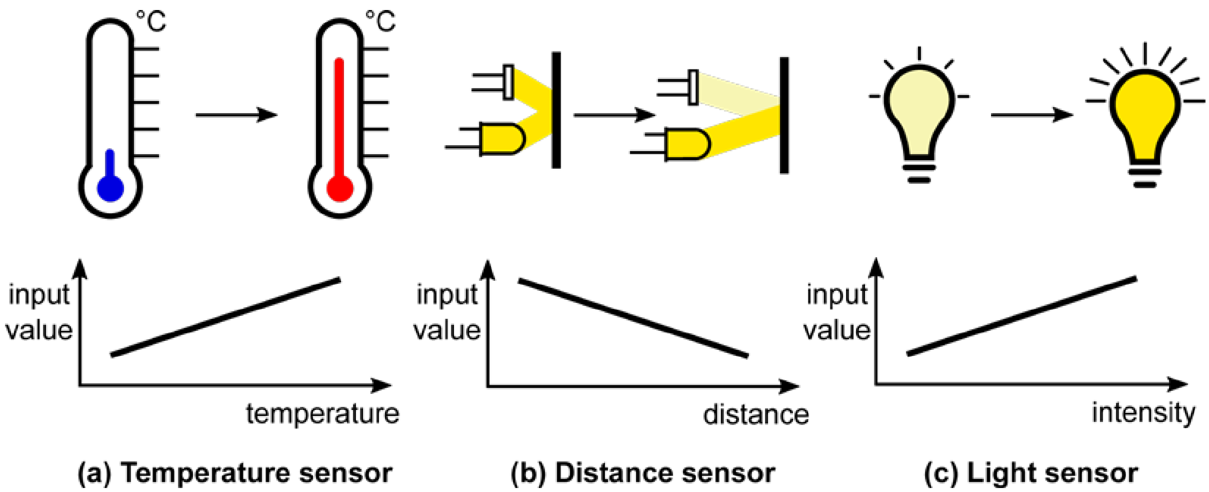


Figure 2.16 Three examples of sensors with an ADC interfaces, also known as analog inputs.

Unfortunately, real sensors often don't have such a nice linear relationship between measured property and their output voltage. We will cover methods to compensate for this kind of nonlinearity in 2.4.

Let's put our knowledge about analog inputs to use. We are programming a mobile robot to follow a light source, e.g. a flashlight, using two light sensors mounted to the robot. The robot is shown in figure 2.17

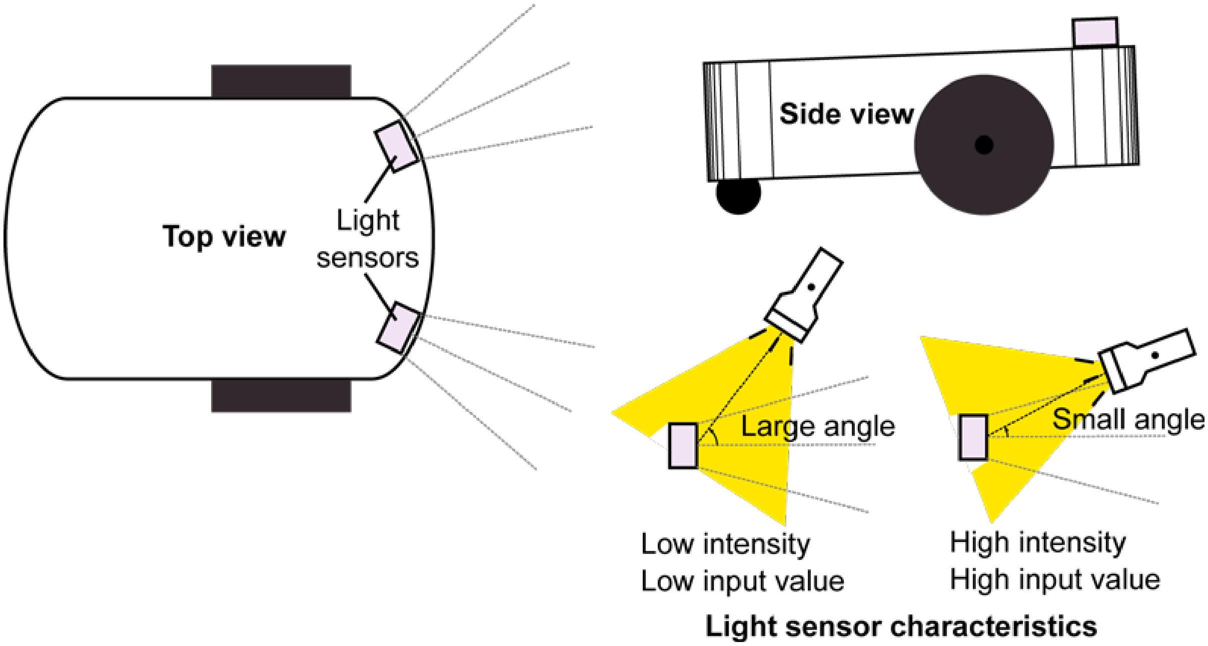


Figure 2.17 A mobile robot with two light sensors providing their data as analog inputs.

The main loop of our light following robot could look like this:[29]

- Read the ADC value from left and right light sensor.
- The sensor pointed more directly towards the light source will receive more illumination and will therefore have a higher ADC value.
- Subtract the two values from each other to provide a positive or negative number in case the light source is to the right of the robots current heading or the left respectively.
- To avoid turning from one side to the other due to minor differences in the sensor readings, a threshold is introduced that is tuned to ignore insignificant differences.
- If the difference value is higher than this threshold, turn the robots towards the direction of the light source. Otherwise, drive straight ahead.

```
left_sensor = 0
right_sensor = 1
threshold = 0.1
while True:
  light_difference = (robot.adc[left_sensor].value()
                      - robot.adc[right_sensor].value())
  if light_difference < -threshold:
    robot.drive.turn_right()
  elif light_difference > threshold:
    robot.drive.turn_left()
  else:
    robot.drive.forward()
```

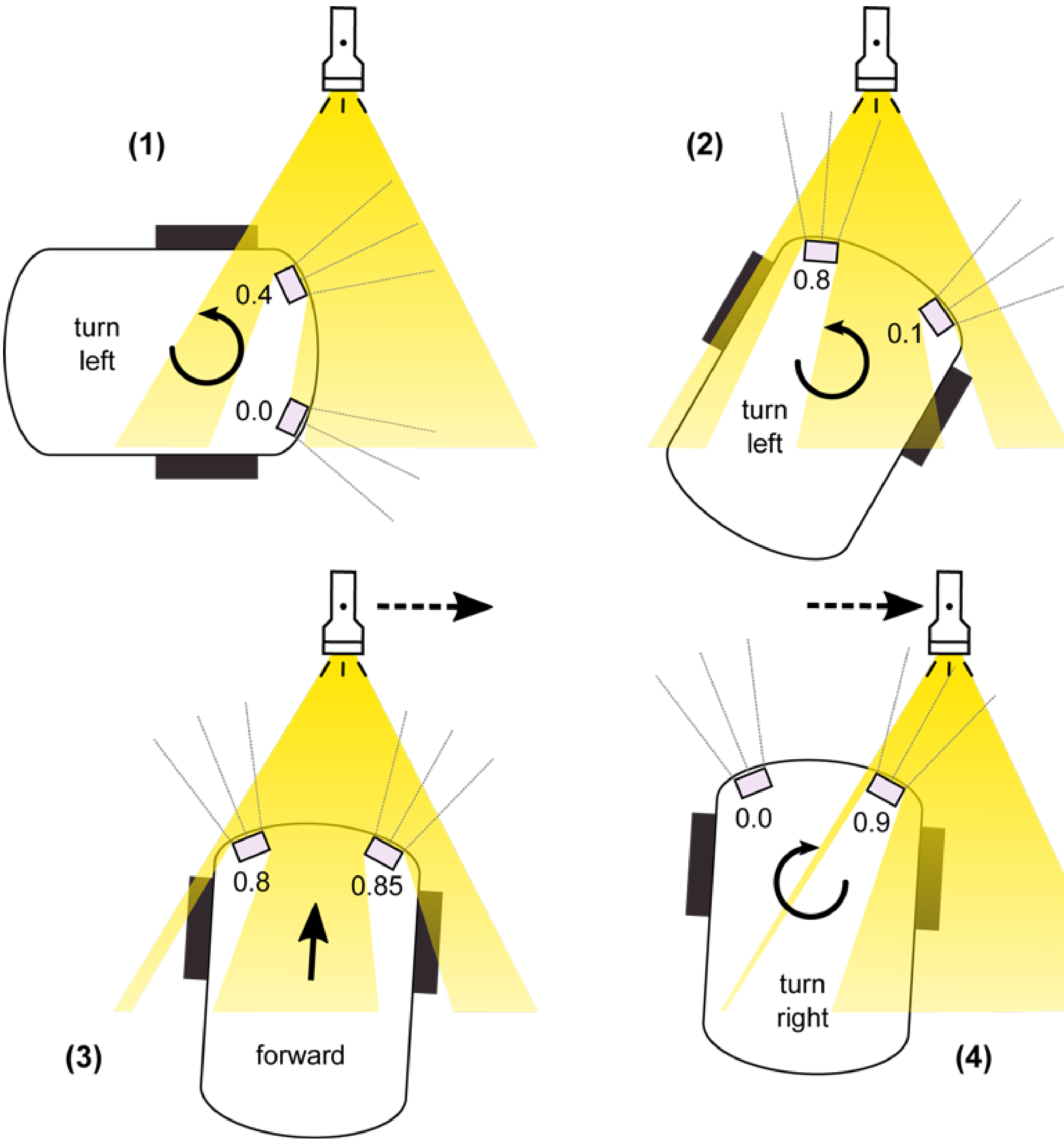


Figure 2.18 Four iterations of the light following robot's main loop.

As with the binary inputs example above, we should not work with raw sensor input in a properly designed robot system. Instead we leave it to the sensing part to deal with sensor hardware specific details and work at a higher and more meaningful level of abstraction in the planning part. A sensible abstraction will be presented in section 2.4. Next we turn our attention to a sensor type that is very important for robots: position sensors.

2.3.3 Position Sensors

Position sensors provide data about the absolute or relative position between two moving parts. Position sensors are also commonly referred to as encoders. In the context of robotics they are most frequently used to measure the position of the robots joint. Because they measure properties in the robot, as contrasted to measuring properties in the environment, they are proprioceptive sensors. Sensors that measure environment properties are called exteroceptive sensors. The distinction between proprioceptive and exteroceptive is made based on the purpose of the sensor, not the sensor type. For example, a temperature sensor can be used to measure both the ambient temperature of the environment (thus exteroceptive) and the temperature of the robot's processor (thus proprioceptive).

Coming back to position sensors, or encoders, we distinguish different types according to three criteria

- Angular/Rotary versus Linear
- Absolute versus Incremental: Absolute encoders directly provide the position value. Incremental encoders only provide the information that a certain incremental position change has occurred.
- For rotary encoders only: Single-turn vs Multi-turn: Single-turn encoders wrap around after a full rotation (360°). Thus, they provide the same value for position 10° and 370° (10° + 360° = 370°). Multi-turn encoders also track the number of full rotations.

We will focus on angular position sensors, or rotary encoders, in this introductory section, as they are the more common type in robots. The reason being that the majority of robot joints are actuated by electrical motors that generate a rotary motion. Even if the rotary motion is translated into linear motion, e.g. using lead screws, it is often still preferable to measure the rotary motion and calculate the linear displacement from it.

Incremental encoders can provide the same information as absolute encoders under the assumption that their data is continuously processed. It is sufficient to know the absolute start position of the incremental encoder and then keep adding the increments (with the correct sign) to the start position in order to get the absolute current position. The same is true for single-turn encoders, where the external processing needs to keep track of full rotations when a wrap around occurs. However, deriving the absolute / multi-turn position in this way has two limitations: It must be ensured that the data processing is never interrupted and that no motion occurs when the device is switched off.

Many different realizations using various physical phenomena have been conceived for each type of encoder. Because we care much more about sensor interfaces, i.e. APIs, than about the electromechanical details of sensor implementation, we content ourselves with two common realizations illustrated in figure 2.19.

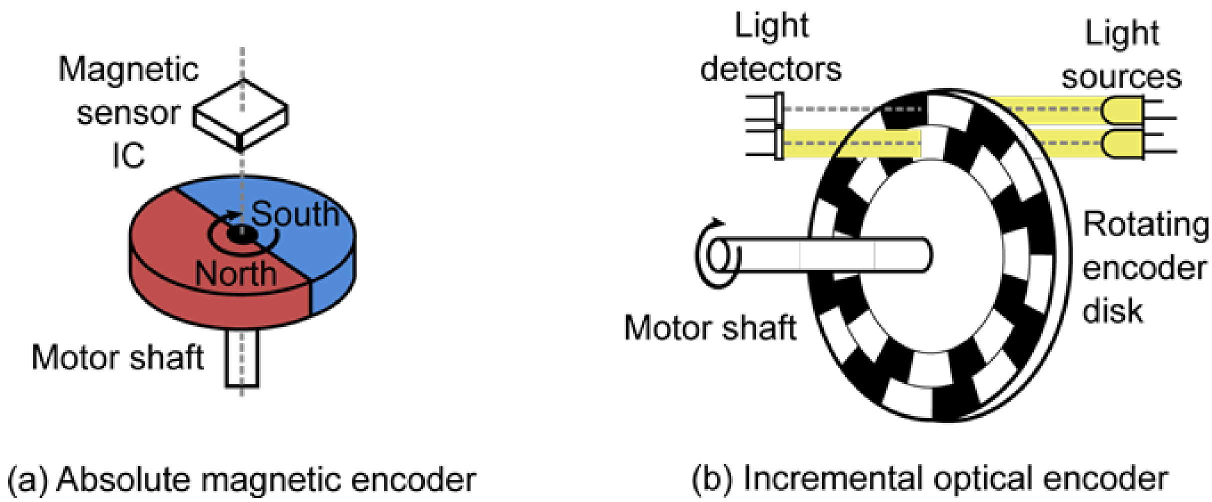


Figure 2.19 Two rotary encoders, one absolute and one incremental, using different measurement principles.

The abstract API for such a position sensor is

```
encoder.position()
```

The return value is the absolute position. The unit is degrees/radian for rotary encoders and meters for linear ones. We do not consider incremental encoders in our abstract API, as we assume they have been upgraded to absolute encoders using the methods described above.

After discussing this important type of proprioceptive sensor, we shift our focus outside the robot and look - quite literally - into the environment.

2.3.4 Cameras

The final type of sensor we want to learn about in this sensor basics overview are cameras. A camera sensor can be seen as a grid of many light sensors together with the necessary optics. Section 12.2.4 will discuss camera sensors in depth. For now it is enough to consider a simple very low resolution grayscale camera as shown in figure 2.20.

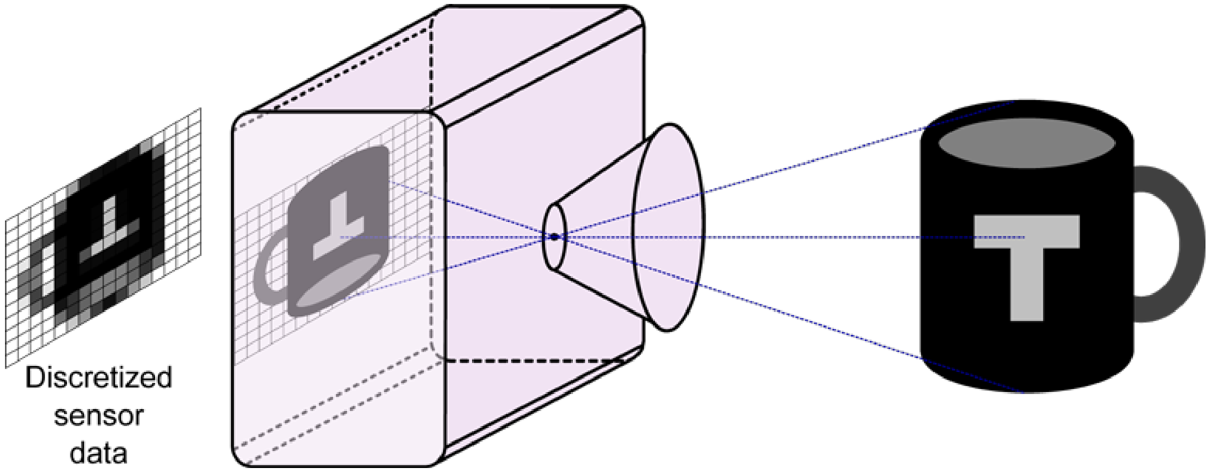


Figure 2.20 A simple model of a low resolution (16x12 pixels) grayscale camera sensor.

The visual appearance of a section of the environment is mapped to an image. The image has a discrete spatial resolution and a discrete intensity resolution. Bright areas result in high intensity values of the grid's sensor elements. Dark areas result in low intensity values. The individual elements in the camera's sensor grid are referred to as pixels, short for picture elements. Our camera in figure 2.20 has a grid of 16 pixels in width and 12 pixels in height. Thus, the resolution of this camera sensor is 16x12 pixels or 192 pixels. Real camera sensors have a resolution in the megapixel range, meaning millions of pixels. Yet, the principle is the same.

Apart from resolution, another important property of cameras is their frame rate. The word frame here is used synonymous to image. Thus, the frame rate is the number of complete images the sensor provides each second. Frames per second (FPS) is the corresponding unit.[30] A common frame rate of camera sensors is 30 FPS, which is in the same range as what human eyes can see. [31] Depending on application demands, lower frame rates might be sufficient or much higher ones might be required.

Given the resolution and frame rate of real camera sensors, they must be connected through computer interfaces with sufficient bandwidth. The most common interfaces are USB and Ethernet together with the Camera Serial Interface (CSI) in embedded systems. Disregarding configuration of the camera sensors, the abstract API to camera sensors is

```
camera.image()
```

The return type of `image()` for a grayscale `camera` is a 2D array of intensity values. For color cameras the array has an additional dimension for colors, usually of size three, giving one 2D array for red, one for green and one for blue. Hence digital color images are often referred to as RGB images due to being made up of data for red, green and blue. Similar to our discussion on analog inputs / ADCs above, each intensity can either be represented as an integral number or as a (normalized) floating point number. The granularity of intensity values, i.e. how many intensity values can be distinguished for each pixel, is referred to as color depth. Although having pixel values normalized to 0.0 to 1.0 independently of color depth makes things easier when dealing with different cameras, similar to what we discussed about ADCs, in practice it is more common to work directly with the integral values. The reason is that the majority of camera sensors used in robotics today have a color depth of 8 bits. With 8 bit color depth 256 intensity values can be distinguished and hence each pixel has a value between 0 and 255.

Given a grayscale camera, the following code snippet would print out the values of the camera image from fig 2.20.

```
image = camera.image()
for v in range(len(image)):
  for u in range(len(image[0])):
    print(f'{image[v][u]:3} ', end='')
  print('')
```

The result reads like this

```
255 255 255 255 200 150  60  15  60 150 200 255 255 255 255 255
255 255 255 110  50 100 128 128 128 100  50 110 255 255 255 255
255 255 255  20  50 100 128 128 128 100  50  20 255 255 255 255
255 255 255  20   0   0   0   0   0   0   0   0  64 100 255 255
255 255 255  20   0   0   0   0   0   0   0  20 230  64 230 255
255 255 255  20   0  80 192 192 192  80   0  20 255 120 120 255
255 255 255  20   0   0   0 192   0   0   0  20 255 120 120 255
255 255 255  20   0   0   0 192   0   0   0  20 230  64 230 255
255 255 255  20   0   0   0 192   0   0   0   0  64 100 255 255
255 255 255  20   0   0   0   0   0   0   0  20 255 255 255 255
255 255 255 110   0   0   0   0   0   0   0 110 255 255 255 255
255 255 255 255 160  30  15  10  15  30 160 255 255 255 255 255
```

If you see the higher values as brighter color, e.g. 255 as white, and lower values as darker color, e.g. 0 as black, you might be able to make out the image directly.[32] Figure 2.21 visualizes this relation between pixel values and image.

255	255	255	255	200	150	60	15	60	150	200	255	255	255	255	255
255	255	255	110	50	100	128	128	128	100	50	110	255	255	255	255
255	255	255	20	50	100	128	128	128	100	50	20	255	255	255	255
255	255	255	20	0	0	0	0	0	0	0	0	64	100	255	255
255	255	255	20	0	0	0	0	0	0	0	20	230	64	230	255
255	255	255	20	0	80	192	192	192	80	0	20	255	120	120	255
255	255	255	20	0	0	0	192	0	0	0	20	255	120	120	255
255	255	255	20	0	0	0	192	0	0	0	20	230	64	230	255
255	255	255	20	0	0	0	192	0	0	0	0	64	100	255	255
255	255	255	20	0	0	0	0	0	0	0	20	255	255	255	255
255	255	255	110	0	0	0	0	0	0	0	110	255	255	255	255
255	255	255	255	160	30	15	10	15	30	160	255	255	255	255	255

Figure 2.21 Visualization of the image data captured by the camera sensor in figure 2.20 with the object's contours overlayed.

Although the object image on the camera sensor is clearly upside down and inverted in figure 2.20 due to point reflection at the focal point of the optics, the camera electronics already compensate for this. Thus, the image data you get from the camera is oriented in the same way you would see the object when looking at it from the camera's point of view.[33] With this we conclude our brief overview of sensor basics and look into the basics of the sensing part that processes the sensor raw data into relevant information.

2.4 Robot Sensing Basics

Sensors deliver raw data to the sensing part that processes this raw data into meaningful information. In the previous section's examples we directly operated on the sensor raw data to decide the next action. Basically we short-circuited the sensors to the planning part, bypassing the sensing part. While this is acceptable for simple (toy) robots and robotics book sections that introduce sensors, it is a bad practice in real robot systems.

We need a proper sensing part in our robot system for the same reason we need proper components in other software systems: Handling complexity. Just imagine how difficult a Software Engineers life would be, if for every file operation one would have to write code that directly talks to the hard disk's registers. This would obviously reduce productivity, quality and maintainability by orders of magnitude. Hence, if you rely on drivers and filesystems provided by an operating system to deal with the low-level details of file access in your applications, you should also rely on sensing to deal with the details of sensor data processing and work at a higher level in the planning part.

Now that we have explained why we want to have a sensing part, let's look at some examples of what it does.

2.4.1 Digital and Analog Inputs

We start with a simple switch as sensor that provides a single digital input via `input.status()`. [34] How can we improve the meaning of something as simple and fundamental as a Boolean value that represents whether our switch is open or closed? Assume your colleague or your future self comes across the following lines of code that you wrote:

```
if not robot.inputs[0].status():
  robot.emergency_stop()
```

It is clear that if *something* connected to input `0` returns status `False` the robot is commanded to do an emergency stop. But why? The code could be improved by adding a comment that states

```
# If the active-low (pressed = 0 V = False) bumper switch connected to
# input 0 is pressed, then the robot hit an obstacle and must stop ASAP.
```

Problem solved? Well, at least things are improved. But only until the next hardware revision either changes the input wiring or replaces the active-low switch with an active-high one.

It is not my intention here to discuss the merits of code comments versus making code readable by giving understandable names and good semantics to variables and functions.[35] Rather, the point is that directly working with the sensor raw data outside the sensing part is not a good idea, not even for digital inputs. A much more readable and maintainable version of the code above could read

```
if robot.collision_sensor.collision():
  robot.emergency_stop()
```

This code does not require a comment and changes to the collision sensor design are properly encapsulated. If this sounds like software design 101, the reason might be that it is software design 101. However, as sensors and their data processing get more complex, it is easy to fall into the trap of ending up with a lack of proper abstractions in the planning code. One layer of encapsulation and abstraction as in the example above are often not sufficient. Consider a robot with multiple collision sensors. Do we really want to deal with this detail in all code related to moving the robot? The answer to this rhetorical question is a clear no. We don't want to deal with such details everywhere. Hence, we might introduce the notion of a collision detection subsystem in sensing that provides a higher level interface. We will soon look at a more complex example of such layering when discussing basic sensing for cameras.

Before going to cameras, let's take a look at sensing basics for analog inputs. Raw data from analog inputs can be processed in many different ways, especially when one takes into account how input values change over time. To give just one example, audio signals are naturally represented as an analog input signal. The processing pipeline between the analog audio input's raw data and recognizing commands given to the robot in a natural language is obviously not a trivial one. In this part of the book, we limit the scope to two common basic operations on analog input signals:

- Conversion to standard units
- Classification by thresholding

We use an analog optical distance measurement sensor as example. The sensor consists of two parts placed next to each other in the same housing: An LED sending out (infrared) light and a light detector (tuned to the LED's wavelength).[36] In a simple realization the output voltage is proportional to the amount of radiated light that makes it back to the detector. Due to both the physics of light and the properties of the light detector, the relationship between distance and output voltage is not linear. Instead, the relationship might look like figure 2.22.

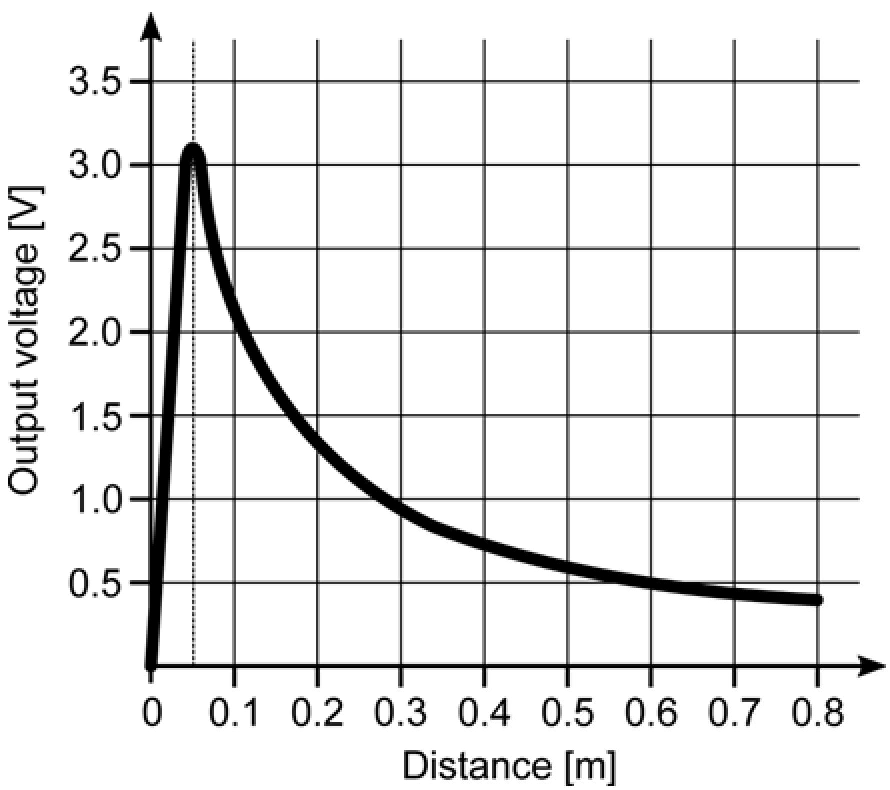


Figure 2.22 Relationship between distance and output voltage of an exemplary analog distance sensor.

To simplify the task, we assume that the sensor is mounted in a way that it is impossible for an object to come closer than 0.05 m. We make this assumption so we do not have to deal with the issue that there are actually *two* distances that result in the same output voltage: One in the range 0.0 to 0.05 m and one between 0.05 and 0.8 m.[37] Furthermore, the data sheet of the sensors tells us that distances beyond 0.8 m provide an output voltage smaller than the value at 0.8 m, which is about 0.4 V. The data sheet also tells us that these values do not provide valid distance information. So, in the range 0.05 m to 0.8 m, how do we get from raw data, analog input voltage, to distance information in the standard unit meters?

The most direct approach is to create a lookup table and add it to the sensing code. The table maps voltage values to distance values. One possible implementation is to discretize the voltage values to integers and fill out an array with the discretized voltage values as index and the distance values as content. The required information can be directly read from the sensors distance-to-voltage diagram (figure 2.22). An implementation could look like this:

```
voltage_to_distance = [0.0] * 32
voltage_to_distance[0] = 0.8  # 0.0 V -> 0.8 m
...
voltage_to_distance[4] = 0.7  # 0.4 V -> 0.7 m
voltage_to_distance[5] = 0.6  # 0.5 V -> 0.6 m
voltage_to_distance[6] = 0.5  # 0.6 V -> 0.5 m
voltage_to_distance[7] = 0.4  # 0.7 V -> 0.4 m
...
voltage_to_distance[31] = 0.05  # 3.1 V -> 0.05 m

def distance():
  discrete_voltage = round(input.value() * 10)
  distance = voltage_to_distance[discrete_voltage]
  return distance
```

This is a valid approach[38], albeit with the disadvantage of having to manually fill out a (potentially large) look up table and having an unnecessarily limited resolution. The limited resolution especially becomes a problem at larger distances, where a small change in voltage results in a large change in distance. Figure 2.23 highlights how the sensitivity between a change in voltage and a change in distance varies.[39] At short distances a change of 0.5 V corresponds to a change in distance of 0.08 m. At larger distances the same change in voltage, 0.5 V, corresponds to a much larger change in distance: 0.30 m.

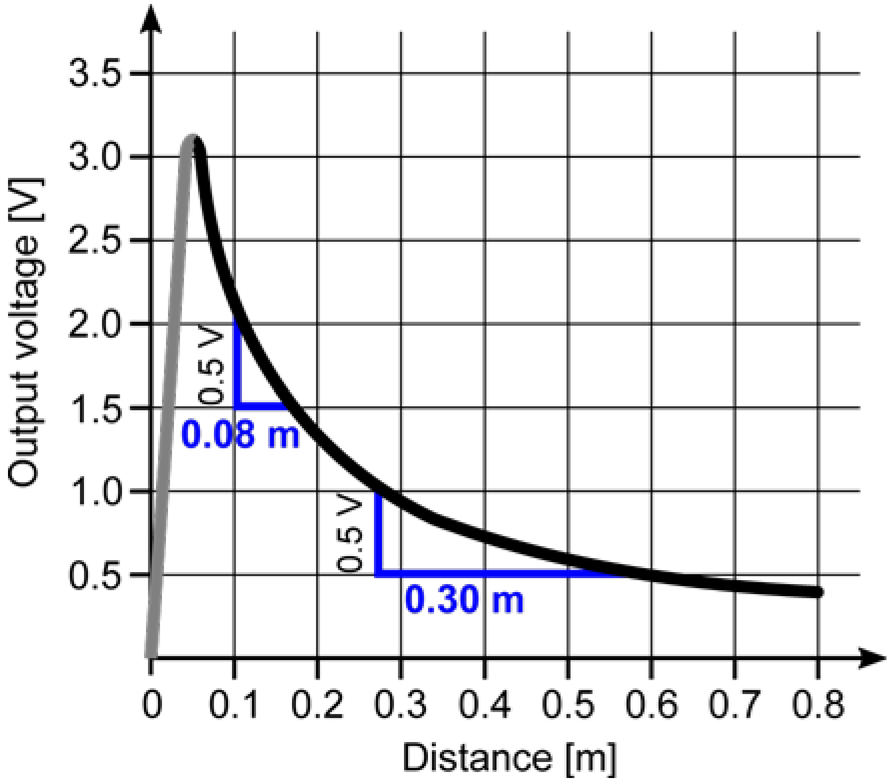


Figure 2.23 Illustration how the sensitivity between output voltage and distance measurement varies over the sensor's range.

With the lookup table approach, we have to increase the array size if we want to increase resolution. Along with it, we increase memory usage and also our manual work. Although all of this is feasible, at least up to a certain point, there are better solutions that we will learn about in later chapters.

When talking about converting to standard units, the important aspect is universal comparability. While the voltage output of our example distance sensor is a valid measure of distance (at least between 0.05 m and 0.8 m), it is not a good one. Others distance sensors will have another voltage scale, other voltage-to-distance relationship or have something other than voltage as output. In contrast, meter is a universally agreed upon measurement of distance[40] that is independent of a specific distance sensor. In this sense meter is a standard unit. One of the first processing operations in sensing should always be to convert data to standard units. Not only is it easier to understand, but it is also a much more future-proof software design that anticipates hardware changes - and the hardware will change, it always does.

Having learned some basics about converting analog inputs to standard units, we will look at classification by thresholding. We continue with our exemplary distance sensor. Its `distance()` function provides the measured distance in meters. For a robot barista we need a mechanism that detects when the coffee beans must be refilled. As the coffee machine does not provide us with a sensor, we add our own. We place the distance sensor on top of the bean reservoir facing downward, as illustrated in figure 2.24.

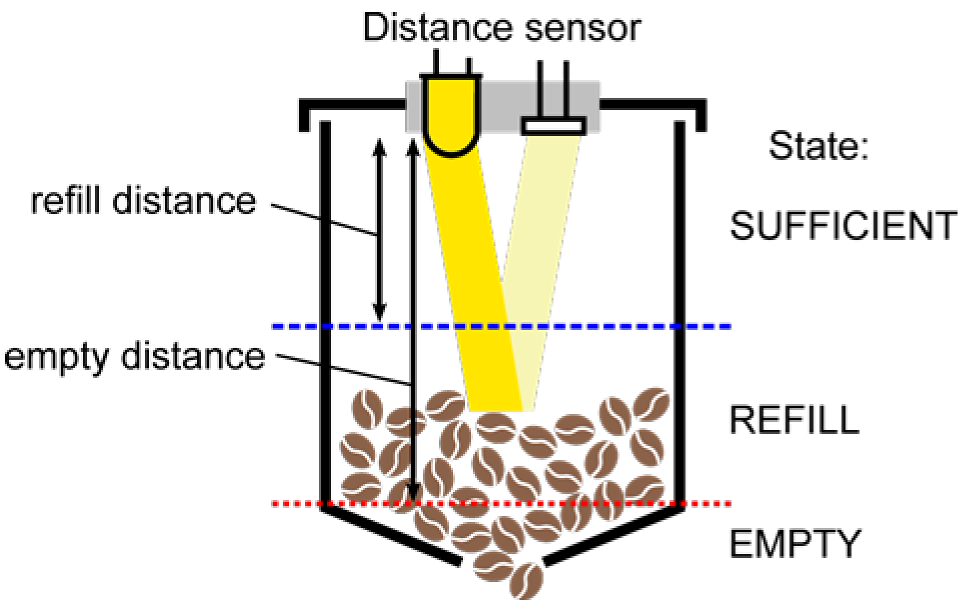


Figure 2.24 Distance sensor used to categorize coffee beans supply state.

Only three states are relevant for us:

- Sufficient coffee beans available
- Request refill, continue to make coffee
- Out of coffee beans, request refill, no coffee can be made :(

A suitable interface towards the planning part of the barista robot is a function `coffee_supply_status()` that returns an enum value for each state. The implementation in sensing that translates from distance measurements to coffee supply states could be:

```
def coffee_supply_status():
  distance = coffee_sensor.distance()
  if distance < refill_distance:
    return CoffeeSupplyState.SUFFICIENT
  elif distance < empty_distance:
    return CoffeeSupplyState.REFILL
  else:
    return CoffeeSupplyState.EMPTY
```

The code is a just a block of conditional statements. However, the concept of thresholding sensor values into discrete states or categories is a common and powerful pattern.[41] With this we leave digital and analog inputs behind for now and focus on basic sensing for cameras.

2.4.2 Cameras

Sensing for cameras, also known as computer vision, machine vision or image processing, is a vast field. Beyond the detailed discussion in chapter 12, we will frequently touch upon the topic of computer vision throughout the book. Each time you will learn additional methods or new applications in robotics. This short basics section is a mere teaser.

We already learned that camera image data can be represented as a 2D array of pixels, each pixel providing a light intensity value. In the sensing part information about objects and situations is extracted from image data and made available to the planning part.

Before we dive right into image processing, this is a good moment for a cautionary tale. Humans have very sophisticated visual perception capabilities. Even young children that have only seen an exotic animal, such as an elephant, in a picture book for kids are likely to instantly recognize the real animal during a visit to the zoo. When we compare the elephant drawing in the children's book with a photo of the elephant at the zoo, we will probably not find anything in common doing a pixel by pixel comparison. Nevertheless, we would agree with the child that both images show an elephant. We can also point out commonalities, such as the elephant having big ears, a long trunk and sturdy legs. Yet, once more, zooming in on the ears in each image to the level that we can see individual pixels, we are unable to describe the relation between the pixel values in the two images although each one clearly shows an elephant ear. What do I want to tell you with this excursion to the zoo in a book about robotics, since the zoo we just visited did not have any robots in it?[42] I want to warn you that having a camera record a pristine crisp image of something that *you* can clearly recognize is fundamentally different from writing *code* that is able to recognize objects and situations in images. Although there has been a lot of progress in recent years utilizing machine learning techniques, especially deep neural networks, for computer vision, we are still far from having a general purpose algorithm with human level capabilities.

This is neither a book on computer vision nor on deep learning. Still, both topics are important for robotics and will get their share of attention. Our focus will be on engineering approaches, as contrasted to data-driven methods or (end-to-end) learning approaches. Having said this, let's turn to a fundamental computer vision algorithm for detecting objects: template matching. We will make a number of assumptions to make the vision task *very* simple. The non-trivial algorithm will correctly solve the posed task. At the same time, it will fall short of most people's initial expectations on what it should also be able to do.[43] Extending the algorithm to even some of these common sense expectations is far beyond the scope of this introductory chapter. You will soon understand why. Later on you will then learn how to overcome these challenges - or at least circumvent them - and build capable robot systems nevertheless.

We want to detect a known object in the camera image. As just mentioned, we are making a number of major simplifications here:

- The object is flat and has a single uniform color.
- The camera and lighting are setup perfectly. The object is in focus and colors are true. There are no shadows, no glare, no underexposure and no overexposure.
- Only a single object is in the camera's view and the background is of a different homogeneous color.

You can translate the word simplification to limitations here. Because if any of the assumed simplifications turn out to be invalid for the actual task, the method might not work reliably or might not work at all. More powerful, i.e. less restricted, methods are the subject of later chapters. Figure 2.25 shows the two inputs to the template matching algorithm, which conform to our assumptions taken here.

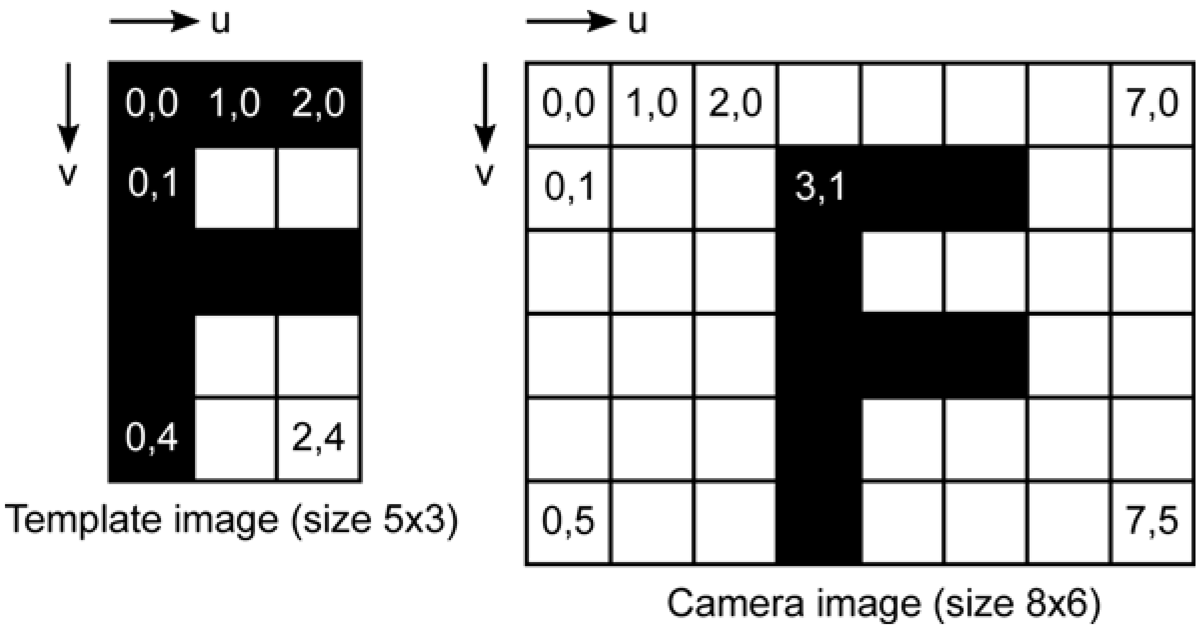


Figure 2.25 Inputs to the template matching method. A cropped image of the object, the template, together with the camera image in which we want to detect the object.

The following API expresses the functionality we want to achieve in code:

- Given the object's `template_image` and a `camera_image`,
- we want to get the `object_position` together with a value, `match_difference` that tells us how good the best match actually is.

```
object_position, match_difference = template_matching(template_image,
                                                      camera_image)
```

The template image is assumed to be smaller than the camera image. Template matching works by pixelwise comparison between the template image and the camera image at all possible offsets in the camera image. We start in the top left corner of the camera image. Using the coordinate system defined in 2.25, the first offset is at (0,0) in the camera image. We calculate a comparison metric between the template image and the equally sized image region underneath it. Then the same calculations are performed at offset (1,0). This process repeats until the template image reaches the bottom right corner, which is offset (5,1) in our example. Figure 2.26 illustrates six steps from this procedure.

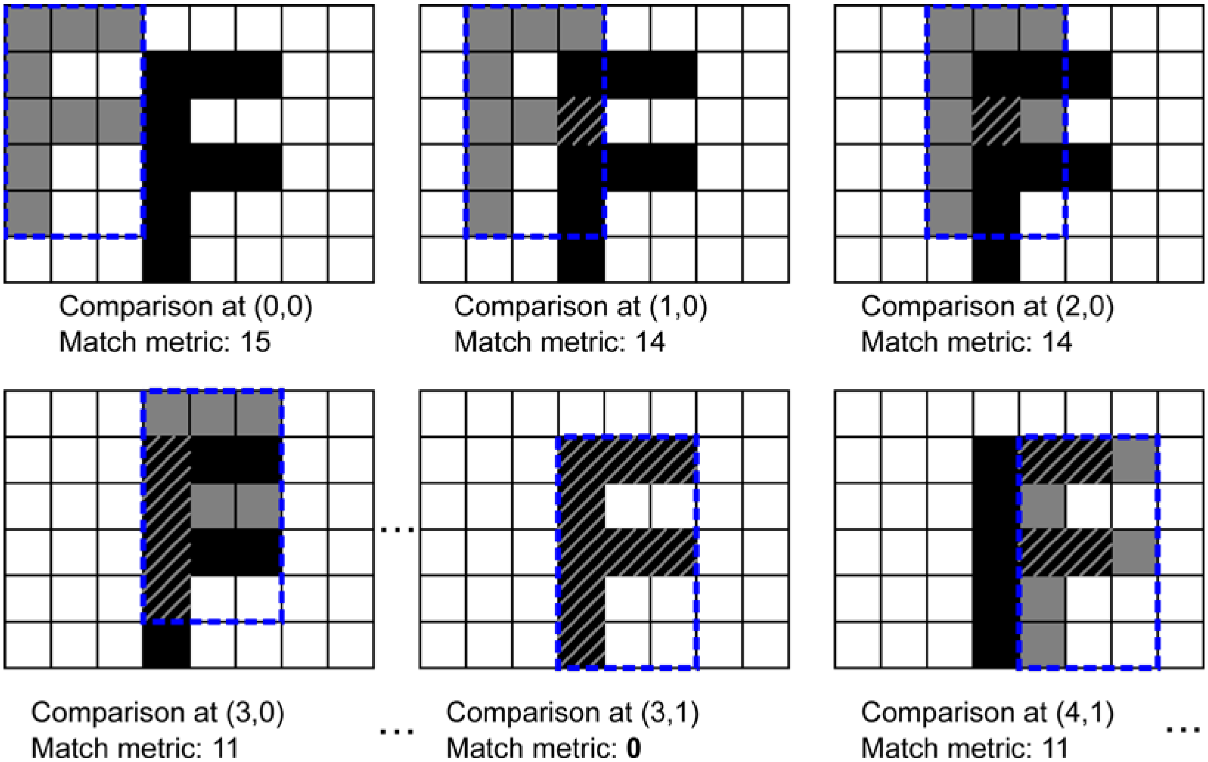


Figure 2.26 The template matching algorithm: The template image is compared to the camera image at every possible pixel offset. At each offset a comparison metric is calculated. The offset with the lowest comparison value is the best match.

In code, the procedure reads as

```
def template_matching(template_image, camera_image):
  best_match_offset = (0,0)
  best_match_difference = math.inf
  for v in range(camera_image.height - template_image.height):
    for u in range(camera_image.width - template_image.width):
      match_offset = (u,v)
      match_difference = match_metric(template_image, camera_image,
                                      match_offset)
      if match_difference < best_match_difference:
        best_match_difference = match_difference
        best_match_offset = match_offset
  return best_match_offset, best_match_difference
```

The two loops iterate over the offsets. In each iteration we calculate the match metric. If the current match metric is better than the best one we have seen thus far, it becomes the new best one. Finally, the function returns the best match once all possible offsets have been evaluated.[44]

There are different ways we can calculate the match quality (`match_metric()`). Each having its advantages and disadvantages in real applications. We will use one of the simpler metrics here: sum of absolute differences (SAD).

To calculate the sum of absolute differences, we subtract each pair of pixels that lie on top of each other when the `template_image` is shifted to `match_offset` in the `camera_image`. This gives us the `difference` value. We take the absolute value of this difference to get the `absolute_difference`. Finally, we sum up all these absolute differences. Hence, the name sum of absolute differences.

```
def match_metric(template_image, camera_image, match_offset):  # SAD
  sum = 0.0
  for v in range(template_image.height):
    for u in range(template_image.width):
      v_cam, u_cam = match_offset[1] + v, match_offset[0] + u
      difference = template_image[v][u] - camera_image[v_cam][u_cam]
      absolute_difference = abs(difference)
      sum += absolute_difference
  return sum
```

This is not the right place to go into more details.[45] Neither is this the kind of book that will provide a formal proof that `match_metric()`, or SAD, is a proper metric in the mathematical sense of the term metric. Anyhow, `match_metric()` does return the lowest value when the template lines up with the object in the camera image, i.e. offset (3,1) in our example.

Just in case you have refrained from getting into robotics in the past because you were overwhelmed by a pile of mathematical formulas, it might be interesting to consider that the `match_metric()` function we just implemented can also be expressed as

$$\sum_{v}^{T_h} \sum_{u}^{T_w} |T(u, v) - I(u + u', v + v')|$$

My message here is that the book takes a code first approach. I will use formulas where they support understanding, but I will always express them in source code as well.[46]

Before we went into object detection in images using template matching, I had cautioned that the result will likely fall short of expectations. Looking at figure 2.27 the limitations of basic template matching are quite obvious.

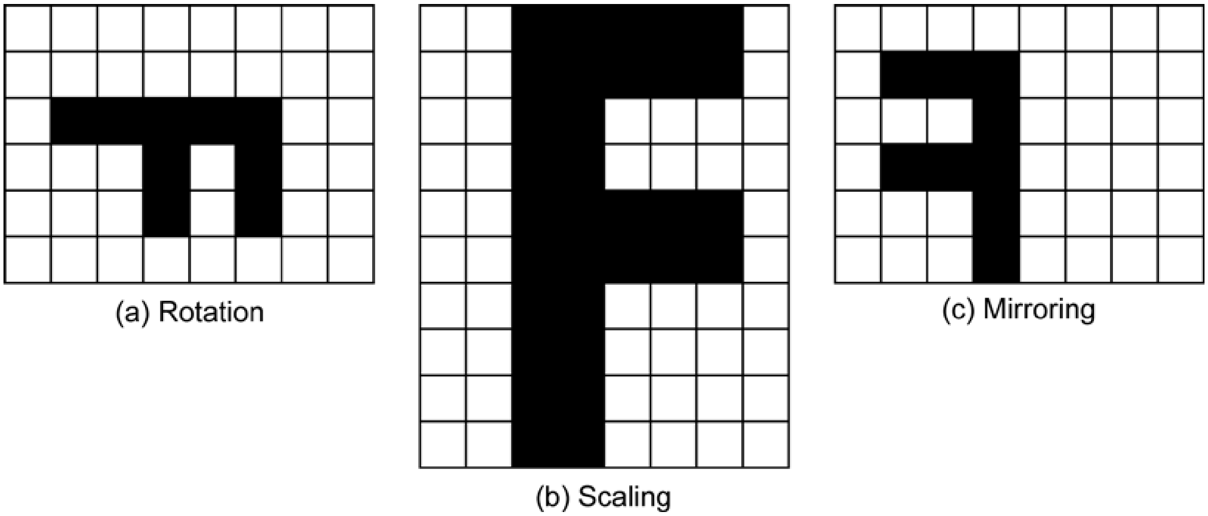


Figure 2.27 Some limitations of the basic template matching method. Rotation, scaling and mirroring result in bad matches between the template and current image.

If we give any of the camera images from figure 2.27 to our `template_matching()` function, it will not properly detect the object. To us it is quite obvious that the object is contained in each image. The only difference is that the object is rotated, scaled or mirrored compared to the template image of the object.

Assuming we know that the object can only appear in a number of predefined ways, we can extend our template matching implementation to accommodate these known variations. An example of this is that the object can only appear rotated in discrete 90° steps and it can appear only scaled by a factor of 0.5, 1 or 2. We can then create a set of additional template images, one for each combination of variations. In the example, we would add one with the unscaled object rotated by 90°, one unscaled at 180°, …, one scaled by a factor of 0.5 with no rotation, …, one rotated by 90° and scaled by a factor of 0.5, etc. Overall we would end up with 12 template images. We pass them one by one to `template_matching()` and use the best overall match:

```
overall = (None, math.inf, 0)    ❶
for i in range(len(template_images)):
  position, difference = template_matching(template_images[i], camera_image)
  if difference < overall[1]:
    overall = (position, difference, i)
print(f'Best match at {overall[0]} with diff {overall[1]} (i={overall[2]})')
```

❶ The best overall match: (position, difference, index).

This approach is feasible and used in practice, if there are only a few known and discrete variations. It becomes unfeasible for many variations or if variations are not known beforehand. We will get to know other methods suitable for these situations.

You have just learned the first method that will help your camera equipped robot to recognize certain objects - albeit still with some limitations. Referring back to our example where the object was detected at (u,v) = (3,1) in the camera image, can we pass this information straight to planning to pick up the object? Unfortunately, the answer is no. Thus far, we only have a 2D image coordinate in the camera's image. We do not yet know how this image coordinate is related to a pose in the environment. Furthermore, from what we discussed about adding proper layers of abstraction, we should not just hand object-in-image information to the planning environment and assume it can deal with this type of information. Rather, we should continue to make our sensing part more capable and have it provider higher level information, such as object-with-properties-in-environment. In other words, there is more to learn about sensing than what we could fit into this basics chapter, at the same time we are already well on our way from raw data to relevant information.

This concludes our introduction to sensors and sensing through which robots receive information about the environment. The next section is all about having an effect on the environment through acting and actuators.

2.5 Robot Actuator Basics

Again, it is worth reiterating from the first chapter: The acting part takes care of transforming planned actions to performed actions. Actuators convert data into physical quantities and thereby influence the environment.

We will take a similar path through actuator and acting basics as we have for sensors and sensing. We start with the most simple actuators - from a software point of view. Those actuators controlled by a single bit via digital outputs. These have just two states: on and off. Next we look at those controlled by a single analog value. Finally, we briefly look at the most important type of actuator for robots today: electric motors.

2.5.1 Digital Electric Outputs

Digital electric outputs are the inverse of digital inputs. Digital inputs provided us with a Boolean value, with `False` (or `0`) representing a low voltage on the input and `True` (or 1) representing a high voltage respectively. A digital output takes a Boolean value and then puts a low voltage (False or 0) on the output or a high voltage (True or 1) respectively.[47] Our abstract API for a digital output is

```
output.set_status(state)
```

with `state` being a Boolean value. Figure 2.28 depicts a few examples of robotics equipment that is controlled through digital outputs.

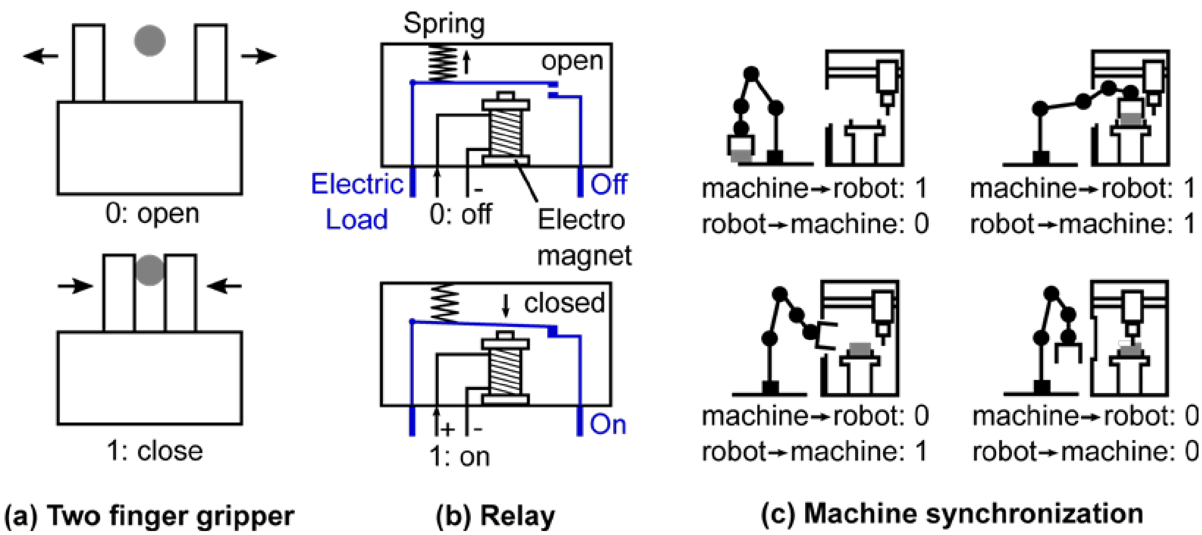


Figure 2.28 Three examples of robot actuators controlled by digital outputs.

You might be surprised to learn that the majority of robot grippers used today is indeed controlled through a single bit. The gripper itself usually allows to manually change the grip force through a setscrew. Gripper fingers are custom tailored to the workpieces handled by the robot. In addition, grippers often provide a digital or analog feedback on whether an object has been grasped. This state of affairs has worked well for robots performing highly repetitive tasks and continues to do so in the future as well. For robots that have to deal with a large variety of objects and perform more varied tasks, more complex grippers with more sophisticated interfaces are available. We will encounter them in later chapters.

Relays are electrically operated switches that enable us to control high voltages or large electric loads using a digital output. Electric voltage is measured in volt (V). Electric current is measured in ampere (A). Electric power and electric load are measured in watt (W).[48] Power can be calculated by multiplying voltage and current. Similar to other electric equipment both digital outputs and relays come in a wide variety of ratings. In order to give you an order of magnitude feeling, here are some common electrical characteristics for both:

- Digital outputs
  - Microcontrollers: Voltage: 1.8 V - 5 V; Current: 4 mA - 40 mA
  - Industrial equipment: Voltage: -10 V - 24 V; Current: 4 mA - 2 A
- Relays: Voltage: 12 V - 480 V; Current: 2 A - 30 A; higher rated ones exist

Electro-mechanical relays, such as the one sketched in 2.28(b), are still widely used. They are called electro-mechanical because the control input energizes an electromagnet, the control coil, which in turn mechanically closes the spring-loaded contacts that switch the connected load. The newer type of relays based on semiconductors, so called solid state relays (SSR), are replacing electromechanical ones in more and more applications. The main advantage of SSRs is their longer lifetime and increased reliability due to the absence of moving parts. Progress in semiconductor technology has also achieved equal or higher voltage and current ratings in same package size.[49]

The last example, using digital outputs together with digital inputs has already been described as part of the digital input basics in section 2.3.1. Our next topic are analog outputs.

2.5.2 Analog Electric Outputs / DACs

Analog electric outputs, also known as digital-to-analog converter (DAC) outputs, are the inverse of analog inputs or ADCs that we discussed in section 2.3.2. We pass a numeric value to the output that sets the output to a specific voltage level The voltage must be within the output's operating range, e.g. 0 V to 5 V. The abstract API for analog outputs is defined as

```
output.set_value(value)
```

with `value` being a floating point value in the range 0.0 to 1.0.[50]

Let's look at a few examples of robot actuators in figure 2.29 that are commonly controlled through analog outputs.

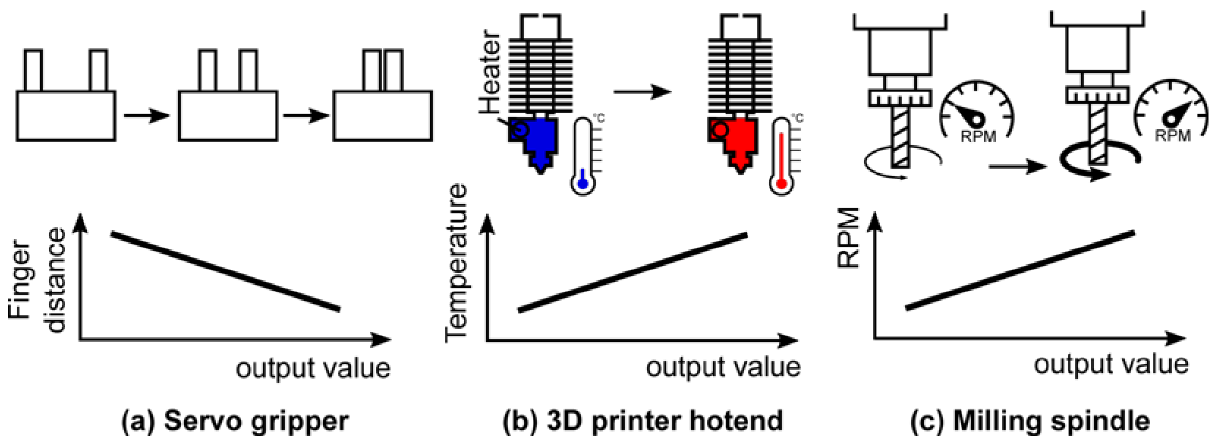


Figure 2.29 Three examples of robot actuators controlled by analog outputs, also known as DACs.

The first example is a servo gripper. A servo gripper can be controlled in a more nuanced way compared to a gripper connected to a digital output. While the latter can only be commanded to open and close,[51] the servo gripper's actuation is more flexible. There are different ways to make use of the analog value, such as:

- Control how far the gripper opens and closes, i.e. the distance between the gripper fingers
- Control the maximum grip force

The next example is a 3D printer hotend that melts plastic filament in fused filament fabrication (FFF) 3D printing. Different filament materials have different melting temperatures. Thus, if we want to use a robot manipulator as 3D printer, it will not be sufficient to simply turn on a heating element via a digital output.[52] Instead, we need more fine grained control over the temperature. Using an analog output value to signal the target temperature to the hotend is a viable interface option.

The third and final example of actuators controlled through analog outputs is a milling spindle. We cannot only use our robot manipulator for additive manufacturing (3D printing), but also for subtractive machining, e.g. cutting, drilling and milling. In all machining operations is essential to keep the relationship between material, tool, tool path and tool speed within a defined range. Otherwise, the either tool will be destroyed or the resulting workpiece has poor quality. Thus, we use an analog output to control the spindle speed in terms of its rotations per minute (RPM).

Although repetition might be the mother of learning, I don't want to bore you either. Let me repeat only one more time what has already been said for senors and sensing: It is essential to build proper interfaces and layers of abstraction into robot systems. Thus, do not work with analog output voltages in the planning part, but use the acting part to bring actuator control to a meaningful level. Instead of having lines such as

```
# Set the hotend connected to analog output 0 to 250° for printing PETG
robot.outputs[0].set_value(0.7)
```

in your planning code, build a proper 3D printer hotend component into the acting part and make use of it. Then the code will be much easier to digest:

```
robot.tools.hotend.set_temperature(NOZZLE_TEMPERATURE_PETG)
```

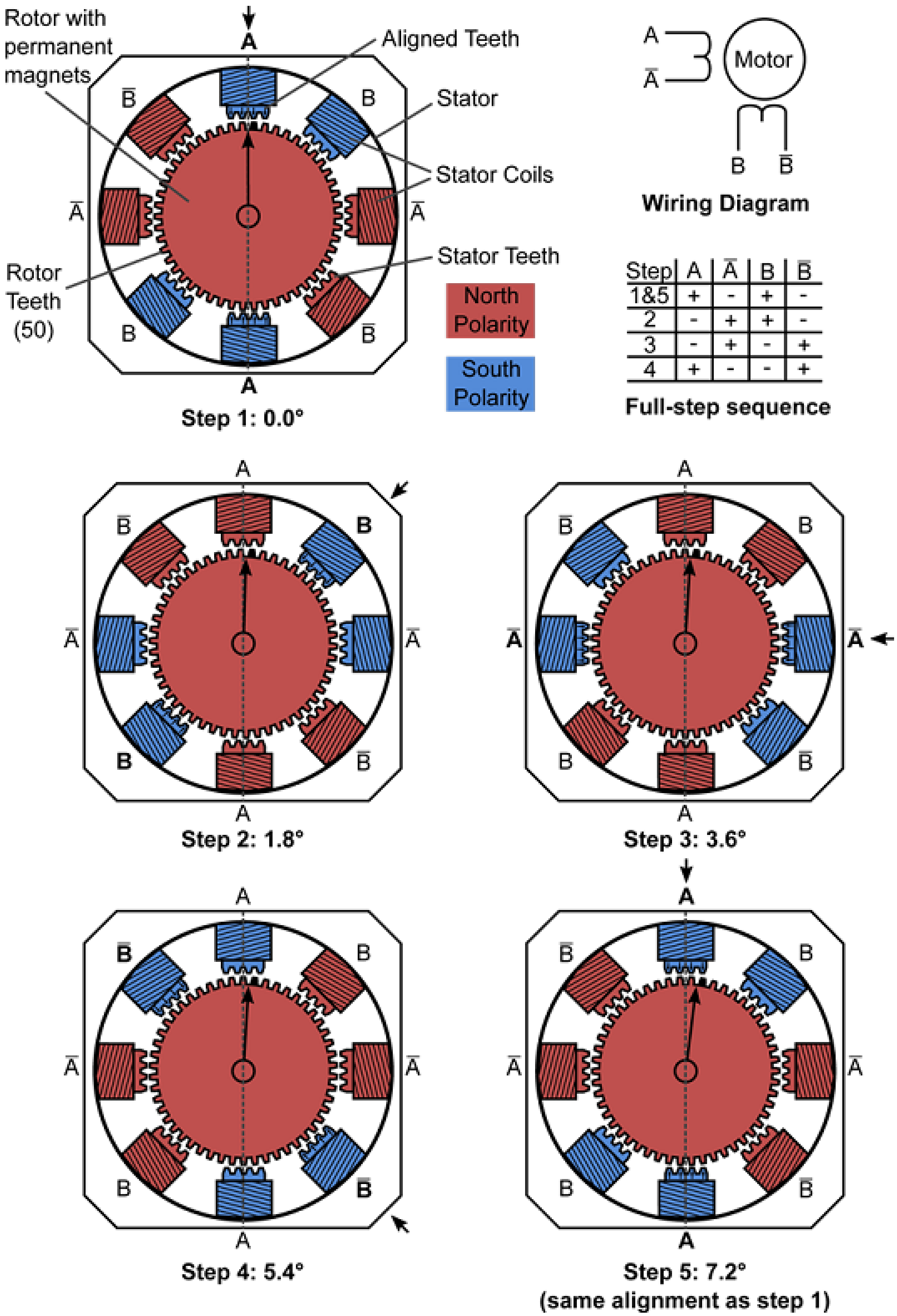
With this we leave digital and analog outputs behind and proceed to electric motors.

2.5.3 Electric Motors

Electric motors are arguably the most important actuator in robots today. Although some robots are driven by hydraulic (pressured oil) or pneumatic (pressured air) actuators, electric actuators are the dominant type since the 1970s. Fundamentally, an electric motor consists of two parts: rotor and stator. The stator is the part of the motor remaining stationary during motion and the rotor is the moving - rotating - part. The stator is usually the motor housing on the outside with the rotor being the rotating motor shaft on the inside.[53] Electric motors convert *electric* energy into motion.

In this short section, we learn about the three most widely used types of electric motors in robotics: stepper motors, (brushed) DC motors and brushless DC (BLDC) motors. Today stepper motors and DC motors dominate the consumer robotics space, just as BLDC motors dominate the industrial robotics and professional service robotics segments. Decreasing costs for the more complex control electronics required by BLDC motors make them increasingly widespread in consumer segments, too. In general, BLDC motors are superior to both stepper motors and DC motors in terms of performance and reliability.

Let's start with stepper motors. From a hardware point of view they are not the most simple type of electric motor, but they are arguably the most simple type from a software perspective. The big advance of stepper motors is that their *position* can be easily controlled. Complete position and speed control can be performed through four digital outputs. No feedback about the current motor status, e.g. current position, is required.[54] Each output controls the voltage of one coil-end wire via (power) transistors. Depending on the sequence in which the coils are energized, the stepper motor will turn left or right. Figure 2.30 depicts a stepper motor and how it generates motion.



Step	A	$\bar{A}$	B	$\bar{B}$
1&5	+	-	+	-
2	-	+	+	-
3	-	+	-	+
4	+	-	-	+

Figure 2.30 The stepper motor and its principle of operation. The bipolar variant is shown.

The discrete stepping behavior of stepper motors is achieved through the toothed permanent magnet rotor in combination with the toothed stator electromagnet. Each step magnetically pulls the rotor teeth to the closest complementary magnetized stator teeth. The motor in figure 2.30 has 50 teeth on the rotor and requires 4 steps to move one tooth forward. Thus, each full revolution

(360°) requires 50 * 4 = 200 steps. Put differently, each step rotates the motor by 360° / 200 step = 1.8° / step. While this is a common stepper motor variant, there are other variants with finer or coarser step sizes.

Note that there are two common types of stepper motors: unipolar stepper motors (5 wires) and bipolar ones (4 wires). The main difference lies in the electronics that control them. Also the exact pattern on the digital outputs needed to make them turn is different. Apart from this, they are sufficiently similar for our purposes that we do not have to differentiate between them. More details on controlling stepper motors will follow in section 2.6.2. Being easy to control is one of the reasons stepper motors can be found in many DIY robotics projects. Furthermore, they provide high torque at slow speeds, which often enables using them without gearboxes. Finally, stepper motors are relatively inexpensive.

The main drawbacks of stepper motors is their low power-to-weight ratio compared to other motor types. Put differently, they are relatively heavy for the torque they produce. This makes them ill suited for articulated robot manipulators. In these manipulators, each joint has to carry the weight of all links and joints, including the motors, that follow it towards the end effector.[55] If the motors are heavy, the previous joint must have an even bigger motor to provide enough torque, hence making the joint before even bigger and heavier. One easily ends up in a robot design that cannot lift any payload - or not even lift itself. The other major disadvantage is that stepper motors deliver much less torque at higher speeds than at low speeds. Finally, stepper motors are relatively inefficient. A lot of electric energy is turned into heat instead of generating motion. Let's move on to the next motor type of interest: DC motors.

Direct current (DC) motors are likely the most well known type of motor in everyday life. You find them in everything that creates motion and runs on batteries. From electric toothbrushes and shavers to RC toys to cordless power tools. DC motors have two important advantages. They are very easy to use in many applications and they are inexpensive. However, DC motors are usually far from easy to use in robotics.

Consider a cordless drill that is based on a (geared) DC motor.[56] When you press the switch on the handle of the cordless drill, the DC motor is connected to the battery and starts rotating. If it is not necessary to adjust the motor speed, the entire electronics required would only consist of the battery, the DC motor and a regular on-off switch in between. No further electronics are required to operate a DC motor, as shown in figure 2.31.

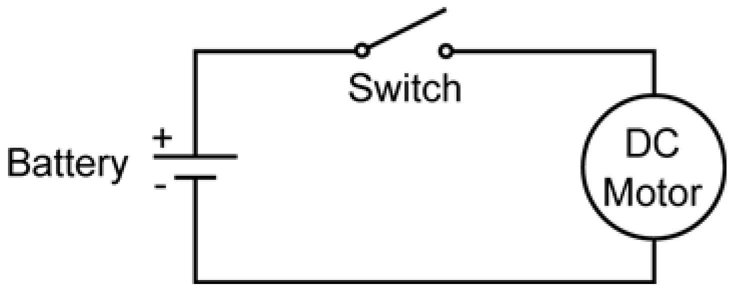


Figure 2.31 Simple electronic circuit diagram (schematics) to control a DC motor.

Adjusting the motor speed via the switch can be achieved in different ways. The switch in a cordless drill is actually not an on-off switch, but a continuous slider. In the most simple implementation, the switch adjusts a variable resistor that controls a transistor in between battery and motor. Better implementations use pulse-width modulation (PWM) control, which is more efficient, i.e. less electric energy is turned into heat. In any case, the motor speed will depend on its mechanical load, i.e. the size of the drill bit, the material and the depth of the drilled hole. Since the cordless drill is used by humans, they can easily adjust the power to the motor via the switch, when it is turning too fast or slow.

What I want to emphasize here is that DC motors are commonly operated without feedback from the motor. If we connect a DC motor directly to a suitable DC power source, e.g. a battery, it will start turning and accelerate until it reaches its no-load speed. When we add load to the motor, e.g. apply friction on the drive shaft, the motor speed will go down. Put differently, we are actually not controlling the motor's speed, but only the power given to the motor. The resulting speed depends on external factors, such as motor load, that are unknown unless we have additional sensors to measure them. Furthermore, I want to highlight that we have been talking about the motor speed here, not the motor's position. In contrast to stepper motors, which we discussed above, it is difficult - actually surprisingly difficult - to control the position of a DC motor.

Let's shortly discuss how a DC motor works and why no more than a simple switch is required to get the motor turning. Figure 2.32 illustrates the internals of DC motors and different states during a rotation.

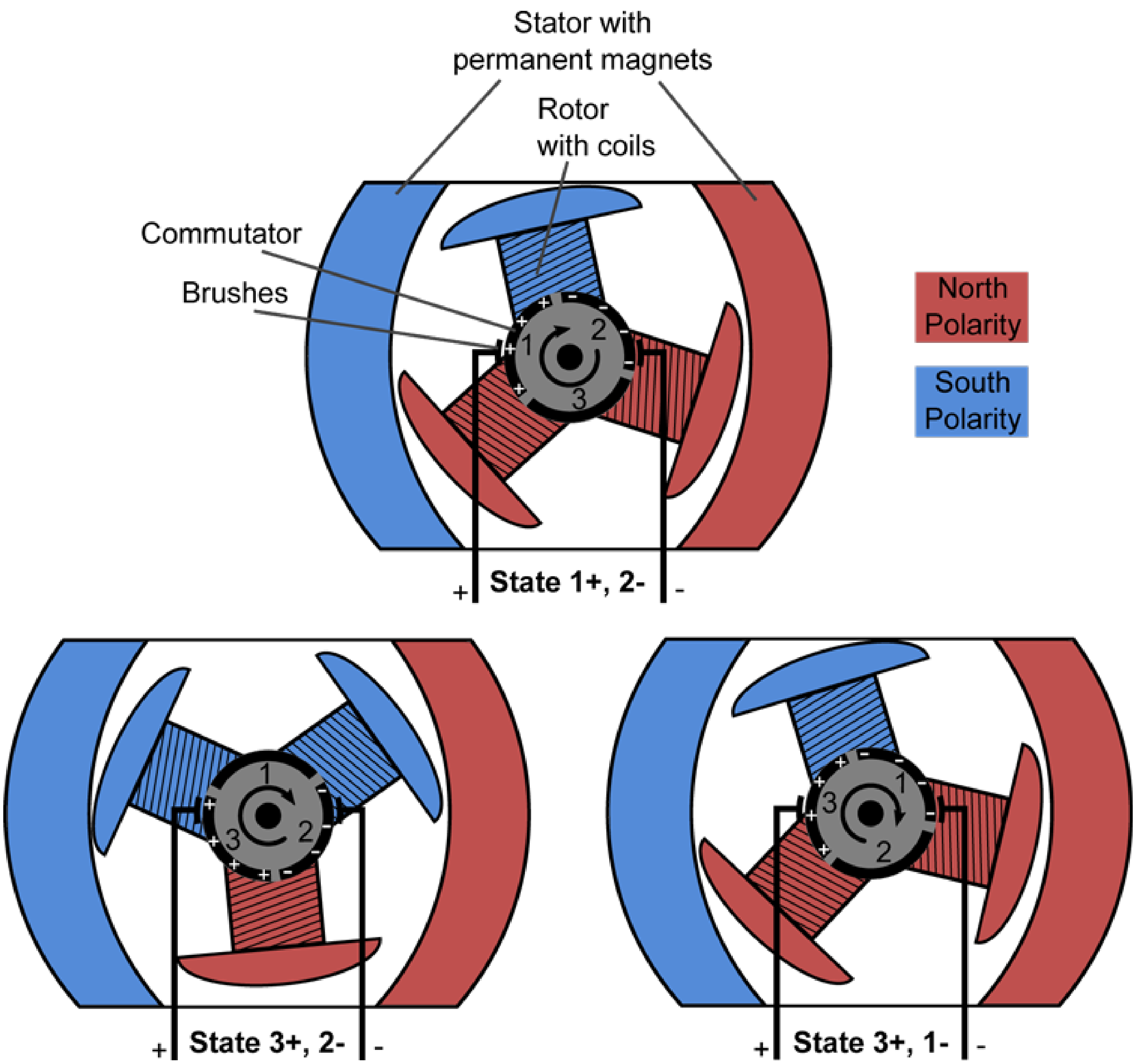


Figure 2.32 The DC motor, its parts and principle of operation.

In the DC motor, the stator contains permanent magnets and the rotor contains coils that act as electromagnets. Because the rotor is rotating during operation, one cannot attach wires to the rotor coils. Therefore, the electric energy is transferred from the stationary connections to the rotating coils via brushes and the commutator. The commutator is simply a round piece of metal subdivided into several electronically isolated sections. The spring-loaded brush, often made of carbon, presses against the commutator and conducts electric energy to it. Together commutator and brush form a rotary switch.

The ingenious part that enables a DC motor to be used without external control electronics is the interaction between the motor's rotation, the commutator that rotates as part of the rotator and the resulting sequence of energized rotor coils. In just the right moment during the rotation, the brushes make contact with a different segment on the commutator and thus change the magnetic polarity of the rotor coils, to continuously keep the rotor spinning via magnetic attraction and repulsion between rotor and stator. One could say that the commutator is a hardware implementation of the control sequence that we have to implement in electronics and software for other motor types. Note that real motors usually contain more coils in the rotor than our illustration in figure 2.32 to improve torque and smoothness of operation. Since proper (position) control of DC motors, which we need for robots, is nearly as complex as control of the superior BLDC motors, we end our discussion of the former here and continue with the latter.

The brushless DC (BLDC) motor has a rotor with permanent magnets and a stator with electromagnetic coils. This is the inverse construction to the DC motor. Because the moving part, the rotor, no longer needs an electric connection, the BLDC motor does neither need brushes nor the commutator. Hence, the parts with the highest wear in the DC motor are absent in the BLDC motor, making it more reliable.

On the other hand, we just talked about the ingenuity of the commutator that switches the electromagnets at just the right moment for rotation. Since the BLDC motor does not commutate its electromagnets by mechanical means, they must be commutated externally. Therefore, BLDC motors are also known as electronically commutated (EC) motors. We need much more sophisticated electronics and software to make them move. This is also the reason why BLDC motors have only become practical with the advent of semiconductor electronics, e.g. transistors and ICs. Although continuously getting cheaper, the required electronics are still a cost driver for simple applications compared to DC motors. Still, in ever more applications the breakeven point between cost and improved reliability and performance is passed in favor of BLDC motors. If we exclude their use in robot toys and DIY robots, DC motors and stepper motors have already been replaced by BLDCs in robotics.

Figure 2.33 illustrates the structure and operation of BLDC motors. Note that the number of permanent magnetic poles on the rotor (here two) and the electromagnetic poles on the stator (here three) can be different - usually higher - in actual BLDC motors.

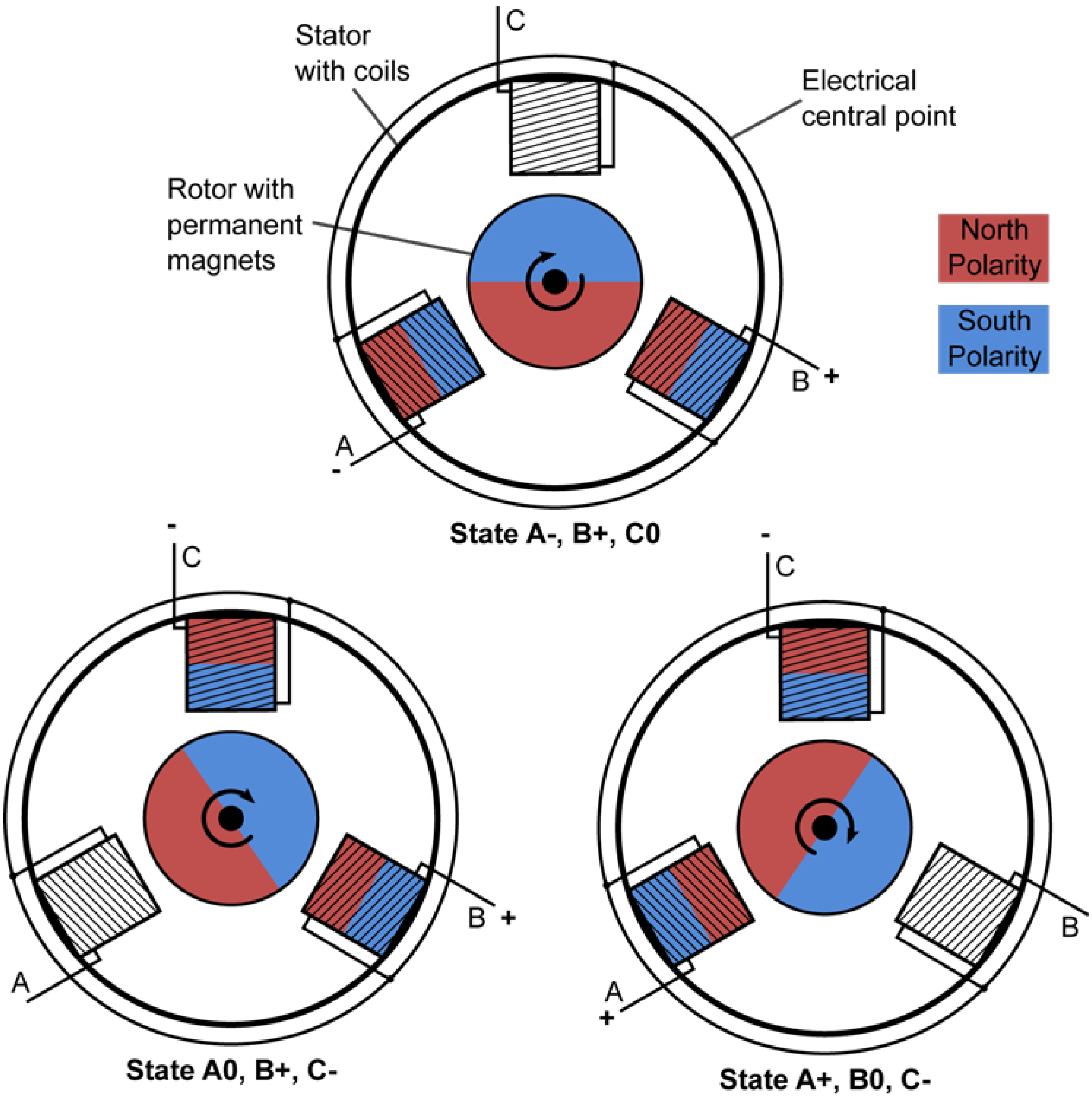


Figure 2.33 The brushless DC (BLDC) motor. A common inrunner variant with a permanent magnet rotor is shown. Three wires (A, B, C) provide the electrical motor interface. Each wire is connected to one stator coil. The other side of all coils is connected internally in the so called central point.

In practice the control electronics for BLDC motors primarily consist of six power transistors in a so called triple half-bridge driver configuration, also known as three-phase driver. Each of the motor's three wires (A, B, C) is connected in between a pair of transistors. One transistor can connect the motor wire to positive supply voltage, the other transistor can connect it to negative supply voltage (ground). Taken together, the electronics enable us to selectively put each motor wire into one of the following states individually:[57]

- Floating: Wire neither connected to positive voltage nor to negative one.
- Positive: Wire connected to positive voltage.
- Negative: Wire connected to negative voltage.

Now it is just a matter of controlling the six transistors in the right sequence at the right time, using six digital outputs.[58] In order to do this properly, we need precise and timely feedback about the motor's current position.[59] The most common sensors used in controlling BLDC motors are additional position sensors mounted to the motor or so called Hall sensors directly integrated into the motor. We already learned about position sensors in section 2.3.3 above. Hall sensors, usually three, are a much simpler and less accurate type of position sensor built into BLDC motors. They only signal which magnetic rotor pole is currently at their location. This is sufficient for speed control of BLDC motors as well as position control with limited accuracy.

We will see in section 2.6.2 and later chapters that profound electrical knowledge and understanding of control theory is required to get BLDC motor control right. This is especially true for the high demands put on motor control by robots. But then again, the likelihood that *you* need this in-depth knowledge of motor control as a robot software engineer is about as low as the likelihood that you need to understand compiler design to use compiled programming languages. It is highly valuable to understand the basics, but only a small group of specialists, those implementing the motor control layer, must really understand the details.

Let's conclude this section on electric motors by acknowledging that we have now seen the three most relevant motor types used in robotics. However, many other variants of electric motors exist. Fortunately, now that you understand the basic function of stepper motors, DC motors and BLDC motors, you will find that the principles you learned from these three types are sufficient to get acquainted with other variants as well.

2.6 Robot Acting Basics

Our journey through the basics of robot sensors, sensing, actuators and acting is slowly coming to an end. In the first section of this chapter, we discussed the complementary nature of sensors and actuators, thus also of sensing and acting.[60] Therefore, what we discussed and exemplified in the sensing section (2.4) about building proper layers of abstraction and internal APIs into robot systems equally applies to acting. We will only look at specific aspects of previously introduced actuators in this section, but will not repeat topics already covered in the sensing section above. Let's start with digital and analog outputs.

2.6.1 Digital and Analog Outputs

When working with digital and analog outputs via their abstract APIs, i.e. `output.set_status(state)` and `output.set_value(value)` respectively, we must also pay close attention to timing. In figure 2.34 two sequences of output states applied on a digital output are shown. Are the sequences the same?

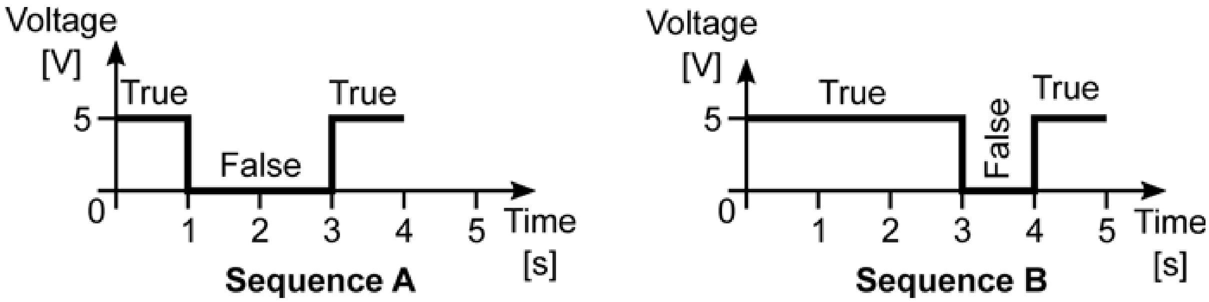


Figure 2.34 Two digital output sequences with same states but different timing.

Both sequences are produced by calling `output.set_status()` with the arguments `True`, `False`, `True` in this order. From this point of view, the sequences are the same. However, if we take time into account, compared to sequence A, the first state (`True`) lasts three times as long in sequence B and the second state (`False`) only half as long. Seen from this perspective, the sequences are quite different.

Whether the difference has any significance depends on the robot system and its application. Yet, it is important to always keep in mind that the difference *can* be very significant for what happens in the environment. If the output in question is connected to a relay turning a toaster on and off, the difference in timing might correspond to our robot chef serving us perfectly browned toast or serving us charcoal instead.

Timing is a crucial aspect of most robot systems, simply because the environment undergoes changes over time. This might be one of the more difficult things to adapt to when entering the field of robotics with a background in software development. In the purely virtual inner world of a computer program, nothing changes on its own. While we aim for programs to execute as fast as possible, the result is correct independently of how fast it is provided. This is not true in robotics. There is a big difference between the robot moving its manipulator to a pose in time to catch a falling object, and reaching the same pose, but only after the object has already crashed on the floor. The point here is not speed, this is not about robot code having to be particularly fast, the point is correct timing.[61] The general topic we are touching on here is the real-time behavior of our robot system, which is discussed more in chapter 3.

The second and also last topic about acting and digital outputs is pulse-width modulation (PWM). PWM clearly falls into the acting part. PWM is not a new type of actuator or output, rather it is a particular use of digital outputs. The idea is to quickly toggle the digital output with a variable ratio between on state (`True`) and off state (`False`). The extreme ratios, 0% on and 100% on, are identical to simply having the output in off or on state with no PWM. The frequency of toggling the output is the PWM frequency with a unit of Hertz (Hz).[62] While frequency expresses the number of cycles per second (Hz = 1 / s), the period expresses the duration of one cycle in seconds. Thus, the relation between the two is period = 1 / frequency and frequency = 1 / period. The ratio between on and off states is known as duty cycle and given as percentage.

The abstract API for PWM is

```
output.set_pwm(frequency, duty_cycle)
```

While it is possible to implement PWM purely in software on top of digital outputs, i.e. using `output.set_status()`, this is often not feasible in practice. The reason is that real PWM frequencies are often in the hundreds or thousands of Hertz. For example, at 1 kHz, we need to run a code snippet that toggles the output state at least 2000 times each second (once to turn the output on and once to turn it off each period), i.e. every 0.5 ms on average. The main issue here is not CPU load, although this can be an issue as well, but precise timing. If the CPU is also used to perform other tasks, it must be ensured that these can be interrupted on time to toggle the output state at the right moment. In practice, the hardware platform on which we run our robot software, provides PWM outputs that can be programmed to run with a given frequency and duty cycle. Our code, running on the CPU only needs to configure the PWM outputs. The cyclic toggling is then performed independently of the CPU.

Figure 2.35 shows PWM signals at different frequencies and duty cycles.

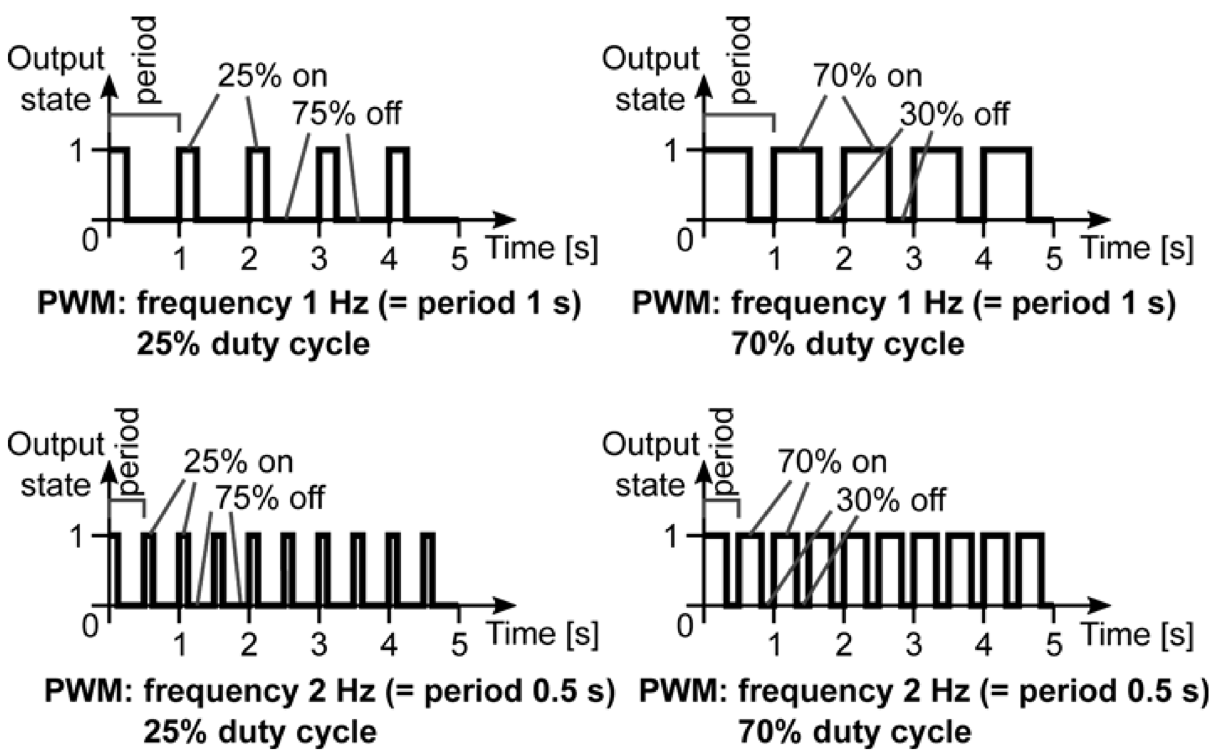


Figure 2.35 PWM signals with different frequencies and duty cycles.

Why is rapidly toggling a digital output so important that it deserves to be treated explicitly in this acting basics section? The answer is straightforward: PWM is frequently used to control various robot actuators. In the next section, we will learn about its use for motor control, but there are many other simpler uses.

Every electric component in our robot system that can not only be turned off completely or operate at full power is a candidate for PWM control. The only requirement is that the component has a certain inertia. Inertia in this context means that the component's state does not instantly switch between completely off and completely on. Instead the component takes some time to change its state. Let's look at two examples: LEDs and a heating element (e.g. a 3D printer hotend). The heating element's inertia is more obvious. When we turn on the heating element, it will take some time until it heats up. Once heated up, it will not instantly cool down when turned off again. Thus by turning it on for a few seconds and then off again for a few seconds, we can control the heat via the duty cycle of our PWM signal. The same also goes for LEDs. While they can blink rapidly, i.e. quickly turn on and off completely, they also don't instantly become bright and dark.[63] Thus we vary the LED's brightness by repeatedly toggling the LED on and off for fractions of a second, i.e. control it by PWM.

The component's inertia can be seen to result in a kind of smoothing or averaging of the PWM signal.[64] Such an averaged PWM signal is more or less equivalent to an analog output signal. But then why not just use an analog output in the first place?

Using PWM instead of an analog output is preferable from a technical point of view due to the characteristics of transistors. Transistors can be either used as electrically operated switches, like relays discussed previously, or as amplifiers. When used as switch the transistor is turned on and off completely, it is controlled by a digital output signal. In this switch mode the electrical losses in the transistors are relatively small. In other words, the transistor turns only a small amount of electrical energy into heat. In contrast, when used as an amplifier of the analog input signal, there a significant electrical losses in the transistor. In summary, because is preferable to operate transistors as switches, PWM control is frequently preferable to using an analog output signal. The higher the electric power involved, the more advantageous is PWM. This leads us directly to control of the electric component with the highest power demands in most robots: motors.

2.6.2 Electric Motor Control

Depending on your background and interests you might be disappointed or you might be relieved that this section will not delve into the details of motor control. Anyway, you will learn some important basics on the topic of (motor) control, but we will not go into details just yet. Let's start with a fundamental distinction: open-loop control versus closed-loop control.

In our established terminology, open-loop control means that we influence a property in the environment through actuators, but do not sense this environment property.[65] Put differently, open-loop control means that we act blindly. This might not sound like a good idea, but it can actually work very well in a controlled environment. Let's suppose we have a simple robot manipulator using stepper motors with a simple two finger gripper. Our robot always performs the same cycle: It waits for a digital input signal, then moves to pose A, closes the gripper, moves to pose B, opens the gripper and then repeats the cycle. This robot neither senses the motor positions, nor the state of its gripper, nor anything else about the environment apart from the one digital input. Hence this is an example of an open-loop control. If there is a machine that always outputs a workpiece at pose A and there is another machine that accepts the workpiece at pose B, our robot fulfills a valuable job by unloading one machine and tending the other.

This example shows that open-loop control is feasible beyond toy examples. At the same time, many things can go wrong in this example. If the stepper motor would miss steps due a temporary overload situation, our robot would move to wrong positions. If there is no part at pose A, our robot would still continue to move to pose B and open its empty gripper there, i.e. it would go through the motions without doing anything useful. We could continue the list of things that could go wrong, but I think the point is made that open-loop control has its downsides, even when it is feasible in the first place.

Closed-loop control means that we influence a property that we also measure. In our machine tending robot example, this could mean that we have motors that provide position feedback or a gripper that detects whether it has actually grasped an object. Given feedback about the motor position, we can use it to compare the commanded motion with the actually executed motion and take action in case they deviate. This straightforward concept of getting feedback about actions and taking the feedback into account for following actions lies at the heart of closed-loop control. Figure 2.36 illustrates the fundamental difference between open-loop and closed-loop control.

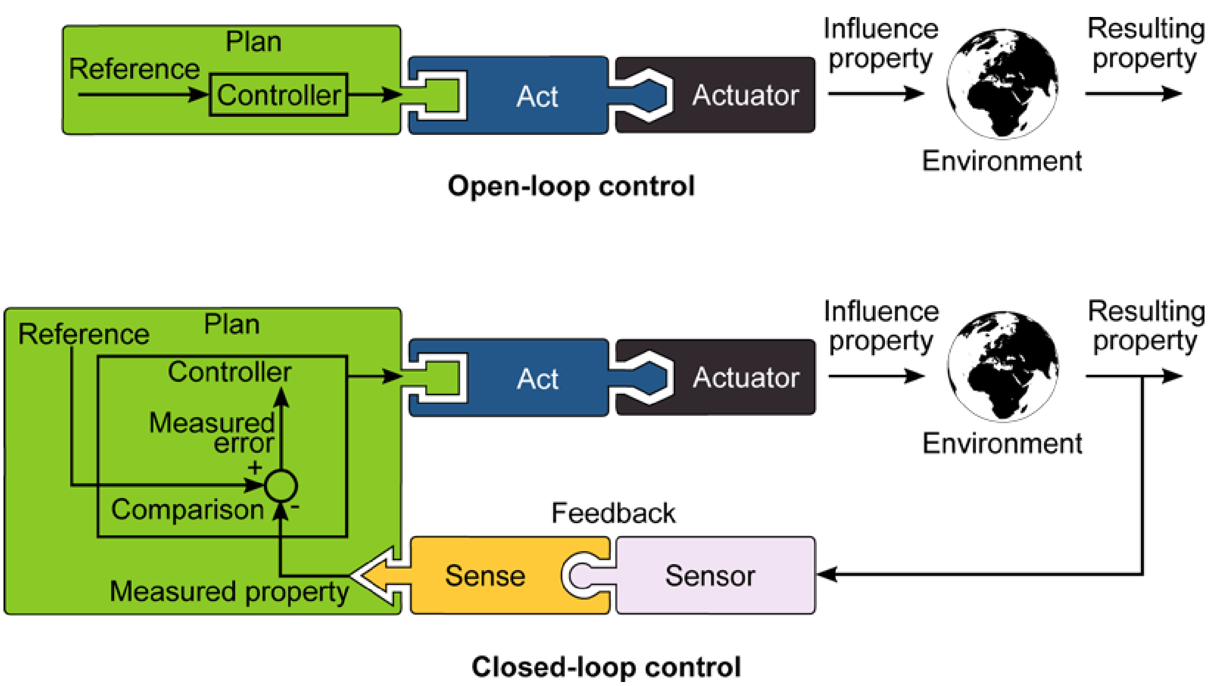


Figure 2.36 Comparison of open-loop control and closed-loop control.

I want to conclude this brief introduction to open-loop versus closed-loop control by explaining that there is usually a hierarchy of nested control loops in robot systems. The loops on each layer can be open or closed. Let's look at our machine tending robot example again to make this notion of nested control loops more tangible. We have outfitted the example robot with actuators that provide position feedback and we use this position feedback for closed-loop control of the motors. Hence, the joint position control layer is now closed-loop. We could regard the grasping of the object as the next layer above. Because we do not sense the objects actual position, e.g. using a camera, the object-grasping layer is open loop. The next layer above could be the manufacturing line in which our robot is installed. We have a digital input that signals to the robot that a part should be moved between the two machines. The robot senses whether the machines are ready to unload and load the next workpiece via this input. Hence, we can regard the machine-tending layer as closed loop. Finally, it is worth noting that there are also layers below motor position control, such as speed control, torque control and current control. Now that we have investigated the hierarchy of nested control loops in this example, it becomes clear why one should not state that the robot system is open-loop or closed-loop. It is unlikely that everything in the robot falls into one category or the other. Rather, one should specify the layer or function in the system that is using open-loop or closed-loop control.

You might be surprised to read that we have already discussed another closed-loop control in this chapter. Take a moment to think about which of our previous examples fits the bill before reading on. The answer is the light following robot in section 2.3.2. Having learned about open-loop and closed-loop control, do you now see it in a different light? The reference was the robot’s heading, towards the light source, encoded into the algorithm itself. The feedback was provided by the light intensity sensors. It was compared against the reference heading, which results in equal readings on both light sensors. Was this very explicit from the example code? Probably not. We could do better now by naming and structuring the code along a closed-loop control.

Let’s wrap up this control primer with a few words about control of electric (BLDC) motors. We have seen the sequence of voltages that must be applied to the three phases (wires A, B and C) of a BLDC motor to make it turn. We also established that we must know the rotor position to control the phases correctly and at the right time. What we did not mention before is that good control requires current measurement on each motor phase. Furthermore, we must control the six transistors via PWM with variable duty cycle. Overall you already know the basic building blocks to start controlling the motor. The missing piece is the actual control algorithm that transforms the sensing input (motor currents and motor position) into the acting output (PWM for the motor phases). There are a number of control strategies for BLDC motors from (nested) proportional-integral-derivative (PID) control to field-oriented control (FOC) to model predictive control (MPC). Each of them goes far beyond the scope of this chapter. You will learn more about control related topics throughout the book. Although often in another context than motor control, your knowledge around the topic of control will continuously grow until we are ready to give our robot’s motors a spin. Before getting there, the next chapter will visit a more familiar and not less relevant topic for becoming a robot software engineer: robot software systems.

2.7 Summary

- Robot software interacts with the real world through APIs that get data from sensors and send commands to actuators.
- From a software perspective, sensors and actuators have a lot in common and play a complementary role in the robot system.
- Although every specific robot software defines its own specific API, they are fundamentally all the same for each type of robot, sensor and actuator.
- Understanding just three fundamental robot types (manipulators, mobile robots and mobile manipulators) is sufficient to work with a large variety of real robots.
- Different types of robots have different abstract APIs. Manipulators move objects in the environment, mobile robots navigate themselves through the environment and mobile manipulators combine these capabilities.
- The inherent complexity of robot systems must be properly handled. It is essential to build good layers of abstraction into the robot software, instead of dealing with low-level details everywhere.
- Many types of robot sensors are connected to the robot through digital inputs and analog inputs.
- Position sensors and camera sensors are further common robot sensors.
- Processing camera data is known as computer vision. While one should not expect to achieve human-level perception, relatively simple algorithms can already provide good application-specific results.
- Digital outputs, analog outputs and PWM outputs are common interfaces to robot actuators.
- Electric motors are important robot actuators. Three relevant motor variants are stepper motors, DC motors and BLDC motors.
- Timing is a crucial aspect in robotics. One reason is that the environment undergoes changes over time. These changes are happening independently of the robot program.
- The notion of open-loop control and closed-loop control is fundamental. The difference between them is the availability and use of feedback data. Control loops are nested in hierarchical layers in robot systems.

3 Robot Software Systems

This chapter covers

- **The robot software stack**
- **The basics of embedded real-time systems, distributed systems, mechatronic systems, embodied intelligence and their relation to robot software**
- **Four abstract example robots as reference robot architectures**

We learned about the six essential parts of a robot system (sensors, sense, plan, act, actuators, environment) in chapter 1. Chapter 2 explained the three fundamental robot types (manipulators, mobile robots, mobile manipulators). Furthermore, the basics of sensors, sensing, acting, actuators and their abstract API were discussed. In this chapter, we take the next step towards implementing concrete robot software components.

Real robot systems have a certain inherent complexity to them as they need to deal with the complexity of the real environment. In order to manage this complexity well, just like with any other complex software system, we need a suitable software architecture. Although, we leave many details of robot software architecture - and software engineering - to later chapters, you will already have a solid understanding of the typical robot software stack by the end of this chapter. It is important to have a feeling for the different layers in the robot software stack, because each specific robot functionality exists in a wider context and not in isolation.

On the one hand, this wider context is the robot software stack. On the other hand, the context also consists of the different fields that have an impact on the robot as a whole. Each field has a different view of the robot system and also uses a different terminology. Ultimately, we are creating *one* robot system. The different fields and views must hence be integrated into one solution. We discuss several of these perspectives and connect them to our robots as software perspective.

We will start this chapter by discussing the robot software stack in section 3.1. In section 3.2 you will then learn the basics of embedded real-time systems, distributed systems, mechatronic systems and embodied intelligence as well as their relation to robot software. Finally, the chapter wraps up with section 3.3 that introduces four robot templates which will accompany us throughout the book.

3.1 The Robot Software Stack

Thinking back to chapter 1, especially figure 1.1 (reproduced below as figure 3.1), you might wonder why we discuss the robot software stack *again*.

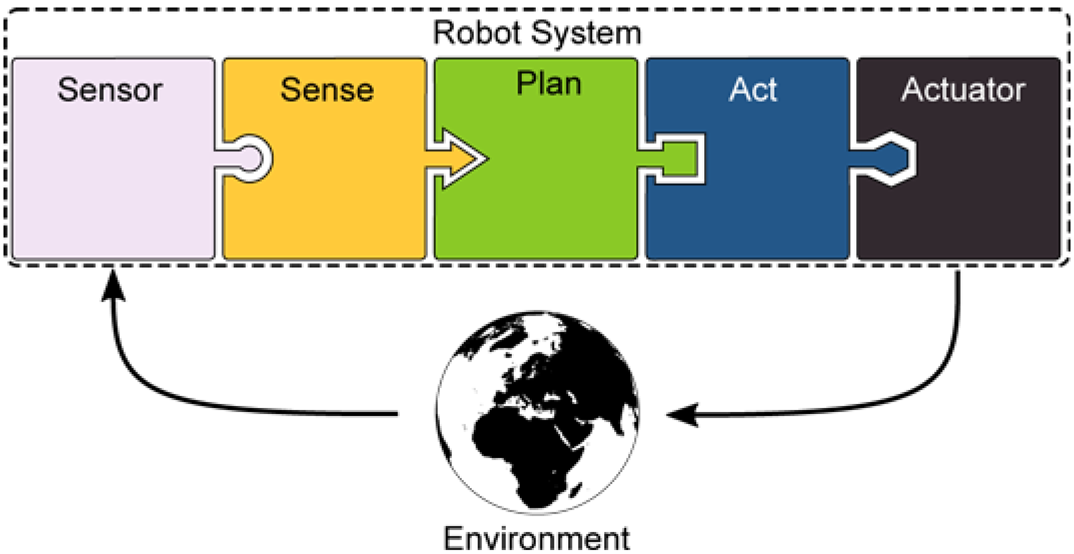


Figure 3.1 The conceptual structure and essential six parts of all robot systems. Reproduced from chapter 1. The horizontal subdivision of robot systems.

The answer is that there are different views on a system's (software) architecture. Each view focuses on different aspects of the same systems. Figure 3.1 focuses on the conceptual parts that make up the casual loop between robot and environment. We will refer to this as the horizontal subdivision. In contrast, figure 3.2 focuses on the layers of the robot software stack. We will use the term vertical subdivision for this one.

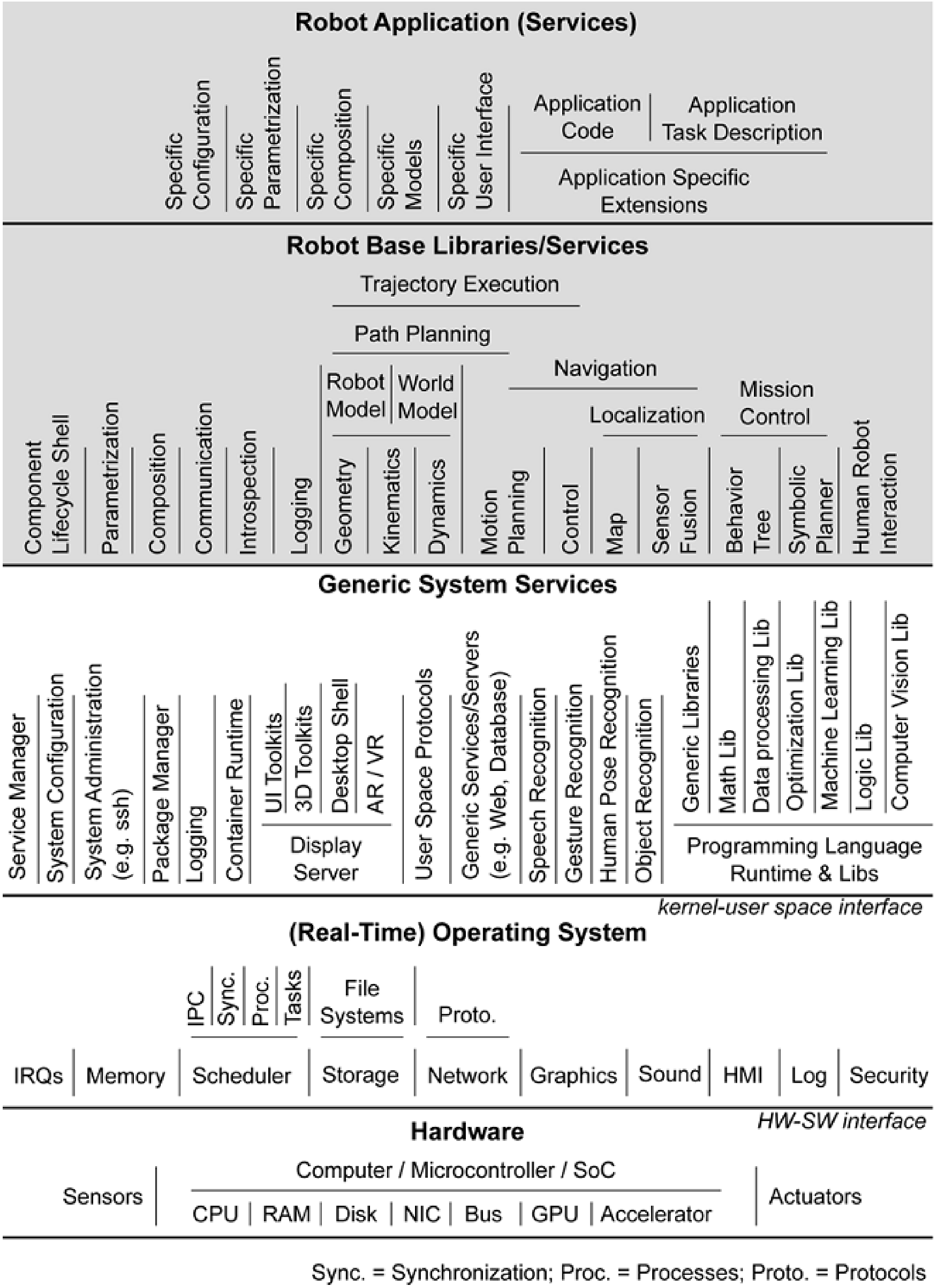


Figure 3.2 The robot software stack. The layers specific to robotics are highlighted. The vertical subdivision of robot systems.

Although 3.2 might contain some parts that are not yet familiar, my assumption is that the general structure of these layers is already familiar to you from working with other software systems. The bottom layer is composed of the hardware devices. These hardware resources are managed by the operating system (OS) layer.[66] Another responsibility of the OS is to encompass drivers that provide a unified API to similar hardware devices. Furthermore, the OS provides the basic mechanisms for multitasking, memory management, data storage and networking. Above the OS layer are the generic services needed in most computer systems. These include managing installed software, performing software configuration and executing software. We also regard software runtimes, general-purpose software libraries and general-purpose services, such as web servers and databases, as part of this layer. Up until this layer, there is nothing robotics specific. Yet, as we will soon discuss, robot software relies heavily on these layers and in some cases uses them in unique ways.

The first robot specific layer is the robot base libraries & services layer. I refer to it as robot *base* layer, because it contains functionality that is independent from concrete robot applications. Not all robots contain all components listed here. For example, manipulators do not need navigation functionality and some robots might not need a symbolic task planner.

At the top of the robot software stack sits the robot application layer. This is the layer that defines what the robot is actually doing, i.e. what task(s) the robot is actually performing. In the most simple case, the application layers consists of a list of instructions that are sequentially executed. In other words, it consists of a (simple) imperative program. In addition, some configuration of the robot base layer is performed here. Such a program would directly call functions in the robot base layer, e.g. a sequence of `move()` instructions for manipulators or a sequence of `navigate()` instructions for mobile robots. The top instruction in this kind of program is an endless loop that performs the same sequence over and over. Usually, there is some kind of start condition, e.g. an external signal, which then starts the next loop iteration. You might be surprised to hear that the majority of all (industrial) robots deployed today work in this way. In more complex robot systems,[67] the application layer contains a set of object, environment, task, skill and other models for the robot base services. Either there is still an imperative program as top-level control or a formalized description of the robot's task is used, that can be interpreted by an automated reasoning component. We will see plenty examples of both ways to realize robot applications.

Before bringing the horizontal and the just discussed vertical view together, I want to direct your attention to the boundary between the robot base layer and the robot application layer. Using the example of a pancake making robot from the previous chapter, let's think about two ways to realize it. In terms of the robot software, the easiest solution is to put all ingredients and kitchen utensils in a well-defined place.[68] Given such an environment, we have nothing more to do than write down a sequence of `move()` commands (and gripper open/close commands) to get pancakes. The robot will only work in the exact environment with the exact utensils for which we have created the program. Therefore, it is clear that in this solution all pancake making capabilities are part of the robot application layer. On the other end of the spectrum, we could add sophisticated object recognition capabilities, reasoning skills and task planning functions for pancake making to our robot. The top level instruction would then just state the number of pancakes to make. Would all this functionality just be usable for making pancakes or could it not also be utilized for other recipes? The likely answer is that we could reuse a lot of functionality for other cooking tasks. So, should we add Chef to the robot base layer in fig 3.2, if we can build generalized functionality around kitchen robots? My suggestion is to leave this Chef component in the robot application layer and rather think of having some layers nested within the application layer. The robot base layer should be reserved for really broadly applicable functionality that is independent of a particular domain. Still, as often with such classifications, there is some gray area, but it need not bother us.

Now, let's bring the horizontal and vertical views from figure 3.1 and figure 3.2 together in figure 3.3:

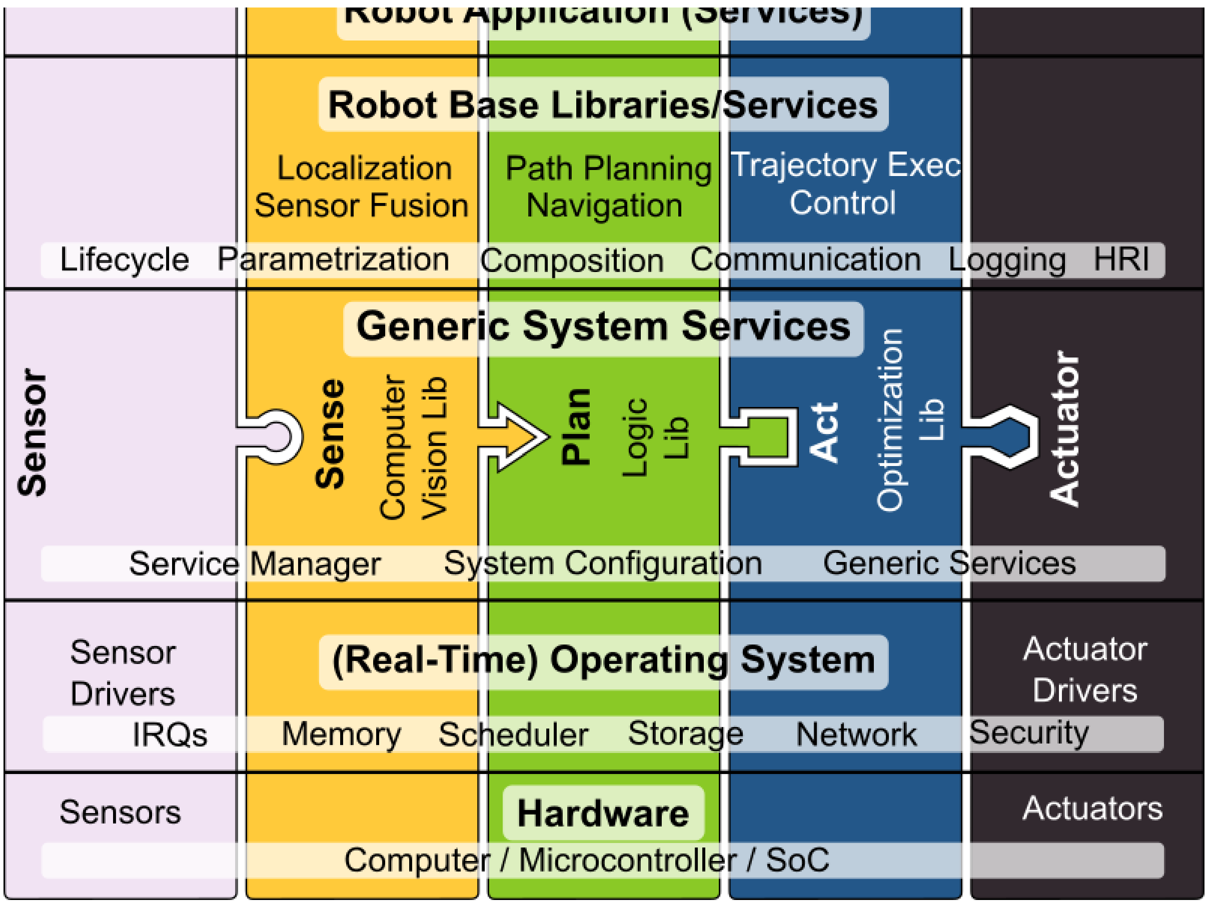


Figure 3.3 Combined view of the horizontal subdivision (parts) and the vertical subdivision (layers).

Some horizontal layers are relevant for all of our vertical parts. Since we assume our robots are controlled by digital computer programs,[69] we need computation capabilities in all parts. This does not only mean the compute hardware, but also a lot of functionality from the operating system and generic system layer. Other components such as specific program libraries are primarily associated with a specific part, e.g. computer vision libraries with sensing.

We see a similar pattern repeated in the robot base layer. Some functions, such as communication and logging, are relevant for all parts. Others have a much closer association with specific parts, such as sensor fusion with sensing, navigation with planning and control with acting. Components related to models tend to be used in multiple parts. For example, the robot model of a manipulator is updated in the sensing part based on encoder data and is also used in acting for the purpose of control. While this non-exhaustive list of components or robot software building blocks might seem daunting at first, they will soon become second nature to you.

We have talked quite a bit about robot *systems* already. Let's briefly talk about the boundaries of these robot systems and how they relate to computer systems. Figure 3.4(a) shows the base case consisting of one computer[70] connected to a set of sensors & actuators and running one robot software stack.

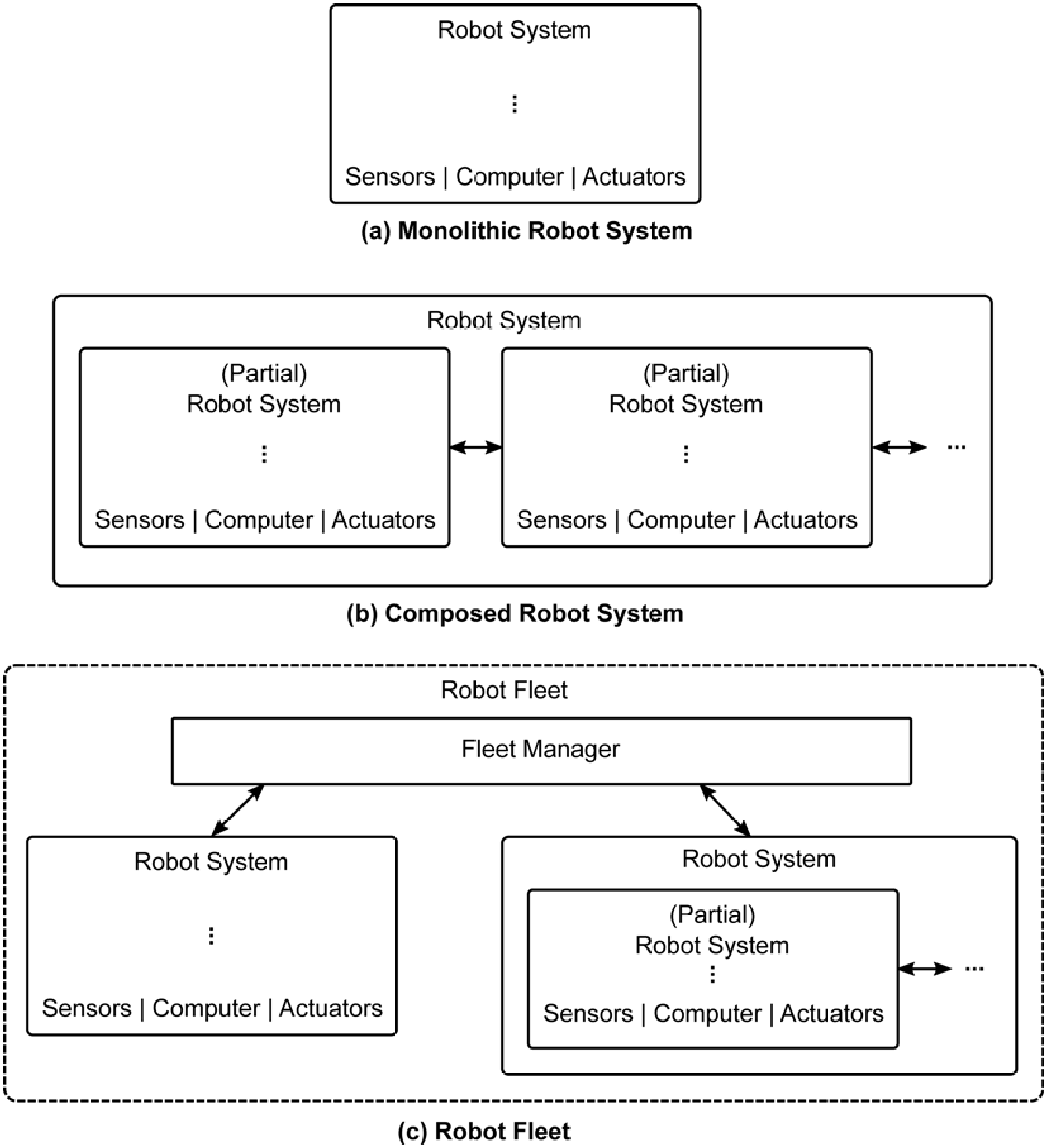


Figure 3.4 Robot systems and their system boundaries.

In figure 3.4(b) the robot system consists of multiple computers connected to each other. Each computer can have its own sensors & actuators and runs (parts of) the robot software stack. But only in combination do these devices make up a complete robot system that can perform intended tasks.

Finally, figure 3.4(c) shows a number of complete robot systems working together. Although each robot system is in itself complete and can perform specific tasks, additional benefits can be gained from coordinating them. Such a coordinated set of robot systems is called a robot fleet.[71] The coordination instance is hence called a fleet manager. Consider a (large) number of mobile robots performing logistics tasks in a warehouse environment. Each robot can independently transport goods from location A to location B. The fleet manager has a global view of transport tasks and vehicle positions. In order to avoid traffic jams and to minimize the overall distance travelled, the fleet manager gives (high-level) orders to the individual mobile robots instead of each mobile robot deciding only based on its local information.

There is another dimension we should keep in mind: the robot system's lifecycle. Everything we discussed in this chapter thus far was about an operational robot system. However, there are multiple phases in the life of a robot solution. These include

- Development (incl. testing and simulation)
- Commissioning
- Operation
- Maintenance

The development phase, for example, will include components such as a developer tools (IDEs, compilers, debuggers, build systems, CI/CD, CAD/CAE, etc.), development machines, simulators and other helpful engineering tools. You will learn about them, specifically the ROS2 development environment and tooling, in chapters 5 and 6.

After we have seen the relationship between the essential robot parts (vertical subdivision) and the layers in the robot software stack (horizontal subdivison), I want to conclude this section with a few more words on having different complementary views in software design. The seminal paper Architectural Blueprints - The 4+1 View Model of Software Architecture by Philippe Kruchten explicates 4 complementary views on software design:

- Logical view: structure of the functionality provided by the system.
- Process view: the system's runtime behavior.
- Development/Implementation view: breakdown of the system into components.
- Physical/Deployment view: physical realization of the system.

These 4 views are illustrated through a number of selected scenarios (the +1 view). I find these views very helpful in talking about complex software systems. Explicating the view under discussion avoids confusion, since they help to clarify the semantics of the arrows and boxes[72] we are usually drawing in architectural diagrams. Fig 3.1 focuses on the logical and process view. Fig 3.2 highlights the development/implementation view. Fig 3.4 is concerned with the physical/deployment view. Please note that while I do not strictly adhere the 4+1 separation of views, I will explicate the diagram's semantics.

Another powerful thinking aid is the C4 model by Simon Brown, which emphasizes the hierarchical structure applicable to software systems. It supports mentally zooming in and out to the right level of abstraction when discussing software design. The four Cs in C4 stand for these zoom levels:

- Level 1, System **C**ontext: relation of overall system to world around it.
- Level 2, **C**ontainer: high-level technical building blocks or subsystems.
- Level 3, **C**omponent: components within a single container.
- Level 4, **C**ode: code-level diagrams (e.g. classes and objects) of a single component.

The concepts from 4+1 and C4 will be used in the diagrams throughout the book.[73] This concludes our introduction to the robot software stack. The next section leads us deeper into robot software and into the various related software disciplines.

3.2 Robots as Software

Robotics is a highly interdisciplinary field. We need at least mechanics, electronics and software to create a robot. But even within the field of software, there are various specialized fields that are important for robot software. In this section, we will take a brief look at each of these fields and get to understand their relation to robotics.

3.2.1 Robots as Embedded Real-Time Systems

The first relevant area is software for embedded real-time systems. Embedded software is software that runs within devices that are not used as general-purpose computers.[74] Frequently these devices have limited computational and memory resources. Another common characteristic of embedded software is that users do not directly interact with it. Embedded software is often referred to as firmware.

Real-time software is software under time constraints. For real-time software it is not sufficient to give the correct result (logical correctness), it must also give the result at the correct time (temporal correctness). Because it is such a widespread misconception that real-time means fast, let me reiterate this point: Real-time software has to guarantee that computations produce a result at a specified point in time.[75] This is the same as stating that the software must exhibit temporal determinism.[76] You can refer back to our discussion on timing in section 2.6.1 of the previous chapter, especially figure 2.34, on the importance of timing.

Combining both aspects, the field of embedded real-time software concerns software that controls devices under real-time constraints. Three examples of industries employing many embedded real-time Software Engineers are aerospace, automotive and communications. A modern car contains on the order of 100 electronic control units (ECUs). Each ECU has its own processing unit running embedded real-time software. They communicate with each other via a bus system[77] and control everything in the car, from various engine parts to brakes to windshield wipers. The software is an integral part of the ECUs and the car user likely doesn't even know about their existence, not to mention directly interacting with them, thus the embedded aspect is evident. The real-time aspect is also easy to figure out, considering not only that fuel needs to be injected into the engine’s pistons at just the right time, but also that one really wants to have a guaranteed time between stepping on the brake pedal and the brake ECU actuating the brake.[78]

We have seen in the few examples above that embedded real-time software is a big and varied field. It has important intersections with robotics as well. As we discussed in the previous section, robot systems are commonly composed of devices running their own software. In many cases it is embedded real-time software that takes care of (low-level) sensor data acquisition and actuator control. I want to repeat what I already stated in the first chapter: Not all of robotics is about close-to-the-metal embedded real-time software. If your main interest is in the higher layers of the robot software stack, you don’t have to become an expert in embedded real-time software. Nevertheless, everyone working in robotics, no matter on what layer, benefits from understanding the basics of embedded and real-time.

Python is the language of choice for this book. Python is a popular modern multi-paradigm language with a clean syntax and comprehensive standard library. It is also widely used in robotics and AI. Unfortunately, it is ill suited for embedded real-time programming. The main drawbacks of Python for this use case are being an interpreted language and having a garbage collector.

In contrast to (pre-)compiled programming languages, interpreted languages, such as Python, are not translated to machine code before execution. Instead, the source code, or an intermediate representation of it, is directly read and executed by the language runtime, the interpreter. For a simple loop, such as

```
iterations = 100000000  # 10 million
sum = 0
for i in range(iterations):
  sum += i
print(sum)
```

the difference in performance compared to the same loop written and compiled in C++ is on the order of 100x, i.e. Python runs a hundred times slower.[79] Please note that this is an extreme example and there are workarounds in the Python ecosystem.[80] The point here is that there is a significant performance penalty for interpreted languages, in particular on resource constrained systems. This is also true for memory usage and memory is often scarce in embedded systems. While memory is in the range of tens of gigabytes (GB) for PCs nowadays, it is still common for microcontrollers to have only a few hundred kilobytes (KB) or a few megabytes (MB) of memory.[81]

Apart from resource utilization, which is mostly an embedded software issue, garbage collection (GC) poses a significant issue for real-time software. In programming languages without GC memory must be managed explicitly. It is the software engineer’s responsibility to manually allocate memory for use and to manually release it after use. Having to perform manual memory management is not only a nuisance, but also a constant source of bugs, e.g. memory leaks. In languages with GC memory management is automated. Memory is automatically allocated and released. The issue for real-time software is the question when these automated memory management operations are performed, especially when the GC becomes active and releases no longer required memory. In non-real-time software we are only interested in average performance. It does not matter when the GC becomes active and delays program execution for some time, as long as things are sufficiently fast on average.[82] However, in real-time systems we care very much about deterministic behavior of each and every program execution, not just the average case. The GC introduces unknown delays in program execution, thus rendering it difficult or impossible to provide real-item guarantees.

We will still use Python for the following examples on embedded real-time programming, because the concepts can be quite well expressed in Python. Just keep in mind that Python should not be your first choice when implementing these parts of the robot software stack.[83]

When developing embedded software, we must pay special attention to resource utilization:

- ROM (non-volatile memory)
- RAM
- CPU

Non-volatile memory (NVM) in embedded systems is commonly not provided by hard disks or solid state drives, but by small amounts of EEPROM or flash memory. The NVM is often not even a separate component, but part of the microcontroller chip itself. For historical reasons, NVM is commonly referred to as read-only memory (ROM) in the context of embedded systems, although it is not strictly read-only anymore.

While most Software Engineers are used to think about required storage space for program data, they usually do not pay much attention to the size of the program itself. And why would they? The code of even large programs usually does not exceed a couple hundred megabytes (MB) with disk space measured in terabytes (TB). In embedded systems the disk, i.e. NVM / ROM, has only a couple hundred kilobytes (KB) or a few MB. Thus, program size matters and sometimes needs to be optimized for.

Many Software Engineers also take memory (RAM) usage into account during development. For example, one should, in general, not read files entirely into memory, but process them in chunks to avoid running out of memory when processing large files. However, the resources in embedded systems are much more constrained than in PCs and servers. Data structures that one would keep in main memory, without thinking about it twice, can easily exceed the available KBs or few MBs of memory in an embedded system. Thus, memory usage must always be kept in mind and some familiar implementations might have to be adapted.

Finally, the CPU performance of embedded systems is at a much lower level than in PCs. To give you an order of magnitude understanding, let's have a closer look at floating point operations per second (FLOPS) as a performance metric: A (high-end) microcontroller has on the order of 0.1 GFLOPS, a (high-end) system on a chip (SoC) about 10 GFLOPS and a (high-end) desktop CPU about 100 GFLOPS. Put differently, there is a performance difference of 1000x (thousands) between PCs and microcontrollers. Thus, keeping an eye on efficient usage of computational resources by selecting the right approach, algorithms and data structures is essential in embedded systems.

Moving from the embedded aspects to the real-time ones, there are a number of topics to pay attention to:

- Worst-case execution time (WCET)
- Stochastic hardware performance
- Schedulers, task priorities and preemption
- Synchronization primitives and inter-process communication (IPC)
- Interrupts and timers

If we want to guarantee that the results of computations are available on time, we need to start by knowing the duration of these computations. While it sounds simple enough to measure how long it takes to run a certain function, there are some pitfalls you should be aware of. Because we must *guarantee* that the deadlines are *always* met, we do not care about the average execution time, instead we are interested in the worst-case execution time (WCET). Functions that contain conditional statements or loops might have a different execution time depending on their input or the overall program state. We are interested in finding the input data and program state that lead to the *worst*, i.e. longest, execution time. Once we have analyzed the program for the worst-case execution paths, it is still not sufficient to measure the execution time only once.

The reason is that most modern CPUs, including those in microcontrollers, have been optimized to provide high performance *on average*. We cannot go into the details of modern computer architecture here. In a nutshell, caches, pipelining, out-of-order execution, speculative execution and other optimizations greatly improve the average performance of CPUs, but they also lead to a huge difference - sometimes several orders of magnitude - between the fastest and the slowest execution of the same code. Unfortunately, it is practically impossible to analyse what execution times will occur from looking at the source code[84] and the CPU datasheet. The practical solution to get the WCET of a function is to run the function many times while putting the overall system into different states and under different loads. The result is an execution time histogram like the one shown in figure 3.5

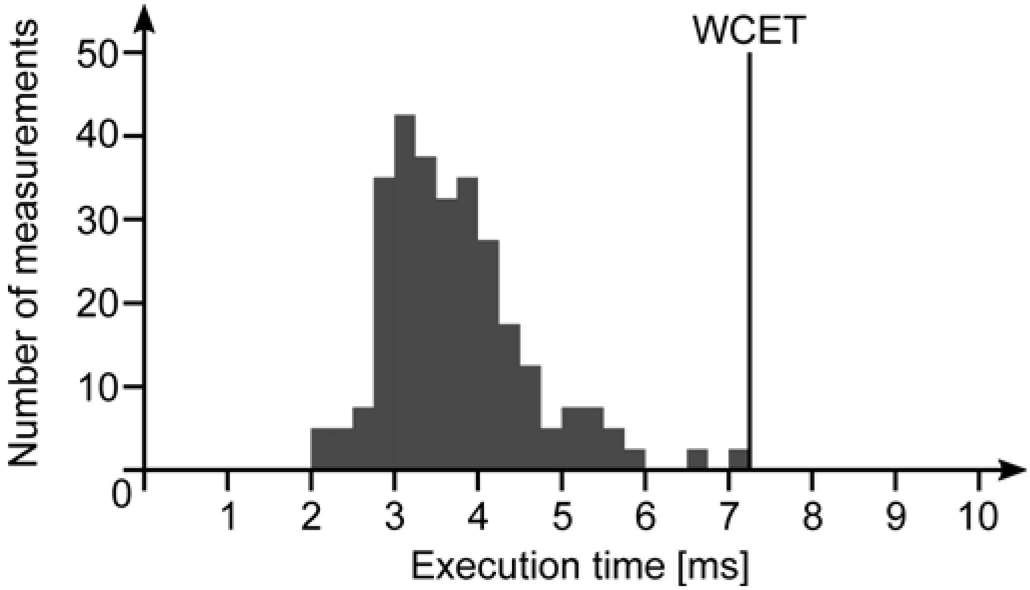
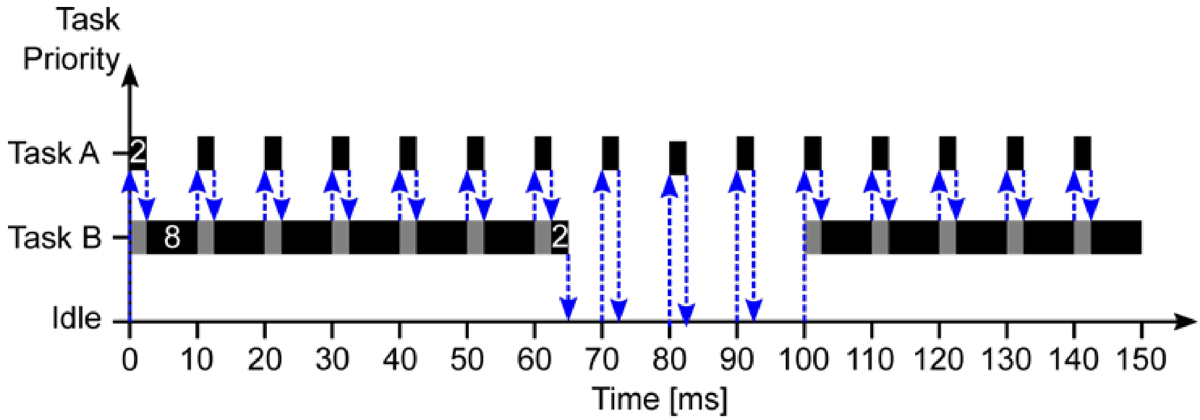


Figure 3.5 Histogram of execution times for worst-case execution time (WCET) analysis.

The horizontal axis shows execution times and the vertical axis shows the number of times this execution time was measured. While more advanced WCET analysis techniques also analyze the distribution of execution times, we limit ourselves to looking at the highest measured execution time. Although we cannot prove that a worse execution time cannot occur, doing a sufficient number of measurements under sufficiently diverse system states is good enough in practice.[85]

The next real-time consideration are schedulers, task priorities and preemption. Let's look at an example system with two tasks, task A and task B, in which task A must be executed every 10 ms and task B every 100 ms. In other words, task A has a periodic deadline of 10 ms and task B of 100 ms. Task A has a WCET of 2 ms and task B of 50 ms. We cannot simply run the tasks sequentially, as we would miss the deadline of task A 5 times while task B is executing. Instead, we need to pause task B every 10 ms, run task A, then continue running task B until the next period of A. Pausing a task and switching to another task is also known as preemption. The software that decides what tasks to run at what time is the scheduler. If a task is preempted by another one, we speak of one task having a higher priority than the other one. In our case, we need to assign a higher priority to task A relative to task B.[86] The resulting system behavior is illustrated in figure 3.6.



	Start time	Period = Deadline	WCET	Priority
Task A	0	10	2	2
Task B	0	100	50	1

Running Waiting

Figure 3.6 Example of preemptive scheduling for two tasks with fixed priorities.

Switching between tasks is not an instantaneous operation. The operating systems needs to store the state of the running task and restore the state of the next task. This process is referred to as context switching. If the switching occurs at a high frequency, the associated overhead and latency can become significant.[87]

Real-time operating systems (RTOSes) provide a suitable scheduler and corresponding task models for these real-time demands. Tasks are often not completely independent, but instead they have to exchange data with each other. Hence, RTOSes provide synchronization primitives, such as mutexes and semaphores, as well as inter-process communication (IPC) mechanisms, such as shared memory and message queues. You will learn more about these when we first make use of them in later chapters.

Finally, real-time systems require interrupts and timers. Interrupts are a hardware mechanism that interrupts the sequential execution of a program and execute a predefined function, the so called interrupt service routine (ISR). Interrupts can either originate from within the system or externally. In either case they signal that some kind of event has occurred. Hardware timers can be configured to cause an interrupt, either after a defined amount of time or periodically with a defined frequency. For example, the scheduler uses timers to regularly interrupt the running program and execute itself in the ISR to determine whether to continue running the current program or to run a different one. Another example for the use of interrupts is avoiding cyclic checking of inputs for changes, so called polling. Instead of occupying precious CPU resources with regularly polling an input, such as

```
while input.status() != True:  # wait for input signal to become true
  pass
do_something()  # then run do_something()
```

we could - given hardware support - register an ISR for the input change interrupt

```
# run do_something() when input changes to True
input.on_change(True, do_something)
```

When an interrupt is triggered, the CPU and operating system have to switch from executing the current task to running the ISR. This process has a non-zero duration. The duration from the interrupt being triggered in hardware to the ISR being executed is known as interrupt latency. Given what we learned about the (non-)determinism of modern computer architectures above, the latency is not always exactly the same. This deviation from the average duration is called jitter. See figure 3.7 for an illustration.

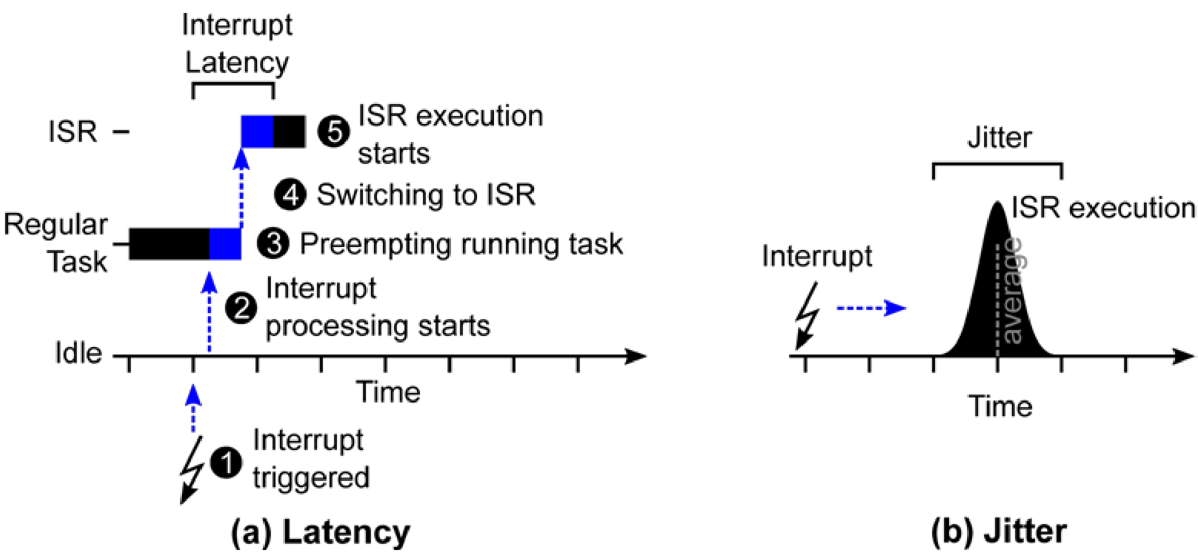


Figure 3.7 Illustration of interrupts their latency and jitter.

This concludes the brief introduction to embedded real-time systems and their relation to robotics software. In the same vein, we will learn about distributed systems in the next section.

3.2.2 Robots as Distributed Systems

Distributed systems are a collection of independent networked computing devices that can be treated as a single coherent system.[88] Let's look closer at each part of this definition. Having independent networked computing devices entails that each device is complete and can execute software independently of the other devices. Furthermore, the devices can communicate with each other. Being able to treat this collection of devices as a single coherent system means that we have a way to work with the devices that goes beyond working with each device directly, i.e. we can perform operations on the collection-level.

This is a bit abstract, let's look at a concrete example. We will assume we have two servers, one running a database and the other running a web backend server. The web backend is configured to connect to the database and use it for data storage and retrieval. Is this a distributed system according to our definition? We clearly have a collection of two independent networked computing devices. But can we treat them as a single coherent system? When accessing the web service, it interfaces with the database in the background and provides us the data we are interested in, without us having to deal with - or know about - the database. Sounds like treating our two devices like a single system. However, this is only user perspective. What about the developer view? From the developer's point of view, there are clearly two distinct systems, the database and the web backend. As developers, we must login to the two servers and run different commands on each to administer the database on the one server and the web backend on the other server. There is no common view of the two devices.

Compare this to having two servers, each one running an identical base software. This base software provides a mechanism to deploy the database and the web backend without requiring to directly interact with the individual servers. In this scenario, we might not even have to know whether there are two servers, one server or 100 servers behind the interface we are using to interact with the system. In other words, we treat the collection of devices as a single coherent system. Hence, our definition of a distributed system is satisfied.

The difference between the two scenarios was the base software that enabled us to work at the collection-level instead of the level of individual devices. What we have referred to as base software is commonly known as middleware. The middleware plays a similar role in distributed systems that an operating system plays for single computers. The middleware manages resources and provides a unified API to them. Furthermore, the middleware makes it easier to implement distributed systems, or, more precisely, to implement applications on top of distributed systems. This is similar to writing an application on top of an operating system that provides the abstraction of a file instead of having to directly deal with disk sectors and the disk controller.[89] Having a middleware, i.e. an API at collection-level, is what distinguishes a collection of computing devices from a distributed system. Figure 3.8 visualizes the difference between having a middleware and not having one using our example above.

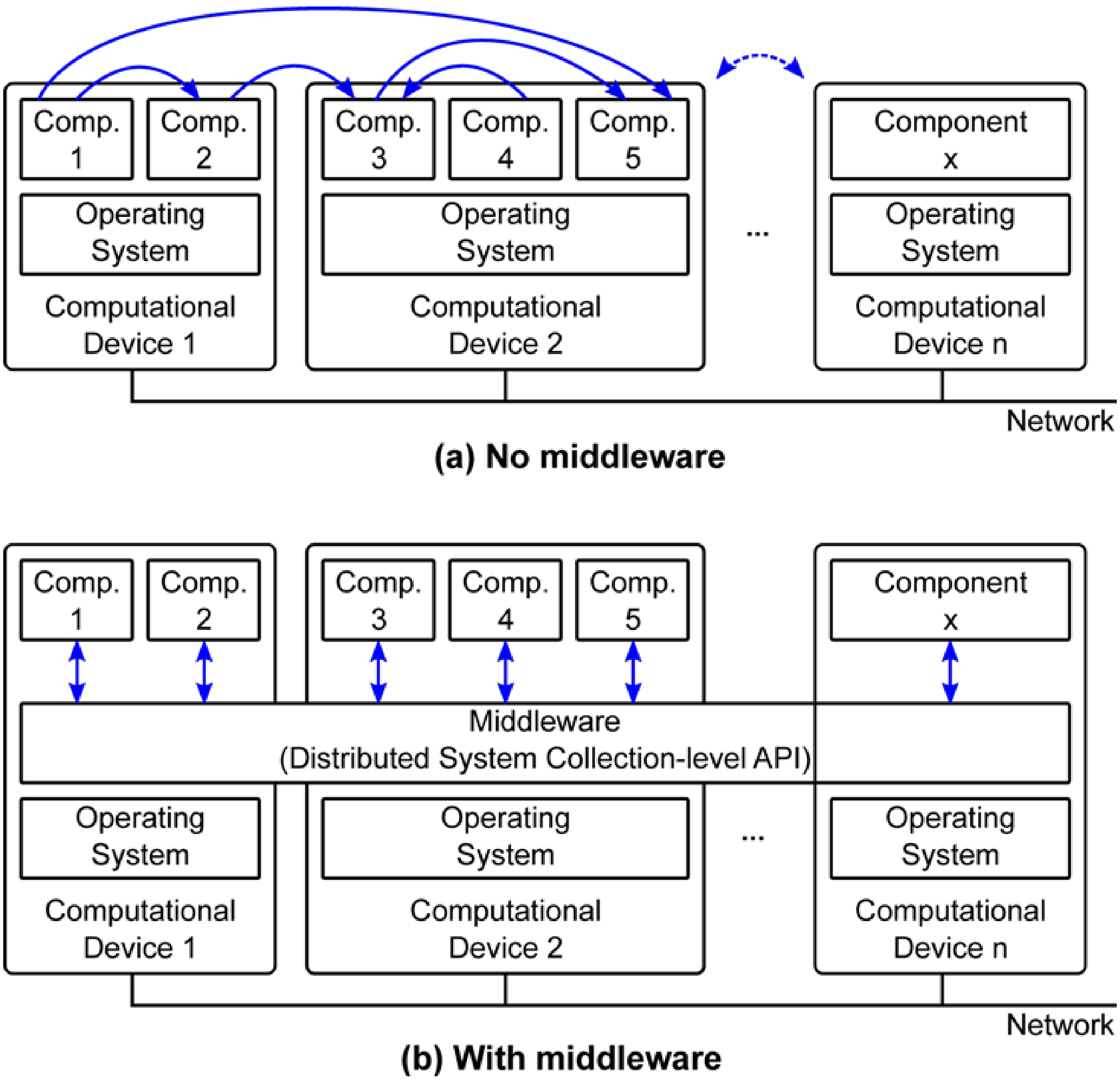


Figure 3.8 (a) A collection of computational devices; (b) A distributed system with a middleware providing a collection-level API. The arrows symbolize the interface and component directly accessed by components.

Essential middleware functions include

- Interoperable and scalable communication mechanisms
- Composition of distributed services
- Transparent distribution of resources
- Tools for deployment, administration and introspection/observation

I will not delve deeper into these functions here. We will discuss them in chapter 5 when learning about the Robot Operating System (ROS) as specific example of a middleware. Instead, I want to make you aware of the fallacies of distributed computing. These fallacies, i.e. *wrong* assumptions or *false* believes, are[90]

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Put differently, don't treat a remote procedure call (RPC) like a regular function call within a program. The same goes for not assuming that sending out data from one component equals another component receiving this data. Neither will the data arrive immediately[91] nor is it guaranteed to arrive at all. We also shouldn't make the assumption that just because the sender is up and running, the same is also true for the receiver. In summary, although a middleware provides us with a collection-level API, we still need to keep in mind that this API behaves inherently different from APIs within a program and that it has additional failure modes.

We should always pay attention to throughput, scalability, overhead and other metrics that characterize the level of performance we can expect from a specific middleware. Put more generally, we want to understand the quality characteristics[92] of the middleware, not just its functionality. In the context of robotics, the temporal behavior and in particular the *guaranteed* temporal behavior is crucial, i.e. the real-time aspects. It is helpful to distinguish between real-time communication used by the middleware and real-time aspects of the overall middleware.

When talking about real-time communication, the terms fieldbus and industrial network are important. We will use the following pragmatic definitions: A fieldbus is any communication mechanism that provides real-time guarantees. An industrial network is a fieldbus used for control of automation equipment, including robots. An important subcategory of industrial networks is industrial Ethernet. Industrial Ethernet is a type of industrial network that uses the widespread Ethernet standard (IEEE 802.3) as its physical layer.[93] Having a real-time communication channel, a fieldbus, is the basis for building a real-time middleware, but the two are not identical.

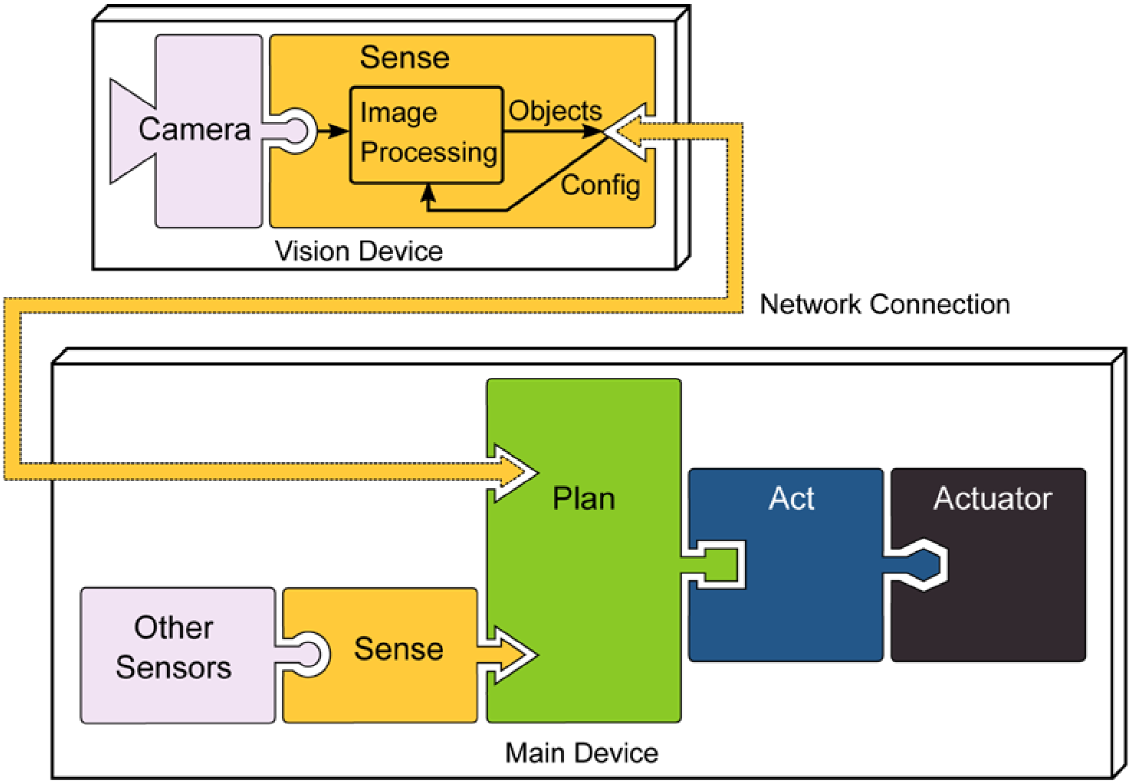
A real-time middleware provides support for building distributed real-time applications. This means the essential middleware functions listed above are based on mechanisms with specified real-time guarantees. For example, the communication would not be based on best-effort. Instead it provides transmission and queuing priorities, quality of service (QoS) options, bandwidth allocation and/or traffic shaping mechanisms. Furthermore, high resolution time synchronization is a common functionality provided by real-time middleware to support building distributed real-time applications.

It is worth mentioning that the layered nature of communication protocol stacks, as captured in the OSI model, allows to build distributed real-time systems even without a real-time middleware. As long as the temporal behavior can be guaranteed end-to-end, it is not necessary to use a real-time middleware or even to use an existing fieldbus. It is just much easier - and highly advisable - to build on top of these instead of effectively reinventing them along with building the actual application. As in non-distributed real-time systems, we only have to provide deterministic timing for the parts of the distributed real-time system that actually require them. In most cases when talking about a (distributed) real-time system, we do not mean that everything is real-time in this system, but only that some parts of it operate under real-time conditions - and that these parts have to obey real-time constraints in order to work properly.

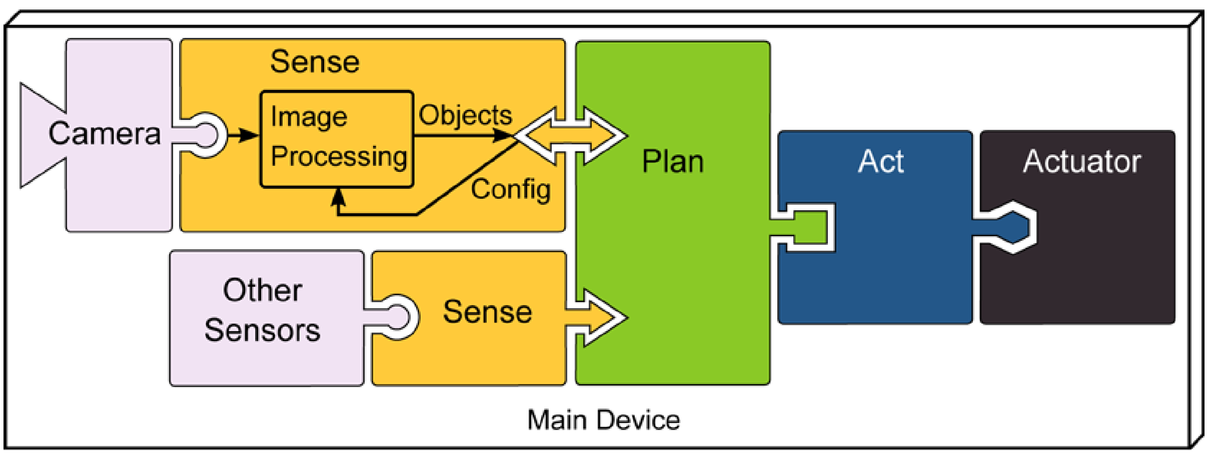
Our example from the previous section, a modern car containing a large number of independent electronic control units (ECUs), also serves well as an example of a distributed (real-time) system. Since it is not feasible, neither economically nor mentally, to manually program the communication between a hundred ECUs, the automotive industry defined a middleware for in-vehicle networks: AUTomotive Open System ARchitecture (AUTOSAR). The AUTOSAR middleware makes it easier to have software components communicate with each other, independently of whether they run in the same ECU or run in different connected ECUs.

Presumably the relevance of distributed systems and middleware for robotics has already become clear from our discussion in this section. Just like cars contain a number of independent networked devices, to increase reuse, decrease cable harnesses and in general improve system modularity and flexibility, robots today are also often composed of interconnected hardware modules. For this reason, you should be familiar with their basics as robot software engineer. While there is no substitute for hands-on experience, we have already mentioned the most relevant aspects here. Let’s conclude by looking at a concrete robotics example of distributed systems.

To address reusability in different robots, we want to build an integrated vision device that contains the camera sensor and performs image processing. Our vision device receives commands specifying the type of objects to detect in the image from the robot’s main computer. It sends back information about the detected objects and their positions. At the same time, we anticipate building a low cost robot at a later point in time. In this case, To save on hardware costs, we want to consolidate all software into the main computer. The camera sensor is then directly connected to the main computer, which also executes the image processing component. Figure 3.9 shows these two system architectures.



(a) Camera and Image Processing deployed in separate device



(b) Camera and Image Processing deployed in main device

Figure 3.9 Two different system architectures and deployments of the Image Processing component.

The question is how we should implement the image processing, so it can run either on the vision device or directly on the main computer. As Software Engineers, many ideas come to mind. We could have two branches in our version control system, one for the remote - as seen from the main computer - image processing and one for local processing. We could implement both options and have a configuration file or command line argument to select how the image processing connects to the rest of the system. There are many further solutions. But do we really want to implement something specific for each software component that has to be flexible in deployment within our (distributed) robot system? Could we delegate solving deployment flexibility and instead focus on what we really want to do (implementing robot applications)? No, we don't want to deal with this topic for each component. Yes, we want to solve it in a generic way. Hence, we integrate a (robotics) middleware into our software stack.

Our middleware of choice provides a publish-subscribe communication pattern that renders communication between components independent from their deployment. Publish-subscribe, or pub-sub for short, introduces the notion of topics.[94] A topic is a named logical channel that receives messages from senders, called publishers, and delivers these messages to receivers, called subscribers. The publisher does not need to know about the subscribers and the other way around. They also do not need to care about whether they run on the same computer or on different ones. Both only need to refer to the topic (by its name). The middleware then takes care of actually delivering the messages from publishers of a topic to the subscribers of this topic. Figure 3.10 illustrates the basic principle.

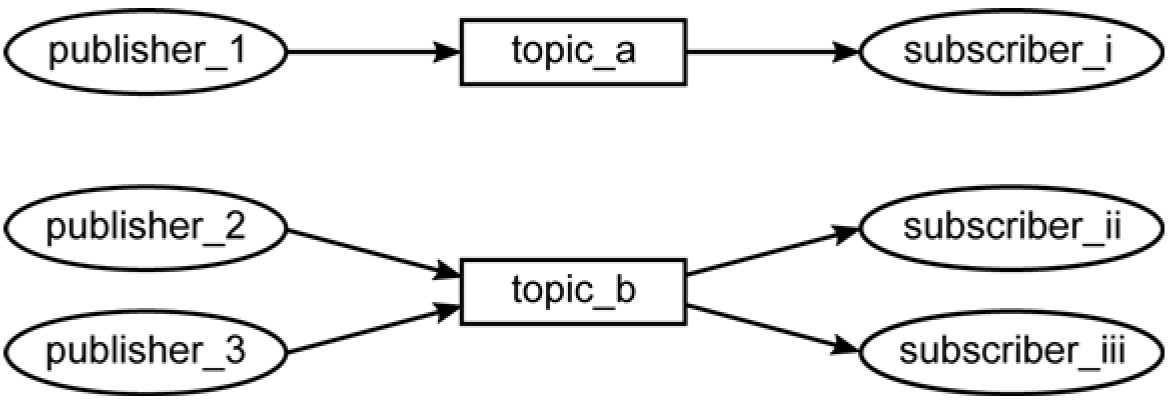


Figure 3.10 The publish-subscribe (pub-sub) communication pattern. In general, each topic can have multiple publishers and multiple subscribers. Topics can have a specific message type.

Since pub-sub is one of the core mechanisms in the ROS middleware, we postpone further details until chapter 5. Let's conclude by looking at a code snippet from the image processing component and from the planning component using the pub-sub mechanism of the middleware.

We start with the image processing component sending out information about detected objects. During initialization we create a `Publisher` instance, named `objects_pub`, that will publish messages of type `ObjectList` on the topic `detected_objects`. In the sensing loop we fill an `ObjectListMessage` with all the detected objects and publish it via `objects_pub`.

```
# During initialization
objects_pub = Publisher(topic='detected_objects', type='ObjectList')
...
# In sensing loop
objects_msg = ObjectListMessage()
for obj in detected_objects:
  objects_msg.push_back({'id': obj.id, 'pose': obj.world_pose, ...})
objects_pub.publish(objects_msg)
```

In the respective planning component, we create a `Subscriber` to topic `detected_objects` with type `ObjectList` and register `camera_objects_callback` as callback function. The callback function will be executed each time a message is published to the `detected_objects` topic and the message content is passed to the callback as argument.

```
# This callback function is executed for every message on subscribed topic
def camera_objects_callback(objects_msg):
  print(f'The camera detected {len(objects_msg)} objects.')
  ...

# During initialization
camera_objects_sub = Subscriber(topic='detected_objects', type='ObjectList',
                                callback=camera_objects_callback)
```

Note that the topic name and message type match between the image processing's publisher and the planning component's subscriber. It is required and sufficient that topic name and message type match in order for publisher and subscriber to communicate with each other. This concludes our introduction to distributed (real-time) systems and middleware. Our next topic is mechatronic systems.

3.2.3 Robots as Mechatronic Systems

The field of mechatronics is best explained by an Euler / Venn diagram, such as the one shown in figure 3.11.

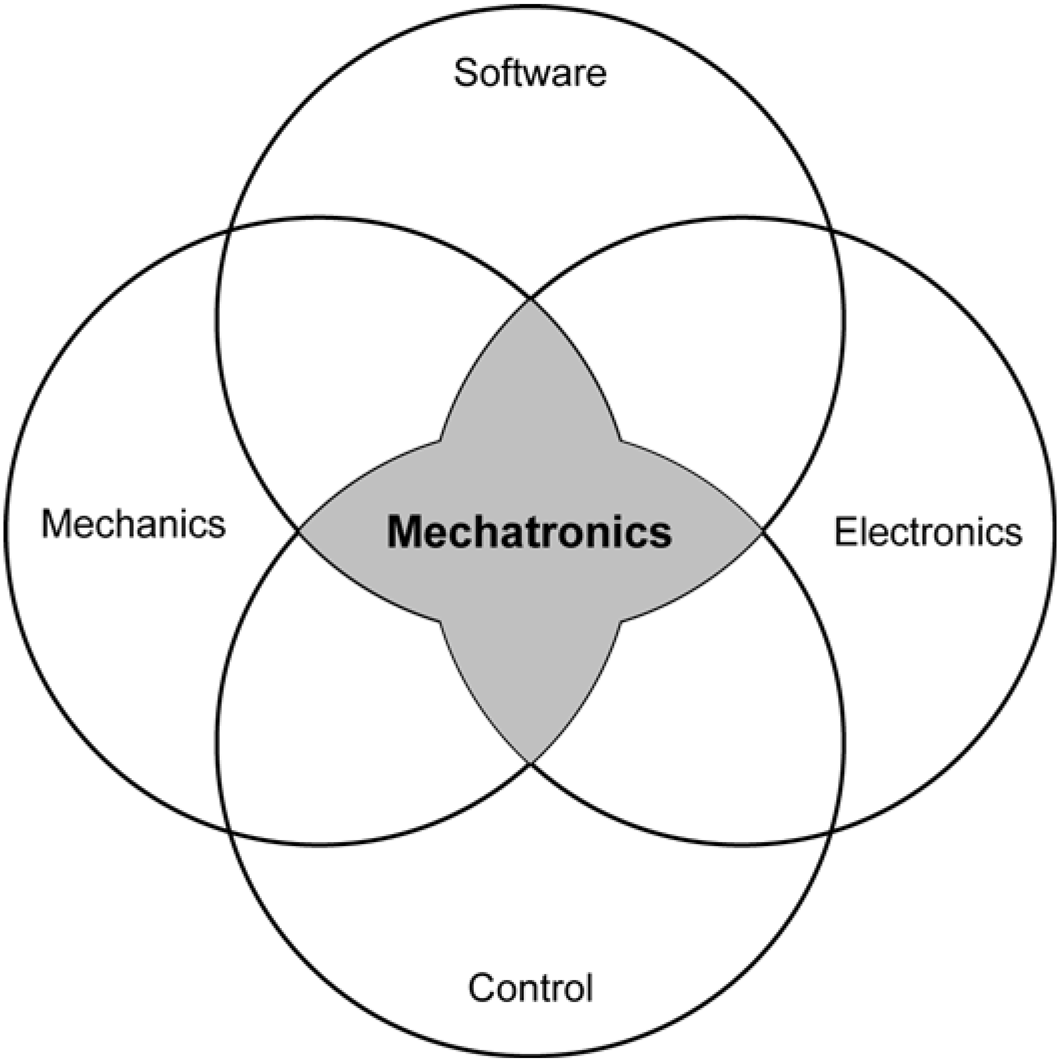


Figure 3.11 Mechatronics is an interdisciplinary engineering approach comprising mechanics, electronics, control and software.

Although there are some variations in which fields are considered part of mechatronics, the usual list comprises

- Mechanics
- Electronics
- Control
- Software

If at this point in the book this list sounds familiar to you, then you are correct. Robots fundamentally are mechatronic devices. However, not all mechatronic devices are robots. Hard disks are a great example of mechatronic devices, but they are definitively not robots. The six essential robot parts (cf. figure 3.1) can also be used to describe mechatronic systems. Still, robots are usually more general-purpose, more complex and exhibit more behavioral variability than ordinary mechatronic systems.

A term closely related to mechatronics is cyber-physical system (CPS). The main difference is that cyber-physical systems are networked and form a distributed system. Hence, robots are often not only used as an illustrative example of mechatronic systems but also of cyber-physical systems.

Apart from the interdisciplinary part, mechatronics can also be said to deal with three flows:

- Matter
- Energy
- Information

In different systems, or different parts of the same system, usually one of these flows dominates. For example, a conveyor belt or a fleet of mobile transportation robots is primarily about the flow of matter. The electric grid has energy as its main flow. A camera system is primarily about information flow. Still, in most systems all three flows can be identified. The conveyor belt requires energy to power the motors and information about the desired speed. The camera also requires energy to operate. Relevant insights can be gained from identifying and thinking about these flows in robot systems.

The main reason I want to highlight the robots as mechatronics systems view to you are the intriguing tradeoffs possible in mechatronic systems. Let's return to the humble hard disk as an example. Hard disk drives (HDDs) consist of a spinning magnetic disk, the platter, an actuated arm that positions the read-write head over the rotating platter and the disk's electronics.[95] Figure 3.12 shows how these components are arranged in an HDD.

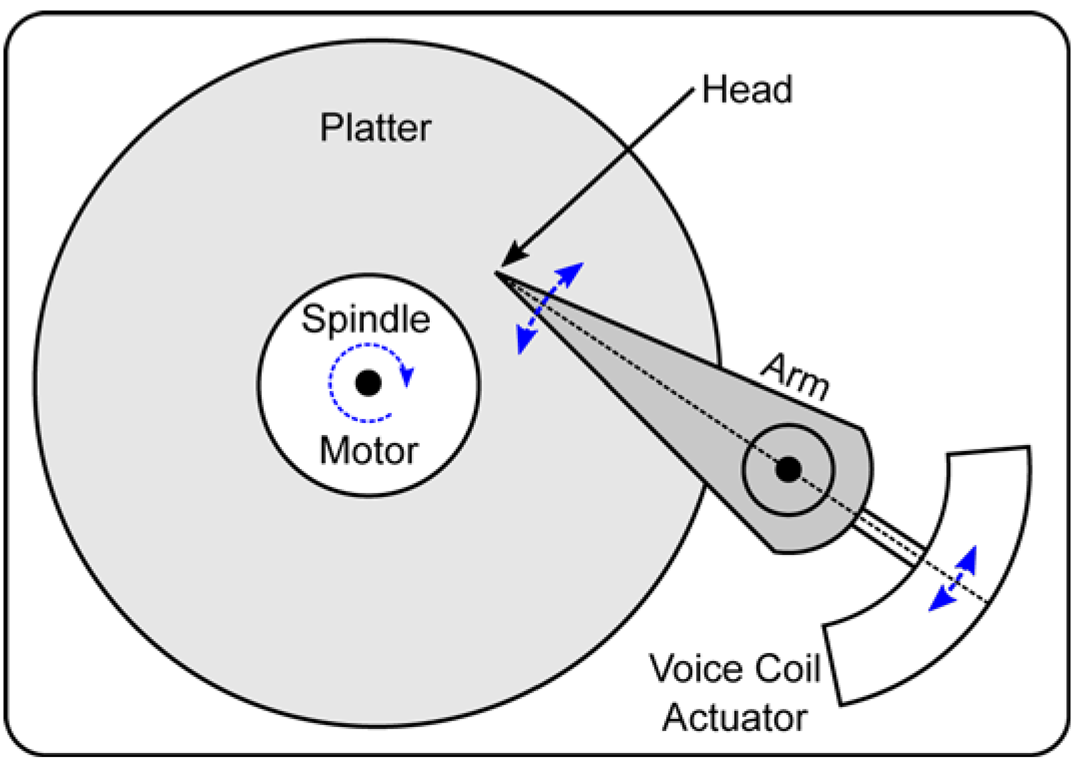


Figure 3.12 Major components of a hard disk drive (HDD). Top view.

The platters are actuated by an electric motor, usually a BLDC, and rotate at a constant speed of typically 5400 or 7200 RPM. The arm is controlled by a type of electric actuator knows as voice coil actuator.[96] The head reads and writes the bits, which are stored as varying magnetization on the platter. The bits are organized into concentric rings on the platter known as tracks. In current HDDs each track has a width of about 100 nm. 100 nm is small - very small. For comparison, a human hair or sheet of paper is approximately 100,000 nm thick. You might ask why I'm highlighting this in an age where chips have a structure width of less than 10 nm. The big difference here is that we have to *mechanically* position the disk head with such high accuracy - and do it rapidly, too. This is a feat close to impossible without turning to mechatronics. Even if it could be accomplished purely mechanically (or purely electromechanically) HDDs would have a cost in the millions instead of being inexpensive commodity devices.[97]

This fascinating capability, to achieve extremely high performance from relatively inexpensive (electro)mechanics by compensating for their lack of precision through closed-loop control and clever algorithms, is what you should take with you on your journey into robotics (software).[98] In mechatronics, a fortiori in robotics, one should always take a wholistic view, i.e. apply system thinking. Finding the right allocation of robot capabilities between mechanics, electronics and software or rather how the combination of them is able to provide the desired capabilities is a critical success factor.

With this we finish our short survey of mechatronics and move to the final perspective on robots in this chapter: embodied intelligence.

3.2.4 Robots as Embodied Intelligence

Already in the beginning of chapter 1, we determined that robots interact with the physical environment and that they are more general-purpose than other types of machines. This can also be stated by saying robots have a body and exhibit intelligent behavior. They are an instance of embodied intelligence. We cannot talk about embodied intelligence without talking about intelligence. Yet, as much as I'd enjoy elaborating on the philosophical question of what intelligence is and whether it makes sense to talk about intelligent robots, it would likely not contribute much to your journey into robot software development. We stick to a functionalistic definition: Intelligence measures an agent's ability to achieve goals in a wide range of environments.[99] The second part, embodiment, emphasizes that interaction with the environment is mediated by the physical makeup of the entity interacting with the environment, i.e. the entity's body. Taken together, embodied intelligence investigates how having a (specific) body affects (a specific form of) intelligence.[100]

There are a number of valuable concepts in embodied intelligence when applied to robotics. We will encounter many of them in later book chapters. However, because these topics will occur in isolation and in a specific context, I want to point out how they all belong together in this short overview.

Let's start with a fundamental, literally philosophical difference of how we see a robot system. The traditional view[101] in robotics and artificial intelligence is to see intelligence as computation. This means that everything internal and external to our robot[102] is represented as symbolic information. Information is the input to computations (algorithms) that produce more information as their output. Some of the input information and output information is causally linked to the robot's body and the physical environment. From this perspective, we can disregard the specific sensors and actuators that couple the computations to the real world.

At the extreme end of the computational perspective, we don't distinguish between dealing with a virtual environment and a physical one. We assume that the robot and its physical environments can be represented like a software object, i.e. as a class instance in an object-oriented programming language. Given a known state of the object, we are able to exactly determine which operations we must perform on the object, e.g. calling its methods with the right parameters, in order to get a desired outcome. Correct computer programs behave deterministically after all.[103] For example, we do not need to know anything about the implementation of Python's `list` class nor about the computer running the Python interpreter in order to make it sorted by calling `sort()`. In the exaggerated computational view, we are in the same principal situation, if our goal is to program a robot to sort socks into pairs. While it is more complex to create the `robot` class with its `sort_socks()` method, it *can* be done in the same manner - it is just a matter of performing computations on data.

The embodied view sees intelligence as emergent from the sensory-motor interaction process between robot and environment. It emphasizes that we cannot make a clean distinction between computation, body and environment. Put differently, we *cannot* fully abstract away from the robot's body and its environment - they always remain entangled. Mapped to our six essential robot parts, this means there cannot be clear boundaries between sensors, sensing, planning, acting, actuators and environment. This can not only be seen as a factual statement, but also a prescriptive statement, i.e. one *should* not (try to) draw clean boundaries between them (because it will not work).

In practice, when doing robotics as an engineering discipline, not as a method in the natural sciences, the two views are much less contradictory than they first appear. The intelligence as computation view focuses on what happens within symbol processing systems, i.e. computers. The embodied intelligence view highlights other aspects required for physical systems behaving intelligently in physical environments. We do not need to decide between adopting one view or the other. Rather we should take both views into account when building and programming robots.

The first topic I want to highlight is the so called symbol grounding problem. Take the symbol mug as an example. How is this symbol connected to the (coffee) mug standing next to me on the table? This is the question behind the symbol grounding problem, how do symbols get their meaning or how do they refer to something in the real world. Obviously, we need to solve this problem for a barista robot. How is it supposed to fill our mug with fresh coffee otherwise? Using what we already learned in the previous chapter, we could build many different robot systems that are able to put coffee into a mug. From having a switch that is pressed when we put the mug in a certain place to using sophisticated computer vision to recognize the mug, we can find various working solutions. However, does a pressed switch really mean mug (present)? Consider someone accidentally - or maliciously - puts a glass where the mug should go. It would still trigger the switch and the barista robot would pour hot coffee into a glass that might shatter as a result. Did we err in naming the digital input signal for the switch `mug_present`? Yes and no.

This is where thinking about a topic that very much sounds like a theoretical problem at first glance, symbol grounding, is really helpful in practice. Because using a camera and computer vision can also not eliminate the issue in our example. Consider what would happen when putting a picture of mug in place of the real mug or when using a mug made of ice. We can easily think of a myriad of further scenarios in which the robot system would fail to act intelligently and end up with spilled coffee. The challenge we *always* have to deal with is that symbols, e.g. variables in our program, cannot refer to the real world unambiguously. We always have to make certain assumptions about the robot and its environment for the system to work properly.[104] If our robot can correctly deal with all *relevant* situations and environments[105], we did a good job - even if the robot would behave entirely unintelligent in other situations or environments.

Another interesting concept is morphological computation. Instead of performing (complex) calculations, we build the robot in a way that these calculations are not necessary. For example, grasping a stiff object with a gripper also built from stiff material is much more difficult compared to having a softer gripper.[106] It is possible, at least in principle, to calculate the exact gripping points and exact gripping forces to pick up hard objects with stiff grippers, but it is difficult and error prone compared to using a softer gripper. Substituting the former with the latter would be considered morphological computation.

Active perception[107] is the next topic of interest. While we usually think of perception as passive, information flowing from the environment to the robot, it also has a more or less active component. Let’s use a camera sensor as an example. First, we decide where to point the camera, this is already an active aspect of the sensing process. Second, we can use the robot’s actuators to influence the environment that we are sensing in a way that we can get additional information about it. A common example of this is object segmentation in a cluttered environment. Without a lot of prior knowledge about the shape of objects, it is often difficult to tell where one object ends and the other begins, especially when the objects have a non-uniform color and overlap each other in the camera image. One way to segment the objects is to gently poke at some part of an object and see what else moves along with the poked part. To give you an illustrative example: Consider a pile of tangled (computer) cables. How long would you have to stare at the cable tangle until you identify what constitutes one specific cable? Depending on the size and messiness of the pile and the cables' color(s), it will take you long - very long. Now consider how much easier the task becomes by allowing to grab one of the cables and pull it out. Third, a special kind of active perception is system identification and calibration. We use prior knowledge combined with sensing and acting to infer properties of the robot system or the environment.[108]

Devising an alternative to the sense-plan-act (SPA) structure, which is also the guiding principle for structuring robot systems in this book, is a further contribution by embodied intelligence to our mental robotics toolbox. This alternative structure is known as subsumption architecture. Under this paradigm, we would not build up symbolic representations of the (entire) robot and the (entire) environment, but instead rely on a much more direct coupling of actions to sensor inputs and a more decentralized control approach. Furthermore, we would break down the intended overall robot behavior into layers of independent sub-behaviors. The higher level behaviors can (partially) inhibit the lower level behaviors, hence subsumption, but in principle all behaviors are always active. For example, in a mobile robot, a low level behavior could be to avoid running into obstacles and a higher level behavior could be to move towards a target destination. If an obstacle is encountered along the way, the low level behavior would perform reactive (local) obstacle avoidance while the higher lever one continuously commands moving towards the (global) destination. When this approach was first proposed, it was in direct opposition to the dominating entirely plan-driven robot control approach. Nowadays most robot architectures are a mix of reactive behavior and plan-driven control. This is also true of the approach we take in this book. We do not lift everything into a fully symbolic representation on which we perform planning, but also integrate sub-symbolic methods into the robot system. Still, we follow the approach that top level behavior is engineered explicitly, instead of emerging from the interaction of independent reactive behaviors.

I want to finish our discussion of embodied intelligence by mentioning three additional topics from this field that we will come back to in the last chapter of this book (chapter 19): developmental robotics, evolutionary robotics and swarm robotics. Developmental robotics aims to build mechanisms that enable a robot to acquire new (sensory-motor) skills over time by learning from its actions and how its body and the environment react to them. A common example is to have a legged mobile robot learn how to walk by (directed) trial and error instead of manually engineering a gait pattern. Evolutionary robotics applies evolutionary algorithms to the design and/or behavior of robots. Instead of an engineered solution, the evolutionary algorithm generates a set of potential solutions (population) that are then automatically evaluated (often in simulation) with respect to optimization goals (fitness). The best solutions (survival of the fittest) are further iterated on (mutation, crossover), until a feasible solution is found or fixed number of iterations (generations) have been completed. Finally, in swarm robotics the goal is to build a number of robots that as a whole exhibit collective intelligence. The focus is often on decentralized control, i.e. not having a central instance that coordinates the individual robots in detail.[109] Instead the overall behavior emerges from the individual interactions of the robots with each other and their environment.

After you have learned about the embedded real-time, distributed, mechatronic and embodied intelligence aspects in robot systems, you now understand the bigger picture and are ready to get hands-on developing the various robot subsystems and robot software components. We end this chapter with an overview of the robots for which we will develop these software capabilities.

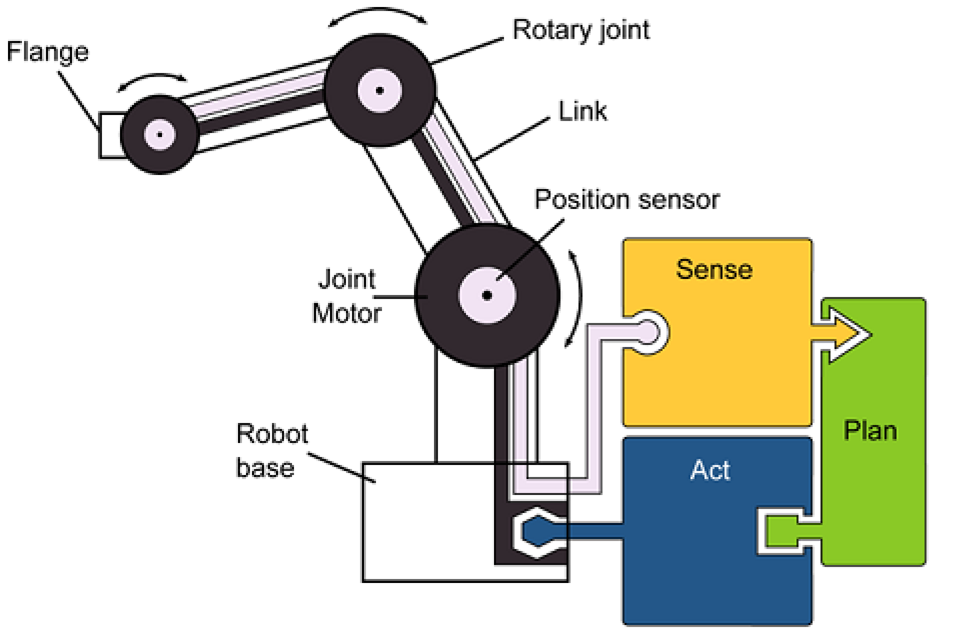
3.3 Introducing AER: The Abstract Example Robot

You have already learned so much about different types of robots and ways of structuring robot systems that we can keep this section quite short. We will have a look at four robot templates, each serving as an abstract example robot (AER) for fundamental robot types. The idea here is that these AERs are concrete enough that we don't have to constantly repeat their basic structure and constituent parts throughout the book. At the same time, they are abstract enough that we can easily specify more detailed variants[110] when we need them.

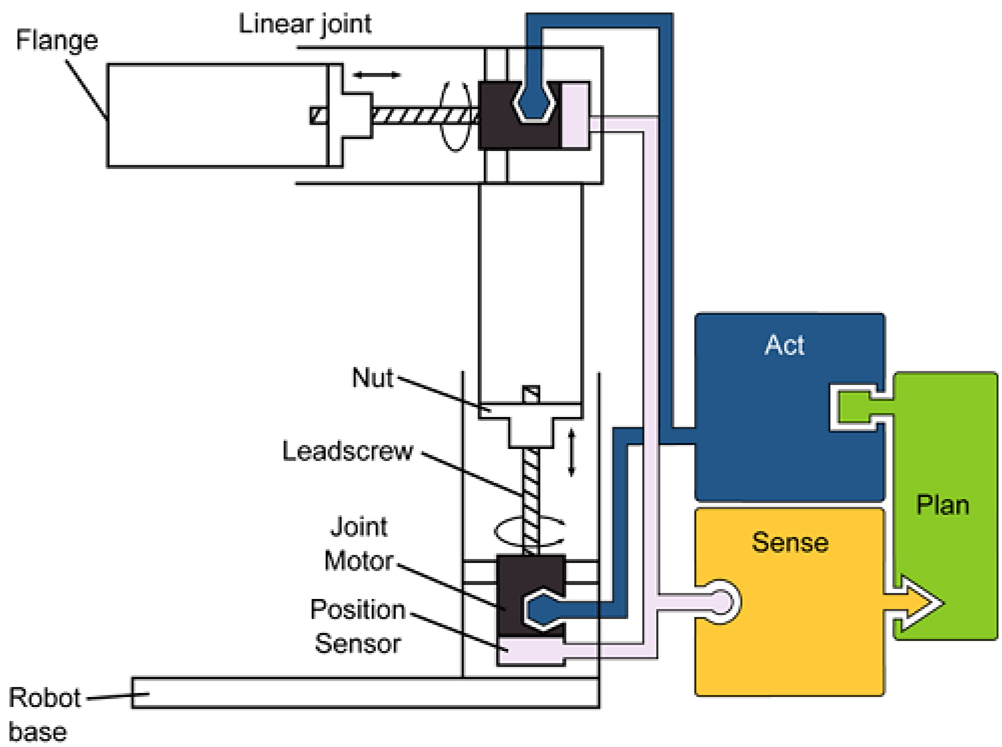
For example, the (industrial) manipulator AER, i.e. $AER_{industrial}$, is always a robot manipulator. Hence, it is always stationary and its purpose is to move objects or tools attached to its flange.[111] However, if I don't explicitly specify that it is a six axis articulated robot, it could also be a three axis Cartesian robot, a SCARA robot or another kind of kinematic.[112] Similarly, unless it is stated that it has a two finger parallel gripper attached, it might as well not have any end of arm tool (EOAT) mounted or it might have a humanoid five finger hand. The point being that if an aspect is not specified, it does not matter for the respective discussion. Therefore, no matter whether your mental image of $AER_{industrial}$ is a massive 2000 kg industrial robot or a small 3D printed DIY manipulator built from three RC servo motors, as long as you have a manipulator in mind, the topic being discussed should be applicable.

3.3.1 $AER_{industrial}$

Following the order from chapter 2.2, we start with manipulators. Figure 3.13 shows two variants of $AER_{industrial}$.



(a) AERindustrial with rotary joints (3 DoF, articulated)



(b) AERindustrial with linear joints (2 DoF, Cartesian)

Figure 3.13 Two variants of AER$_{industrial}$.

The following applies for all AER$_{industrial}$ variants:

- It is a manipulator.
- All joints are actuated by their own motor and each has a position sensor.[113]

Unless specified otherwise, $AER_{industrial}$ has the following properties by default:

- Its pose has six degrees of freedom: three translations and three rotations.
- The pose refers to the flange if no tool is mounted and to the tool center point (TCP) otherwise.
- Rotary joints have a range of motion of less than a full revolution, i.e. less than 360 degrees or 2 radians, i.e. only the single-turn position must be considered.
- The joints (incl. gearboxes) and links are perfectly stiff, i.e. there is no deflection due to load.
- The robot has no exteroceptive sensors, such as cameras.

3.3.2 AER_{mobile}

Our next robot type are ground-based mobile robots. Figure 3.14 depicts AER_{mobile} with a selection of potential sensors and potential actuators.

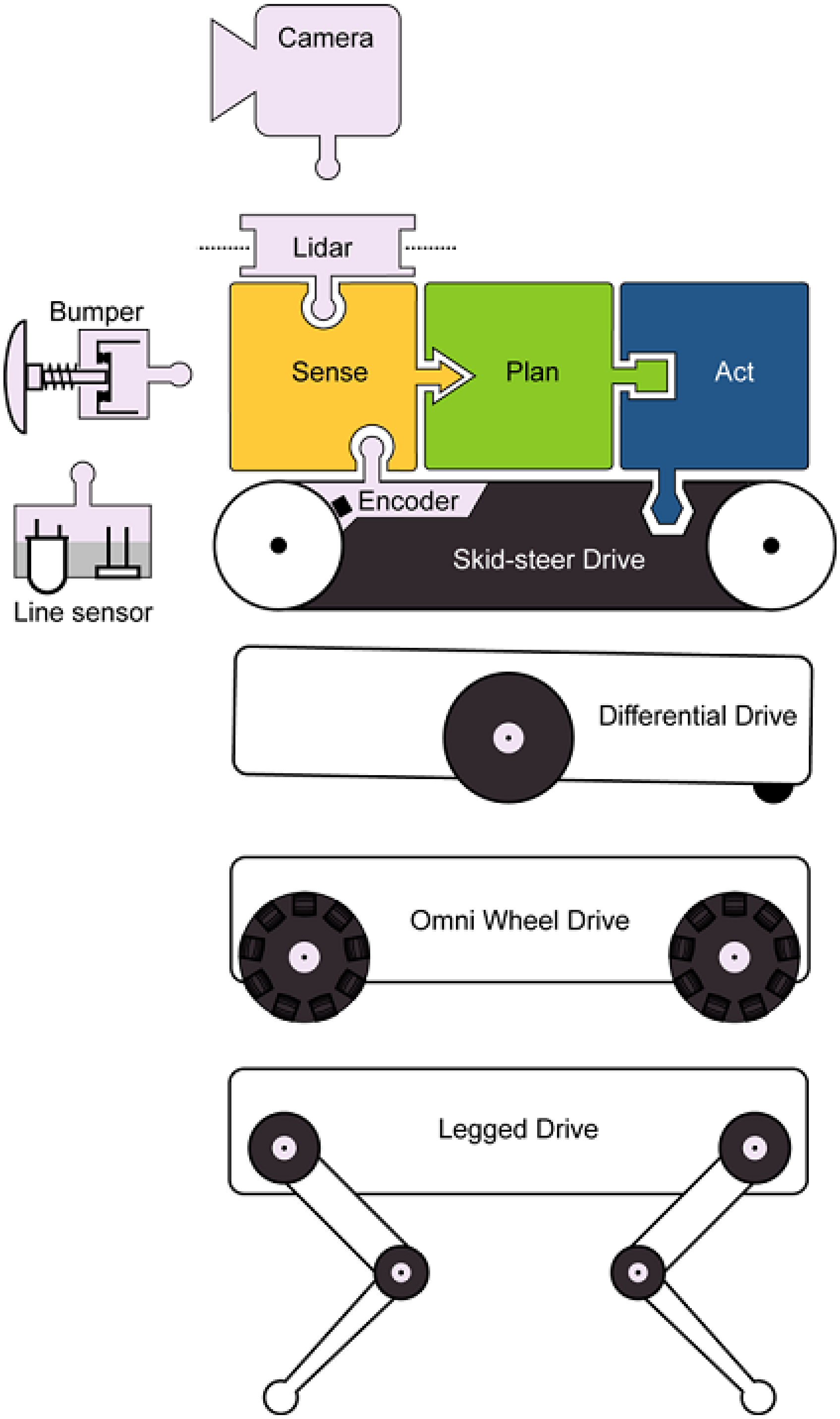


Figure 3.14 Variants of AER$_{mobile}$ as combination of environment sensors and drive mechanisms.

The following applies for all AER$_{mobile}$ variants:

- It is a ground-based mobile robot.
- It has no manipulator.[114]
- The drive actuators can be position controlled and have position sensors.
- It has at least one exteroceptive sensor, such as bumper switches, line following sensors, Lidar sensors or cameras.

Unless specified otherwise, AER_{mobile} has the following properties by default:

- It can only move on a 2D plane, e.g. on a building floor or on flat terrain.
- Its pose has three degrees of freedom: two translations and one rotation.
- It is holonomic / omnidirectional. It can move in all three degrees of freedom, i.e. it can move backward and forward, left and right, turn on the spot and simultaneously move along all three degrees.[115]

3.3.3 AER_{drone}

We continue with mobile robots, *flying* mobile robots. In this robot category, we will limit ourselves to helicopters and multirotors aka quadcopters aka drones.[116] The reason is that other types of aircrafts, e.g. planes, cannot stand still, i.e. they cannot hover in place. This limitation results in significantly different control schemes and planning algorithms compared to ground-based mobile robots and flying mobile robots, which can hover in place. Figure 3.15 illustrates the AER_{drone}.

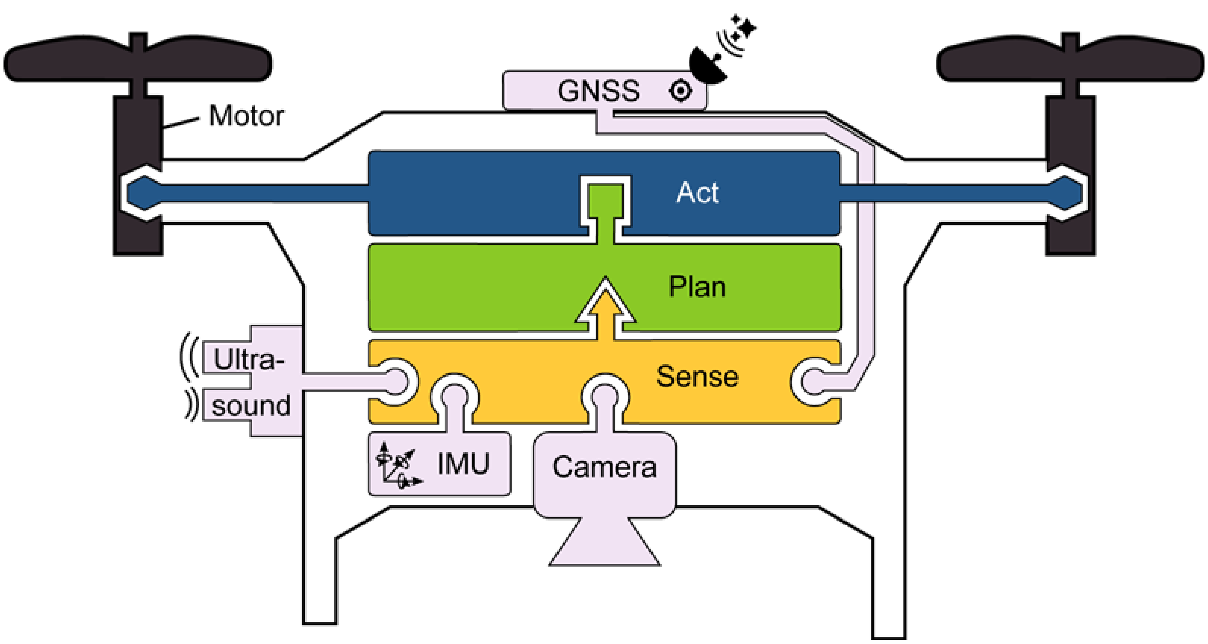


Figure 3.15 The AER_{drone} with different (optional) sensors.

The following applies for all AER_{drone} variants:

- It is a flying mobile robot.
- It has an inertial measurement unit (IMU) as proprioceptive sensor. The IMU at least provides three axis linear acceleration and three axis angular velocity data.[117]
- It has at least one extroceptive sensor that provides (relative) position information with respect to a fixed environment coordinate system.

Unless specified otherwise, AER_{drone} has the following properties by default:

- Its *stable* pose has four degrees of freedom: three translations and one rotation. Although all six degrees of freedom (3 translations, 3 rotations) are relevant to control motion, we assume that the only rotation axis that can be freely controlled while hovering is the yaw axis. The other two rotational axis, roll and pitch, are zero while holding position.
- It is holonomic. It can independently move in all four degrees of freedom.

3.3.4 AER_{united}

Our final main AER is a mobile manipulator. We call it AER_{united} because it brings together manipulation and mobility capabilities in a single robot. Due to combinatorics, a large number of AER_{united} variants are possible when combining $AER_{industrial}$ (figure 3.13) and AER_{mobile} (figure 3.14). Figure 3.16 shows one of these many variants.

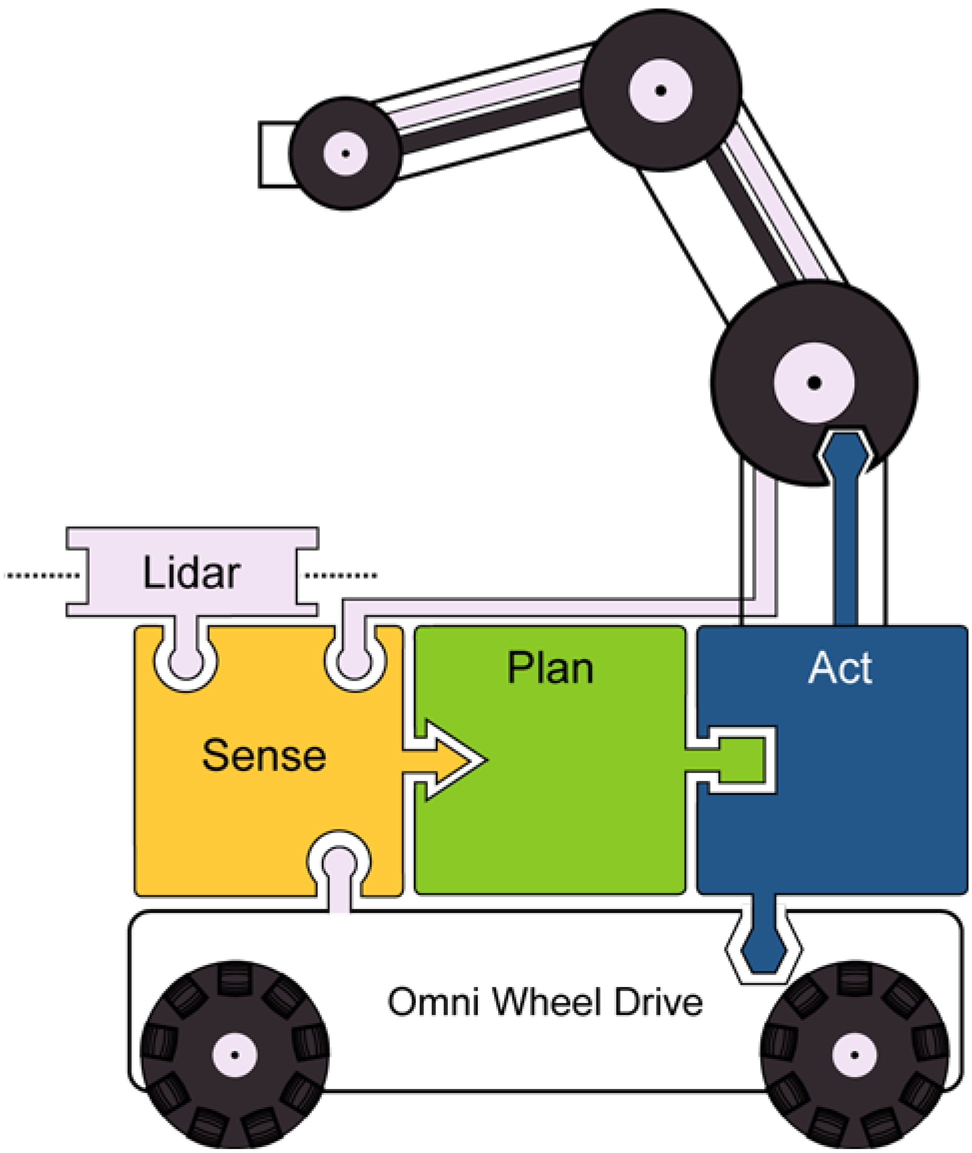


Figure 3.16 One variant of AER_{united}.

The properties of AER_{united} result from combining the properties of AER_{mobile} as the mobile base with those of $AER_{industrial}$ as the manipulator mounted on the base. Thus, I will not repeat all of these properties, but only mention potentially unclear ones.

The following applies for all AER_{united} variants:

- It is a mobile manipulator.
- It can move itself through the environment and utilize its manipulator, also concurrently.

Unless specified otherwise, AER_{united} has the following properties by default:

- It has a single manipulator.
- It has at at least one extroceptive sensor in order to sense the environment for mobility. No additional extroceptive sensors, e.g. for manipulation, are assumed.

3.4 Summary

- The complete robot software stack reaches from low-level real-time embedded software to core robot algorithms running within the robot to external large scale distributed cloud software.
- Robot software utilizes a large number of general-purpose software libraries, frameworks and services. Beyond these common components, there are a variety of robotics-specific software components.
- The vertical subdivision of the robot software stack into layers can be combined with the horizontal subdivision of the five essential robot parts. Each part (sensor, sense, plan, act, actuator) has an internal structure that mirrors the structure of the overall robot software stack.
- Robot systems can be monolithic or composed of multiple independent subsystems. Coordinating a number of robots to work towards a common objective constitutes a robot fleet.
- A robot's software stack quickly becomes large and complex. Good software design is of essence to manage robots' inherent complexity. It is advisable to adopt complementary architectural views, e.g. 4+1 and 4C, of the robot system, instead of trying to combine them all into a single diagram.
- The field of embedded real-time systems deals with controlling resource-constrained devices under real-time constraints. Robots - or their subsystems - often fall into this category. Worst-case execution times (WCET) analysis, priority scheduling, synchronization primitives with real-time guarantees and interrupts are some of the embedded real-time topics highly relevant for robot software.
- Distributed systems are a collection of independent networked computing devices that can be treated as a single coherent system. Middleware provides functions to simplify communication, composition, resource distribution, deployment and analysis of networked devices by providing a collection-level API. Yet, the fallacies of distributed computing must be kept in mind when dealing with these systems. The publish-subscribe communication pattern is one way to create software components with deployment flexibility. Fieldbus protocols and industrial networks provide real-time communication mechanisms between networked components. Most robots today are build as distributed systems, especially those using ROS.
- Mechatronics is an interdisciplinary engineering approach comprising mechanics, electronics, control and software. Adopting a mechatronics view helps in finding optimal tradeoffs in allocating capabilities among them. Not only can such a wholistic approach improve robot cost and performance, but it is often mandatory to find robust solutions in the first place.
- Embodied intelligence highlights many important aspects of the interaction between the robot's physical makeup, its body, and its environment. These include the problem of grounding symbols, i.e. having symbols refer to real world objects, avoiding difficult computational problems by solving them in the robot's hardware (morphological computation), recognizing the active element in perception and utilizing sub-symbolic reactive behaviors as a complement to symbolic computations.
- The four abstract example robots (AER), $AER_{industrial}$, AER_{mobile}, AER_{drone} and AER_{united}, are robot templates that simplify creating specific robot systems.

4 Robot Motion 1: Geometry

This chapter covers

- **Working with objects in space: shapes and poses**
- **Coordinate transformation trees**
- **Common geometry representations and how to operate on them**

In the previous chapter we concluded the big picture overview of robotics. In this chapter we start to zoom in on specific parts of the robot system. We stated before that robots are programmable general purpose machines that directly interact with the real world to perform tasks in it. A less complete, but still illustrative description depicts robots as general purpose motion machines. Motion is the topic of this and the next chapters. In other words, we focus our attention on the acting part.[111] At the same time, the concepts that you will learn here, together with their concrete realizations and the mathematics behind them, are also important in other parts.

There are four foundational concepts involved in robot motion: geometry, kinematics, dynamics and control. You will learn about them in this order. This first chapter, out of four chapters on robot motion fundamentals, is all about geometry. Section 4.1 introduces geometric notions such as dimensions, shapes, poses, pose parameters and coordinate systems. Coordinate transformation trees facilitate specifying poses in the right frame of reference, as we will discuss in section 4.3. In section 4.4 we discuss common geometry representations and learn about important mathematical tools such as vectors and matrices. Section 4.5 provides you some practical advice for handling geometry in your robot software.

The chapter concludes (section 4.6) with a number of short standalone exercises in Python.[112] I encourage you to take the time to utilize your newly acquired skills right away in these exercises. Gaining confidence with geometry and the associated topics from this chapter is time well invested, as the next chapters build on them.

4.1 Objects in Space

The term *geometry* might bring back memories from long gone school math lessons. And, maybe, positive or negative emotions along with them. No matter how it is for yourself, I want to tell you up front that we will not have geometry lessons here. Our focus is on what is needed to develop robot software, not on passing math exams.[113]

The purpose of geometry is to describe space, objects in space, their properties and the relations between them. Common notions are sizes, distances, positions, angles and dimensions. Common objects are points, lines, planes, triangles, rectangles, other polygons, circles, pyramids, cuboids, other polyhedrons, cylinders and spheres. The description we are looking for is a formal one that can be manipulated in a computer program. We require geometry in order to model our robot and its environment in a way that allows to define what we want (not) to happen.

For example, we want the robot to `move()` and `navigate()`.[114] We must be able to express the targets in a precise numerical manner. We also have to encode geometric information about the robot and the environment to plan and execute the robot's actions. The robot should not collide with itself or the environment, but it should nevertheless physically interact with other objects in a purposeful way.

An object is defined by two geometric aspects:

- shape
- pose

The *shape* includes information about the space occupied by the object, such as size and outline. Examples are "a line of 2 m length", "a circle with a diameter of 0.3 m" or "a cuboid of size 0.5 x 2 x 0.2 m". Shape specifications such as "a large dog", "a small building" or "the Eiffel tower" are, of course, also valid. However, they are ill suited for our purposes due to being natural language expressions that are hard to operate on in software as well as due to their lack of precision.

The *pose* tells us the location of the object relative to some reference or coordinate system.[115] Everyday examples are "at the intersection of park avenue and oak street", "on the moon", "10 m above ground" and "in the second drawer from the top". While each of these location specifications may well serve its purpose to locate an object in a given context, it is very hard to work with them in software. What we are looking for is a more uniform way to specify all possible poses by a set of numeric values that make it easy to calculate with them, e.g. get a distance given two poses. For this reason, examples we are interested in are rather along the lines of "at (4, 2)" or "at position (5, 2, 0) with orientation (1.57, 0, -3.14)".[116]

As the examples of suitable and less suitable specifications for shapes and poses have illustrated, for robot software, we are interested in geometric descriptions that are universal, precise and can be easily (and efficiently) processed in software. Before continuing to learn about these suitable representations, it is worth noting that when robots interact with humans, it is highly important to create a mapping between the way humans usually perceive, structure and communicate about objects in space and the robot's internal representation of them. At least I'd rather ask my future robot assistant to "hand me the big flat screwdriver from the workbench" than asking it about "the object with a midsection of cylindrical shape with 5 mm diameter and 15 cm length that ends on one side in a wedge … from the planar surface at (6.3, 4.7, 0.8) in room coordinates". The topic we are touching here is semantics. We will get back to it in the final section of this chapter when discussing robot and environment models.

Let's now look at a few examples of how we can represent the shape of geometrically simple objects, so called geometric primitives. Here we are only interested in the object's shape, but not its pose (position and orientation). See figure 4.1 for an illustration.

- A line's shape is fully determined by its length, i.e. it has only a single parameter.
- A circle's shape is fully determined by its radius (or diameter), i.e. it also has only one parameter.
- A triangle's shape can be determined in a number of ways, all of which uniquely define its shape. All of these definitions consist of three parameters. Examples: Length of all three sides. Length of two sides and the angle between them. Two angles and the length of the side between them.
- A rectangle's shape is fully determined by the length of two adjacent sides, usually named width and length.
- A cylinder's shape is fully determined by the circle and the line along its height, i.e. two parameters.
- A sphere's shape, just as a circle's shape, is fully determined by its radius (or diameter).
- A cuboid's shape is fully determined by the length of its three sides, or in other words the width and length of the rectangle at the bottom and the line along its height.

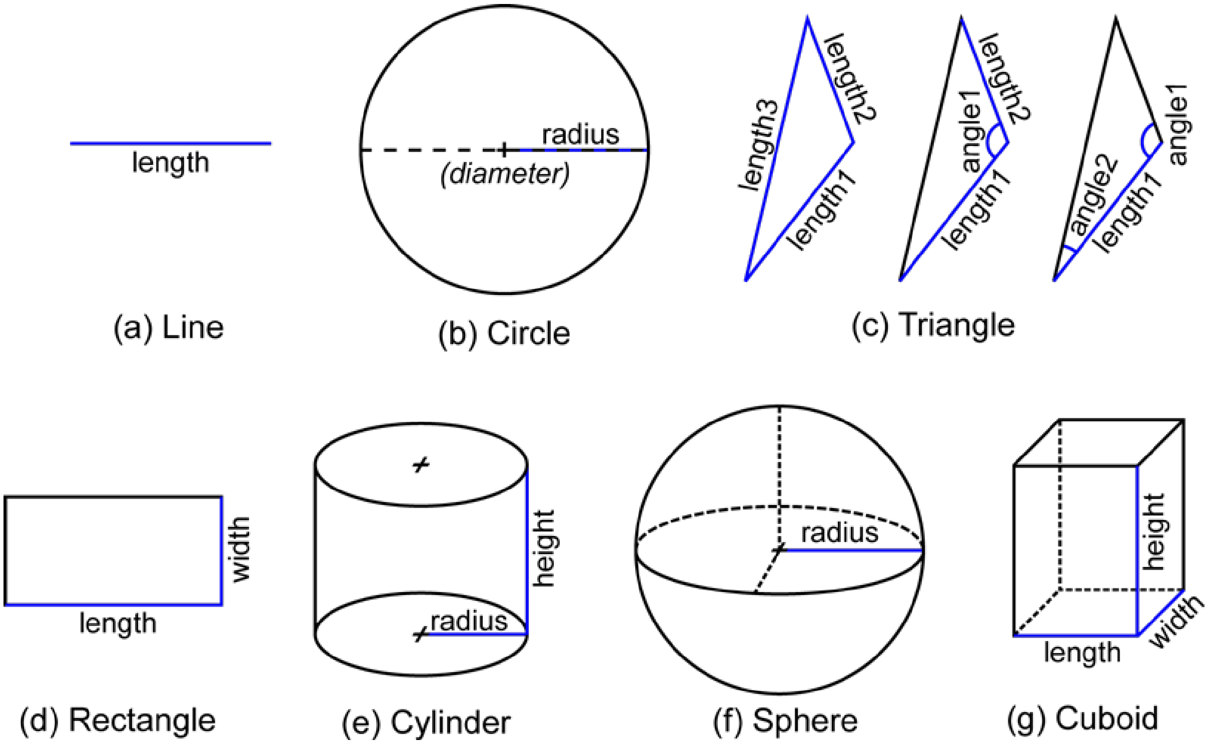


Figure 4.1 Illustration of the parameters that define the shape of simple geometric objects.

Now that we know some methods to represent shapes, we can continue with the representation of poses.[117] The pose of an object fully determines its whereabouts, i.e. both position *and* orientation. Whenever we want to specify the pose of an object, our first consideration must be the dimensions of the space in which we want to locate the object. A dimension can be defined as the minimum number of values, or coordinates, required to specify any point within it. Because robotics deals *a lot* with objects and their motion in various spaces, keeping the different concepts straight will save you many troubles. Unfortunately, there is no universally accepted terminology in robotics that prevents all misunderstandings. We can still minimize issues as much as possible - in this book and beyond - by adhering to the following distinctions.

- The term *dimension* when used in relation to objects and spaces refers to the number of independent coordinates required to specify the position - and only the position - of a point in space. We abbreviate a one dimensional space or object as “1D”. A two dimensional one as “2D”. Finally, a three dimensional one as “3D”. The space of physical reality is a 3D space. No higher dimensional spaces, in the sense defined here, appear in this book.
- The term *pose parameters (PP)* refers to the number of independent values required to specify the pose - position and orientation - of an object in space. It is simply the number of independent position coordinates plus the number of independent orientation coordinates.[118]
- The term *degree of freedom (DoF)* refers to the number of (independently actuated) joints in a robot. We will only use this term when talking about joints, joint coordinates and joint space.

This might sound a bit abstract at first, but quickly becomes second nature when working with robot software. A few examples will already make it much more tangible.

The most simple object and space is a point on a line (cf. figure 4.2(a)). This space has only one dimension (1D). Because it is not meaningful to talk about the orientation of a point, there is no orientation. Hence, we can specify the pose with one parameter leading to pose parameters of one (1 PP): x.

Let's look at a point on a flat surface next (cf. figure 4.2(b)). This space has two dimensions (2D) and also two pose parameters (2 PP): x and y. When we replace the point by an object, orientation becomes relevant (cf. figure 4.2(c)).[119] The space is still two-dimensional (2D), but the pose now has three parameters (3 PP): two position parameters (x, y) and one orientation parameter (theta, θ).[120]

This brings us to the highest dimensional space, three-dimensional (3D) space. A point in this space (cf. figure 4.2(d)) has three pose parameters (3 PP) for the three position values: x, y and z. An object in 3D space in addition has three orientation values (cf. figure 4.2(e)): roll, pitch, yaw. Hence, in general, an object in 3D space has six pose parameters (6 PP): three position parameters (x, y, z) and three orientation parameters (roll, pitch, yaw).[121]

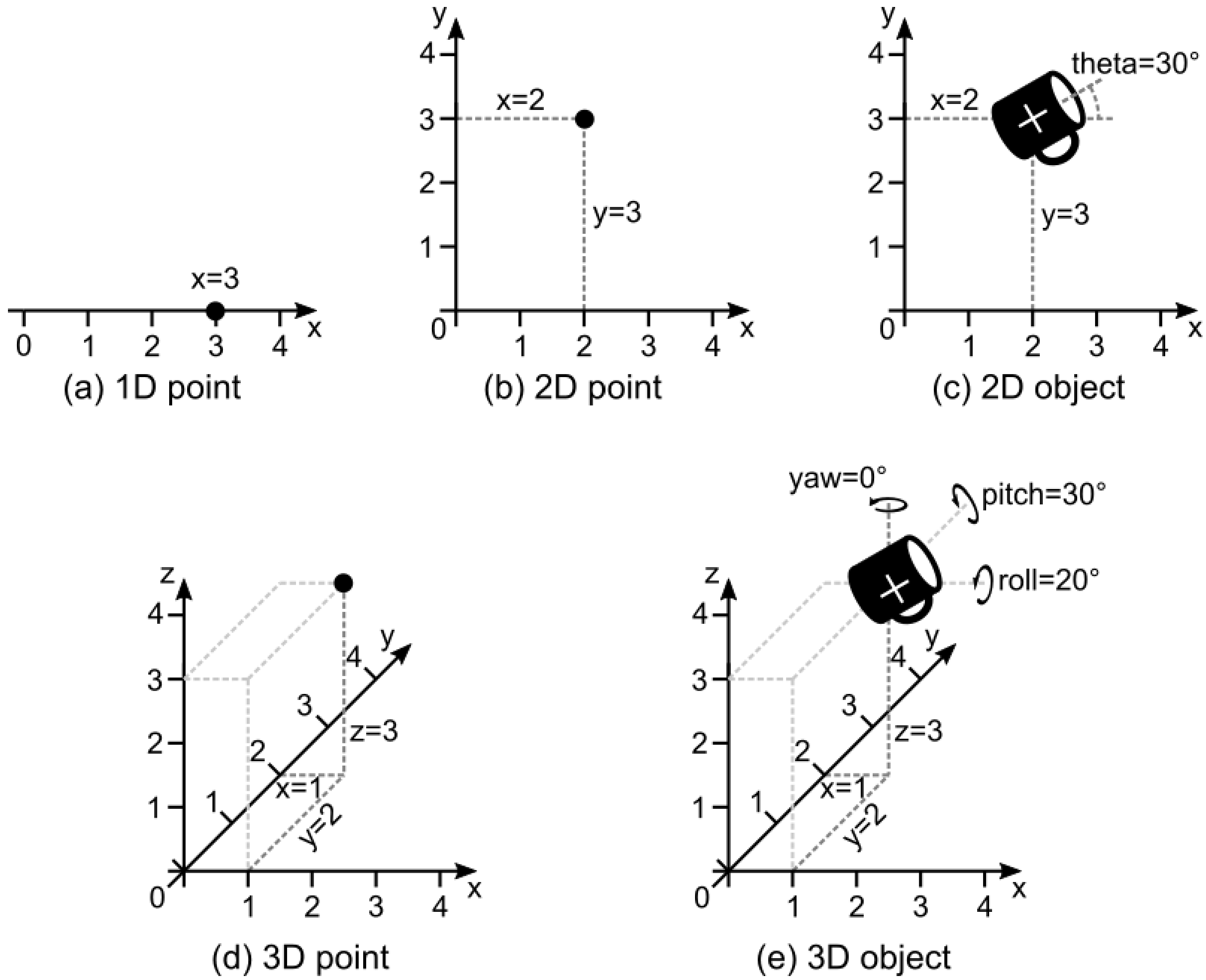


Figure 4.2 Objects in spaces with different dimensions and pose parameters.

Table 4.1 gives some examples of dimensions, pose parameters (PP) and degrees of freedom (DoF) for different robot types. The goal is to make the point that dimension, pose parameters and degrees of freedom are independent concepts. You are not expected to already know what values they take on for each robot.

Table 4.1 Relation between dimensions, pose parameters and degrees of freedom for different robot types.

Robot Type	Dimension	Pose parameters (PP)	Degrees of Freedom (DoF)
Ground-based mobile robot in warehouse	2D	3 PP	Depending on drive type (e.g. differential: 2 DoF; Mecanum: 4 DoF; active steerable: 8 DoF)[122]
Industrial robot	3D	6 PP	Most commonly 6 DoF
Collaborative robot with 7 joints	3D	6 PP	7 DoF
Mobile manipulator with two 7 DoF arms on omnidirectional mobile base	3D	15 PP: 6 PP per arm and 3 PP for base	18 DoF: 7 DoF per arm and 4 DoF for base
Quadrotor drone	3D	6 PP	4 DoF

The dimension and pose parameters have a task-specific element to them. For example, describing the ground-based mobile robot as 2D with 3 PP, makes it impossible to model the situation where it is flipped over. We need to consider the environment and situations in the environment, in terms of objects and their poses, that we want to describe in our robot software.

Before continuing, let's briefly summarize what we already learned about objects in space together with our terminology. Geometrically speaking, an object is defined by its shape and pose. A pose is specified in a space of a certain dimension and consists of a number of values that together provide information about position and orientation in this space. Pose parameters (PP) is the number of values in a pose (position values plus orientation values). For robots, the degree of freedom (DoF) refers to the number of (independently actuated) joints.

The only thing left for this section is to learn about coordinate systems, also known as *frames of reference* or just *frames*.[123] For our purposes, it is sufficient to define a coordinate system as a set of numbers that uniquely determine the pose of an object in a given space. Referring back to figure 4.2, we have already seen 1D, 2D and 3D coordinate systems. They are defined by the reference lines, also known as axes, of the coordinate system. The 1D coordinate system in figure 4.2 had only one axis, the x axis. The 3D coordinate system had three axes: x axis, y axis and z axis.

The coordinate systems we have already seen in this book are *right-handed Cartesian coordinate systems*. This means all axes have a right angle between them, i.e. they are orthogonal to each other. The term right-handed refers to the relative orientation of the coordinate system's axes. There is a very helpful mnemonic for it: the human hand, in particular the right hand. As shown in figure 4.3, we identify the x axis with the index finger, the y axis with the middle finger and the z axis with the thumb of the right hand. When these three digits are held at right angles to each other, as shown in the figure, they form a right-handed Cartesian coordinate system.

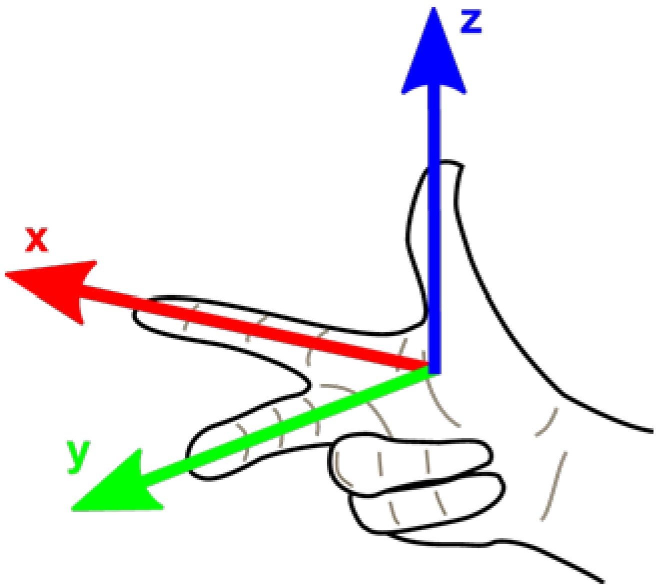


Figure 4.3 The right-hand rule (without rotations).[124]

The colors used for x axis, y axis and z axis are a common convention:

- x axis: red
- y axis: green
- z axis: blue

You can easily remember the correspondence by comparing the order of axis xyz and the order of colors in RGB (red, green, blue). RGB being a common way to specify colors, e.g. for the pixel in a digital image.[125]

It is likely that you will often hold your right hand in this manner during the remainder of this book, especially in this chapter. In combination with a convention for coordinate systems used for the robot, objects and the environment, the right-hand rule is an invaluable thinking aid. I did actually not yet show you the entire right-hand rule. The missing aspect is rotations. These are added in figure 4.4(a).

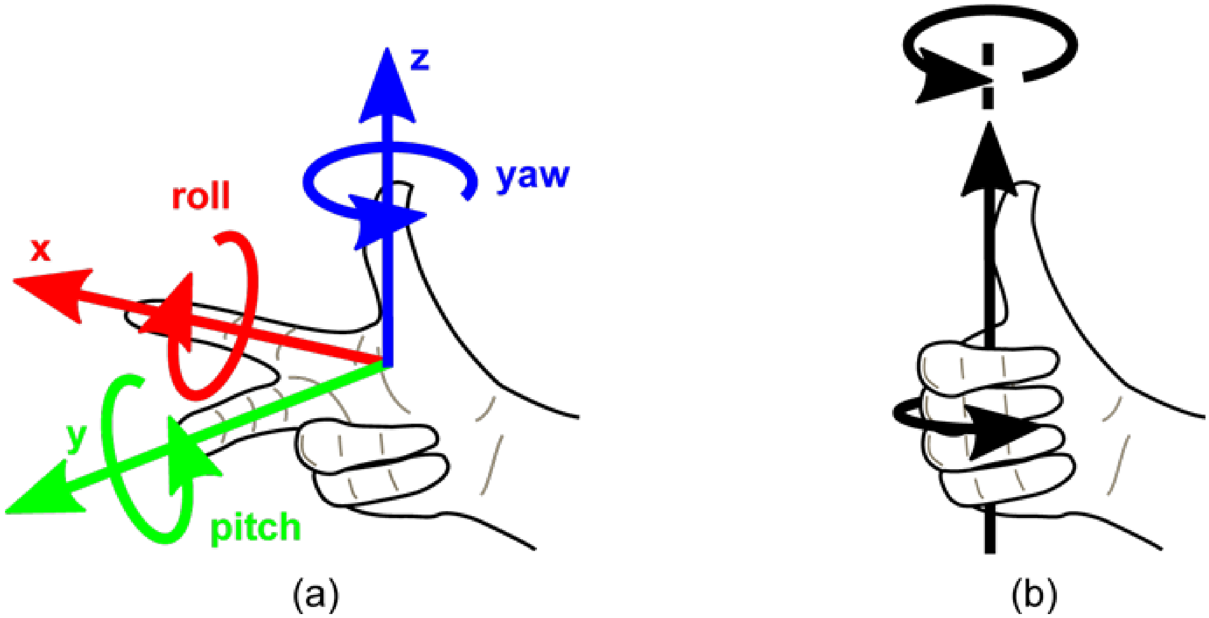


Figure 4.4 The right-hand rule.

We will use the following naming convention for these three rotations:[126]

- Rotating along x axis: roll = (gamma)
- Rotating along y axis: pitch = (beta)
- Rotating along z axis: yaw = (alpha)

One common mistake is to get the direction of positive rotation wrong. The positive direction of rotation is indicated by the arrow heads in figure 4.4(a). Luckily there is also a simple physiological mnemonic for this, again using the right hand. As shown in figure 4.4(b), pointing your thumb along an axis while curling your fingers inward leads to your finger tips pointing towards the direction of positive rotation.

Although the Cartesian coordinate system is the most important and most widely used type of coordinate system in robotics, there are three other coordinate systems you should know about. First, the *2D polar coordinate system*, second the *3D spherical coordinate system* and third the *3D cylindrical coordinate system*. You should at least remember that they exist, that they can greatly simplify calculations under certain conditions and that one can easily transform between all these coordinate systems.[127] Cartesian coordinates for a point are based on length measurements to the coordinate system's origin, where all axes intersect and all their values are zero. In contrast, polar, spherical and cylindrical coordinates are based on a mixture between lengths and angles to determine the position of a point (see figure 4.5). We will not go into details here, instead we will discuss how to work with them and transform back and forth to Cartesian coordinates when we first utilize each.

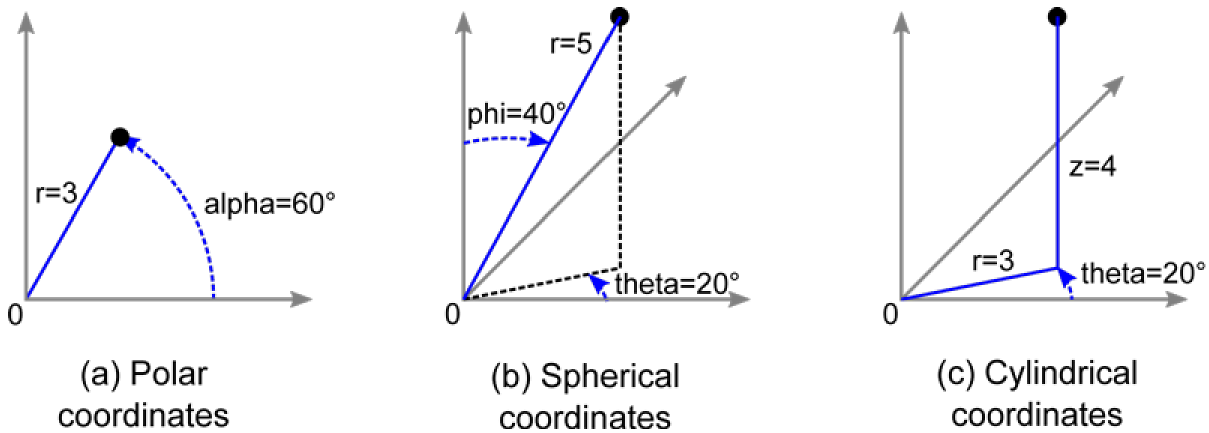


Figure 4.5 The 2D polar coordinate system, the 3D spherical coordinate system and the 3D cylindrical coordinate system.

After a short digression on math and physics in robotics, we will look at how we can use multiple coordinate systems in the same space to greatly simplify working with motions and thus greatly simplify modeling robot systems.

4.2 Math and Physics in Robotics

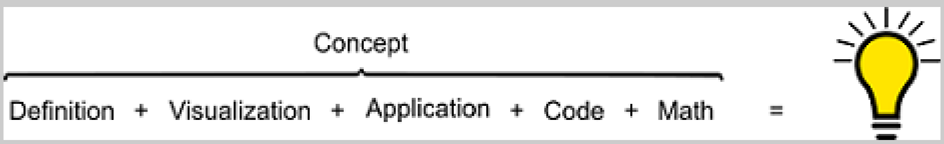
Before we dive into the remainder of the chapter, let me say a few words on the subject of mathematics and physics in robotics. You already know how to program computers to transform input data into output data via a sequence of elementary operations (an algorithm) using programming languages such as Python. Depending on your prior experience, you might have more, less or no knowledge of the theoretical underpinnings in computer science, semiconductor technology and related fields. While it is always an advantage to have more background knowledge, e.g. to understand concepts such as Turing machines, Lambda calculus and complementary metal oxide semiconductors (CMOS), this knowledge is not required to be a productive software developer. However, one cannot be proficient in programming without understanding basic computer science concepts behind the program syntax, such as conditional statements (`if`), logical operators (`and`/`or`), loops (`while`), references/pointers, functions, etc. Depending on your role, you require a more or less in-depth knowledge of these topics. Yet, everyone who develops software is expected to have a conceptual grasp of them. The situation in robotics is similar with respect to math and physics.

Concept summary sidebars on key math and physics concepts are provided for you throughout this and the following chapters. As shown below, each sidebar summarizes a concept in terms of a definition, a visualization, its application in robotics, a Python code snippet and a mathematical description. The concepts are introduced, exemplified and elaborated outside the sidebars. I strongly encourage you to invest the time to learn each concept properly, in the order they are presented in the book. Consider each of the learned concepts as a new tool, built from math and physics, for your robot software engineering skill set. By the end of the book, at the latest, I anticipate you will have gained the ability to go back and forth between concept, a visualization, code and potentially also its mathematical description. It makes robotics much easier and much more fun to have this (basic) understanding of first principles. My presumption is also that you will highly appreciate these skills for your work in robotics and beyond. Nevertheless, as long as you understand the idea behind these math and physics tools, you can skip the more detailed and more formal explanation and revisit them when we utilize them in various robot applications later on.

SIDEBAR **Structure of concept summary sidebars**

Definition: A brief definition of the math or physics concept/tool.

Visualization:



Application in robotics / Tasks solved:

- **Short description of why understanding the concept is important in robotics.**
- **List of common tasks in robotics you can solve by applying the concept.**

Code:

```
# Code snippet for the concept
```

Math:

Formula(s) for the concept

Related concepts: List of related concepts.

To give you an example for at least “getting the gist of it”, let’s look at the relation between a robot’s position and its velocity. At the end of the next chapter, you should understand that we can calculate position from velocity by means of integration and calculate velocity from position by means of differentiation. Furthermore, you should know that there are symbolic and numerical methods to implement these calculations. It is not essential that you can write down formulas for the derivative or integral, nor that you can implement them from memory.

4.3 Coordinate Transformations

Every pose and all coordinates are always specified with respect to a specific coordinate system. There is not *the* coordinate system. Rather one usually works with a number of different coordinate systems. Each defined in a way that either simplifies the calculations within the robot system or makes it easier for the robot developer to understand the system. Take the scene in figure 4.6 as an example.

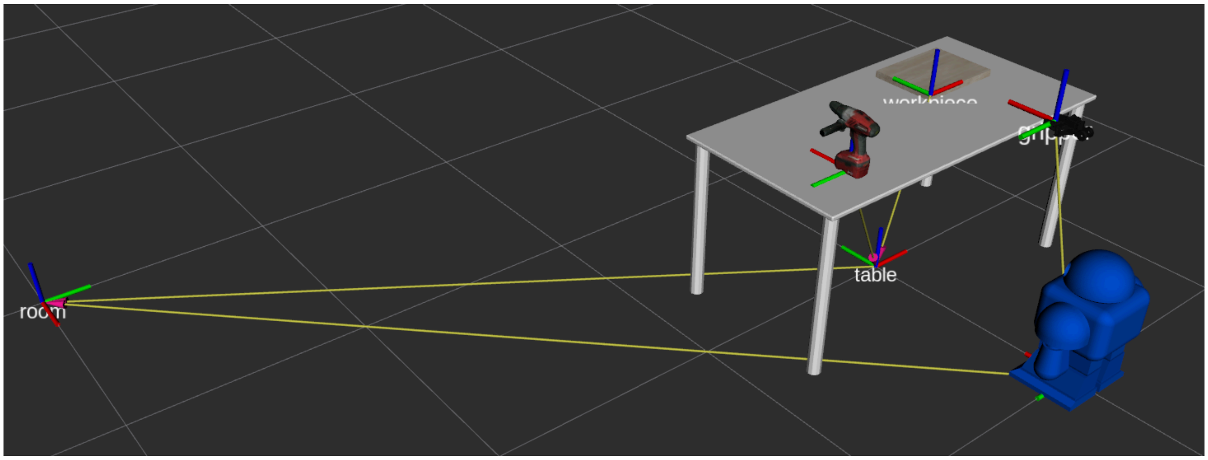


Figure 4.6 Example scene with six coordinate systems: room, table, tool, workpiece, robot and gripper.

There is a coordinate system defined for the room, the table within in, the power drill (= tool) and wooden board laying on the table (= workpiece), the robot and its gripper[128]. Of course we can specify the pose of all objects in the room coordinate system. It is a coordinate system after all and can thus by definition specify the pose of any object within the space. Nevertheless, it would be quite awkward to specify the task of drilling a hole in the middle of the workpiece in room coordinates. One would have to specify all poses in room coordinates: the power drill, the gripper holding it as well as the path of the drill. Not only must we figure out these pose values relative to the corner of the room in the first place, but we must change all these poses when moving the table, moving the workpiece on the table or using a different power drill. There must be a better way than doing this - and there is.

The basic idea is quite simple. We specify all poses in the most appropriate coordinate system. In our example, the hole of the workpiece in the workpiece's coordinate system, the position where to grip the power tool in the power tool's coordinate system and the location of the workpiece in the table's coordinate system. Then we describe the position of each coordinate system relative to another coordinate system. As long as a chain of these relations exists between two coordinate systems, we can *transform* any pose in one coordinate system into a pose of the other coordinate system. Both poses will refer to the same absolute pose in space. Let's look at a simple 2D instance to illustrate this topic. Figure 4.7 shows a simplified 2D top-down view of our example 3D scene.

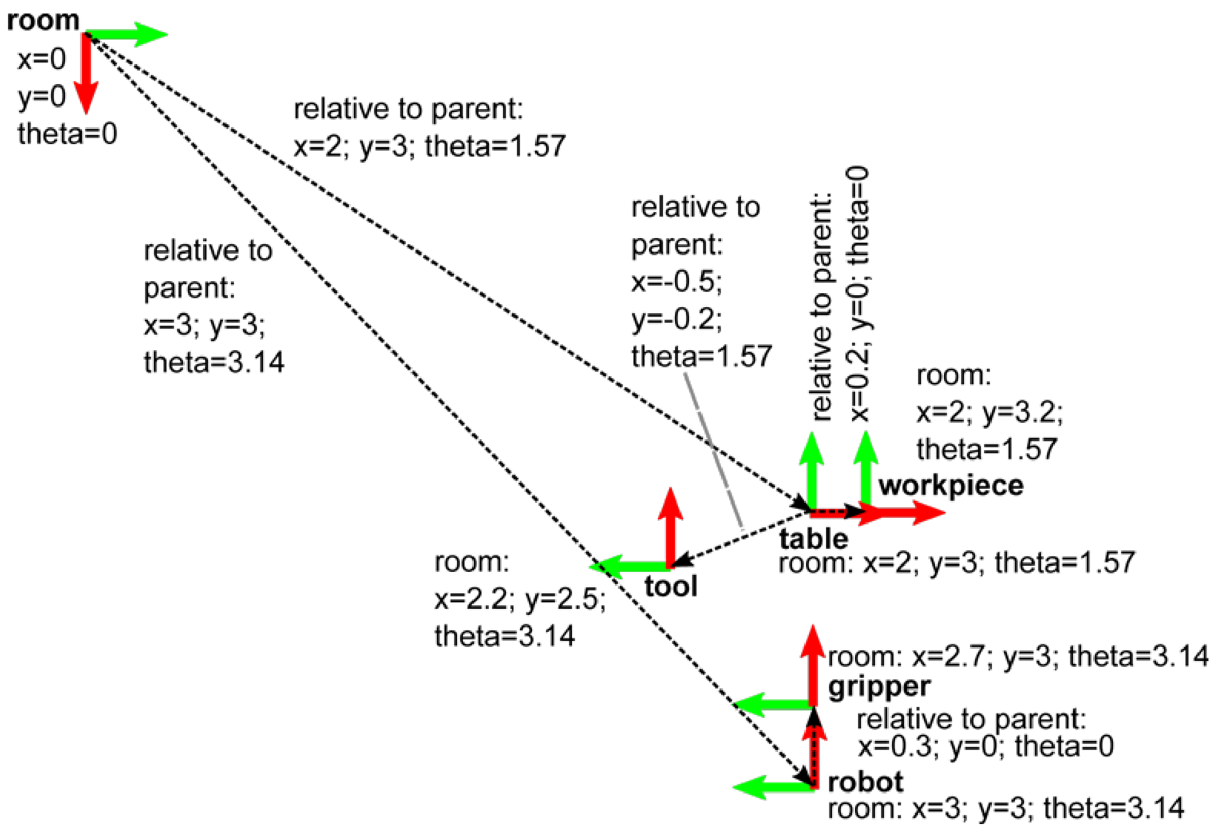


Figure 4.7 Simplified 2D top-down view of the scene in figure 4.6. The known relative relationships between the room, table, tool, workpiece, robot and gripper coordinate systems are shown.

Because this point is important, let me repeat. We only need to know the relation between the coordinate systems to know how to transform any pose within one coordinate system into a pose in the other. Furthermore, we can use a chain of relations to do this, i.e. we do not need to know the relation between each pair of coordinate systems, rather it is sufficient if all "hang together" in a web of relations. This web of relations between coordinate systems forms a graph. Actually a very specific class of graphs: a tree.[129] Figure 4.8 visualizes the transformation tree[130] of figure 4.7.

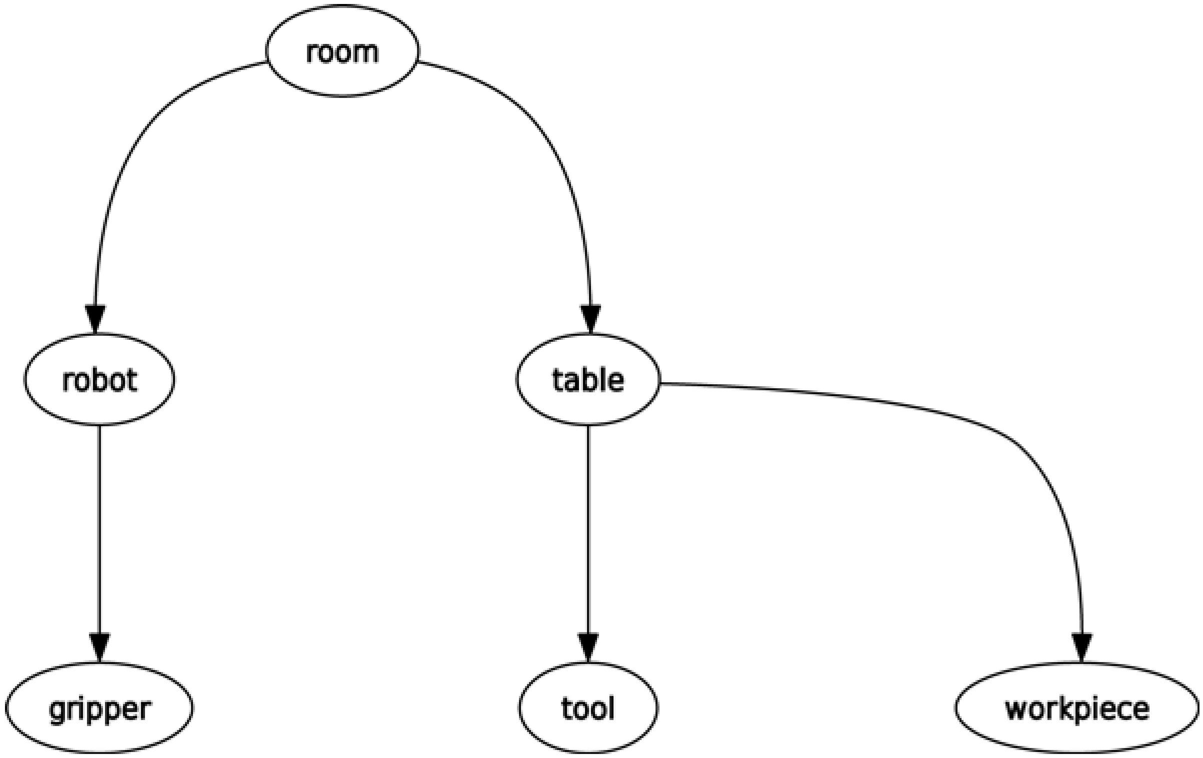


Figure 4.8 A transformation tree, visualizing the relationship between the coordinate systems of figure 4.7.

Coming back to our example, given we know - be it from manual setup or sensor measurement - where the table is located in the room (as defined by the transformation from the room coordinate system to the table coordinate system), where the workpiece is located on the table, where the robot gripper is in the room and where it grips the tool, we can easily specify the task of drilling in the workpiece coordinate system and leave the rest to coordinate transformations. If you do not already appreciate how much this facilitates robot software, I'm confident that you will appreciate it by the end of the book.

In order to implement handling of geometry, coordinate systems and coordinate transformations in software, we still have to define data structures and algorithms, an API, for them. We will get to this right away in the next section. I provide a few exercises for you to train and test your new skills afterwards.

4.4 Common Geometry Representations

The leading question for this section is how can we represent poses and shapes in our robot software so it easy to operate on them? Let's start with positions.

In 1D space (cf. figure 4.2(a)) a position can be represented by just a single number. Performing basic mathematical operations on this number, such as addition, subtraction and multiplication, each have a corresponding geometric meaning. Adding two numbers corresponds to either

- joining two distances/lengths or
- moving a position by a certain distance.

See figure 4.9 for an illustration.

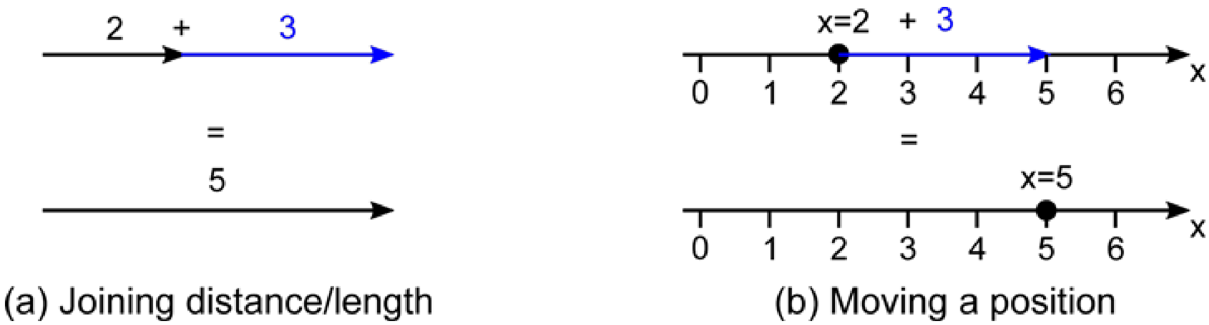


Figure 4.9 In 1D the addition of two numbers corresponds to either (a) joining two distances/lengths or (b) moving a position by a certain distance.

It is essential to understand this ambiguity between positions and distances/lengths in 1D because the same is true, but harder to point out, in higher dimensional spaces. The simple code snippet

```
a = 3
b = 2
c = a + b
```

can mean two very different things when it comes to 1D geometry. We need to keep these different meanings, their different *semantics*, apart in our robot software. The following two code snippets perform the same computations, but their semantics are clear. We are working with distances/lengths.

```
a_to_b_distance = 3
b_to_c_distance = 2
a_to_c_distance = a_to_b_distance + b_to_c_distance
```

```
arm_segment1_length = 3
arm_segment2_length = 2
overall_arm_length = arm_segment1_length + arm_segment2_length
```

Again performing the same computations, this one unambiguously is about calculating a new position:

```
pick_up_position = 3
pick_up_to_drop_off_distance = 2
drop_off_position = pick_up_position + pick_up_to_drop_off_distance
```

The reason why one easily becomes confused is not only a question of (primitive) data types representing many different concepts. One has already become used to the latter as a software engineer.[131] The confusion, when it comes to spatial representations, rather originates from the *valid* conversions between positions and distances/lengths. These valid conversions are

- Length + Length results in a Length (same for subtraction)
- Position + Length results in a Position (same for subtraction)
- Position - Position results in a Length

It does not make sense to add two positions.[132] Always name your variables in a way to keep positions apart from distances/lengths. Using more general terms, always distinguish between poses and transformations.[133]

There is one more basic operation that can be performed on distances/lengths: scalar multiplication. A scalar is simply a single number. The term scalar multiplication is derived from operations in higher dimensions where positions and transformations are not single numbers, but vectors - as we will discuss soon. In 1D, scalar multiplication means multiplying a distance/length with a unitless number:

```
a_to_b_dist = 3
scaling_factor = 0.5
scaled_a_to_b_dist = scaling_factor * a_to_b_dist
```

To repeat, this operation only makes sense on distances/lengths, but not on positions. You will see more use cases for scalar multiplication in 2D and 3D. Let's now move to dimensions beyond 1D, but still disregard orientation for now.

4.4.1 Vectors

In 2D and 3D space (cf. figure 4.2(b,d)) we need 2 respective 3 numbers to describe a position. Using the right "data structure" along with a suitable definition of operations on it allows us to write code very similar to the 1D case. The mathematical tool we need for this is the *Euclidean vector* or simply *vector*. For our purpose, a vector is an ordered tuple/array of numbers with (at least) addition, subtraction, scalar multiplication and length/magnitude/norm as defined operations.[134]

The *vector operations* are defined as follows:

- Addition: Add the individual numbers in the vector with each other, i.e. perform the operation element-wise.
- Subtraction: Perform element-wise subtraction of the individual vector elements.
- Scalar multiplication: Multiply the individual numbers in the vector by the scalar value, i.e. perform multiplication element-wise.
- Length / magnitude / (Euclidean) norm: Take the square root of the sum of all squared individual numbers in the vector. More details below.

Note that addition and subtraction are only defined for vectors of equal size, i.e. consisting of the same amount of numbers. Before looking at how to operate on vectors in Python using numpy as well as looking at the mathematical notation for vectors, let's first implement all three vector operations in basic Python.[135]

```
import math

a_vec = [1, 2, 3]
b_vec = [4, 5, 6]
scalar = 2

def vector_addition(a, b):  # a + b
  if len(a) != len(b):
    return None
  res = [0] * len(a)
  for i in range(len(a)):
    res[i] = a[i] + b[i]
  return res

def vector_scalar_multiplication(s, a):  # s * a
  res = [0] * len(a)
  for i in range(len(a)):
    res[i] = s * a[i]
  return res

def vector_norm(a):
  sum = 0
  for val in a:
    sum += pow(val, 2)
  return math.sqrt(sum)

print(vector_addition(a_vec, b_vec))  # prints: [5, 7, 9]
print(vector_scalar_multiplication(scalar, a_vec))  # prints: [2, 4, 6]
print(vector_norm(a_vec))  # prints: 3.741...
```

While we could implement our own vector class, including operator overloading, we really should make use of existing implementations instead. The most widely used library for vectors and matrices[136] in Python is numpy:

```
import numpy as np

a_vec = np.array([1, 2, 3])
b_vec = np.array([4, 5, 6])

print(a_vec + b_vec)  # prints: [5, 7, 9]
print(scalar * a_vec)  # prints: [2, 4, 6]
print(np.linalg.norm(a_vec))  # prints: 3.741...
```

Let's briefly look at mathematical notation for vectors. Afterwards we use them for working with 2D and 3D geometry. Vectors are represented by bold lowercase letters and the individual numbers by normal font lowercase letters subscribed by their position in the vector.[137] For example, a 3D vector would be written as $\mathbf{a} = (a_1, a_2, a_3)$ or

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The former would be a row vector and the latter a column vector. Although the difference between them can be important when performing operations between the two kinds or when matrix operations are involved, we can ignore this aspect for now. Using this notation, vector addition can be written as

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$$

Scalar multiplication as

$$s * \mathbf{a} = \begin{bmatrix} s * a_1 \\ s * a_2 \\ s * a_3 \end{bmatrix}$$

Vector norm (Euclidean) as

$$|\mathbf{v}| = \left\| \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \right\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

After this short introduction to vectors, their notation and implementation, we can start working with them. The following code corresponds to the geometric operation shown in figure 4.10.

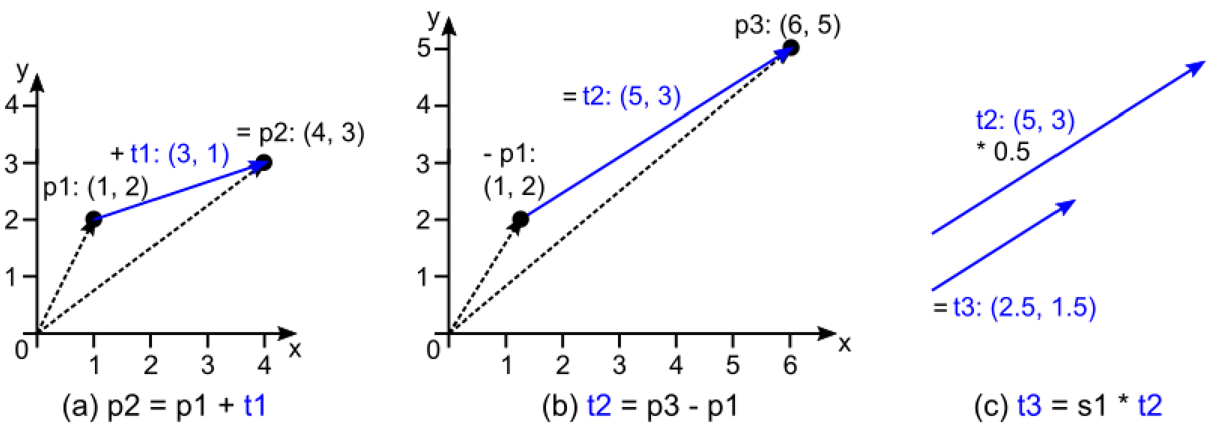


Figure 4.10 (a) Addition of a pose and a translation results in a pose; (b) Subtraction of two poses results in a translation; (c) Multiplication of a translation with a scalar results in a translation.

```
import numpy as np

position1 = np.array([1, 2])
translation1 = np.array([3, 1])
position2 = position1 + translation1

position3 = np.array([6, 5])
translation2 = position3 - position1

scalar1 = 0.5
translation3 = scalar1 * translation2
```

Inverting or reversing a translation can be done by scalar multiplication with the value -1 or by subtracting instead of adding it. Vectors behave like regular numbers in this respect.

$$\mathbf{a} + (-1) \cdot \mathbf{a} = \mathbf{a} - \mathbf{a} = \mathbf{0}$$

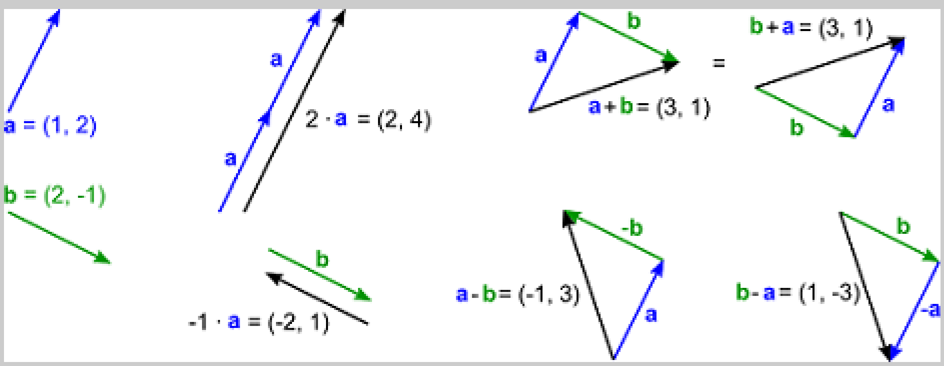
Apart from the vectors having three coordinates instead of two, the code would be identical when working in 3D instead of 2D. As in the 1D case, addition/subtraction of two displacements/translations[138] results in a displacement/translation, addition/subtraction of a position[139] and a translation results in a position and subtraction of two positions results in a translation.

You have now learned how to work with positions and translations. Next you will learn how to work with orientations and rotations. Taken together you will then be able to succinctly express geometric relations via poses and transformations in your robot software.

SIDEBAR **Vector (Euclidean)**

Definition: A vector describes quantities that cannot be expressed in a single number (scalar), e.g. a position or direction in 2D/3D. Important operations on vectors are addition/subtraction, scalar multiplication and calculating their length (norm).

Visualization:



Application in robotics / Tasks solved:

- **Describe positions p in 2D and 3D.**
- **Describe orientations o in 3D.**
- **Define rotation axis.**
- **Work with quaternions.**
- **Model generalized coordinates, e.g. joint values q.**
- **Describe gradients.**

Code:

```
import numpy as np

s = 2
a = np.array([1, 1, 1])
b = np.array([2, 3, 4])

c = a + b  # c = [3, 4, 5]
d = a - b  # d = [-1, -2, -3]
e = s * b  # e = [4, 6, 8]
f = np.linalg.norm(a)  # f = 1.732...
```

Math:

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}$$

$$s * \mathbf{a} = \begin{bmatrix} s * a_1 \\ s * a_2 \\ s * a_3 \end{bmatrix}$$

$$|\mathbf{v}| = \left\| \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \right\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

Related concepts: Matrix, Function

4.4.2 Matrices

In comparison to position and translation, orientation and rotation are more difficult to understand and reason about - at least for most people. Don't get discouraged by "not seeing it" at first. Not only success will come from experience with rotations, but also a sense of familiarity and a more intuitive grasp.

Let's start in 2D, which is a lot more intuitive than 3D when it comes to rotations. In 2D there is only a single rotation value (cf. figure 4.2(c)).

What actually happens when an object is rotated, i.e. when it changes orientation? The relative positions of points on the object stay the same. This is also true of the relative angles between parts of the object. The change is all about the absolute position[140] of the object's points. In other words, both preserve the object's shape. Our coffee cup doesn't grow or shrink when moved, nor does is it deformed. Translations also preserves distances and angles within the object and change the absolute position of the object's points. The difference only becomes apparent when considering how multiple points on a rigid object are transformed by translations versus by rotations. Figure 4.11 illustrates this.

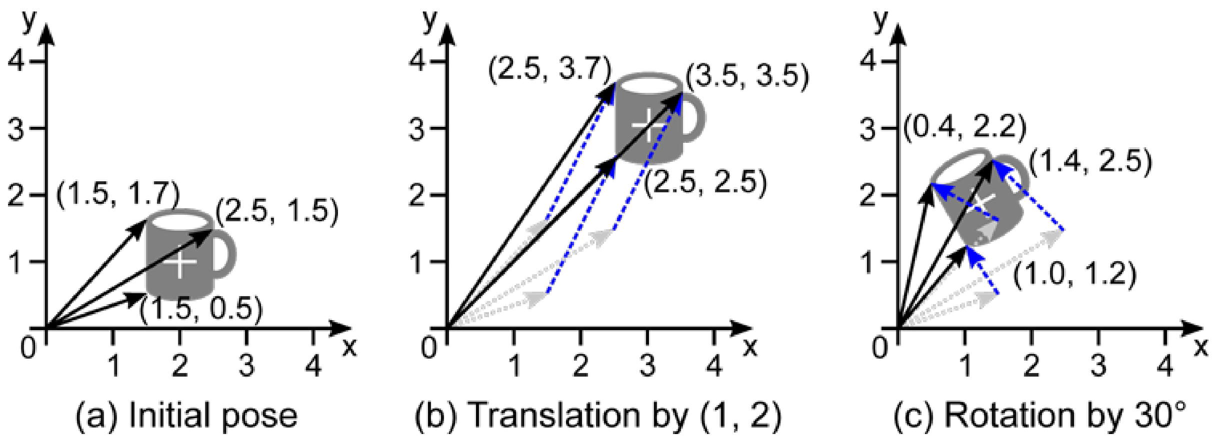


Figure 4.11 Both translations and rotations keep the relative distances and angles between points on an object constant. Both change the position of the points in the external coordinate system. However, they move them in different ways.

I want to draw your attention to the dashed arrows. In a translation (b), all the dashed arrows, which visualize the change between old and new position, have the same length and direction for all points on the object. We could actually use this as a definition for translations. In case of a rotation (c), the object's points are moved into different directions and by different distances - while preserving the distances and angles between them.

We learned that translations are expressed by the *addition* of a position vector and a translation vector. This happens independently for all points of an object. Expressed in code:

```
for i in range(len(points)):
  points[i] = points[i] + translation
```

A rotation would be expressed by a *multiplication* of each point with a *rotation matrix*. The `@` -operator below is the numpy operator for matrix multiplication. Using the more common `*` -operator would lead to a wrong result because it is used for element-wise multiplication in numpy.

```
for i in range(len(points)):
  points[i] = rotation_matrix @ points[i]
```

This leads to a number of questions. What is a (rotation) matrix? What is matrix multiplication? What values in the rotation matrix correspond to what rotation? Let's answer them one by one.

From a software engineer's point of view a *matrix* is simply a two dimensional array.[141] An important matrix property is its size. Like tables, matrixes have a number of rows and columns.[142] It is convention to first name the number of rows then the number of columns. A 3×4 matrix has 3 rows and 4 columns. In more general terms, a m×n matrix has m rows and n columns. In mathematical notation (bold) upper-case letters are usually used for matrices. The individual elements in the matrix are referred to by a lowercase letter and two indices (row, column).[143] A 3×4 matrix can be written as

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{bmatrix}$$

Vectors are a special case of matrices. They are matrices with either only one row (1×n, row vector) or only one column (m×1, column vector). Since one of the indices would always have the value 1, this part of the index is dropped, resulting in the vector notation we used above.

Addition and subtraction are performed element-wise, as for vectors. Also the scalar multiplication works as already explained for vectors. Beyond these operations we have already seen, there is also a new one: matrix multiplication.

Matrices of compatible sizes can be multiplied with each other. This multiplication does *not* take place element by element. Instead, *matrix multiplication* is defined as follows. Given two matrices **A** and **B**, we calculate each entry $c_{i,j}$ in the result matrix **C** = **AB** by element-wise multiplication of **A**'s row i with **B**'s column j. Expressed as a function:

```
def matrix_multiplication(A, B):
  # The number of columns in matrix A
  # must be equal the number of rows in matrix B
  if len(A[0]) != len(B):
    return None

  # C: Zero initialized matrix of size len(A) x len(B[0]),
  #    i.e. A rows x B columns
  C = ...
  for i in range(len(C)):       # iterate rows in C
    for j in range(len(C[0])):  # iterate columns in C
      for k in range(len(B)):   # iterate rows in B (= columns in A)
        C[i][j] += A[i][k] * B[k][j]
  return C
```

Using numpy, we can write this simply as

```
C = A @ B
```

I will not explain why matrix multiplication is defined in this way as this would take us on a major detour into linear algebra. Instead, let’s get back to the topic of rotations and rotation matrices.

A rotation matrix is a matrix that rotates vectors by a certain angle along a defined axis of rotation. We will not derive the entries of rotation matrices from trigonometry here.[144] At this point, it is sufficient for you to know that the following 2×2 rotation matrix **R**(θ) correctly rotates 2D vectors by theta (θ) degrees, when multiplying them with **R**(θ).

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Let’s validate figure 4.11(c) using our newly gained knowledge together with Python and numpy:

```
from math import pi, cos, sin, radians

points = [np.array([1.5, 0.5]), np.array([1.5, 1.7]), np.array([2.5, 1.5])]
theta = radians(30)
rotation_matrix = np.array([[cos(theta), -sin(theta)],
                            [sin(theta),  cos(theta)]])

for i in range(len(points)):
  points[i] = rotation_matrix @ points[i]

print(points)
# values rounded: [1.0, 1.2], [0.4, 2.2], [1.4, 2.5]
```

Performing the opposite of a given rotation can be done by inverting the rotation matrix. The opposite rotation or inverse rotation for a given rotation is the rotation that rotates back to the starting point. Performing a rotation and then performing the corresponding inverse rotation is identical to not performing any rotation. Expressed in matrix notation, given a rotation matrix **R** and its inverse rotation matrix $\mathbf{R}^{-1}$ the result of multiplying them is the identity matrix **I**:

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

The identity matrix, when used as a rotation matrix, performs no rotation. It has the value 1 on its diagonal and only contains zeros otherwise. The 2D identity matrix is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Matrix inversion in general is not a simple operation.[145] Fortunately, rotation matrices belong to a class of matrices that render inversion simple, even trivial. The inverse of a rotation matrix is equivalent to its transpose. The transpose of a matrix is a matrix with its rows and columns swapped. Stated differently transposition flips the matrix along its diagonal. Let's use a 3×2 matrix **M** and its transpose $\mathbf{M}^{\top}$ as an example:

$$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \quad \mathbf{M}^{\top} = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}$$

Using mathematical symbols, for any rotation matrix **R** it holds that the inverse $\mathbf{R}^{-1}$ is identical to the transpose $\mathbf{R}^{\top}$, i.e.

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

Thus given a rotation matrix `rotation`, we only have to transpose it to get the inverse rotation matrix `inverse_rotation`:

```
rotation = np.array(...)
inverse_rotation = rotation.transpose()
```

We can now translate and rotate in 2D:

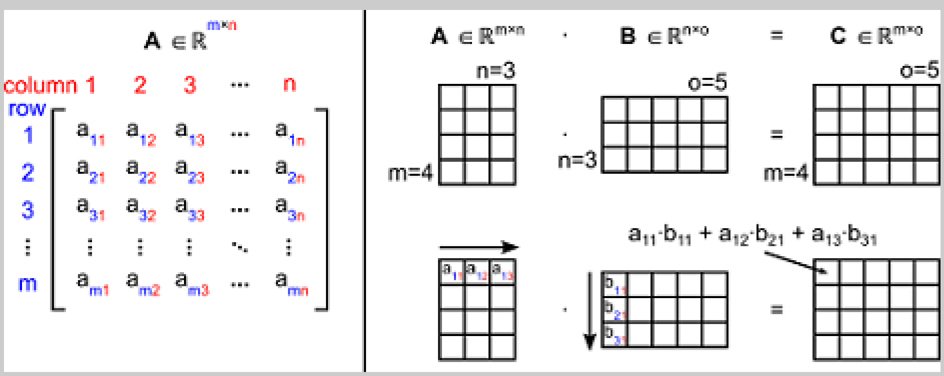
- To translate we add a translation vector to the vector.
- To rotate we multiply the vector with a rotation matrix.

We can combine both operations by executing them sequentially to translate and rotate objects. The order of operations is important, in general, you will get a different result when first translating and then rotating compared to first rotating and then translating.[146] The only thing we still have to discuss in order for you to have all the required tools to work with coordinate transformations in 2D is to combine position and orientation into a pose respectively translation and rotation into a transform.

SIDEBAR **Matrix**

Definition: A matrix is a rectangular array of numbers/symbols arranged in rows and columns. A m×n matrix has m rows and n columns. Important operations on matrices are addition/subtraction (element-wise), scalar multiplication (element-wise), matrix multiplication, transposition and inversion. Multiplication of matrices with vectors is a special case of matrix multiplication. Vectors can be seen as matrices with only one column (m×1) or only one row (1×n).

Visualization:



Application in robotics / Tasks solved:

- **Describe rotations and orientations using rotation matrices R.**
- **Model poses P and transformations T in homogeneous coordinates.**
- **Work with Jacobians J in kinematics and trajectory planning.**

Code:

```
import numpy as np

def matrix_multiplication(A, B):  # A.shape[1] == B.shape[0]
  C = np.zeros((A.shape[0], B.shape[1]))
  for i in range(C.shape[0]):
    for j in range(C.shape[1]):
      for k in range(A.shape[1]):
        C[i][j] += A[i][k] * B[k][j]
  return C

C = matrix_multiplication(A, B)
C = A @ B  # numpy matrix multiplication
```

Math:

$$\mathbf{A} + \mathbf{B} : \mathbf{A}_{i,\ j} + \mathbf{B}_{i,\ j} = (\mathbf{A} + \mathbf{B})_{i,\ j}$$
$$s \cdot \mathbf{A} : s \cdot \mathbf{A}_{i,\ j} = (s \cdot \mathbf{A})_{i,\ j}$$
$$\mathbf{A} \cdot \mathbf{B} : (\mathbf{A} \cdot \mathbf{B})_{i,\ j} = \sum_{k\ =1}^{n} a_{i,\ k} b_{k,\ j}$$
$$\mathbf{A}^T : (\mathbf{A}^T)_{i,\ j} = \mathbf{A}_{j,\ i}$$
$$\mathbf{A}^{-1} : \mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}$$

Related concepts: Vector, Function

4.4.3 Homogenous Coordinates

A pose in 2D with orientation has 3 PP (pose parameters): x, y, θ (theta).[147] The same is true for a transformation in 2D with orientation. To repeat once more, poses and transformations have the same representation and the same operations, but they have different semantics.[148] The symbol **P** will be used for poses and the symbol **T** will be used for transformations. There is no difference between the mathematical structures - such as vectors and matrices - representing poses (**P**) and transformations (**T**) in the same space. We distinguish between them only for semantic reasons.

We can describe the entire figure 4.7 using five transformations. However, your code might look less concise than you want it to, if you perform translations and rotations separately.[149] *Homogeneous coordinates* help us perform the same operations in a more - well - homogeneous manner.

Instead of separately dealing with position and orientation resp. translation and rotation, homogeneous coordinates perform both operations via a single matrix multiplication. For this to work, we will use *transformation matrices*, also known as *homogeneous matrices* to represent poses and transformations. Again without derivation, one can create a homogenous coordinates matrix **T**(θ, **t**) by extending the size of a rotation matrix **R**(θ) by one row and one column and inserting the translation vector **t** as shown:

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

All 3 PP (x, y, θ) are encoded in the 2D transformation matrix $\mathbf{T}(\theta, \mathbf{t})$. The only operation we need to perform in order to translate *and* rotate a pose is to represent it in homogenous coordinates and multiply it with the transformation that is also represented in homogenous coordinates:

$$\mathbf{P}_{\text{new}}(\theta_{\text{new}}, \mathbf{t}_{\text{new}}) = \mathbf{P}_{\text{old}}(\theta_{\text{old}}, \mathbf{t}_{\text{old}}) \cdot \mathbf{T}(\theta, \mathbf{t})$$

expressed in Python

```
old_pose = np.array(...)
transformation = np.array(...)
new_pose = old_pose @ transformation
```

Initializing a 2D homogeneous matrix from the pose parameters in Python reads:

```
position = np.array([1.0, 2.0])
theta = radians(90)
pose = np.array([[cos(theta), -sin(theta), position[0]],
                 [sin(theta),  cos(theta), position[1]],
                 [0,           0,          1]])
```

I encourage you to check by pen and paper that the multiplication of two homogenous matrices with only positions/translations ($\theta = 0$) results in the same position values as a translation by vector addition. Furthermore, verify that a rotation-only transformation matrix ($\mathbf{t} = (0, 0)$) when multiplied with a homogenous pose is equivalent to rotating using a rotation matrix.

Inverting a transformation represented as a homogenous matrix works in the same way as it does for rotation matrices - we use the inverse homogenous matrix. However, in contrast to rotation matrices, the inverse matrix is *not* identical to the transpose matrix. For this reason, the inverse for a given transformation $\mathbf{T}$ is $\mathbf{T}^{-1}$. Note that the inverse of a homogenous matrix can be calculated in a much more efficient way than the general matrix inverse. Thus, you should use special methods for inverting homogenous matrices if they are provided by the library of your choice.[150]

Having learned how to handle 2D, we can now proceed to 3D. Fortunately, there is not a lot of new concepts to learn for 3D. Our focus is rather on handling the additional complexity and ambiguities inherent in 3D space.

Positions and translations do not need a separate discussion. Our vectors have 3 (x, y, z) instead of just 2 entries. Otherwise everything works the same way as in 2D. For most people this also goes for their intuitive grasp. The only added difficulty is that books and screens are 2D and can thus only show a 2D projection of the 3D scene, which can make it difficult to comprehend the 3D relations.[151]

You have likely already guessed that this next paragraph will address orientation/rotation - and that they are, once again, more involved in 3D as well. This assumption is correct, 3D orientations and rotations pose some challenges. While you end up with the same position when performing multiple translations *independently of the order you perform them in*, this is not true for rotations.[152] Figure 4.12 exemplifies this point.

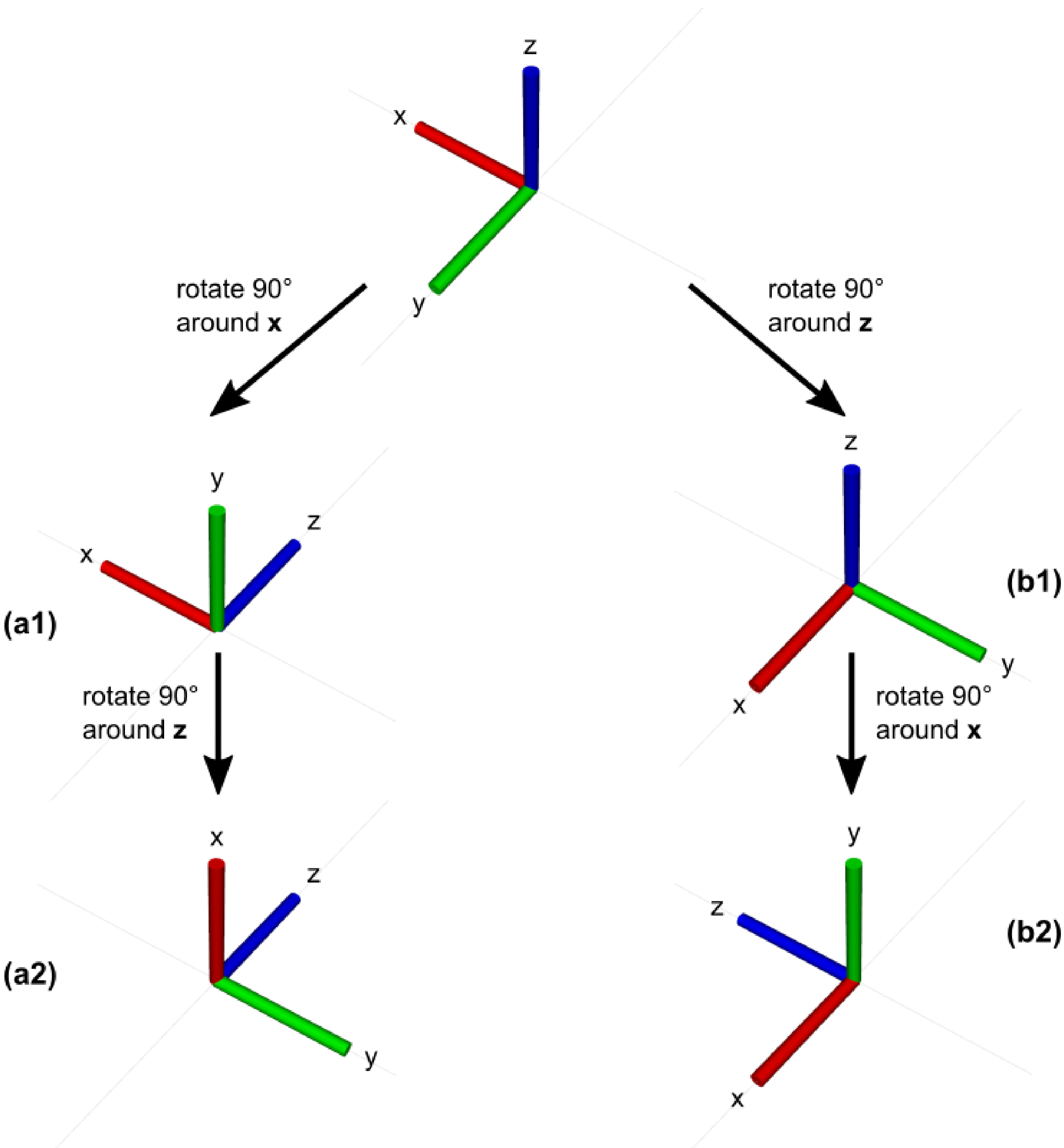


Figure 4.12 Performing the same two rotations, but in different order, leads to different results.

For this reason, we need to agree on a convention for the order of rotations. Furthermore, we also need to define whether we use fixed/static axis (extrinsic rotation) or rotating axis (intrinsic rotation). Each convention has its benefits and drawbacks and is more or less suitable in different situations. There is no best universal convention, the best choice rather depends on what (part of a) robot system one wants to describe. One can convert back and forth between different conventions using well known formulas. The problem when mixing conventions in the same codebase is rather the comprehension on the developer's side. It is very easy to become confused and end up with an incorrect implementation.

By default, unless explicitly noted otherwise, we will follow the convention used in Robot Operating System (ROS)[153], e.g. for the Unified Robot Description Format (URDF) `rpy` parameter:

- Rotation around fixed axes.
- First roll () around x axis, then pitch () around y axis and finally yaw () around z axis.[154]

Using this convention, we can create a universal 3D rotation matrix (roll, pitch, yaw) by combining the three 3D rotation matrices for rotations around each single axis (x, y, z):

$$\mathbf{R}(\text{roll}, \text{pitch}, \text{yaw}) = \mathbf{R}(\gamma, \beta, \alpha) = \mathbf{R}_Z(\text{yaw})\mathbf{R}_Y(\text{pitch})\mathbf{R}_X(\text{roll})$$

I will not print the content of this matrix here. If you really need it, you can look it up. My general advice to you is to make use of existing implementations for these basic constructs.[155]

We composed the 3D rotation matrix **R**(roll, pitch, yaw) from three rotations (roll, pitch, yaw) around the three axis (x, y, z). It is guaranteed that any 3D rotation can be described in this manner. There are also other ways - quite a few actually - to express 3D rotations. One that I want to mention here is the *axis-angle* representation. According to Euler's rotation theorem, every rotation in 3D space can be expressed as one rotation around a single axis. We will come back to axis-angle and how to work with them in later chapters. Yet, please take some time to think about why axis-angle can describe any 3D rotation.

Now that we have dealt with 3D orientation/rotation, the step to homogenous coordinates for 3D is straightforward. We extend our 3D rotation matrix in the same way as we have done in the 2D case.

$$\mathbf{P}(\text{roll}, \text{pitch}, \text{yaw}, \mathbf{t}) = \begin{bmatrix} & & & t_x \\ & \mathbf{R}(\text{roll}, \text{pitch}, \text{yaw}) & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using 3D homogenous coordinates is identical to 2D ones:

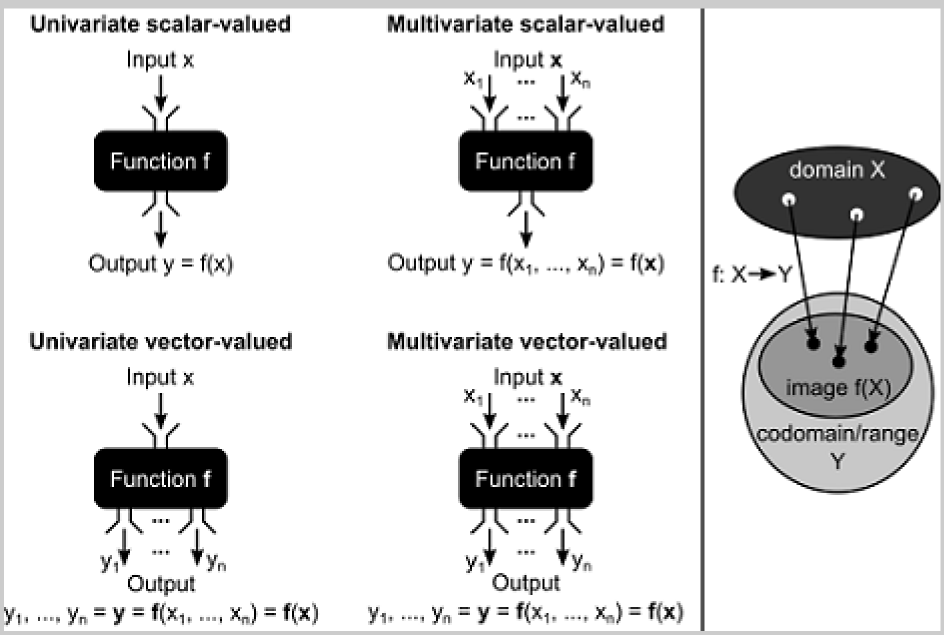
```
old_pose = np.array(...)
transformation = np.array(...)
new_pose = old_pose @ transformation
```

Equipped with this knowledge, you are ready to work with 3D poses and transformations. However, while homogenous coordinates are a complete formalism, there is another one you will frequently encounter in robotics: quaternions. The next section provides a brief introduction to them.

SIDEBAR **Function (multivariate, vector-valued)**

Definition: A function assigns one output value to every input value. A function with multiple inputs is called multivariate or multivariable. A function with multiple outputs is called vector-valued.

Visualization:



Application in robotics / Tasks solved:

- **Robotics is fundamentally multidimensional (e.g. 3D, 6 PP, n DoF). Hence, multivariate functions and vector-valued functions are everywhere.**
- **(Multivariate) (vector-valued) functions serve as common abstraction to encapsulate a quantitative relationship - just like in programming languages.**

Code:

```
def univariate_scalarvalued_function(x):
  return 2 * x

def multivariate_scalarvalued_function(x, y):
  return x + y

def univariate_vectorvalued_function(x):
  return np.array([x, 0, 0])

def multivariate_vectorvalued_function(x, y, z, s):
  return s * np.array([x, y, z])
```

Math:

$$f : \mathbb{R}^3 \rightarrow \mathbb{R}$$
$$f(\mathbf{x}) = f(x_1, x_2, x_3) = \sqrt{x_1^2 + x_2^2 + x_3^2}$$

Related concepts: Vector, Matrix, Derivative, Integral

4.4.4 Quaternions

Quaternions are an alternative way to represent 3D orientations and rotations. While they are certainly not the most intuitive representation, their computational efficiency and favorable mathematical properties render them a good implementation choice. Much could be said about the mathematics behind quaternions, but we will focus only on using them.[156] Using quaternions instead of rotation matrices, and hence homogenous coordinates, has a number of advantages and some disadvantages.

Advantages:

- More compact representation. Quaternions only require to handle 4 numerical values compared to 9 values in a 3D rotation matrix.[157]
- Better numerical stability, i.e. less issues with errors due to the finite precision of floating point numbers.
- Simplified interpolation between two orientations/rotations (SLERP) as well as averaging of multiple orientations/rotations.
- No singularities (Gimbal lock).

Drawbacks

- Position/Translation and orientation/rotation are handled separately.[158]
- Less intuitive to use.

A lot of robot software has adopted the use of quaternions in their internal representation of poses and transformations, but exposes more “human friendly” representations, such as roll-pitch-yaw, to users (and developers). In these cases the translation vector and the quaternion is part of a “Pose” or “Transform” data structure / class, which allows you to work with them in the way you would with homogenous coordinates.[159]

You can think of quaternions in similar ways as of the axis-angle representation mentioned above. The quaternion encodes both the rotation axis as well as the angle of rotation around it. However, be aware that the similarities end there. The mathematics behind quaternions are quite different compared to those of axis-angle.

A quaternion is represented as a normalized vector **q** with 4 entries.

$$\mathbf{q} = \begin{bmatrix} q_i \\ q_j \\ q_k \\ q_r \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Note that x, y and z are not directly related to position coordinates. It is just a common choice of symbol names for the individual entries of the quaternion vector, same as in case of q_i, q_j and q_k. Normalized means that the vector norm[160] of the quaternion vector equals 1, i.e. $|\mathbf{q}| = 1$. Put differently, quaternions are unit vectors.

In line with treating quaternions as a mathematical black box with a well defined API[161], we will focus on operating on them in code instead of in mathematical notation. To combine two rotations represented as quaternions, the quaternions must be multiplied: `quaternion_multiply()`. To rotate a 3D position/translation vector $\mathbf{a} = (a_1, a_2, a_3)$ using quaternions, one must extend the 3D vector by a fourth value with the value 0 (i.e. w = 0): $\mathbf{a}_q = (a_1, a_2, a_3, 0)$. This prepared vector must then be multiplied with the rotation quaternion **q** in the following manner:

$$\mathbf{b}_q = \mathbf{q}\mathbf{a}_q\mathbf{q}^{-1}$$

This means, we multiply the prepared vector on the left side with the rotation quaternion **q** and multiply the result on the right side with inverse of the rotation quaternion $\mathbf{q}^{-1}$. We can get the inverse quaternion via `quaternion_inverse()`. To interpolate between two rotations represented as quaternions, we use `quaternion_slerp()`. There are functions to convert from other representations, e.g. more human friendly ones, to quaternions, e.g. `quaternion_from_euler()`, `quaternion_about_axis()` and `quaternion_from_matrix()`.

Let’s wrap up quaternions with a short example:[162]

```
import numpy as np
from tf_transformations import *

roll = 1.57
pitch = -1.57
yaw = 0
rotation_q = quaternion_from_euler(roll, pitch, yaw, axes='sxyz')   ❶

position1 = np.array([0, 1, 0])
position1_q = np.append(position1, 0)  # extend by w = 0

position2_q = quaternion_multiply(quaternion_multiply(rotation_q,
                                                      position1_q),
                                  quaternion_inverse(rotation_q))

position2 = position2_q[:3]
print(position2)
# values rounded: [-1.0, 0.0, 0.0]
```

❶ We need to provide the parameter `axes='sxyz'` to `quaternion_from_euler()` in order for the function to interpret the `roll`, `pitch` and `yaw` parameters in the same convention as the one used in this book.

You can graphically verify the correctness of the result in figure 4.13.

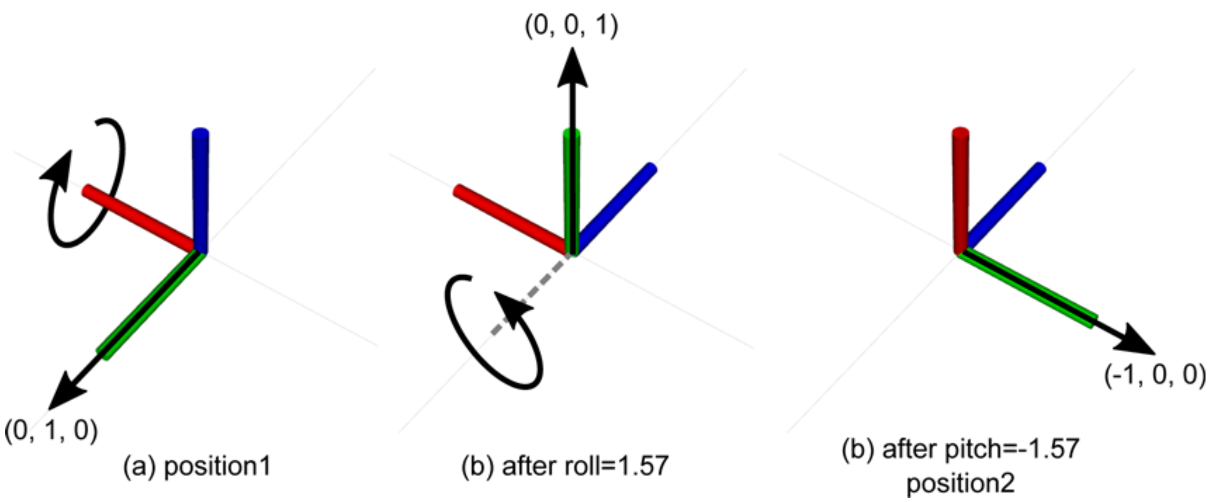


Figure 4.13 Rotating position1 (0, 1, 0) by (roll, pitch, yaw) = (1.57, -1.57, 0).

After we have now covered common representations for poses, let's look at common representations for shapes.

4.4.5 Shapes

Primitive geometric *shapes*, such as those shown in figure 4.1, can be described with parametrized formulas. These formulas enable us to render the shape and its outline. Furthermore, they allow us to efficiently query properties, e.g. whether a point is inside the shape or whether two shapes intersect. Let's use circles to illustrate this.

A circle has one parameter, its radius r (or equivalent its diameter). A circle centered at the origin, i.e. (0, 0), is defined as $x^2 + y^2 = r^2$. This can also be expressed in parametric form by a set of two equations

$$x = r\cos(t)$$
$$y = r\sin(t)$$

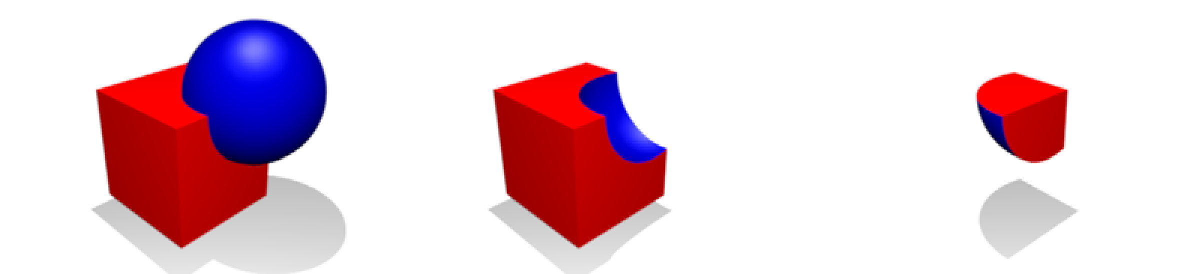
with t in the range from 0 to 2π, i.e. 0 to 360°. We can use this formulas to draw a circle by simply iterating over t with a sufficient small step size and coloring the pixels at the resulting (x, y) values.[163] Using the initial circle formula, we can answer the point-in-shape query for a 2D point **v** by the following condition

$$|\mathbf{v}| = |(x,\ y)| = \sqrt{x^2 + y^2} \le r$$

In words, the point is within the circle (with radius r centered on the origin) if the length of the vector, i.e. the distance of the point to the origin, is less or equal to the circle's radius.

There are several ways how we can represent more complex geometric shapes. Let's look at four important ones that you will likely encounter in the context of robotics.

First, we can create more complex geometric shapes by combining several (more) primitive shapes. Shapes can be combined through Boolean operations on them, such as union, difference and intersection. A union of two objects consists of all the volume enclosed in either one, i.e. the two objects are merged into a bigger one. The difference removes the parts of one object that overlap with the other object, resulting in a smaller object, i.e. we cut away from one object using the other. The intersection only leaves the volume where both objects overlap as its result, also resulting in a smaller object. This method is known as *constructive solid geometry (CSG)* and the basis of computer-aided design (CAD) software. See figure 4.14 for an illustration of these Boolean operations on shapes.



(a) Union (b) Difference (c) Intersection

Figure 4.14 Illustration of three Boolean operations on shapes used in constructive solid geometry (CSG).[164]

The second method combines simple lower dimensional shapes to form a surface in higher dimensional that defines the complex shape. Two widely used shape representations that fall into this category are meshes (polygon models) and curve models. The most common type of *meshes* consist of triangles in 3D space (faces) whose corners (vertices) and sides (edges) overlap to form a connected net.[165] Meshes play an important role in computer graphics and robot simulators. See figure 4.15 for an illustration.

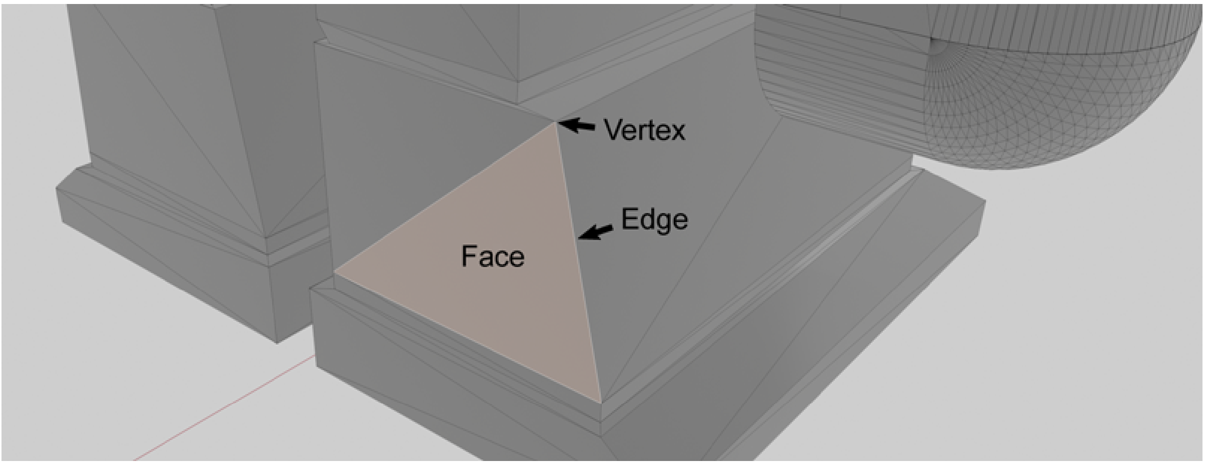


Figure 4.15 Example mesh with one face (triangle), one of its three edges and one of its three vertices highlighted.

Curve models are more commonly found in CAD systems and other computer-aided engineering (CAE) tools. One prominent member of this model family are non-uniform rational basis spline (NURBS) models. Each curve defines parts of the shape's surface. The shape is made up of these curve patches. The patches are defined so there are no discontinuities at their intersection, i.e. the resulting surface is smooth.

A third representation for complex shapes are *volumetric models*. Here a complex shape is represented by a discrete set of volumes. One common method is to subdivide 3D space into small discrete volumes, *voxels*, usually cubes, and define the shape by the voxels it occupies. We have already seen the same concept in 2D, namely pixels, when learning about digital images and cameras in chapter 2.3.4. Voxels are basically 3D pixels. Their main drawback is the fixed resolution in which the voxel grid discretizes space. This is aggravated by the fact that while the number of pixels grows quadratic (power of 2) with the wanted resolution, the number of voxels grows cubic (power of 3). An example of shapes represented by voxels is shown in figure 4.16.

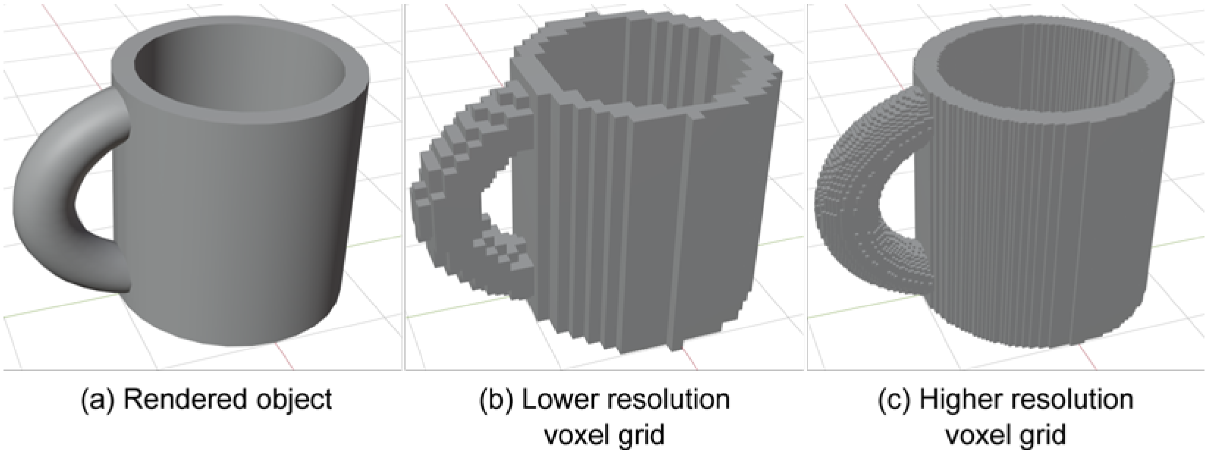


Figure 4.16 Voxel grid representation of a mug in two different resolutions.

The fourth and final representation that I want to mention here are point clouds. A *point cloud* simply consists of a set of 3D points. These 3D points often have additional properties associated with them apart from their position, such as color information. Because a point has no size, no extend, no volume and no surface, point clouds are usually visualized by representing the points as small spheres or cubes. However, even when giving a small volume to each point, the volume and surface of the model are given only implicitly. Thus, point clouds are often converted to one of the other representations that we discussed to perform operations on them. Nevertheless, point clouds are widely used in robotics for sensor data, object detection and visualization. You will learn more about them in later chapters.

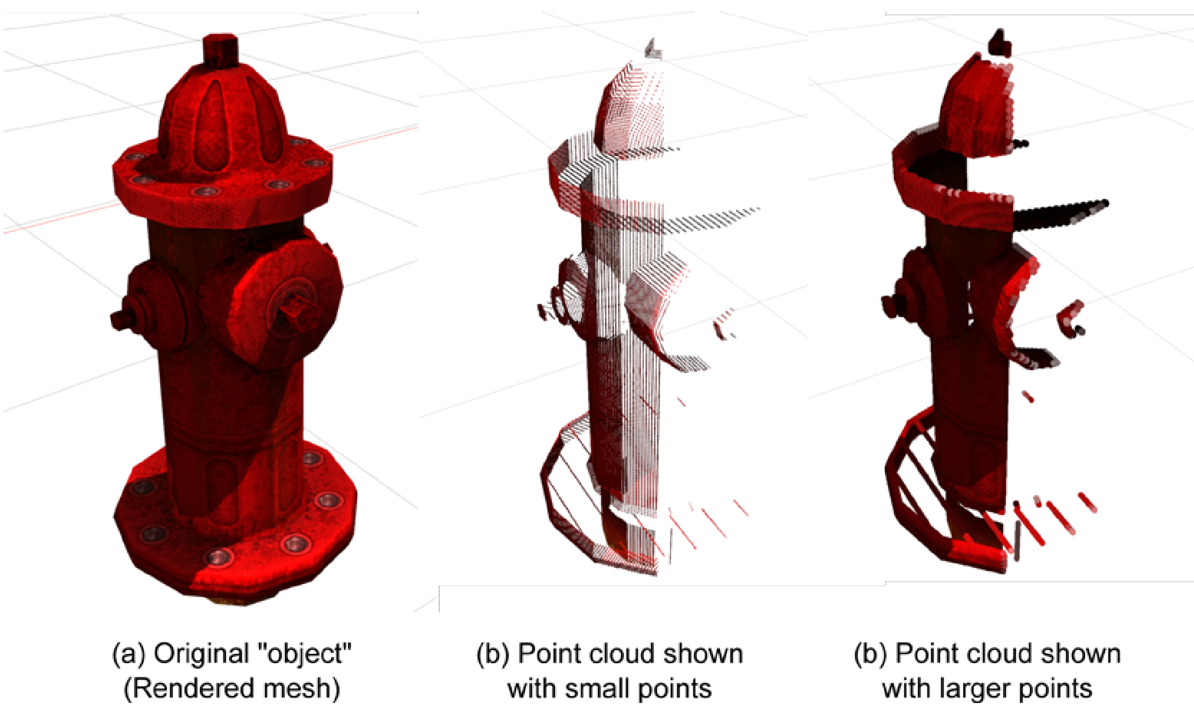


Figure 4.17 Visualization of a RGB point cloud generated by a RGBD camera. The camera is looking at the object from the left, thus there are no points from the object side facing away from the camera.

4.5 Practical Advice

You have now learned all the required tools to describe poses and transformations in formal ways. While expertise will come along with experience using them in your own projects, I want to provide you with some practical advice from my own experience:

- Use your visual cortex and spatial reasoning skills.
- Use unambiguous naming conventions for handling poses and transformations.
- Use coordinate transformation libraries.

Working with coordinate transformations without proper visualization is difficult, very difficult.[166] Imagine this chapter without any figures, expressing everything you have already seen in words and numbers only. I think you will agree that it would be much harder to grasp poses and transformations, in particular the combination of 3D and orientations/rotations. The advice, hence, is to make good use of existing visualization tools in your robot software framework of choice or add visualizations in case they are not readily available.

A second point related to making things graphical is a visual aid that helps you in combining poses/transformations in the right order as well as determining when you have to use the inverse transformation. The rule is simple: follow the arrows. It means when you have a transformation tree in front of you and you have to correctly combine transformations, it is sufficient to follow the path between the two coordinate systems of interest in the tree. The sequence of arrows you traverse on this path together with their direction provides you with all required information. If we want to calculate the transform between the tool coordinate system and the gripper coordinate system in our example scene above, figure 4.18 shows us the solution.

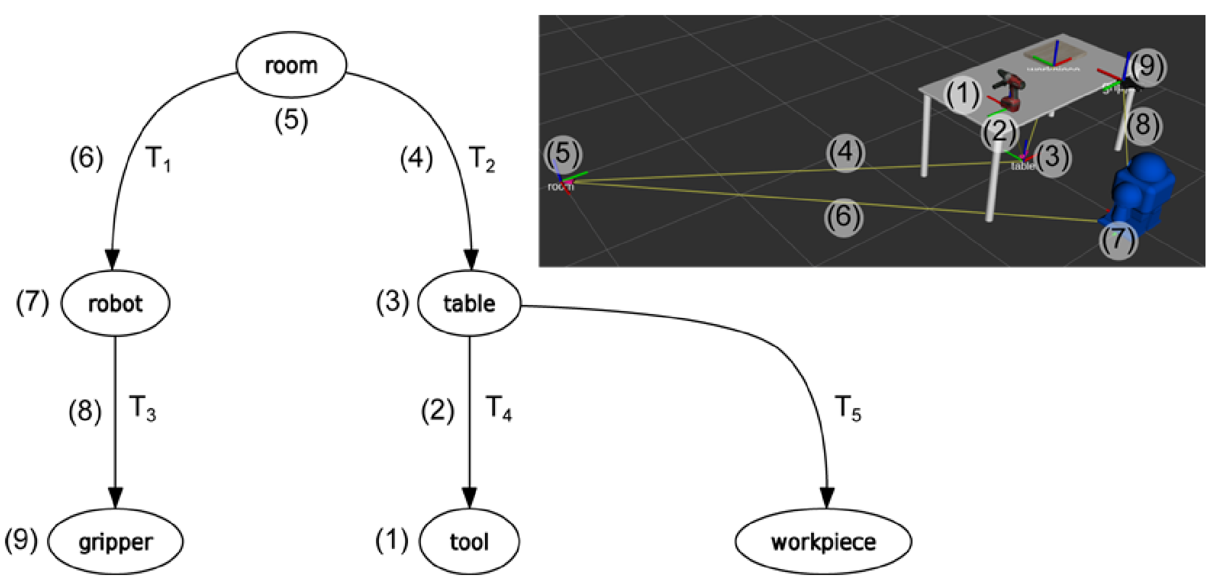


Figure 4.18 “Following the arrows” to determine how to calculate the transformation from tool to gripper coordinate system.

Traveling along the path from (1) to (9), we build up our transformation from left to right. When crossing an arrow in forward direction, we multiply on the right side with the transformation $\mathbf{T}_i$ associated with this arrow. When traversing an arrow in opposite direction, we multiply on the right side with the inverse transform $\mathbf{T}_i^{-1}$. In our example this results in the final transformation

$$\mathbf{T} = \mathbf{T}_4^{-1}\mathbf{T}_2^{-1}\mathbf{T}_1\mathbf{T}_3$$

In mathematical notation, we will denote the reference coordinate system of a pose $\mathbf{P}$ as a superscript prefix. For example, if pose $\mathbf{P}_1$ is given in the world coordinate system w, we will write ${}^{w}\mathbf{P}_1$. For a pose $\mathbf{P}_2$ in the robot frame r, the symbolic expression is ${}^{r}\mathbf{P}_2$. We can extend this notation to transformations. The pose of coordinate system B in coordinate system A can be written as ${}^{A}\mathbf{P}_B$. This is commonly expressed as “pose of B with respect to (frame) A”. As we discussed, the pose ${}^{A}\mathbf{P}_B$ can also be understood as describing the transformation from coordinate system B to A. Hence, we will write ${}^{A}\mathbf{T}_B$ for the transformation from coordinate system B to A. A spatial location given in frame B, ${}^{B}\mathbf{P}$, can be converted to a representation in frame A by ${}^{A}\mathbf{P} = {}^{A}\mathbf{T}_B \cdot {}^{B}\mathbf{P}$. This notation has the advantage that one can visually validate the correctness of a chain of transformations by observing that superscript and subscript of adjacent transformations need to match:

$$^{A}\mathbf{T}_E = {}^{A}\mathbf{T}_B \cdot {}^{B}\mathbf{T}_C \cdot {}^{C}\mathbf{T}_D \cdot {}^{D}\mathbf{T}_E$$

The inverse of ${}^{A}\mathbf{T}_B$, i.e. ${}^{A}\mathbf{T}_B^{-1}$, is ${}^{B}\mathbf{T}_A$.

When it comes to variable names and coordinate systems, there are two ambiguities we want to avoid. First, the meaning or usage of a transformation. A transformation can either be used to describe a change in position or a change of coordinate system. One can use the following terminology to avoid mixing up these concepts. Moving (translating and/or rotating) an object uses an *active* transformation. Changing the coordinate system uses a *passive* transformation. Second, we need to be able to keep track of the coordinate system a pose is specified in the source code. With keeping track I mean we, as robot software engineers, need to understand what coordinate is used for each pose from reading the code.

My recommendation is to explicitly spell out what you are doing in your code as part of variable and function names. For example, I find the following naming conventions helpful:

- Suffix all poses with their coordinate system / basis, e.g. `p1_world`, `p2_table`, `p3_robot`.
- Moving (active transformation)
  - Use a three part name `frompose_mv_topose`. `mv` stands for “move”. Examples: `drillstart_mv_drillstop`, `tool_mv_handle`.
  - A move does not change the basis: `p2_base = p1_base @ frompose_mv_topose`
  - Composition: `frompose_mv_topose = frompose_mv_intermediatepose @`

intermediatepose_mv_topose

- Change of basis (passive transformation)
  - Use a three part name `oldbase_cbt_newbase`. `cbt` stands for "change basis to". Examples: `world_cbt_table`, `robot_cbt_gripper`.
  - A change basis operation has the structure `p1_newbase = oldbase_cbt_newbase @ p1_oldbase`
  - Composition: `oldbase_cbt_newbase = oldbase_cbt_intermediatebase @ intermediatebase_cbt_newbase`
- Don't use a `cbt` variable for a `mv` operation or the other way around. If you want to use the same transform with both meanings, create an alias variable.

Note that in the change basis operation `oldbase_cbt_newbase` is multiplied on the left side and in the move operation `frompose_mv_topose` is multiplied on the right side. As stated previously, the order of matrix multiplications is significant, in general

$$\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$$

In other words, you will get the wrong result if performing the operations in a different order.

Instead of repeatedly writing code that multiplies various poses and transformations in order to get the right pose/transformation in the right coordinate system, it is much better to have a generic coordinate transformation library/subsystem that hides these details.[167] There are really only a few basic operations required:

- Update the coordinate transformation tree[168]
- Retrieve a pose or transformation between two coordinate systems
- Calculate a new pose/transformation

An abstract API for these operations can be defined as

```
# Set transform between parent_frame and child_frame
# to parent_frame_cbt_child_frame
set_transform('parent_frame', 'child_frame', parent_frame_cbt_child_frame)

# Retrieve transform between parent_frame and child_frame
parent_frame_cbt_child_frame = get_transform('parent_frame', 'child_frame')

# Change basis of pose_some_frame to target_frame
pose_target_frame = to_frame(pose_some_frame, 'target_frame')

# Calculate with poses
# Example: Calculate pose for robot gripper, given tool handle in tool
#          coordinates, offset grasp point to gripper pose
#          and offset gripper to TCP
tool_grip_pose_tool = (tool_handle_pose_tool @ grasp_point_mv_gripper
                       @ gripper_mv_tcp)
manipulator.move(to_frame(tool_grip_pose_tool, 'robot'))
```

This concludes our journey through the most common representation of coordinate systems, poses and transformations used in robotics. Let's close our introduction to geometry for robot software with a few exercises before continuing with motion in the next chapter.

4.6 Exercises on Geometry

For the exercises, we use `numpy` (NumPy) and `transformations` (also known as transformations.py). Both can be easily installed from the Python Package Index (PyPI) using `pip`.[169] Make use of the numpy and transformations documentation.[170] Please include numpy, transformations and some math functions in the following manner for the exercises:

```
from math import radians, sin, cos
import numpy as np
from transformations import *
```

SIDEBAR

Exercise 4.1: Calculate the result of moving the 2D point at x = 1 and y = 3 by x = 3 and y = 2 using vectors.

Solution 4.1:

```
p = np.array([1, 3])
t = np.array([3, 2])
print(p + t)  # [4 5]
```

SIDEBAR

Exercise 4.2: Combine two 3D translations t_1 = (2, 0, 0) and t_2 = (0, 1, 3) into one translation t_3 using two different calculations and verify that they have the same result.

Solution 4.2:

```
t1 = np.array([2, 0, 0])
t2 = np.array([0, 1, 3])
t3a = t1 + t2
t3b = t2 + t1
print(np.allclose(t3a, t3b))  # True
```

SIDEBAR

Exercise 4.3: Rotate the 2D vector v = (3, 0) by 45 degree, print the result, rotate the resulting vector by 45 degree again, print it, rotate the resulting vector by 270 degrees and print the result.

Solution 4.3:

```
def rotate(v, theta):
  rotation_matrix = np.array([[cos(theta), -sin(theta)],
                              [sin(theta), cos(theta)]])
  return rotation_matrix @ v

v = np.array([3, 0])
v1 = rotate(v, radians(45))
print(v1)  # rounded: [2.12 2.12]
v2 = rotate(v1, radians(45))
print(v2)  # rounded: [0.0 3.0]
v3 = rotate(v2, radians(270))
print(v3)  # rounded: [3.0 0.0]
```

SIDEBAR

Exercise 4.4: Create a pose in homogenous coordinate representation for position x = 3, y = 2, z = 0 and orientation roll = 90° pitch = 0° yaw = -90° and print it.

Solution 4.4:

```
p = translation_matrix([3, 2, 0]) @ euler_matrix(radians(90), 0,
                                                 radians(-90), axes='sxyz')
print(p)
# rounded:
# [[ 0.0  0.0 -1.0  3.0]
#  [-1.0  0.0  0.0  2.0]
#  [ 0.0  1.0  0.0  0.0]
#  [ 0.0  0.0  0.0  1.0]]
```

SIDEBAR

Exercise 4.5: Show by example that the order of 3D rotations matters, i.e. show that 3D rotations are not commutative.

Solution 4.5:

```
r1 = euler_matrix(radians(90), 0, 0, axes='sxyz')
r2 = euler_matrix(0, 0, radians(90), axes='sxyz')
r1r2 = r1 @ r2
r2r1 = r2 @ r1
print(np.allclose(r1r2, r2r1))  # False
print(r1r2)
# rounded:
# [[ 0.0 -1.0  0.0  0.0]
#  [ 0.0  0.0 -1.0  0.0]
#  [ 1.0  0.0  0.0  0.0]
#  [ 0.0  0.0  0.0  1.0]]
print(r2r1)
# rounded:
# [[ 0.0  0.0  1.0  0.0]
#  [ 1.0  0.0  0.0  0.0]
#  [ 0.0  1.0  0.0  0.0]
#  [ 0.0  0.0  0.0  1.0]]
```

The same rotations were used in figure 4.12. When you read the columns of the rotation matrix as vectors, you can see that the results match. In `r1r2` the first column is (0, 0, 1), which you can read as the new direction of the x axis, which points in along the previous z axis. The second column is (-1, 0, 0). This tells us that the former y axis points along the old negative x axis. Finally the third column (0, -1, 0), gives us the new z axis as pointing along the old negative y axis. This matches figure 4.12(a2). You can check for yourself that `r2r1` matches figure 4.12(b2).

SIDEBAR

Exercise 4.6: Calculate the transformation T from workpiece to tool in figure 4.8 given T_4 = (x, y, z, roll, pitch, yaw) = (-0.5, -0.2, 0.75, 90°, 0, 0) and T_5 = (0.2, 0, 0.78, 0, 0, 0).

Solution 4.6:

```
t4 = (translation_matrix([-0.5, -0.2, 0.75])
      @ euler_matrix(radians(90), 0, 0, axes='sxyz'))
t5 = translation_matrix([0.2, 0, 0.78])
t = inverse_matrix(t5) @ t4
print(t)
# rounded:
# [[ 1.0  0.0  0.0 -0.70]
#  [ 0.0  0.0 -1.0 -0.20]
#  [ 0.0  1.0  0.0 -0.03]
#  [ 0.0  0.0  0.0  1.00]]
```

Take a close look at figure 4.6, in particular the workpiece coordinate system. Try to see that the signs of the translation vector (-0.7, -0.2, -0.03) are correct, i.e. check where the tool is relative to the workpiece coordinate system.

You have now learned enough about *static* objects and their geometric relationships. In the next chapter we continue with *moving* objects - a robot that doesn't move wouldn't really be a robot after all.

4.7 Summary

- An object has two main geometric properties: shape and pose.
- The shape includes information about the space occupied by the object, such as size and outline.
- The pose specifizes the location of the object, position and orientiation, relative to a coordinate system.
- An essential property of spaces is their dimension. Dimension can be defined as the minimum number of values, or coordinates, required to specify any point (position only) within a space. The dimension is abbreviated as 1D, 2D or 3D.
- A coordinate system is a set of numbers that uniquely determine the pose of an object in a given space.
- Right-handed Cartesian coordinate systems are commonly used. The right-hand rule is a great mnemomic and thinking aid for right-handed Cartesian coordinate systems.
- Other important coordinate systems include the 2D polar coordinate system, the 3D spherical coordinate system and the 3D cylindrical coordinate system.
- Pose parameters (PP) refers to the number of independent values required to specify the pose, i.e. position and orientation, of an object in space. It is the number of independent position coordinates plus the number of independent orientation coordinates.
- In a robotics context, the degree of freedom (DoF) is the number of (independently actuated) joints in a robot.
- Using multiple coordinate systems to describe poses greatly simplifies working with robot systems. A coordinate transformation tree relates the different coordinate systems to each other and enables changing easily between them.
- Vectors can be used to represent 2D and 3D positions and translations.
- Matrices, more specifically rotation matrices, can be used to represent 2D/3D orientations and rotations.
- Axis-angle and quaternions are alternative representations for 3D rotations.
- The order of rotations is important in 3D. Performing the same rotations in a different order can lead to a different result.
- Homogenous coordinates can represent poses and transformations in a more uniform manner, compared to handling position/translation and orientation/rotation separately.
- Common representations for shapes include constructive solid geometry (CSG), polygon meshes, curve models (e.g. NURBS), voxel grids and point clouds.
- Using a coherent naming convention helps to avoid confusion about the coordinate system for a given pose.
- Variable names should also clearly indicate the intended semantics for each pose/transform. They should either represent a change in pose (active transformation) or a change of coordinate system (passive transformation).
- The implementation details of handling poses, transformations and coordinate transformation trees should be hidden behind a good API.

5 Robot Motion 2: Kinematics

This chapter covers

- Kinematics: Trajectories and the conversion between joint and task space
- Analytical and numerical approaches to kinematics
- Differentiation and integration as essential mathematical tools

While the previous chapter on geometry laid the foundation for robot motion, nothing actually moved yet. In this chapter, we will set things into motion. Motion at its core is the change of geometry, especially poses, over time. We need a formalism that allows us to concisely describe change (over time). Calculus provides us with the required mathematical tools to model change in our robot software. Thus, you will become familiar with concepts from calculus, *derivatives* and *integrals*, throughout the chapter.

The goal of this chapter is to give you a solid basic understanding of the topics mentioned. When finishing the chapter you will have gained the *general* knowledge and skills in calculus and kinematics. This enables you to work with and learn more about the kinematics of *specific* robots in later chapters. The following two chapters on dynamics and control take a similar approach: Solid overview up front, details and refinements later on, e.g. when discussing specific robot applications.

5.1 Kinematics introduction

The previous chapter was all about working with objects in space. This chapter is all about the motion of these objects.

For our purposes, it is sufficient to define motion as the change of an object's pose over time. Similar to poses, motion is always relative to something, usually a (relatively) fixed coordinate system. Hence, we are aiming for a description that enables us to calculate *changes* of object poses *over time*, instead of describing constant stationary poses as we have done before.

In robotics, *kinematics* comes up in two major contexts:

- Trajectories, i.e. time parametrized paths
- Kinematic equations that describe the relation between joint space and 3D space

A *path* is simply an ordered sequence of poses. Paths are geometric construct.[171] They do not describe motion because there is no notion of time in a path. A *trajectory* adds the dimension of time to the sequence of poses. For this reason, trajectories describe how an object moves along the poses over time. These time parameters result in concepts beyond geometry such as velocity, acceleration and jerk. We will soon look at them one by one, but let's first summarize the second meaning of kinematics in a robotics context.

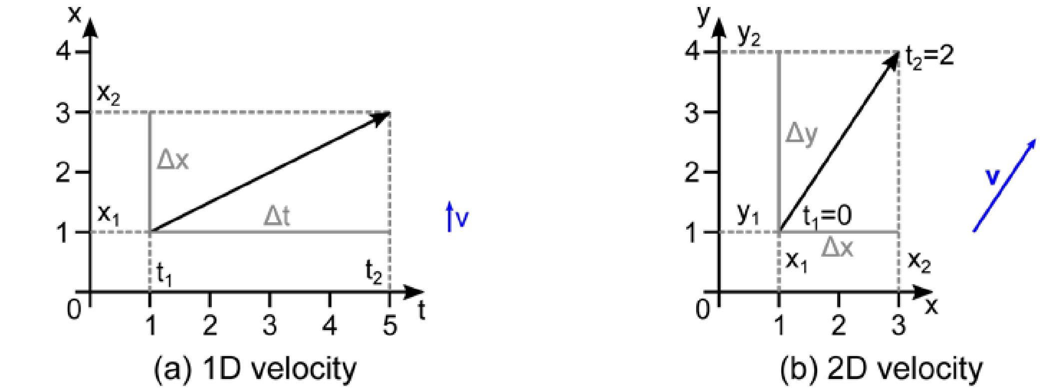
As we discussed in chapter 2, robots consist of links and joints. Some or all of the joints are actuated by motors, allowing us to purposefully move the robot in its environment. A natural way to think of robot motion is the position and motion of the individual joints, i.e. the robot's degrees of freedom (DoF). At the same time, the real world and objects in it are most naturally described in 3D with 6 pose parameters (PPs). Robot kinematics, in the sense of kinematic equations, give us the tools to convert back and forth between these two descriptions. The joint perspective is described in terms of joint space with a number of DoF. The 3D perspective, familiar from the previous chapter, is often referred to as Cartesian space or task space in this context.

In the next sections, we acquire the tools to describe changes of the robot's state in software, especially changes of pose over time. Afterwards, we make use of these tools to write code that translate between joint space and task space. You will find that these robot base layer functions are pervasive in sensing, planing and acting. The robot needs to know where it is, where its end effectors are and along which trajectory it must move itself and/or its EEs in order to achieve a given objective.

5.2 Velocity and Acceleration

While I assume you already have an idea about velocity, acceleration, perhaps also jerk, either from everyday life or from formal education, I want to at least give you a brief introduction / recapitulation of these three key terms.

Velocity describes the *rate of change* in an object's pose.[172] It takes geometry and time into account. In the most simple case, constant velocity, we can calculate the velocity by dividing the distance between two poses by the time it takes to go from one to the other. In terms of notation, *time* is associated to the symbol t. Figure 5.1 illustrates such a constant velocity in 1D and 2D.



$$v = \frac{\Delta x}{\Delta t} = \frac{x_2 - x_1}{t_2 - t_1} = \frac{3-1}{5-1} = \frac{2}{4} = 0.5 \qquad \mathbf{v} = \begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix}\frac{1}{\Delta t} = \begin{bmatrix}3-1\\ 4-1\end{bmatrix}\frac{1}{2-0} = \begin{bmatrix}2\\ 3\end{bmatrix}\frac{1}{2} = \begin{bmatrix}1\\ 1.5\end{bmatrix}$$

Figure 5.1 Illustration of constant velocity in 1D and 2D. Note the different visualizations and axis labels: (a) 1D, x over time; (b) 2D, (x, y) at times t_1 and t_2.

Given the start pose $\mathbf{P}_1$, the end pose $\mathbf{P}_2$ together with the time at the start t_1 and at the end t_2, we can express the calculation of velocity **V** in a formula

$$\mathbf{V} = \frac{\mathbf{P}_2 - \mathbf{P}_1}{t_2 - t_1}$$

Because poses consists of both position (in units of meters) and orientation (in units of degrees or radians), the velocity of moving between them cannot be expressed in a single number. Instead position and orientation are treated separately. This also results in a separation of *linear velocity* **v** (derived from change in position) and *angular velocity* **ω** (omega) (derived from change in orientation). You will learn more about *twists* **V** that represent linear and angular velocity together later on. For now, we focus on either one of the two aspects in isolation. Thus, we restate the above formula for linear velocity **v** and positions $\mathbf{p}_1$ and $\mathbf{p}_2$ only

$$\mathbf{v} = \frac{\mathbf{p}_2 - \mathbf{p}_1}{t_2 - t_1}$$

Note that **p** is a frequently used symbol for positions in this context.[173] We are dividing a vector $\Delta\mathbf{p} = \mathbf{p}_2 - \mathbf{p}_1$ by a scalar $\Delta t = t_2 - t_1$, thus the result is a vector **v**.[174] The notation Δ**p** (Delta **p**) and Δt (Delta t) is just a common way of naming variables that represent a finite difference, such as the difference between the two positions respectively the two times.

Besides this *spatial* linear velocity expressed as vector, there is also the *scalar* linear velocity, which might be the more familiar one.[175] The good thing is that the scalar velocity is simply the norm of the spatial velocity: $v = |\mathbf{v}|$. While only few people - at least outside robotics - would state that they are “currently driving (60 km/h north, 25 km/h west)”, it is only a more precise description of “currently driving 65 km/h (north-northwest)”.

If the velocity is not constant, but also varies over time, we need to look at acceleration. *Acceleration* describes the *rate of change* in an object's (spatial) *velocity*. *Linear acceleration* is denoted by **a** and *angular acceleration* is denoted by **ω'** (omega'). We can go through exactly the same considerations for acceleration as we just did for velocity, but with poses replaced by velocities.

The common concept here is that of *derivatives*. The derivative of a function describes the "instantaneous rate of change" of the function. *Differentiation* is the term used for the process of calculating derivatives.

Let's connect derivatives with kinematics step by step. First, we describe motion as a function. Second, we study an intuitive explanation of derivatives. Third, we learn how pose, velocity, acceleration and jerk are connected by derivatives.

We can define motion as a function with one parameter t for time that specifies the object's pose at time t. For a stationary object, i.e. one that does not move, we can write $\mathbf{p}(t) = \mathbf{p}$. The function has the same value, independent of the time parameter. A little more interesting is an object moving at a constant velocity **v**: $\mathbf{p}(t) = \mathbf{p}_0 + t \cdot \mathbf{v}$. For t = 0, we get $\mathbf{p}(0) = \mathbf{p}_0$. Thus we call $\mathbf{p}_0$ the initial position. Figure 5.2 shows a 2D example, which can, for example, describe a vacuum robot traversing a room.

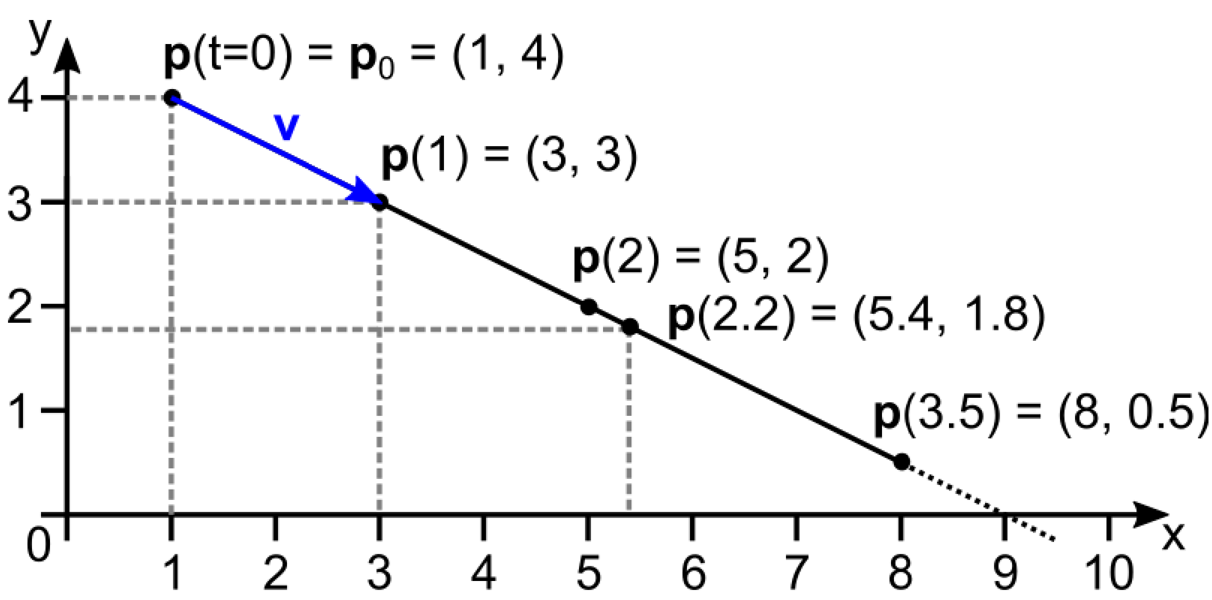


Figure 5.2 Example of a 2D motion $p(t) = p_0 + t \cdot v$ with constant velocity v = (2, -1) and initial position p_0 = (1, 4).

Although the figure only provides values for the position at certain times t = 0, 1, 2, 2.2, 3.5, the change of position is assumed to be continuous. The position is specified for *any* time t. This is illustrated by the continuous line in the figure. As stated above, we can use poses instead of positions here - but we stick to positions only as it eases explanation. If you think of poses or orientations, one difference you would likely notice is the periodic nature of orientations.[176]

Taking the next step towards making our motion equation more general, we allow for velocities to change over time: $\mathbf{p}(t) = \mathbf{p}_0 + \ldots \mathbf{v}(t) \ldots$. Now the velocity is no longer a constant value, but in itself a function of time $\mathbf{v}(t)$. In Python we can express this as the difference between `position1()` and `position2()`:

```
# Constant velocity
initial_position = ...
constant_velocity = ...
def position1(time):
  return initial_position + time * constant_velocity

# Varying velocity
initial_position = ...
def velocity(time):
  return ...
def position2(time):
  return initial_position + ... velocity(time) ...
```

We will resolve the ellipses (…) in the above formula and code in a few paragraphs when introducing integrals. For now it is more important to see how we can express motion as a function over time and how rates of change come into the picture. We already defined acceleration as the rate of change in velocity. Hence the velocity at time t can be calculated from an initial velocity and acceleration, i.e. $\mathbf{v}(t) = \mathbf{v}_0 + \ldots \mathbf{a}(t) \ldots$. Again acceleration can be constant, $\mathbf{a}(t) = \mathbf{a}$, or also change over time.

5.3 Derivatives

Let's now learn how *derivatives* can help us to calculate the rate of change for a given function. My goal here is not to formally teach you calculus, since this would require a books of its own. Rather, independently of your prior knowledge of this subject, I want to ensure that you have an intuitive grasp of the involved concepts, so we can use them in this and in later chapters.

As example we select the 1D position of a robot moving down the sidewalk along a (straight) road, e.g. a last mile delivery robot. Given the motion is 1D and along the x-axis, we use the symbol x to denote position. Figure 5.3 visualizes a few seconds of the sidewalk robot's journey.

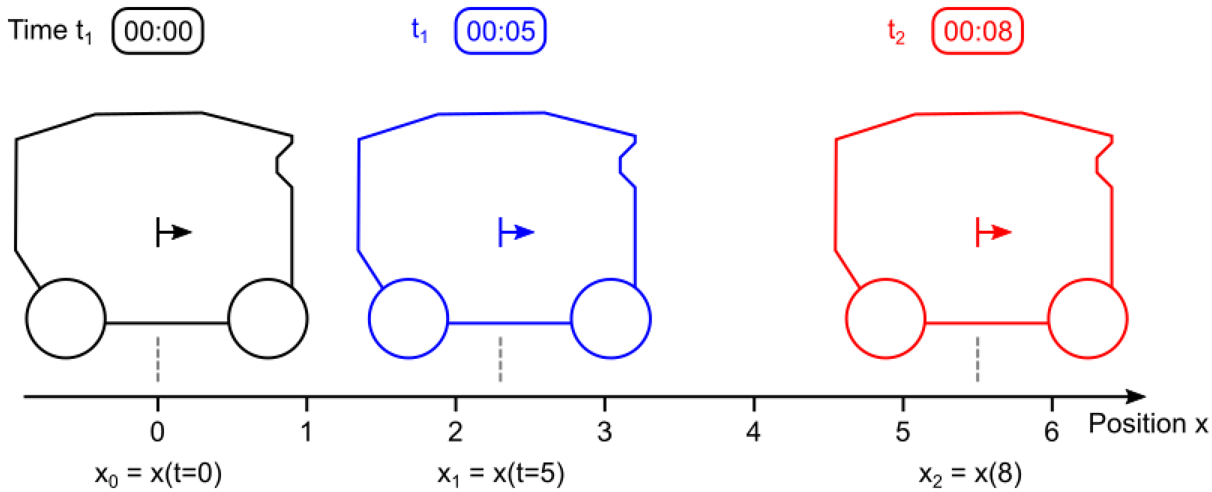


Figure 5.3 Illustration of the sidewalk robot and the two parameters of interest: position x and time t.

In order to program the robot, we need to be able to answer questions such as

- Given the robot's velocity over time, at which position is it right now?
- Given the target position and an arrival time, how fast must the robot move to get there on time?
- Given the maximum deceleration of the robot's brakes and the range of our obstacle sensors, how fast can we allow the robot to move and still ensure that it is able to stop before hitting an obstacle?

Answering these and similar questions requires a basic understanding of how to work with positions, velocities and accelerations. The mathematical tools that enable us to code the answers into our robot's software are derivatives and integrals.

The velocity of the robot is not constant because it first needs to speed up from standstill and it has to slow down at the end. Furthermore, it also needs to adapt its speed to obstacles along the way, e.g. pedestrians. The position over time thus looks as shown in 5.4.

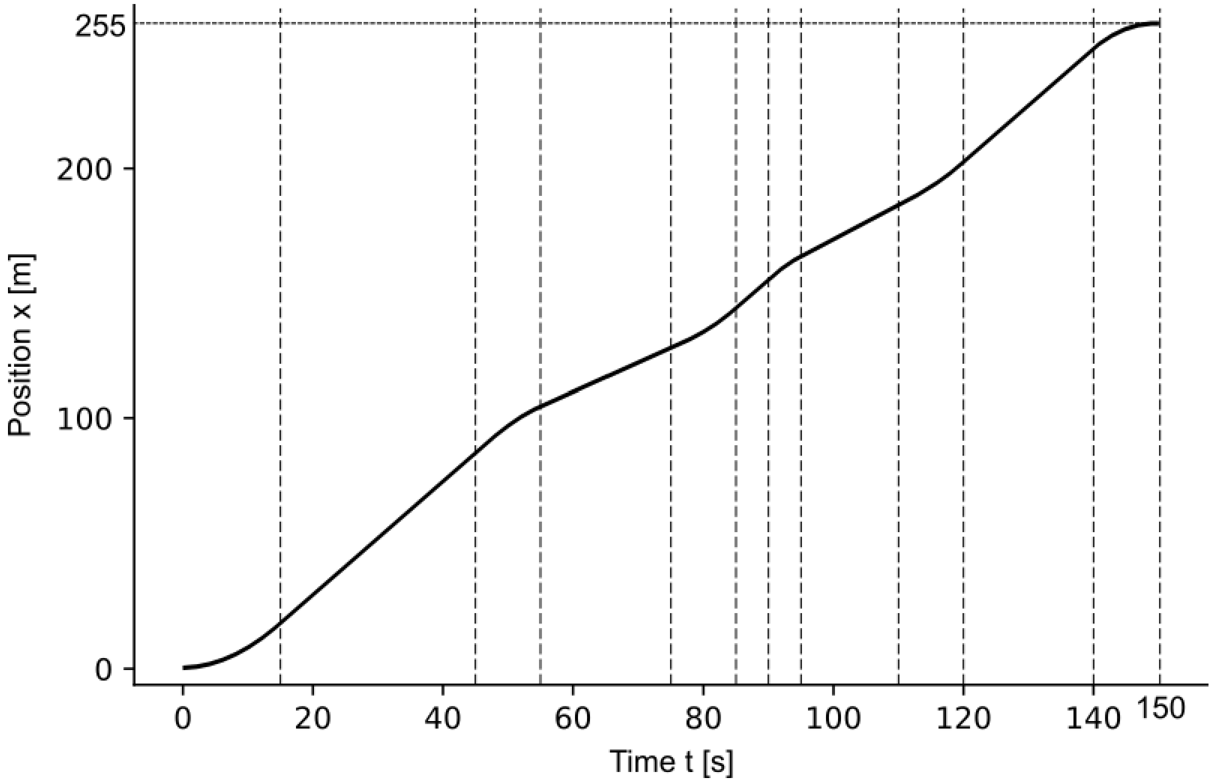


Figure 5.4 Change of the sidewalk robot's position x over time t.

In this example, the position always increases, but we can see by the different slopes of the curve that it increases at different rates. We also observe that the slopes do not have the same shape. Some are line segments, while others are curved. We can easily calculate the *average* velocity between $t_1 = 0$ and $t_2 = 150$ using the formula

$$v = \frac{x_2 - x_1}{t_2 - t_1}$$

At t = 0 the robot's position is x(0) = 0 and at t = 150 it is x(150) = 255. Thus, we get v = (255 - 0)/(150 - 0) = 1.7.[177]

However, we are not only interested in the average velocity, but also in the *momentary* velocity for each point in time. In the time interval between t = 15 and t = 45 the change in position is a straight line, which means that the velocity is constant in this interval. We can compute the velocity of this segment by the same formula as before. The only difference is our choice of t_1 and t_2. Reading the positions from figure 5.4, we get v = (85.5 - 18)/(45 - 15) = 2.25. Figure 5.5 visualizes this calculation.

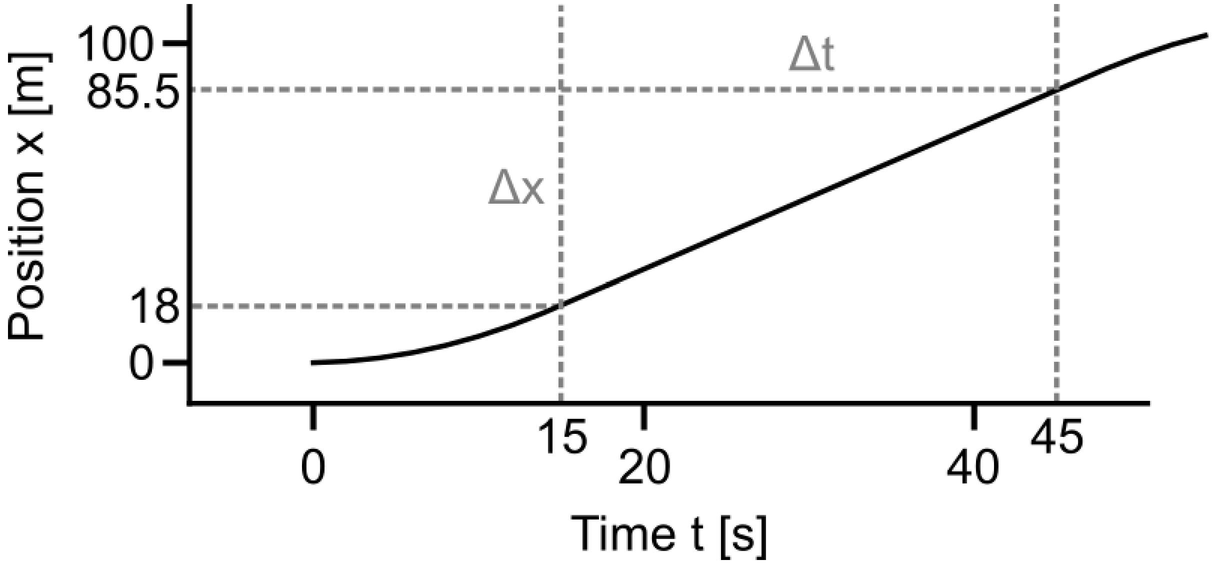


Figure 5.5 Calculation of velocity v for segment t_1 = 15 and t_2 = 45.

We can repeat the same calculation for all constant velocity segments of the trajectory, such as t = [55, 75] and t = [120, 140].[178] But what about the curved segments that originate from changing velocities, i.e. where acceleration is different from zero? This is where derivatives will help us.

We want to know the velocity and acceleration values for the first segment t = [0, 15]. If we use our approach thus far, we end up with the result shown in figure 5.6.

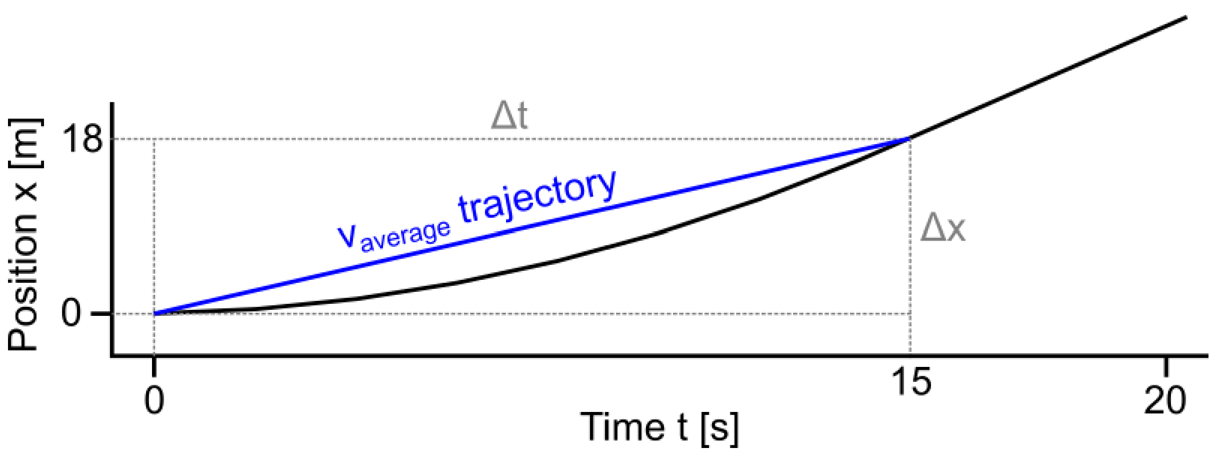


Figure 5.6 Calculation of (average) velocity v for segment t_1 = 0 and t_2 = 15, i.e. t = 15.

We are actually calculating the average velocity of the entire segment, instead of getting the robot's momentary velocity for each point in time. The critical insight now is that we can reduce the time interval to smaller and smaller intervals around our time point of interest and that each reduction in the interval brings us closer to the true momentary value. Figure 5.7 illustrates this process for the (momentary) velocity at t = 10.

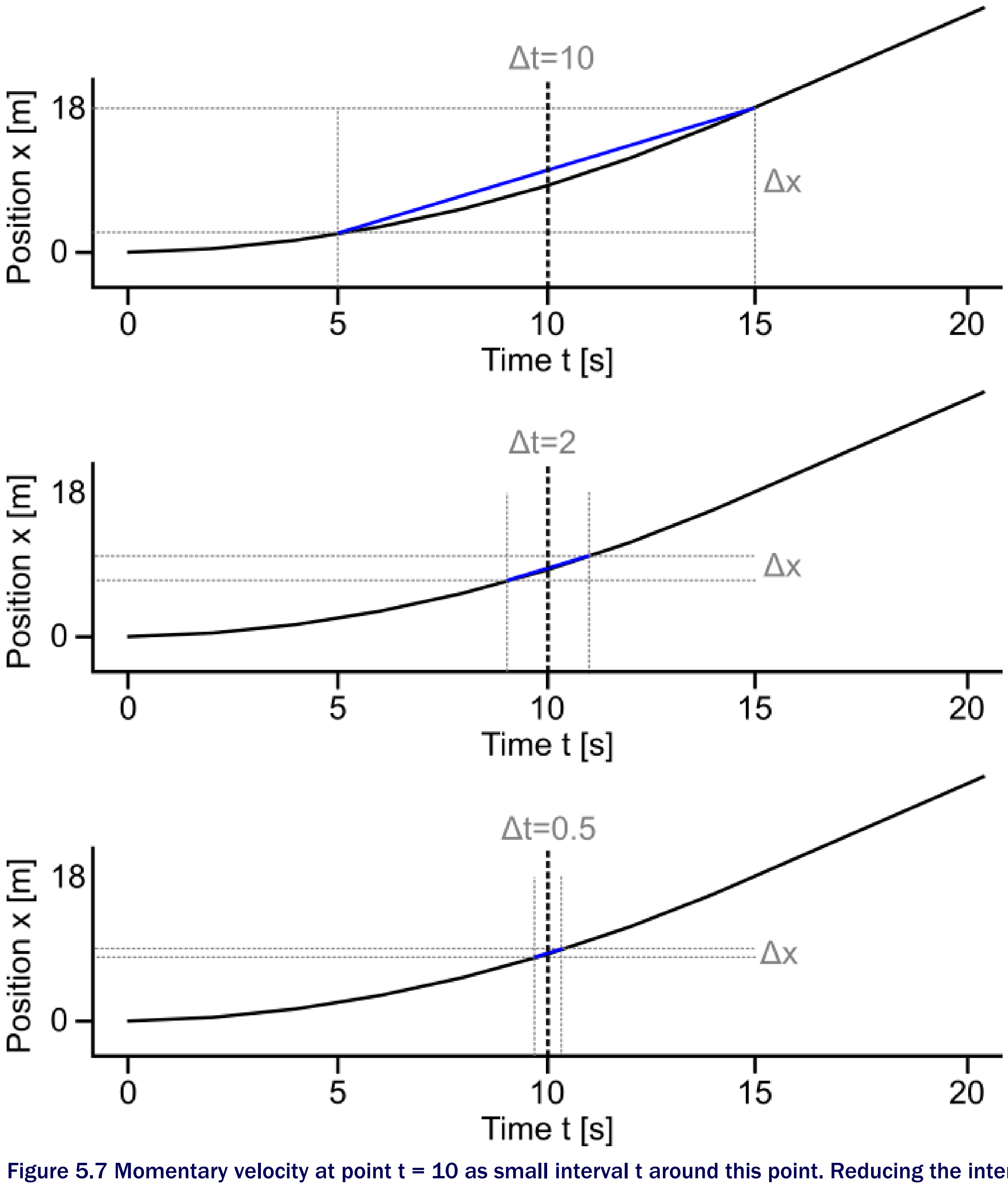


Figure 5.7 Momentary velocity at point t = 10 as small interval t around this point. Reducing the interval size further and further is the intuitive explanation for differentiation.

With each successive reduction of the interval size, the slope of the velocity line approaches more and more closely the local slope of the position change. If we make the time interval small enough around the time point of interest, the velocity line will exactly touch the position curve in this point. Put in mathematical terminology, the velocity line becomes the tangent line at our point of interest.

Repeating this process for all time points along the trajectory, gives us the velocity function v(t) that we have been looking for. You can see the result in figure 5.8.

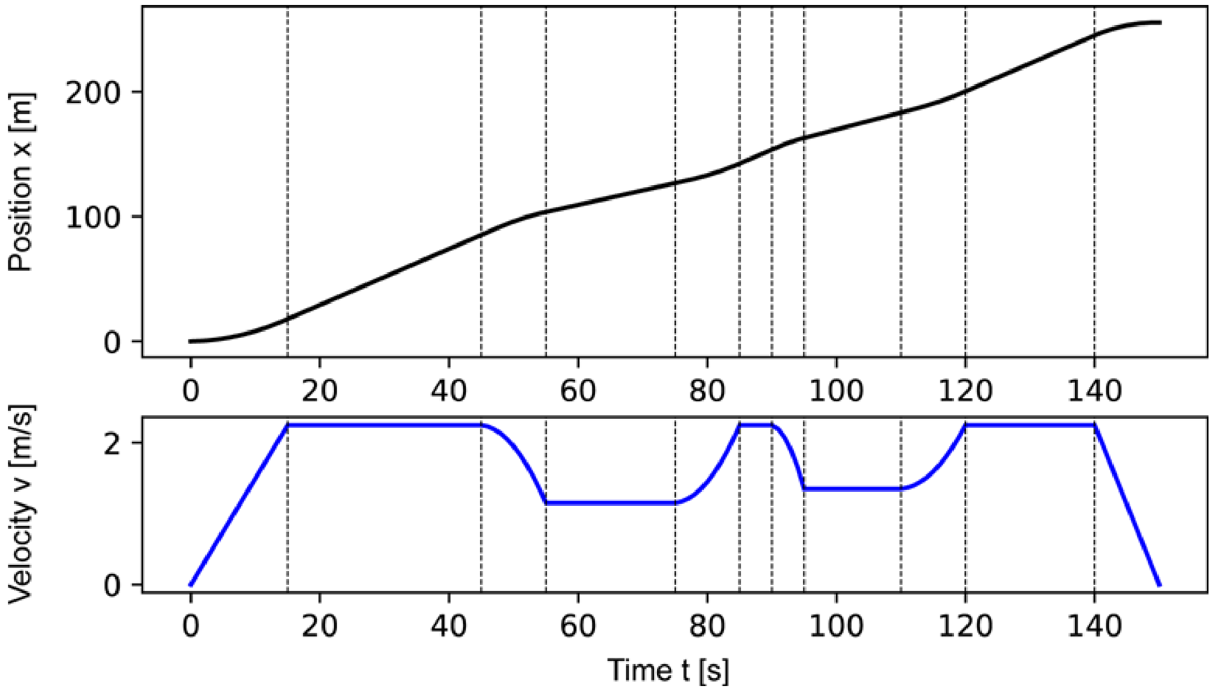


Figure 5.8 Position function x(t) and the corresponding velocity function v(t). The velocity function is the derivative of the position function.

The relation between the position function x(t) and the velocity function v(t) is a derivative relationship. v(t) is the derivative of x(t). In calculus notation, the derivative of a function is commonly denoted in one of the following manners:

- Prime mark (') notation: x'(t) = v(t)
- Leibniz's notation: $\frac{d}{dt}x(t) = v(t)$

We need to discuss two practical challenges with this intuitive approach to differentiation. First, how small is "small enough" for the interval. Second, how to calculate derivatives in practice.

In theory, we need to calculate with *infinitesimal* small intervals, that is intervals of a size approaching zero. This is expressed as a limit function

$$\frac{d}{dt}x(t) = v(t)$$

We need not go into more details here, but keep in mind that whenever we are using a finite interval, we are actually calculating an approximation.[179]

This brings us to the practical question of how to perform differentiation, especially in software. Obviously it is neither practical to work with arbitrarily small intervals nor with arbitrarily many points as the required effort approaches infinity. Thus, we need to go with either of the following solutions.

- Symbolic differentiation
- Numerical differentiation

Both ways of doing differentiation are frequently used in robotics. We will make use of symbolic differentiation in order to come up with the derivative of a given symbolic function. For example, given a formula that relates the position of our robot's joints to the robot's pose, symbolic differentiation enables us - using pen and paper - to come up with a formula that relates joint velocities to the overall robot velocity. We then program this derivative formula into our robot system. Numerical differentiation on the other hand is mostly used to calculate derivatives values during the runtime of the robot system. For example, given a wheel encoder that regularly provides the wheel's current position, numerical differentiation can be used to calculate wheel velocity.

Symbolic differentiation uses the fortunate situation that we know the derivative function for many common functions, such as polynomials, trigonometric functions, exponential functions, logarithms as well as certain combinations of them (e.g. products, quotients and chains). To give you some examples of these known relationships:[180]

$$\frac{d}{dx}x^2 = 2x$$

$$\frac{d}{dx}sin(x) = cos(x)$$

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$$

Using this knowledge, we can calculate the derivative function "on paper" and insert the solution into our software. Another possibility is to use software that can operate on mathematical formulas presented in a symbolic way by applying the same rules as a mathematician would utilize manually.

Numerical differentiation is an alternative, especially when the function we want to differentiate is given as data instead of in a symbolic form or when working with formulas for which it is too complex to find a symbolic solution. The numerical derivative is simply calculated by using finite intervals, instead of infinitesimal ones. Thus, repeating the calculations, which you have learned above, for each interval. The drawback is that the solution in only an *approximation* of the true values. Furthermore, it is not always easy to know in advance how small we need to choose the intervals in order to get a sufficiently accurate result.[181]

In general, differentiation can not only be performed on the original function, but can be repeated on the derivative. This gives us a second-order derivative of the original function. When talking about position and velocity, the second-order derivative of position is the same as the first-order derivative of velocity, i.e. acceleration. Expressed in mathematical symbols, we can state x'(t) = v(t) and x''(t) = v'(t) = a(t) or in Leibniz's notation:

$$\frac{d}{dt}x(t) = v(t)$$
$$\frac{d^2}{dt^2}x(t) = \frac{d}{dt}v(t) = a(t)$$

Continuing our example, calculating the derivative of the velocity function v(t), we get the acceleration function a(t) as shown in figure 5.9

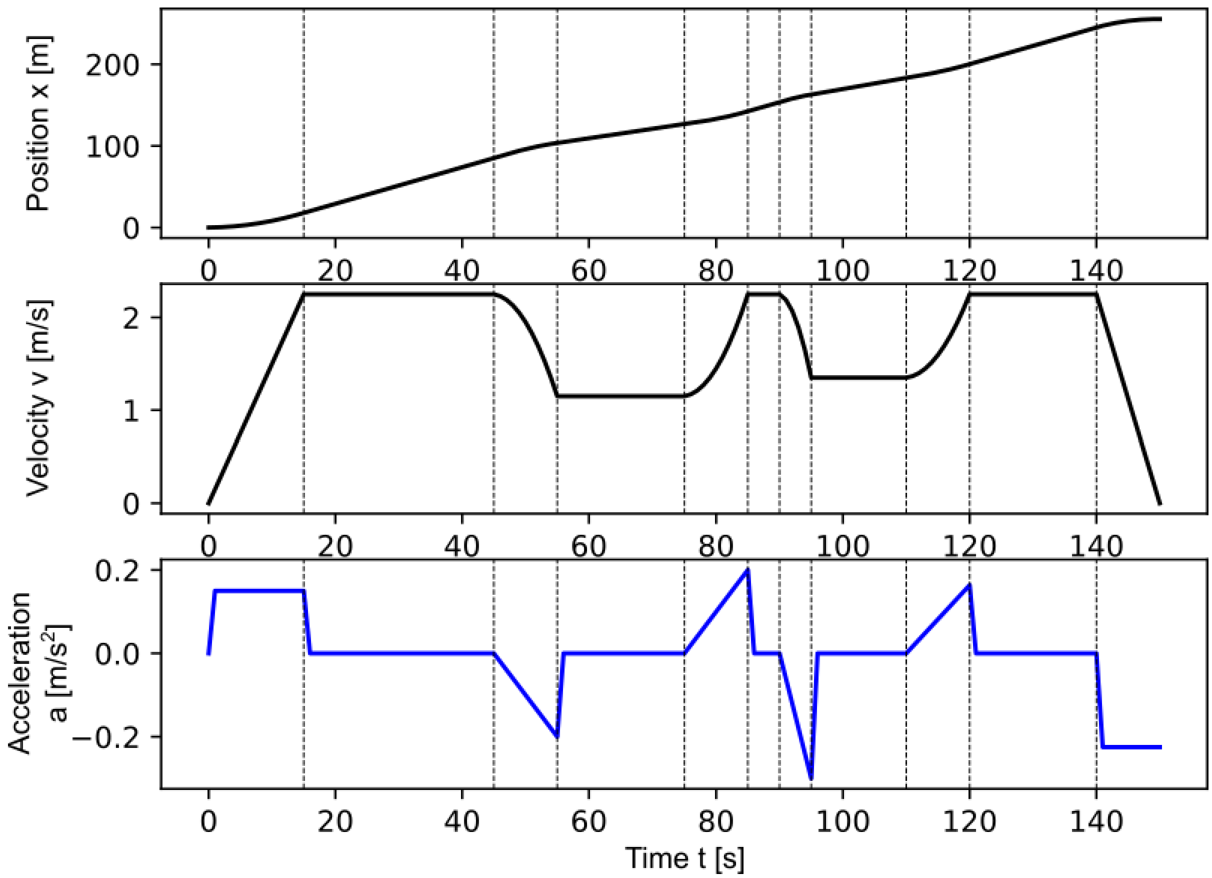


Figure 5.9 Position function x(t), velocity function v(t) and acceleration function a(t). The acceleration function is the second-order derivative of the position function and the first-order derivative of the velocity function.

Let's look at the units of the position, velocity and acceleration functions. Position x(t) is given in meters [m]. Velocity v(t) is the rate of change in position, therefore its unit is meters per second [m/s]. Acceleration a(t) is the rate of change of velocity, thus, we get meters per second per second, [(m/s)/s] = [m/(s·s)], or meters per second squared [m/s^2]. While position (meters) and velocity (meters per second) is familiar to most people - albeit perhaps more as miles per hour or kilometers per hour,[182] acceleration (meters per second per second) might not be as familiar. You'll likely have an easier time to get an intuitive grasp of [m/s^2] by following this line of thinking: If velocity answers the question "How quick is the change in position?", acceleration answers the question “How quick is the change in velocity?”. The quantitative answer to both questions is “… per second”. For position, the unit of position [m] is used for the blank. For velocity, the unit of velocity [m/s] is used.

Acceleration is not the final rate of change, which is relevant in robotics. There is at least one more: *jerk*. Jerk j(t) is the rate of change in acceleration, or put differently, the derivative of the acceleration function a(t), i.e. a'(t) = j(t). Jerk has the unit [m/s^3]. We could continue by looking at the derivative of jerk. However, it is unlikely that you will encounter derivatives beyond jerk in robotics.

After a lot of visual explanations accompanied by their mathematical expression, let’s look at derivatives in code. We start with a simple position function `pos1()`:

```
def pos1(t):  # t >= 0
  return 5 + 0.5 * t
```

Expressed using our mathematical notation convention, we could write `pos1()` as $x(t) = 5 + 0.5 \cdot t$. Figure 5.10 shows a plot of `pos1()`.

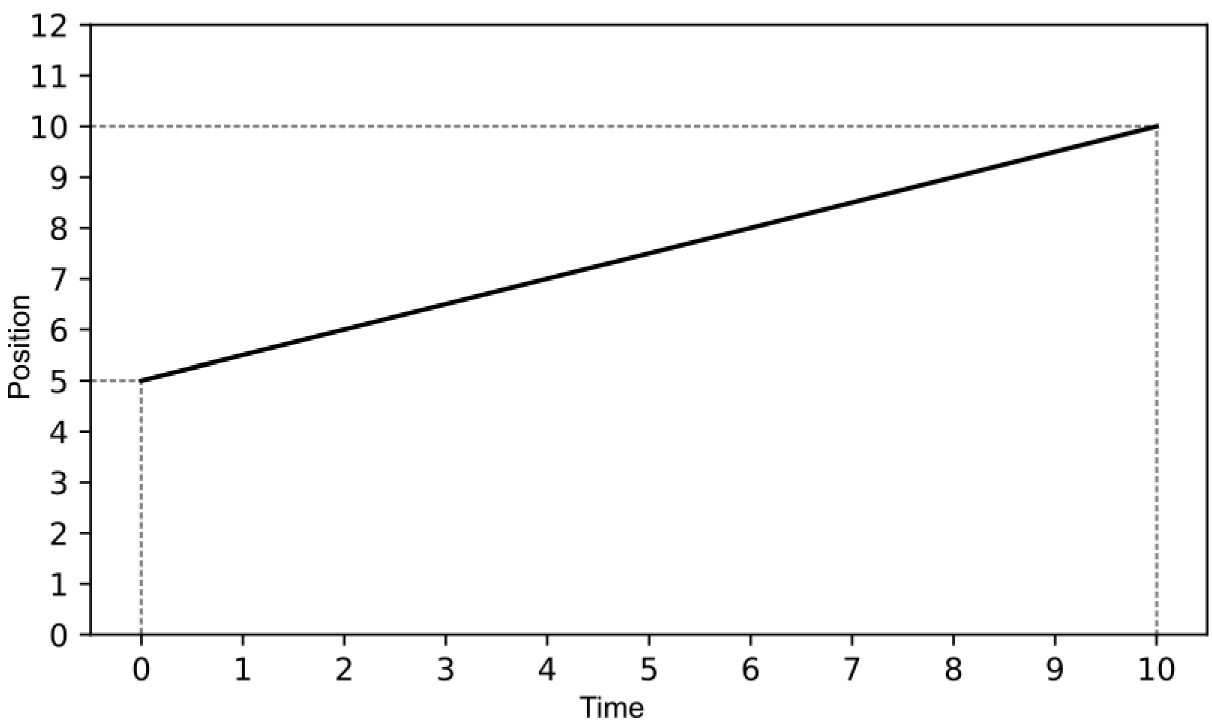


Figure 5.10 Plot of `pos1()` function for values between 0 and 10.

SIDEBAR **Matplotlib**

In case you have never worked with Python matplotlib, I encourage you to familiarize yourself with its basics.[183] Only 4 lines of Python are required to create a plot of `pos1()`, similar to the one shown in figure 5.10:

```
import numpy as np
from matplotlib import pyplot as plt

def pos1(t):  # t >= 0
  return 5 + 0.5 * t

t = np.linspace(0, 10, 100)   ❶
positions = pos1(t)   ❷

plt.plot(t, positions, 'k-')   ❸
plt.show()
```

❶ **Create array `t` with 100 equally spaced values between 0 and 10.**

❷ **Evaluate `pos1()` for each value in array `t` and store the results in array `positions`.**

❸ **Plot values in `t` (x axis) and `positions` (y axis).**

We will now calculate the derivative of the position function `pos1()`, i.e. the corresponding velocity function `vel1()`. More specifically, we will perform *numerical* differentiation in Python.

```
# Calculate f'(x):
# The numerical derivative of function f at x using interval size delta
def derivative(f, x, delta=0.01):
  return (f(x + delta/2) - f(x - delta/2)) / delta

def vel1(t):  # t >= 0   ❶
  return derivative(pos1, t)
```

❶ Actually t must be greater than or equal to the chosen value for delta (here 0.01) in order to satisfy the condition of `pos1()` to only receive values for parameters t with t >= 0.

Figure 5.11 shows a plot of `vel1()`.

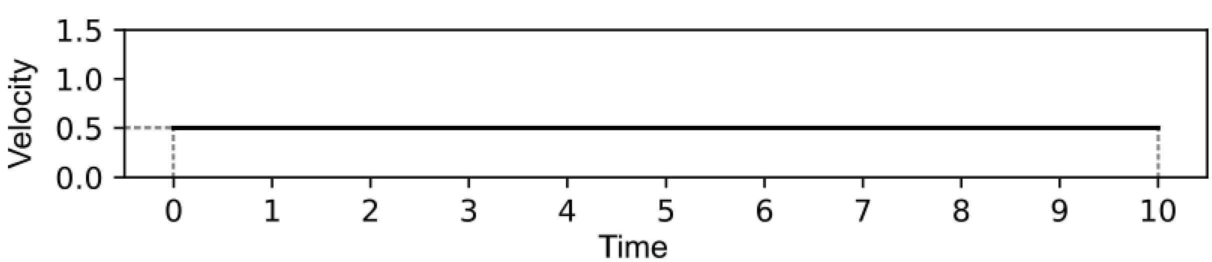


Figure 5.11 Plot of `vel1()` function for values between 0 and 10.

`vel1()`, the derivative of `pos1()`, is a constant function with the value 0.5.

Let's briefly look at a more complex position function `pos2()`:

```
def pos2(t):  # t >= 0
  if t <= 3:
    return 5 * t
  elif t <= 7:
    return 7.5 + 0.83 * t**2
  else:
    return 21.1 + 0.079 * t**3
```

Using the function `derivative()` defined above, we can directly define `vel2()` in the same manner as before:

```
def vel2(t):  # t >= 0
  return derivative(pos2, t)
```

Performing the same operation, we can also differentiate `vel2()` to get the acceleration function `acc2()`, i.e. the second-order derivative of `pos2()`:

```
def acc2(t):  # t >= 0
  return derivative(vel2, t)
```

Figure 5.12 shows a plot of `pos2()`, `vel2()` and `acc2()`.

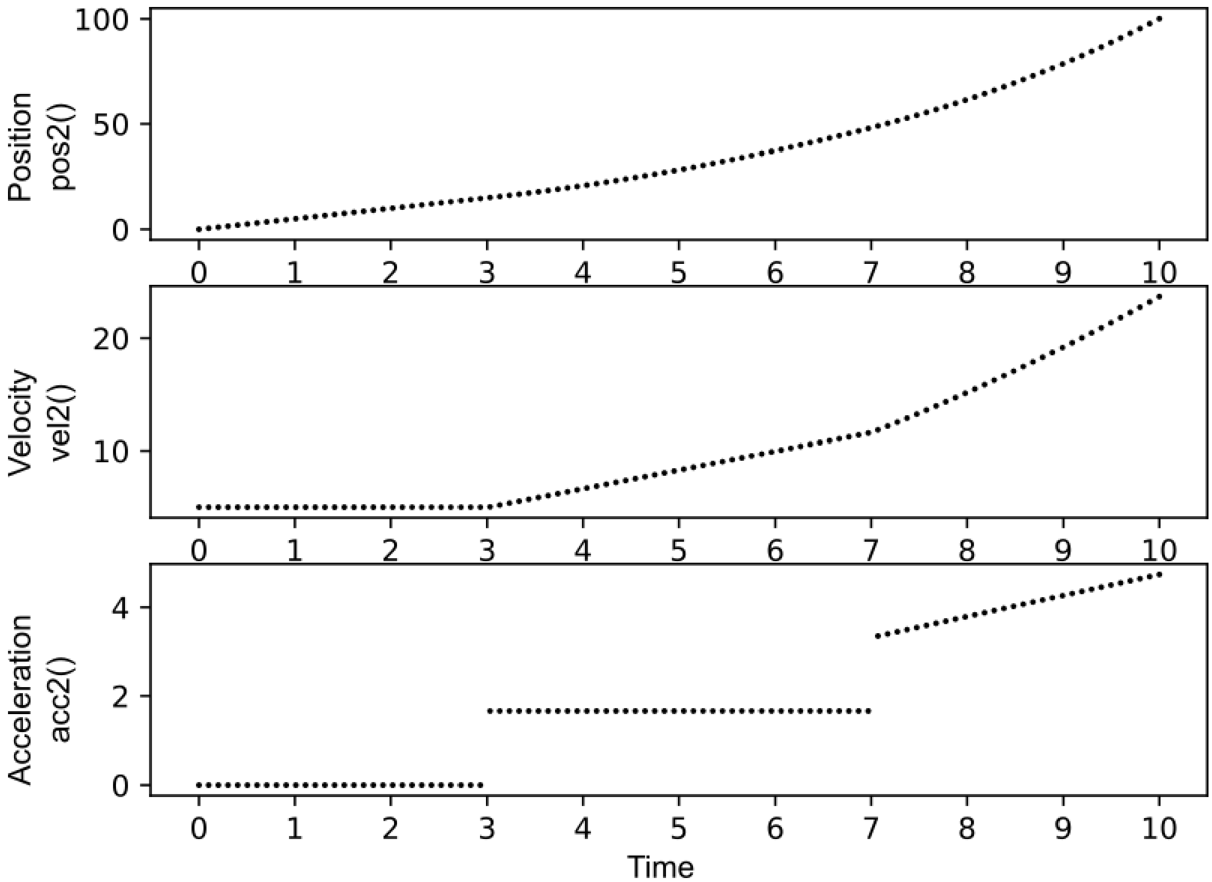


Figure 5.12 Plot of the functions `pos2()`, `vel2()` and `acc2()` for values between 0 and 10 with a resolution for time t of 0.1.

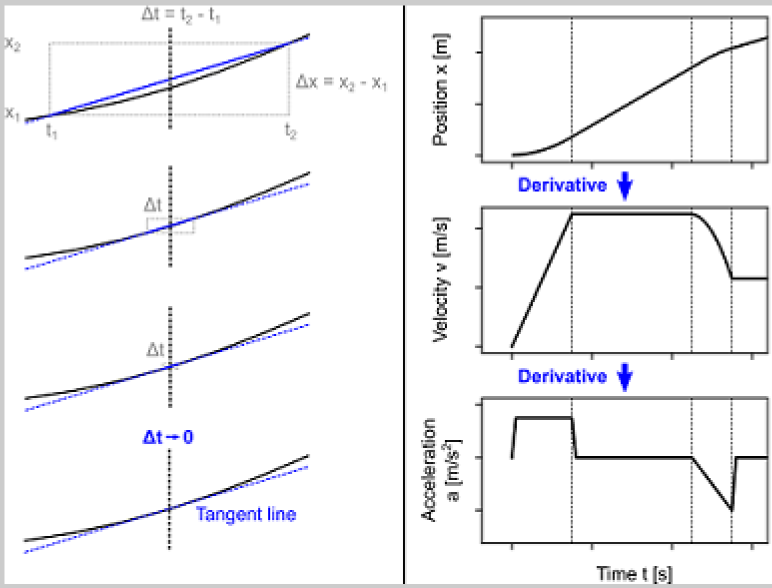
In principle, numerical differentiation is exactly what we have just seen: Dividing the difference between two function values from "near" function parameters by the distance between the parameters. However, there are a number of pitfalls. These are related to mathematical preconditions, on which we do not have the room to elaborate here, e.g. continuity and differentiability. Furthermore, other pitfalls result from the limited precision of computer floating point numbers. You will learn more about some of these details throughout the book. But there is much more to learn about these subjects than a single book can cover.[184]

We have now made the journey from position to velocity to acceleration to jerk by performing repeated differentiation. Now we turn around and move into the opposite direction from jerk to acceleration to velocity to position by means of integration.

SIDEBAR **Derivative**

Definition: The derivative of a function describes the "instantaneous rate of change" of the function. Differentiation is the process of calculating derivatives.

Visualization:



Application in robotics / Tasks solved:

- **Calculate robot velocity from robot position and acceleration from velocity.**
- **Find the required acceleration and velocity to have the robot reach a target at a specific time.**
- **Calculate Jacobians J that relate joint velocities to the velocity of the robot or its EE.**

Code:

```
def derivative(f, x, delta=0.01):
  return (f(x + delta/2) - f(x - delta/2)) / delta
```

Math:

$$f'(x) = \frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Related concepts: Integral, Partial derivative, Gradient

5.4 Integrals

Referring back to figures 5.9 and 5.12, going from top to bottom in both figures is a matter of differentiation. Going from bottom to top is the process of integration.

Using the sidewalk delivery robot as an example again, we want to calculate the position of the robot, given its velocity over time. Intuitively, we know that the faster the robot is moving, the faster its position changes as well. Furthermore, it is easy to see that we need to “add up” the distance traveled in each time interval to get the overall position. More precisely, to get the robot’s position x at time t, i.e. x(t), we need to sum up all the distances from time zero to time t. The distance traveled in each time interval depends on the velocity v(t) at each point in time.

We could hence describe *integration* as the inverse operation of differentiation.[185] Stating the same differently, an *integral* can be seen as an *antiderivative* and the other way around. Given a function and its derivative, the integral of the derivative is the original function. Going back to the sidewalk robot makes this abstract description more concrete and visual - and relates it to robotics.

Let’s first discuss a time interval with a constant velocity. We again use the segment t = [15, 45] from figure 5.9, shown in figure 5.13 below.[186] Looking at the position x(t) in discrete time intervals of Δt = 1 s, we see that x(1) is increased by 2.25 m compared to x(0). The change between x(1) and x(2) has the same magnitude. This is true of all time intervals of size Δt = 1 s in this segment, i.e. the change between x(t) and x(t + 1) is always 2.25 m, independently of the value t.

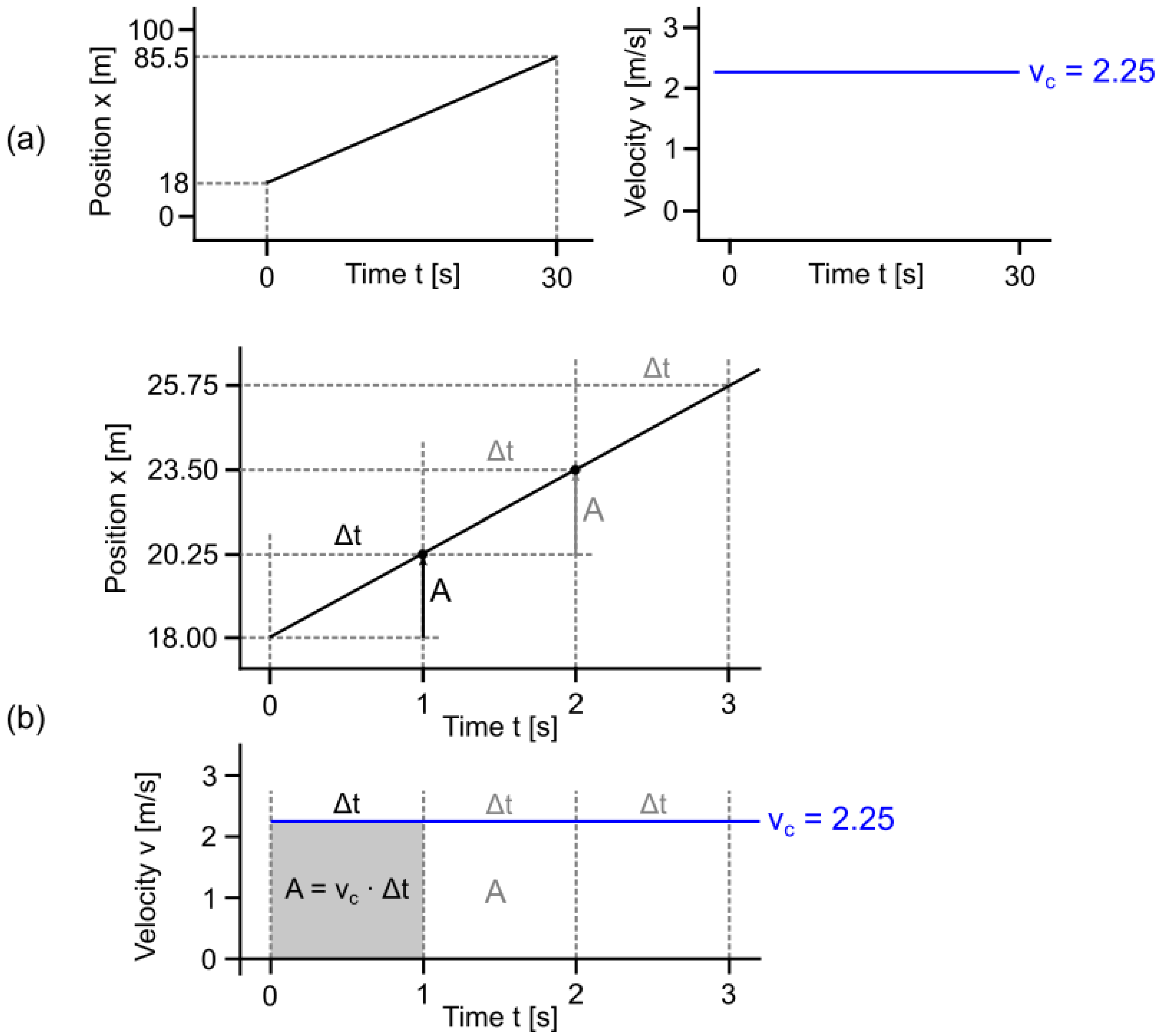


Figure 5.13 Calculation of position x(t) for a segment with constant velocity v(t) = v_c = 2.25 m/s.[187]

The constant velocity in this segment is v(t) = v_c = 2.25 m/s. We defined velocity as the rate of change in an object's pose. Since this rate of change is constant here, we can simply multiply the rate of change per second by the time interval in seconds: $v_c \cdot \Delta t = 2.25$ m/s $\cdot$ 1 s = 2.25 m. Thus, for constant velocity v(t) = v_c, we can calculate the position x(t) as $x(t) = x_0 + v_c \cdot t$. We need x_0, the starting position at time t = 0, to express that the robot is located at x_0 at time t = 0, i.e. x(0) = x_0. In our example, the values are x_0 = 18 and v_c = 2.25 m/s, resulting in x(t) = 18 m + 2.25 m/s · t.

The formula $v_c \cdot \Delta t$ can be interpreted as calculating the area of a rectangle with height v_c and width Δt. In figure 5.13(b) this area is marked with the symbol A. As you can see, A is the *area under the curve* v(t) for an Δt interval. Furthermore, this area corresponds exactly to the change of position x(t) over the course of the Δt interval. This is not a coincidence. It is rather the intuitive explanation of integrals. If the rate of change, here velocity, is smaller, the area A under the curve is also smaller. If the time interval is shorter, A is also smaller. The same goes when considering larger rates of changes or longer time intervals. Furthermore, a multiplicative relationship makes sense here as doubling the velocity or doubling the time also doubles the distance traveled. Let's apply this intuitive understanding to a more complex case and then formalize our intuition.

We pick the interval t = [0, 15] from the sidewalk robot's journey (figure 5.9), where the velocity v(t) is not constant, but instead grows linearly. Figure 5.14 shows the first 4 seconds of this interval.

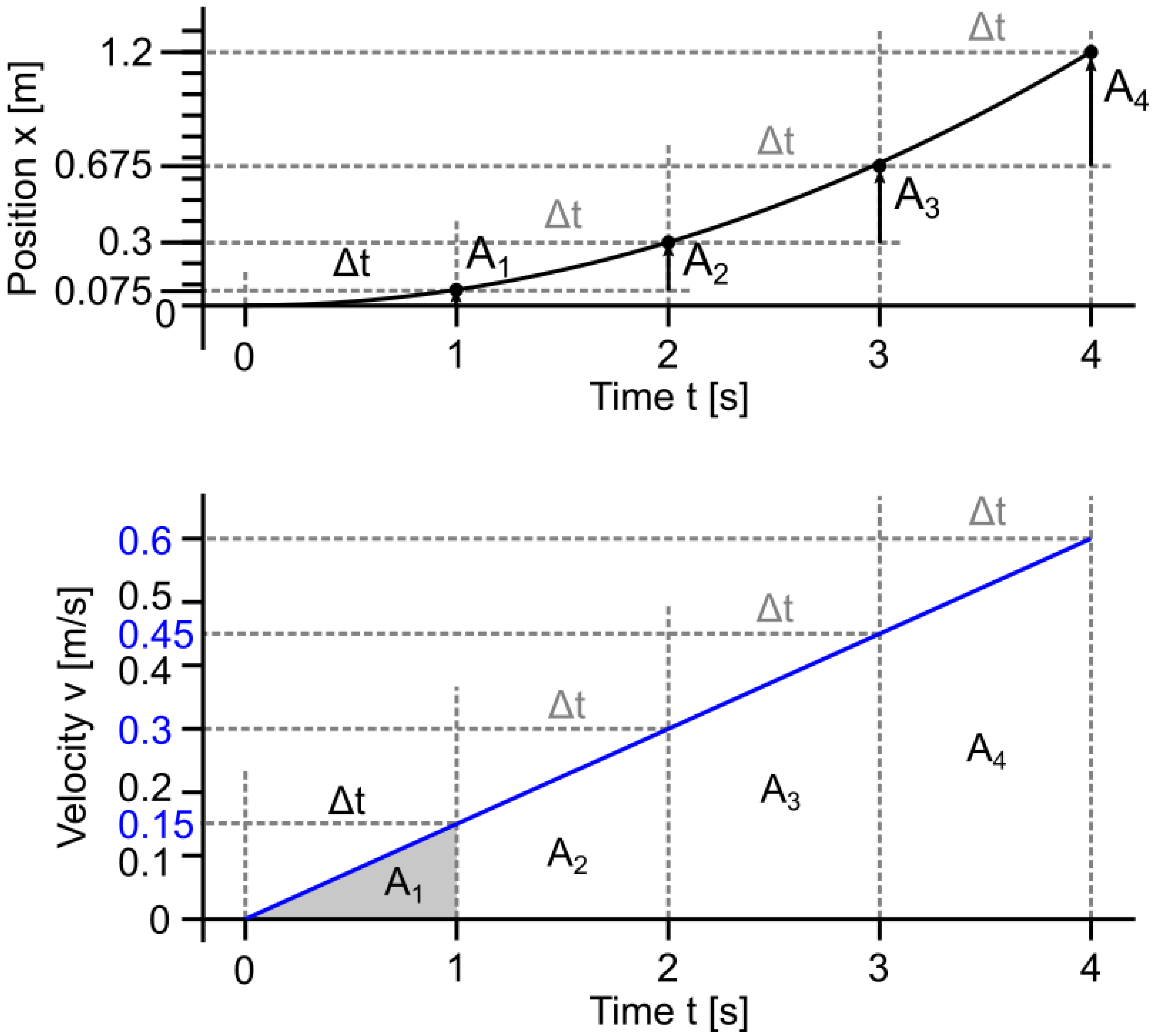


Figure 5.14 Calculation of position x(t) for a segment with variable velocity v(t).

The area under the curve A_i for the intervals is shown as before. However, this time the area is not a rectangle. Using A_1 as example, performing the same operation as before, i.e. multiplying the function value $v(t = 1 \cdot \Delta t = 1) = 0.15$ with the interval size $\Delta t = 1$ would give us a wrong result of 0.15 instead of the correct 0.075. This is true for all intervals A_i. Fortunately, we can follow a very similar process to the one we utilized for differentiation in order to decrease this error: decrease the interval size Δt.

Figure 5.15 illustrates that the error for using rectangles $A_i = v(i \cdot \Delta t) \cdot \Delta t$ decreases along with a decreasing interval size Δt. The figure shows an exemplary curve segment from an arbitrary function v(t) to make the point that this works for any v(t). It does not depend on a particular shape of function v(t).[188]

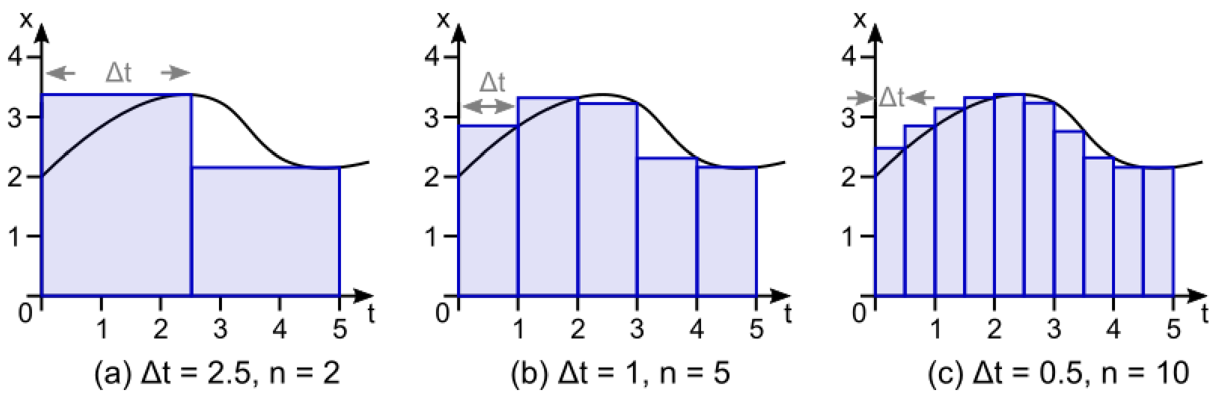


Figure 5.15 Illustration of numerical integration by means of the rectangle method. The approximation converges to the true integral value with decreasing rectangle width t (= increasing number of rectangles n).

In other words, we *approximate* the true area under the curve via a number of small rectangles. This is the same fundamental insight to (numerical) integration as the one for (numerical) differentiation.[189] To get the true integral value, we again need to transition from finite small intervals to infinitesimal small intervals, i.e. again have Δt approach zero:

$$\lim_{\Delta t \to 0}$$

The number of rectangles n increases with the decreasing width Δt of each rectangle, thus we can also state that n must approach infinity:

$$\lim_{n \to \infty}$$

Everything stated above about practical considerations when performing numerical differentiation also holds true for numerical integration. Using symbolic integration as an alternative is one of these considerations. The examples given for symbolic differentiation above can - with some caution - be read right to left, giving us examples for symbolic integration. This only leaves two more aspects of integration to discuss before we can wrap up the topic for now: mathematical symbolism and Python implementation.

Integrals are denoted by the *integral sign* ∫ together with information about the variable of integration, the so called *differential*, e.g. dt for variable t or dx for variable x. If we want to express that the position function p(t) is the integral of the velocity function v(t), we can write

$$p(t) = \int v(t)dt$$

We stated before that the derivative of the position function p(t) is the velocity function v(t), i.e. p'(t) = v(t) or in Leibniz's notation

$$\frac{d}{dt}\left(\int v(t)dt\right) = \int\left(\frac{d}{dt}v(t)\right)dt = v(t)$$

Combining these makes it clear that differentiation and integration are inverse operations of each other:

$$\int_a^b v(t)dt$$

This is also known as the *fundamental theorem of calculus*.

It is crucial to keep two types of integrals apart. So called *indefinite* integrals are used to *represent a function* whose derivative is another function, i.e. antiderivatives. The other type of integral are *definite* integrals that *provide a value* that corresponds to the area under the curve of a *specific interval* [a, b]. When looking for *the function* that represents the integral (e.g. p(t)) of another function (e.g. v(t)), one is dealing with indefinite integrals. When looking for *the value* that corresponds to the total change of a quantity (e.g. position) within a specific interval, which is represented by a rate of change function (e.g. velocity), one is dealing with definite integrals.

Definite integrals are also denoted by the integral sign ∫ and the differential (e.g. dt). In addition, definite integrals have the interval [a, b] written as a subscript and a superscript to the integral sign. For example the definite integral, a number, of the function v(t) integrated over variable t in interval [a, b] is written as

$$\int_a^b v(t)dt$$

There is a simple relation between indefinite integrals and definite integrals, which is highly relevant in practice. Given the function p(t) is the indefinite integral of the function v(t), we can calculate the definite integral for any interval [a, b] by a simple subtraction:

$$\int_a^b v(t)dt = p(b) - p(a)$$

Let's spell out in words what is written here as a formula:[190]

- (left-hand side:) The aggregate change in position, i.e. the distance traveled, due to moving with the position change rate (= velocity) v(t) at each moment in the interval between time a and time b
- is equivalent to
- (right-hand side:) the position p(t) at the end of the interval (time b) minus the position at the start (time a), i.e. to the distance traveled.

This is not a proof, but at least an intuitive confirmation.

The challenge with indefinite integrals is that they cannot easily be calculated numerically. In contrast, calculating definite integrals numerically can be done by following the procedure explained around figure 5.15 above.

Let's wrap up integrals with numerical integration in Python. As stated before when discussing differentiation, the right approach in production software is to utilize a numerical library for integration.[191] The following code snippets are not intended as a reference implementation, it is rather a "naive" verbatim implementation of what I explained above to help you understand the principles of performing integration in software.

We use the velocity function $v(t) = 0.15 \cdot t$ from above as example. Using symbolic integration, it is possible to calculate that

$$p(t) = \int v(t) dt = \int 0.15 \cdot t\, dt = 0.075 \cdot t^2 + C$$

In case you wonder what the constant C is about, here is a short intuitive explanation: When we calculate the derivative of a function, e.g. velocity from position, the absolute value of the position function does not matter. Only the *relative* change of position is relevant. The derivative function thus contains no information about the *absolute* value of its originating function, i.e. its integral function. To compensate for this when calculating an indefinite integral, the so called *constant of integration* C is added. Using the example of velocity and position again, integrating velocity over time tells us about the distance traveled over time. However, without additional information, we don't know where this distance is traveled, i.e. its offset from zero. This position offset is represented by the constant C.

In Python, we calculate the numerical definite integral of a `vel()` on the interval [`a`, `b`] as

```
def vel(t):
  return 0.15 * t

# Calculate the numerical definite integral of function f
# in interval [a, b] using rectangles of width delta
def integral(f, a, b, delta=0.01):  # a <= b
  sum = 0
  x = a
  while x <= b:
    sum += f(x) * delta
    x += delta
  return sum

print('The integral of vel in interval [0, 4] is', integral(vel, 0, 4))
# result (rounded): 1.203
```

Using the symbolic integral of v(t) and the relationship between indefinite integrals and definite integrals above, we can verify this numerical result:

$$\int_0^4 v(t)dt = p(4) - p(0) = 0.075 \cdot 4^2 + C - (0.075 \cdot 0^2 + C) = 0.075 \cdot 16 + C - 0 - C = 0.075 \cdot 16 = 1.2$$

Notice the numerical error of 0.003 for a `delta` value of 0.01.

We can express an equivalent implementation of `integral()` as

```
def integral(f, a, b, delta=0.01):  # a <= b
  n = int((b - a) / delta)
  sum = 0
  for i in range(0, n + 1):
    sum += f(a + i * delta) * delta
  return sum
```

This `integral()` code can be translated one to one in the mathematical sum notation Σ as

$$\sum_{i=0}^{n} f(a + i \cdot \Delta x) \cdot \Delta x$$

The number `n` corresponds to the number of rectangles that we sum up. You will frequently encounter the Σ *sum notation* (Sigma) in robotics literature. It is therefore a useful skill to be able to mentally translate it into a loop construct - at least until Σ feels as familiar as a `for` or `while` statement.

There are a number of improvements of the basic numerical integration method presented. One of the simpler ones uses trapezoids instead of rectangles to better approximate the shape of the function with the same number of segments. However, going into the details of numeric algorithms for calculus is out of scope for this book. I recommend you to use readily available numeric libraries instead of implementing your own integration functions. For example, SciPy offers `scipy.integrate.quad()`:

```
from scipy import integrate

val, _ = integrate.quad(vel, 0, 4)
print(val)
# returns: 1.2
```

Often the function we want to differentiate or integrate is not given as a function, but as an array [192] of values. The values in the array represent the function by providing the function value for discrete parameter values. For example, our sidewalk robot's velocity function v(t) for the interval t = [0, 15] with a temporal resolution of Δt = 1 could be given as an array `vel_arr` of length 16 with each entry at index `i` providing the function value for v(t = `i` · Δt):[193]

```
import numpy as np
vel_arr = np.array([0.00, 0.15, 0.30, 0.45, 0.60, 0.75, 0.90, 1.05, 1.20,
                    1.35, 1.50, 1.65, 1.80, 1.95, 2.10, 2.25])
```

The reason you will often encounter such discretized function representations, i.e. arrays of function values, in robotics is due to many functions only being available implicitly. The software running on our example mobile robot might get an updated value from the wheel position sensors every 0.1 s. Thereby, the position function is only given as an array of function values for t = 0, 0.1, 0.2, 0.3, …. Furthermore, we can calculate its derivative only with Δt greater or equal to 0.1.[194]

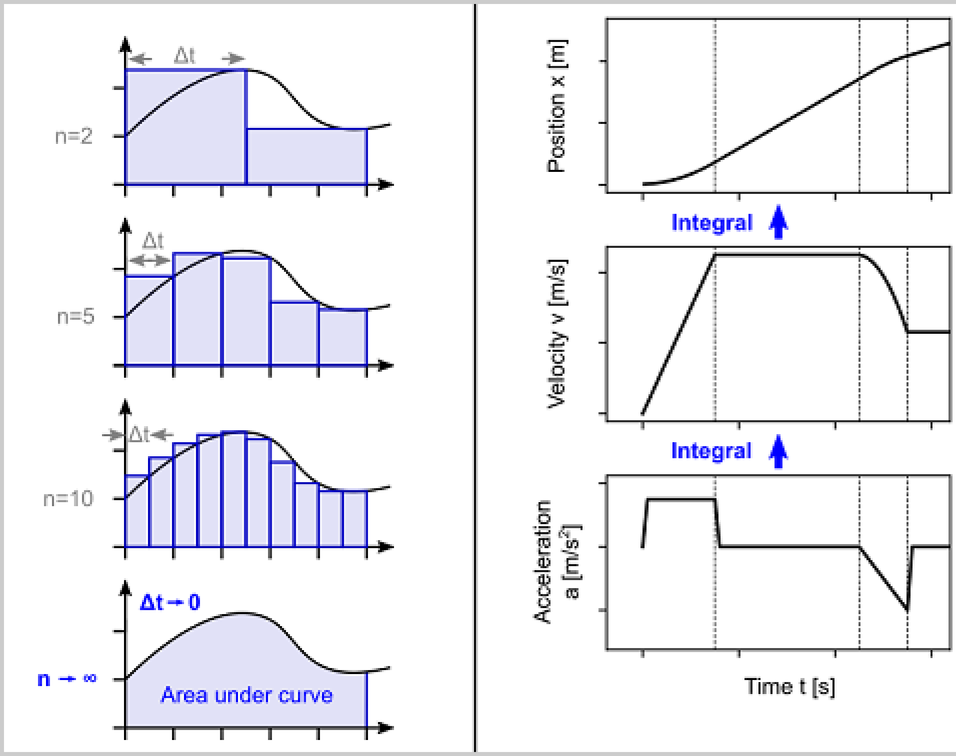
Apart from the `delta` parameter in the `derivative()` and `integral()` functions being determined by the Δt associated with the given array, we can still calculate the numerical derivative and integral using these functions - just as we did above. Nevertheless, we should follow the advice of preferring library functions for such calculations. Using NumPy (`numpy.gradient()`, `numpy.sum()`, `numpy.trapz()` and `numpy.cumsum()`), we can calculate the derivative and integral of `vel_arr` for t = [0, 15]:

```
# Derivative: acceleration (acc_arr) from velocity (vel_arr)
acc_arr = np.gradient(vel_arr)
print('acc_arr: ', acc_arr)

# Integral: position (pos_arr) from velocity (vel_arr)
pos_arr1 = np.cumsum(vel_arr)  # rectangles, cumulative
print('pos_arr1:', pos_arr1)

pos2 = np.sum(vel_arr)  # rectangles
print('pos2:', pos2)

pos3 = np.trapz(vel_arr)  # trapezoids
print('pos3:', pos3)

# Result (rounded and formatted for readability):
# acc_arr:  [0.15 0.15  0.15  0.15 0.15 0.15 0.15 0.15 0.15 0.15 0.15 0.15
#            0.15 0.15  0.15  0.15]
# pos_arr1: [0.00 0.15  0.45  0.9  1.5  2.25 3.15 4.2  5.4  6.75 8.25 9.9
#            11.7 13.65 15.75 18.0]
# pos2: 18.0
# pos3: 16.875
```

SIDEBAR **Integral**

Definition: An integral of a function describes the “area under the curve” of a function. It describes the accumulated overall change of a quantity. Integration can be seen as the inverse operation of differentiation, i.e. an antiderivative.

Visualization:



Application in robotics / Tasks solved:

- Calculate robot position from robot velocity and velocity from acceleration. For mobile robots, this comprises odometry.
- Derive the area, volume, mass or other quantity for complex geometric shapes.
- Combine rate-of-change data to get the overall resulting change, e.g. aggregate current energy consumption into consumed battery life.

Code:

```
def integral(f, a, b, delta=0.01):
  sum = 0
  x = a
  while x <= b:
    sum += f(x) * delta
    x += delta
  return sum
```

Math:

$$\int_a^b f(x)dx = \lim_{n \to \infty} \sum_{i=0}^{n} f(a + i \cdot \Delta x) \cdot \Delta x \quad \text{where } \Delta x = \frac{b-a}{n}$$

$$\int_a^b f(x)dx = F(b) - F(a) \quad \text{where } f(x) = F'(x)$$

This concludes our discussion of basic calculus for the moment. You now posses the required mathematical tools to work with kinematics and dynamics.[195] Figure 5.16 summarizes the relationship between position, velocity, acceleration and jerk in terms of derivatives and integrals.

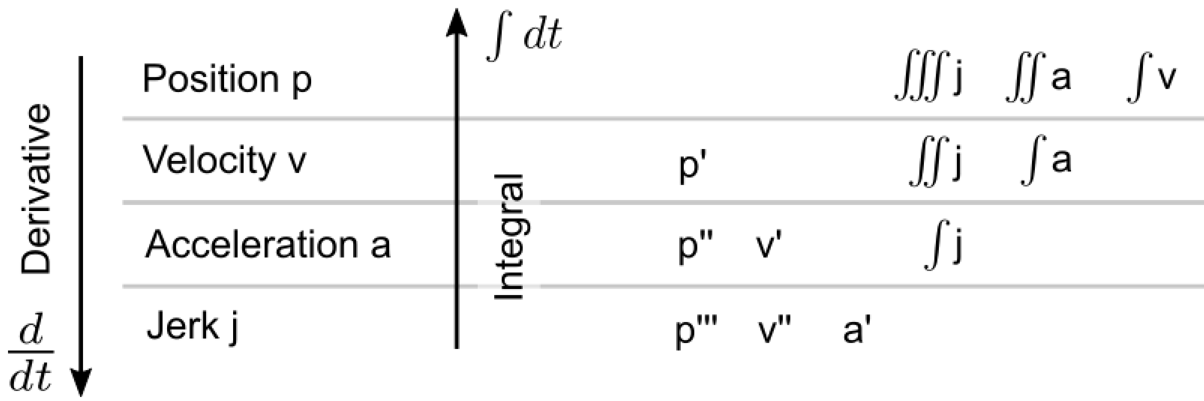


Figure 5.16 Relationship between position p, velocity v (= p'), acceleration a (= p'' = v') and jerk j (= p''' = v'' = a').

5.5 Joint Space and Task Space

The two terms *joint* and *link* are of essence when talking about kinematics. A brief recap:[196]

- Joint: The moveable parts in the robot hardware. The most common joint types are those that can rotate, like a hinge, and those that can extend linearly, like a drawer slide or telescopic pole. Joints contain the actuators that make the robot move.
- Link: A rigid mechanical part connecting the joints. The links give the manipulator its shape and make up the bulk of what is visible when looking at a robot.

Our goal is to formally describe the relation between a robot's joint positions and a robot's 3D pose. This is not only important for positions (poses), but also their derivatives, i.e. velocities (twists) and accelerations (linear and angular). Having such a description enables us to work with the most appropriate representation in each part of our robot software stack. For example, we usually want to program manipulation tasks in 3D, but we have to command the individual actuators using joint values. When working with joint values, e.g. joint positions, joint velocities or joint accelerations, we refer to this as working in *joint space*. When dealing with 2D space or 3D space, we either refer to this as *2D/3D space* or *Cartesian space* or as working in *task space*.

We use forward kinematics and inverse kinematics to convert between joint space and task space. *Forward kinematics (FK)* corresponds to the direction of conversion from joint space to task space. *Inverse kinematics (IK)* corresponds to the opposite direction of conversion, i.e. from task space to joint space. This is illustrated in figure 5.17.

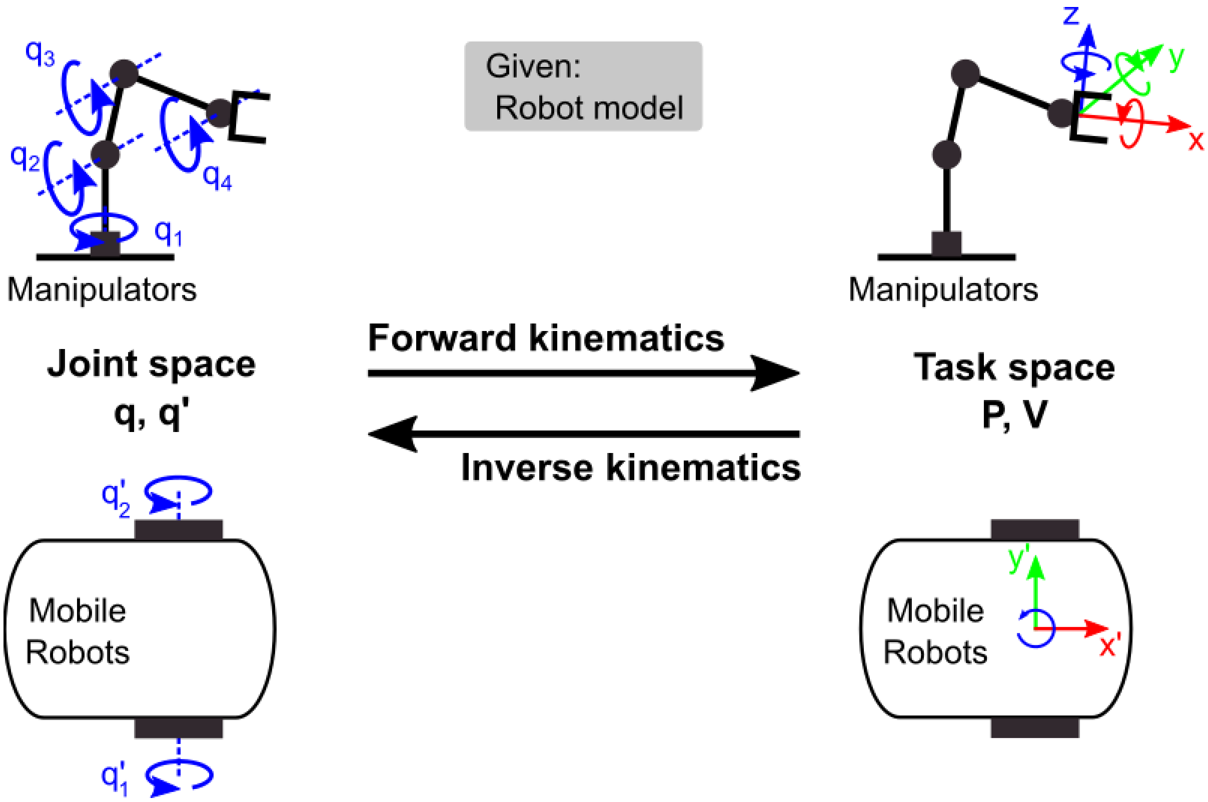


Figure 5.17 The relationship between joint space, task space, forward kinematics and inverse kinematics.

Although the majority of examples and illustrations in this section are about the kinematics of manipulators, the same principles apply for the kinematics of mobile robots. Since this chapter is an introduction to the topic, we will only discuss serial manipulator kinematics, differential drive kinematics for driving mobile robots and quadrotor kinematics for flying mobile robots. There are many details to be learned about each specific type of kinematic beyond the scope of this chapter.

One aspect of inverse kinematics is worth highlighting up front: solvability. *Inverse kinematics*, i.e. the set of joint values corresponding to a given 3D pose, can have

- no solution,
- one solution or
- multiple (incl. infinite) solutions.

The 2 degrees of freedom (DoF) 2D example in figure 5.18 shows these three possibilities.

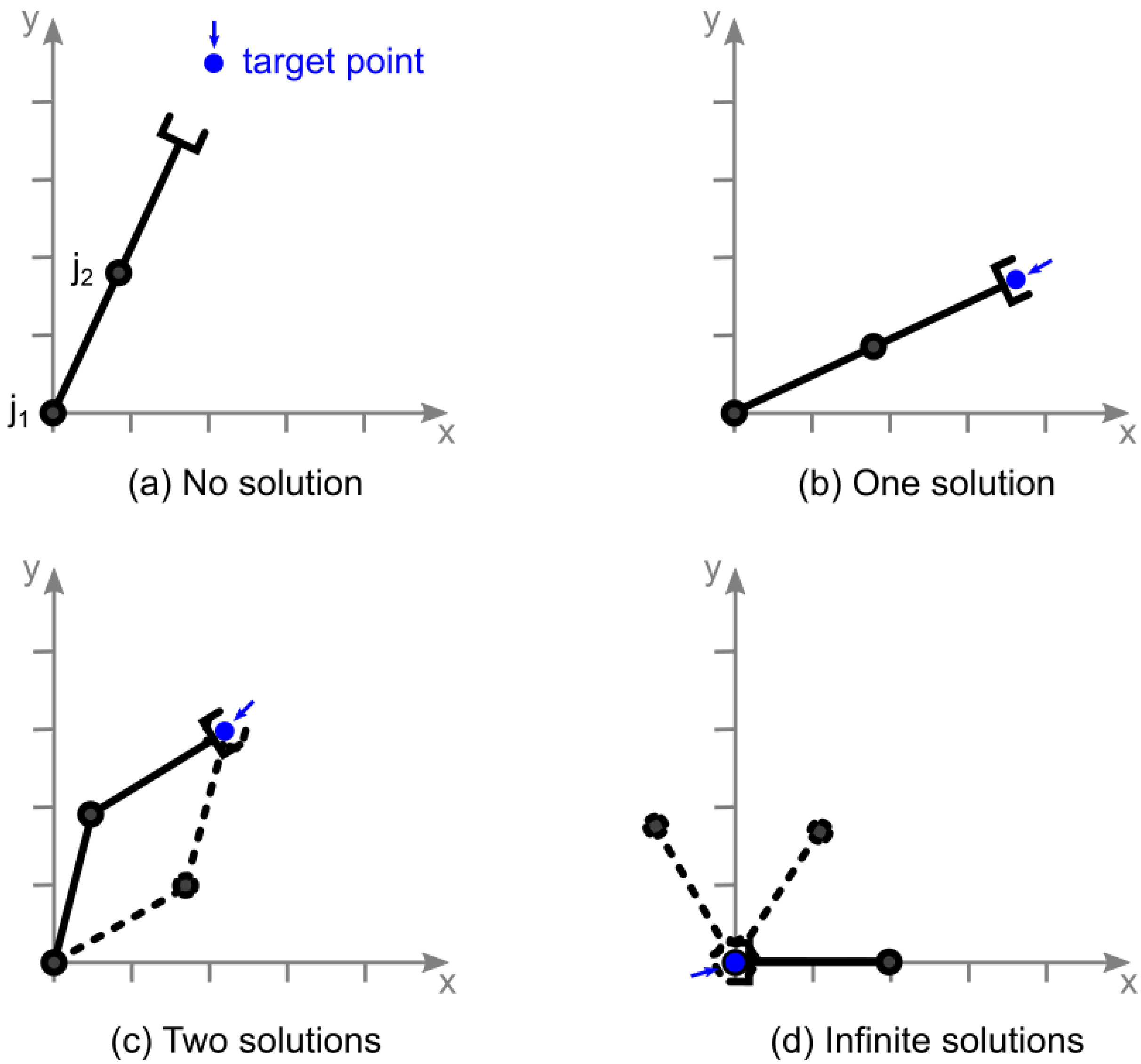


Figure 5.18 Different number of inverse kinematic solutions for a simple 2 DoF manipulator in 2D.

There is no set of joint values that allows the example robot to reach the task space point in 5.18 (a). Hence, there is no inverse kinematics solution. In 5.18(b) there is exactly one set of joint values (j_1 = 25°; j_2 = 0°) for the given point in task space. For the 2D point in 5.18(c) there are two solutions: (j_1 = 75°; j_2 = -45°) and (j_1 = 30°; j_2 = 45°). Finally, there are infinitely many solutions for 5.18(d). The value of j_1 does not change the resulting task space position of the example manipulator when j_2 = 180°.

Certain types of robots, especially the widespread 6 DoF articulated (industrial) manipulator, have a special multiplicity of inverse kinematics solutions. These are known as *robot configurations*, *arm configurations* or simply *configurations*. The common 6 DoF manipulator made up of six rotary joints has eight inverse kinematic solutions in general.[197] They can be characterized by intervals for the values of the robot's joints. Figure 5.19 depicts eight different configurations for the same 3D pose.[198]

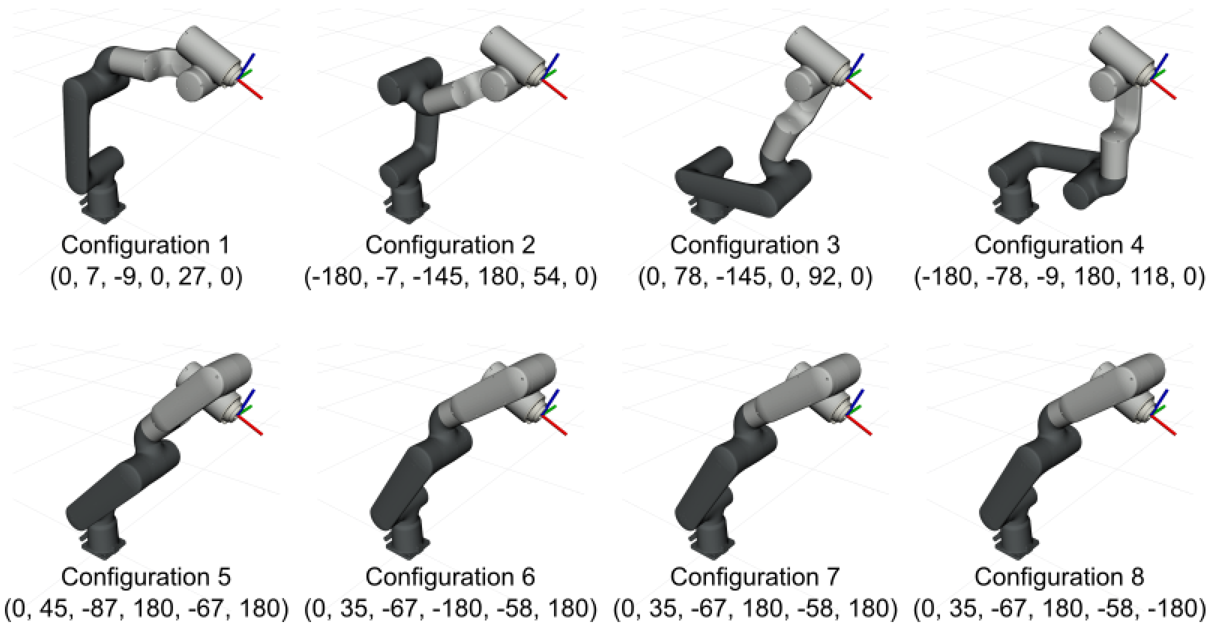


Figure 5.19 Eight different arm configurations, each with different joint values, resulting in the same pose of the manipulator flange.

For manipulators in an industrial environment, we usually want them to always remain in the same configuration. Although all shown joint values result in the same task space pose, they are far apart in joint space. Thus, changing between configurations potentially results in a large motion of the manipulator.[199] Not only is it wasteful in terms of time required to perform such large movements, it might not even be possible due to obstacles in the robot's vicinity. Furthermore, certain motions in task space, e.g. following a straight line, are impossible if a configuration change is required between starting pose and target pose. Figure 5.20 visualizes the robot's motion when changing from configuration 1 to configuration 4 in figure 5.19. It also shows the large swept volume of this re-configuration motion.

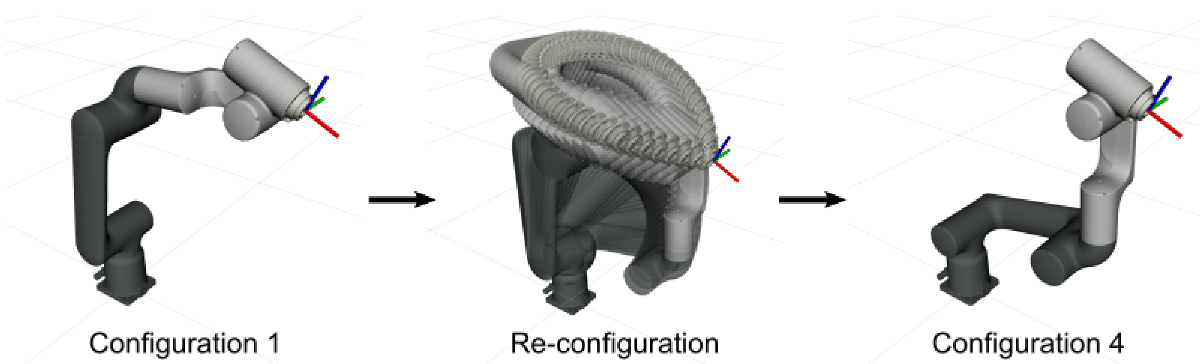


Figure 5.20 Illustration of the large motion and the resulting swept volume when changing between two configurations from figure 5.19.

In case of multiple inverse kinematic solutions, we need to decide which set of joint values we command to the actuators. We can often express this as an optimization problem. The goal is to minimize or maximize some quantities. For example, we might want to minimize time, minimize path length and/or maximize the distance to the closest obstacle. In addition to minimization and maximization, there can be hard constraints, such as remaining in the same configuration or limiting joint velocities. We'll further investigate resolving inverse kinematic ambiguity later on.

Another important concept - and a challenge - in kinematics are singularities. A *singularity* occurs when several robot joints line up in a way that motion in one joint can be canceled out by another joint.[200] Two typical singularities of 6 DoF manipulators are shown in figure 5.21.

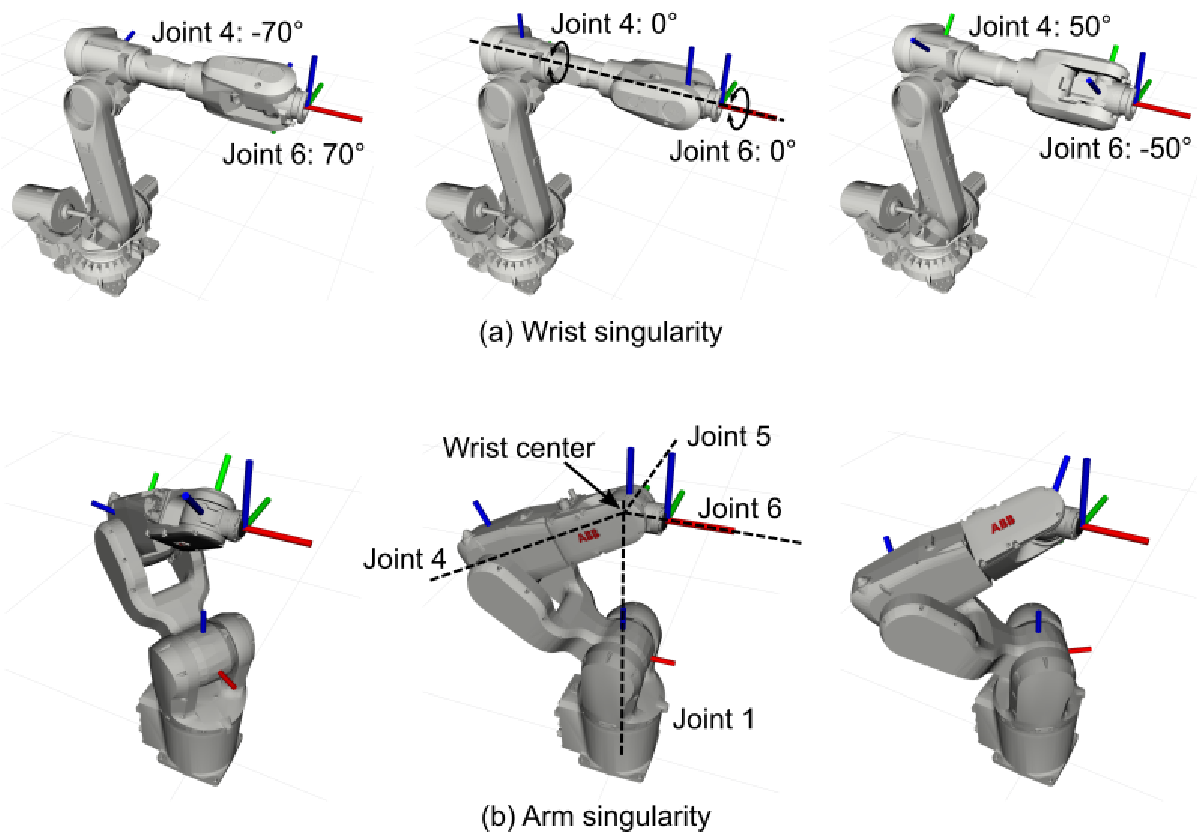
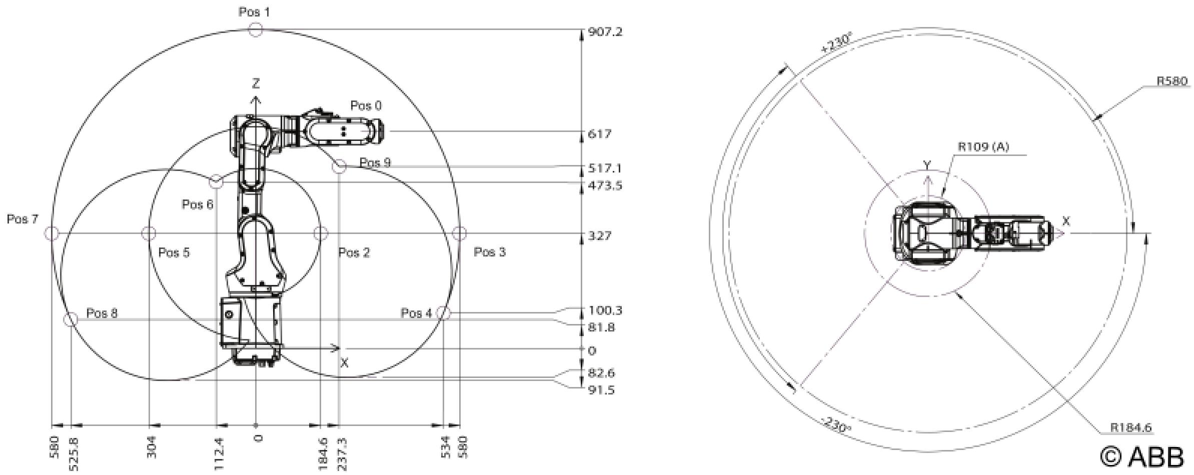


Figure 5.21 Two kinds of singularities common for 6 DoF robots: (a) A wrist singularity occurs when joint 4 and joint 6 are aligned. (b) An arm singularity occurs when the wrist center (intersection of joints 4, 5 and 6) intersects with the rotation axis of joint 1.

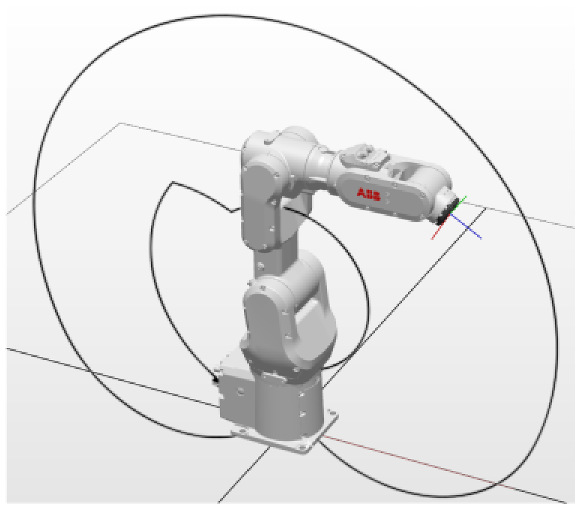
The result is that the two joints can change their values without the robot pose changing. Thus, there are an infinite number of IK solutions for singularity poses. Handling this infinite number of IK solution is the first challenge when a singularity occurs. The second effect is that the robot loses a degree of freedom in the singularity. An illustrative explanation of this builds on what we just stated. Two joints can cancel out the effect of each other's motion on the overall robot pose. Therefore, they basically become a 1 DoF joint with two actuators. If two joints together just have 1 DoF in a singularity, the robot loses 1 DoF. The mathematical explanation derives from looking at the effect of a singularity on the *Jacobi matrix* **J**, which relates joint velocities to Cartesian velocities - and which we will discuss in later chapters. In a singularity the Jacobi matrix loses rank and becomes singular, hence the term singularity, and cannot be inverted.[201] Independently of the explanation, the loss of a degree of freedom means that the robot can no longer move in one or more directions. The only mitigation is to avoid moving through or close to singularities. As changing the robot configuration requires passing through a singularity, let me phrase this more precisely. We need to avoid moving through or close to singularities, when the end effector pose is supposed to follow a precise path in task space. When commanding the robot motion in joint space, singularities are of no concern. They are only relevant in task space.

In summary, *forward kinematics* (FK) provides the task space pose given the joint values. For each set of joint values, we get one definite pose from FK. Inverse kinematics (IK) provides the joint values, given a task space pose. There can be either no IK solution (no set of joint values results in the pose), one IK solution or multiple IK solutions for a given pose. This depends on the robot's kinematics, i.e. its joints and links, and the requested pose.[202]

The overall volume in task space that a given robot can reach is referred to as *workspace*, *work volume* or *work envelope*. The workspace contains all reachable points, which is to say that there is at least one inverse kinematics solution for each point in the workspace.[203] You will find visualizations similar to figure 5.22(a) for the robot workspace in most manipulator datasheets. [204] Furthermore, one can also visualize the workspace volume in 3D as shown in figure 5.22(b). Knowing a robot's workspace is very helpful in setting up a robot application. Looking at the workspace makes it easy to determine unreachable poses - and hence unreachable objects, because everything outside the workspace is unreachable by definition. However, keep in mind that it is not sufficient for an object to be inside the workspace to guarantee it can actually be reached in the way required for the application.



(a) Standardized workspace characterization



(b) Workspace visualization in virtual 3D environment.

Figure 5.22 (a) Standardized workspace visualization for a 6 DoF industrial manipulator. (b) Workspace visualizations in 3D.

The relation between joint space degrees of freedom (DoF) and task space pose parameters (PP) is an important consideration in robotics.[205] We can distinguish three cases

- DoF < PP: The robot is *kinematically deficient* or *non-holonomic*.
- DoF = PP: The robot is *holonomic*.
- DoF > PP: The robot is *kinematically redundant* or simply *redundant*.

In case of kinematically deficient manipulators, each position in the workspace can only be reached in certain orientations.[206] For mobile robots, being non-holonomic, also means that they cannot reach all positions in all orientations within their workspace or at least they cannot reach them on a direct path. Holonomic robots can in principle[207] reach all poses within their workspace. Again, for manipulators this means being able to reach them at all and for mobile robots reaching them on a direct path. In case of redundant robots, all poses can in principle be reached with multiple sets of joint values - in general with infinitely many. The difference between deficient, holonomic and redundant is best explained with two concrete examples.

The first example are 4 DoF, 6 DoF and 7 DoF manipulators operating in 3D space. A 3D pose has 3 position parameters and 3 orientation parameters, hence 6 pose parameters (PP). Given this task space, the 4 DoF manipulator is kinematically deficient, the 6 DoF one is holonomic and the 7 DoF manipulator is kinematically redundant. Figure 5.23 shows three real-world examples of such manipulators.

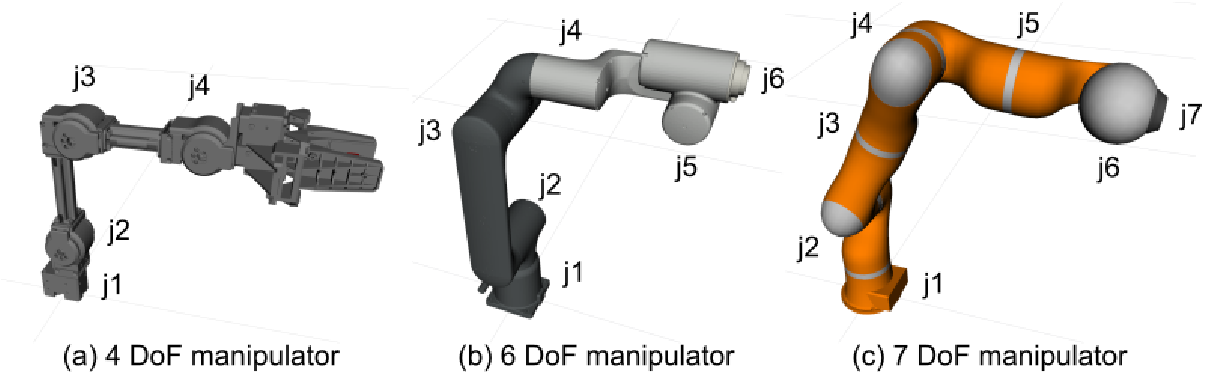
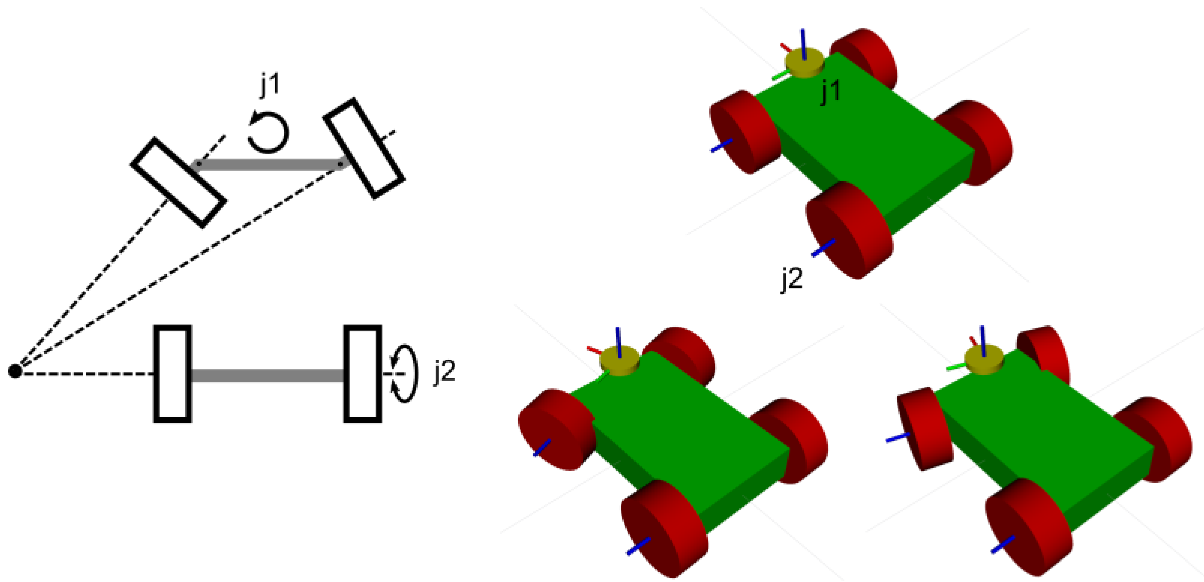
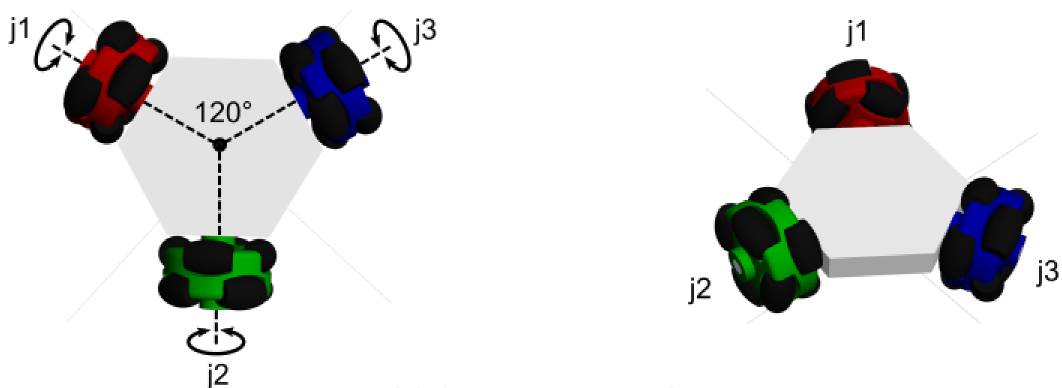


Figure 5.23 (a) Kinematically deficient 4 DoF manipulator. (b) Holonomic 6 DoF manipulator. (c) Kinematically redundant 7 DoF manipulator.

The second example are an autonomous car and a mobile robot with an omni wheel drive. Both are described by a 2D pose with 2 position parameters and 1 orientation parameter, i.e. 3 PP. A car's joint configuration is known as *Ackermann steering* or *Ackermann kinematics*. Ackermann kinematics have 2 DoF: the rotation of the wheels generating forward/backward motion and the steering angle of the front wheels. Parallel parking is difficult because a car is non-holonomic (2 DoF < 3 PP). If it were holonomic, one could simply drive "sideways" into the parking space. Given the parking space is wide enough for maneuvering, it is still possible to get the car into the parking space, but it requires an indirect path.[208] In case of the mobile robot, each of its three omni wheels is actuated independently, resulting in overall 3 DoF. The *omni wheel drive*, also known as *Kiwi drive*, makes it possible to move and rotate in any direction. It is an holonomic drive (3 Dof = 3 PP). See figure 5.24 for an illustration of Ackermann and omni wheel kinematics.



(a) Ackermann steering drive

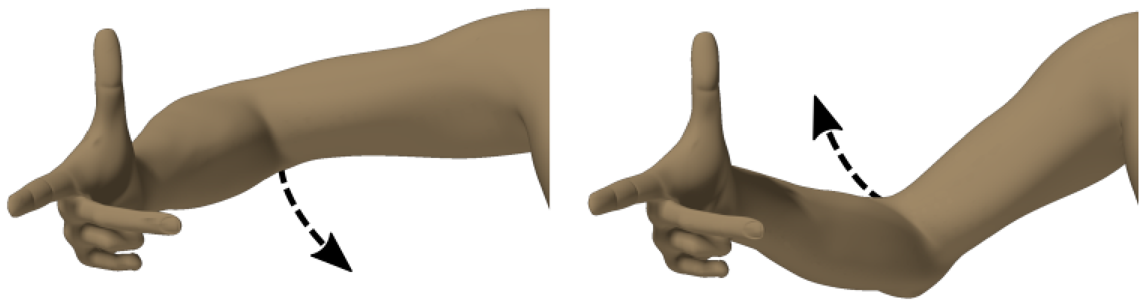


(b) Omni wheel drive / Kiwi drive

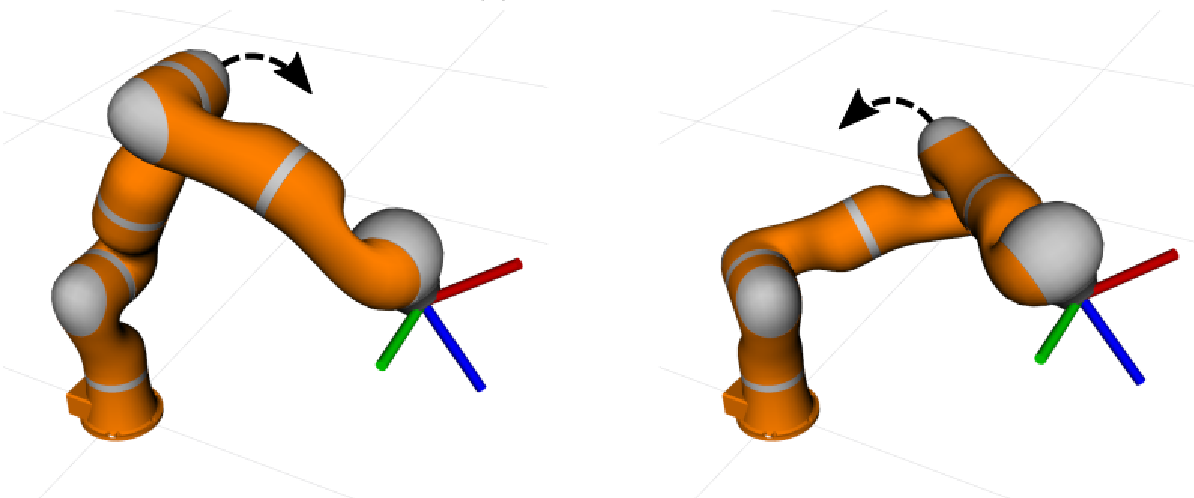
Figure 5.24 Two types of mobile robot kinematics: (a) Non-holonomic Ackermann steering drive and (b) Holonomic omni wheel / Kiwi drive.[209]

There is another concept often mixed up with deficient, holonomic and redundant kinematics. This other concept is about underactuated, fully-actuated and overactuated robots. The question behind this other concept is whether each degree of freedom in the robot's joints has an independent actuator. *Underactuated* robots have less actuators than DoF. This either means some actuators influence multiple DoF at the same time or that some DoF are not actuated and their position cannot be directly controlled. *Fully-actuated* robots have exactly one actuator per DoF. *Overactuated* robots have more than one actuator per DoF. You will get to know examples for each of these throughout the book. If nothing to the contrary is explicitly stated, we assume a fully-actuated robot, i.e. the degrees of freedom are equal to the number of independent actuators.

Having multiple IK solutions is not only an additional problem to be solved, but also an opportunity to improve robot behavior. Redundant robots, DoF > PP, are frequently used not in order to improve direct reachability, but to reach the same pose with different joint values. Being able to go around obstacles in the robot's workspace and in general increased motion flexibility are the motivating factors. Yet, we also need to keep in mind that each additional joint adds complexity, weight, failure points and also cost to the robot. In case of manipulators, the inspiration for redundancy stems from the kinematics of the human arm, which can be considered to have 7 DoF. When you put your hand in a determined pose, e.g. put it flat on a table, you can still change the position of your *elbow* without moving the rest of your body. This literal elbow room is a result of your arm having more than 6 DoF, i.e. it being kinematically redundant. Figure 5.25 illustrates this kinematic flexibility.



(a) Human elbow room



(b) Robot elbow room

Figure 5.25 Kinematic redundancy provides additional motion flexibility (a) for humans and (b) for robots.

Now that you know the terms joint space, task space, forward kinematics, inverse kinematics, singularities, kinematic deficiency, holonomy, kinematic redundancy and robot workspace, we can dive into kinematic computations.

5.6 Forward Kinematics

The input to forward kinematics (FK) are joint values (joint space) and the output is a pose (task space). Geometry provides us with the connection between the two spaces.

Let's start with a very simple 1 DoF robot in a 2D space. We only care about the 2D position, not the orientation. The pose hence has 2 PP: x, y. Our 1 DoF robot consists of one rotary joint j_1 with a motion range of [0°, 180°]. The robot has two links. The base link l_0 is fixed to the environment at position (x = 0; y = 0). The arm link l_1 has a length of 2 [m]. We consider the distal end of l_1 as the end effector (EE). Figure 5.26 visualizes our example robot.

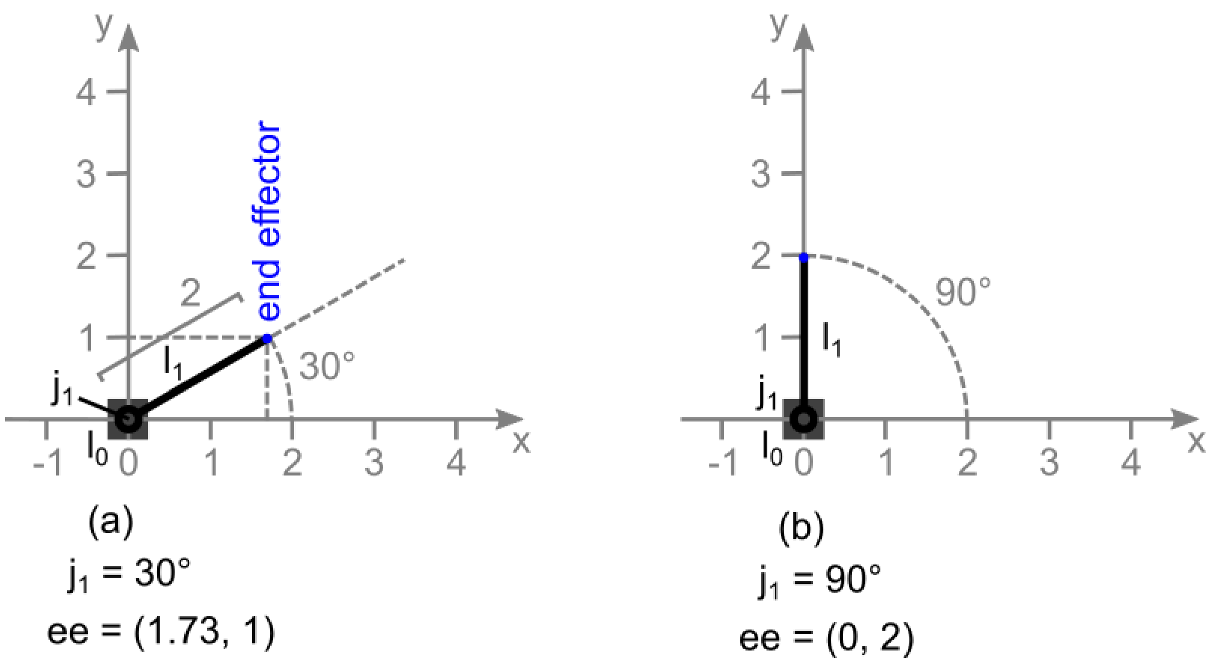


Figure 5.26 Two positions of the exemplary 1 DoF 2D robot.

We can now calculate the forward kinematics using the robot model (joints and links) together with the current joint value. The base link l_0 is always at its fixed position (0, 0). For this reason, the joint j_1 is also always at position (0, 0). This also determines the absolute position of one end of the arm link l_1, the end of l_1 attached to joint j_1. The only unknown position is the other end of link l_1, which coincides with the end effector. From figure 5.26 we can see that the determining factor for the position of l_1 (and the EE) is the angular value of joint j_1. Using *trigonometry*, we can mathematically describe the relation between the position of the EE and the value of j_1.

SIDEBAR **Trigonometry**

A brief refresher on trigonometric functions. In a right-angled triangle, the following equations help to calculate side lengths from angles and the other way around.

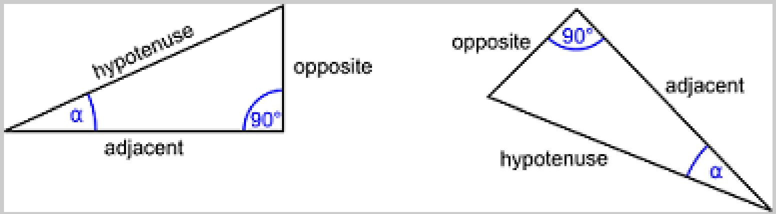


Figure 5.27 A right-angled triangle.

- Sine: sin() = opposite / hypotenuse
  - opposite = sin() · hypotenuse
  - Inverse sine (sin^{-1} or arcsin): = arcsin(opposite / hypotenuse)
- Cosine: cos() = adjacent / hypotenuse
  - adjacent = cos() · hypotenuse
  - Inverse cosine (cos^{-1} or arccos): = arccos(adjacent / hypotenuse)
- Tangent: tan() = opposite / adjacent = sin() / cos()
  - opposite = tan() · adjacent
  - adjacent = opposite / tan()
  - Inverse tangent (tan^{-1} or arctan): = arctan(opposite / adjacent)
  - Inverse tangent, two-variable arctan: = atan2(opposite, adjacent)
- Pythagorean theorem: $adjacent^2 + opposite^2 = hypotenuse^2$
  - $\text{hypotenuse} = \sqrt{\text{adjacent}^2 + \text{opposite}^2}$

Note that the hypotenuse is the side that is not touching the right angle. Furthermore, the adjacent is the side touching the angle we are interested in. The opposite is the only side not touching the angle .

Link l_1, joint j_1 and the x axis form a right-angled triangle. With the angle at j_1, the hypotenuse is l_1, the opposite is the y segment from j_1 to the EE and the adjacent is the x segment from j_1 to the EE. Figure 5.28 shows this mapping between robot model and trigonometry. Thus, $\sin(j_1) = y / l_1$, which is equivalent to $y = \sin(j_1) \cdot l_1$. Similarly, $\cos(j_1) = x / l_1$, hence $x = \cos(j_1) \cdot l_1$.

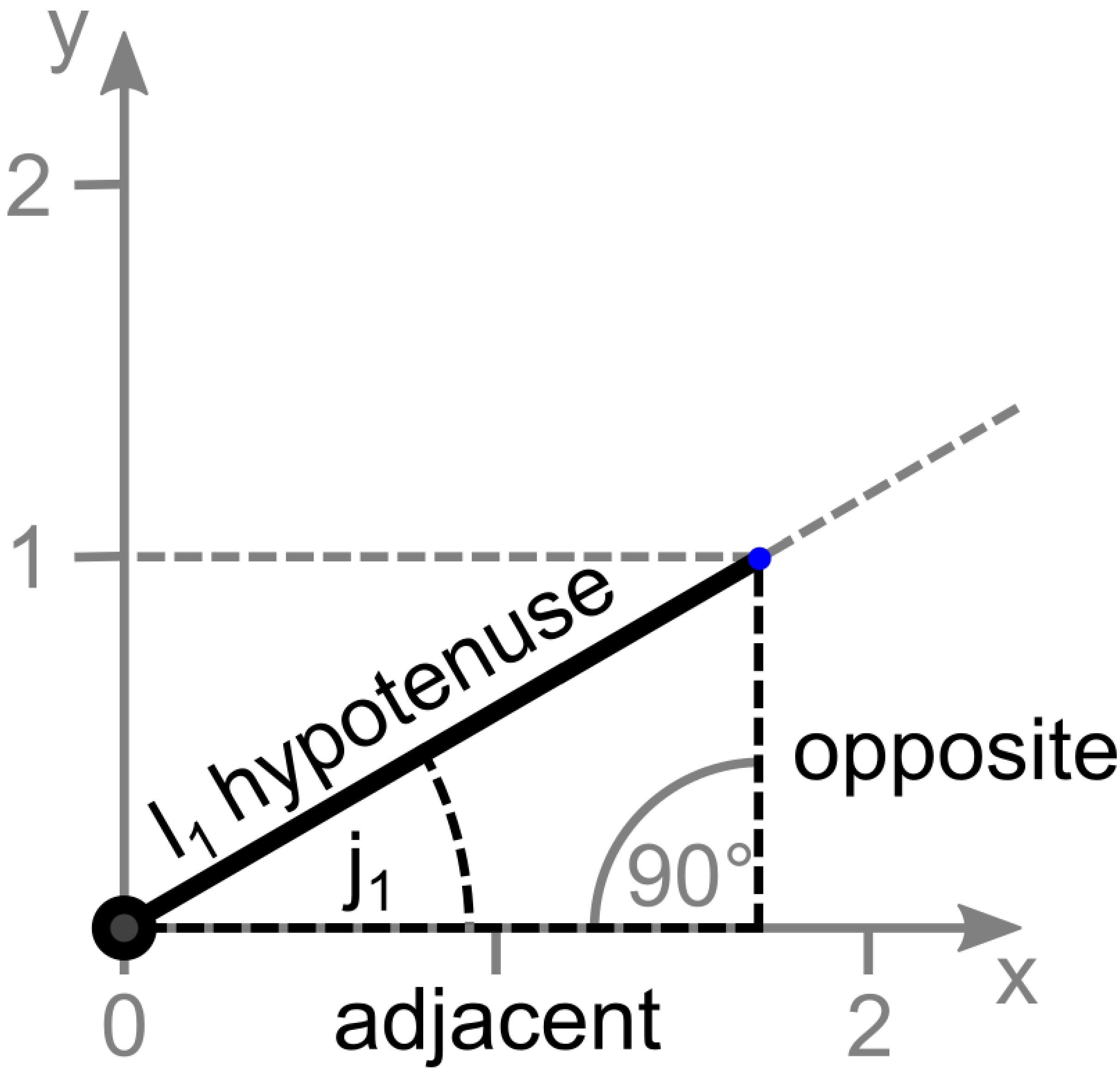


Figure 5.28 Applying trigonometry to the 1 DoF 2D robot from figure 5.26.

Inserting the joint value j_1 = 30° from figure 5.26(a) into these equations, we get the correct rounded result x = 1.73 and y = 1. Repeating this for j_1 = 90° (figure 5.26(b)), we also get the correct values for the EE position: (0, 2).

Using this knowledge, we can implement the forward kinematics `fk1()` for our 1 DoF example robot:

```
def fk1(j1_value, l1_length):
  x = cos(j1_value) * l1_length
  y = sin(j1_value) * l1_length
  return np.array([x, y])

l1_len = 2
print('30°:', fk1(radians(30), l1_len))  # result (rounded): 30°: [1.73 1]
print('90°:', fk1(radians(90), l1_len))  # result (rounded): 30°: [0    2]
```

Let's stay with our 1 DoF robot for a little longer, but still make the model more generic. We add a positional offset between the robot's base link l_0 at (0, 0) and joint j_1. The shape of link l_0 is changed so that j_1 attaches at position (2, 1) relative to the origin of l_0. Figure 5.29 depicts the modified robot.

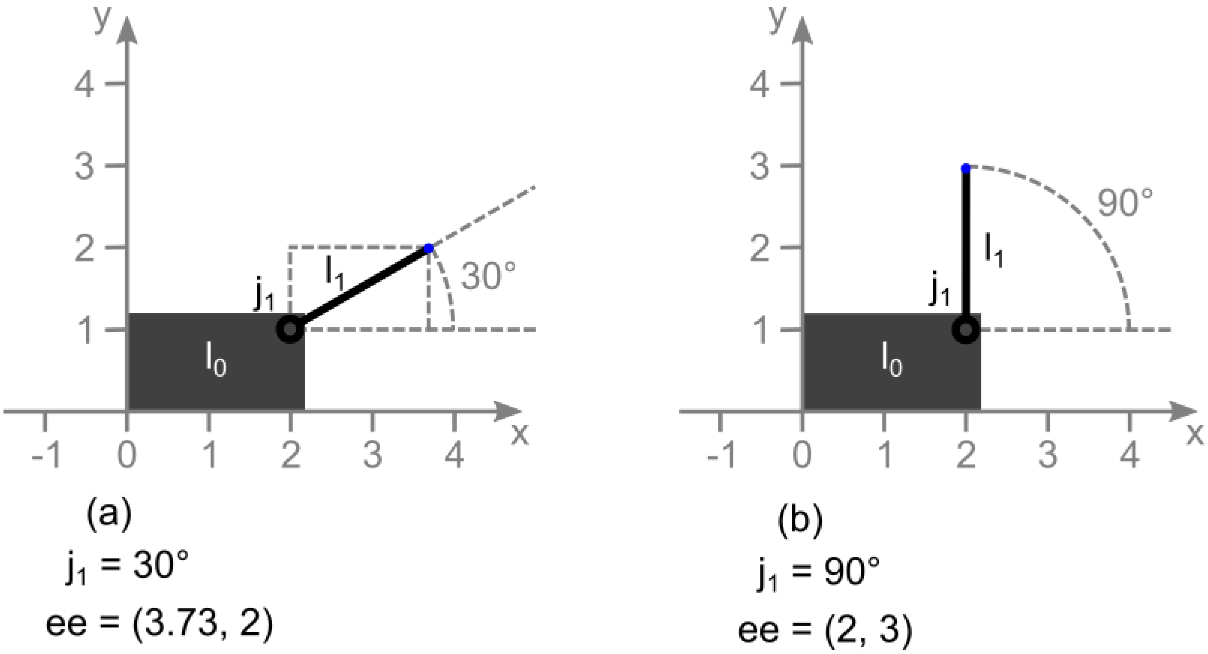


Figure 5.29 1 DoF 2D robot with modified base link l_0.

Modifying the base link l_0 and where joint j_1 attaches to it influenced the position of the EE - it is now in a different position given the same joint value. We need to adjust our forward kinematics for the new j_1 offset. If we consider only the part of the robot between joint j_1 and the EE, nothing has changed. This is a central insight for serial kinematics: We can segment the kinematic chain from base link to end effector into joint-link segments. Breaking up the complete end to end kinematics into independent segments is similar to breaking up complex coordinate transformations into a transformation tree.[210] Both actually follow the same underlying principle. The only difference being that our coordinates here are joint values instead of poses.

These non-Cartesian coordinates are known as *generalized coordinates*. Non-Cartesian simply means that we are not using our usual description for the pose of a rigid body in 3D space, i.e. not describing the pose by 6 parameters (3 position, 3 orientation). Instead we use another set of parameters that can uniquely describe the pose of an object - or a set of objects - in space. The reason for working with generalized coordinates is reducing the overall number of parameters and simplifying calculations. Above we used the expression "set of joint values" when talking about the robot pose in joint space. This set of joint values describes the pose of all robot parts, its links, in generalized coordinates. A description of the robot in regular Cartesian coordinates requires one 3D pose with 6 PP for each link. Thus, for a 7 DoF robot, we need $7 \cdot 6 = 42$ parameters using Cartesian coordinates compared to the 7 joint values.[211]

We can reduce the number of required parameters so significantly because each joint imposes *constraints* on the motion of the two bodies it joins together. An example of such *motion constraints* is a revolute joint.[212] A *revolute joint* only allows motion in one parameter: rotation along its axis. Two independent 3D bodies have $2 \cdot 6 = 12$ parameters (figure 5.30(a)). When coupling two 3D bodies by a revolute joint, the coupled body can still freely move in space, i.e. 6 (Cartesian coordinate) parameters are required to describe it. But the relative motion between the two bodies can be described with a single parameter, the joint value. Put differently, we can write down a function taking the joint value as its only parameter, which provides the full 3D transformation between the two bodies. The pose of the entire mechanism consisting of the two coupled bodies can hence be described using only $6 + 1 = 7$ parameters (figure 5.30(b)): 6 Cartesian coordinate parameters to describe the pose of the first body and 1 generalized coordinate parameter for the joint.

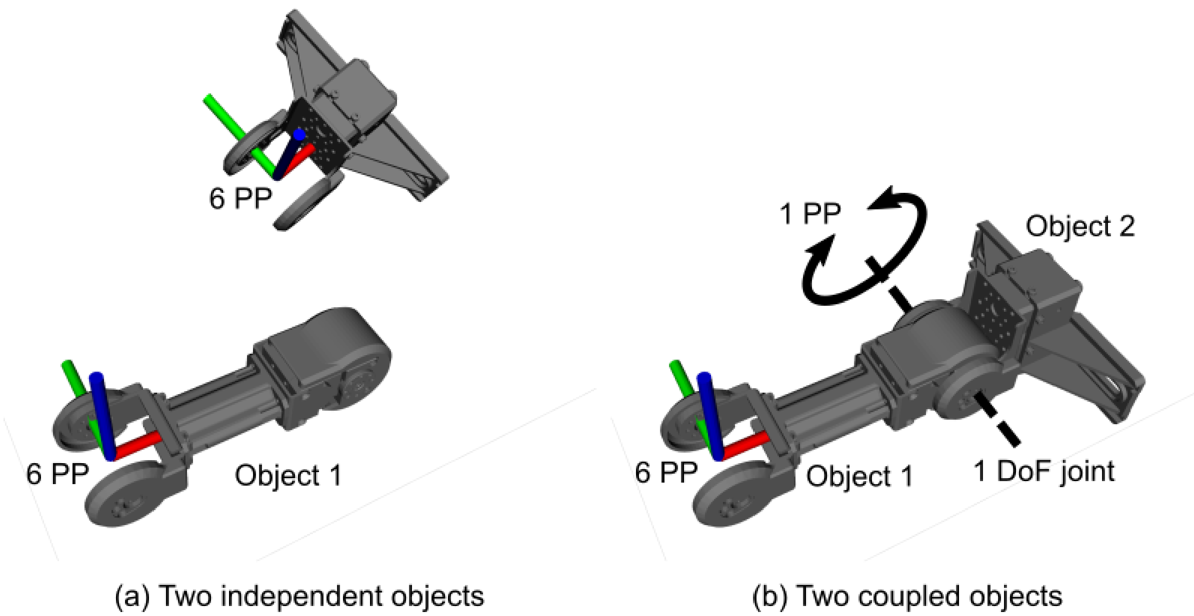


Figure 5.30 Two independent 3D objects (a) requiring 12 PP (Cartesian coordinates). Two 3D objects coupled by a 1 DoF rotational joint (b) requiring 7 PP (generalized coordinates).

The forward kinematics of a serial manipulator can be calculated step by step, going through the kinematic chain from base link to end effector. In case of the modified 1 DoF robot (figure 5.29), the position of j_1, relative to l_0 is (1, 2). The next segment from j_1 to EE is defined by the link l_1, whose pose depends on the value of joint j_1. We already know how to calculate the FK for this segment (`fk1()`). We only need to combine the two transformations, l_0 to j_1 and j_1 to ee to get the overall FK:

```
def fk1mod(j1_position, j1_value, l1_length):
  return j1_position + fk1(j1_value, l1_length)

j1_pos = np.array([2, 1])
l1_len = 2

print('30°:', fk1mod(j1_pos, radians(30), l1_len))
# result (rounded): 30°: [3.73 2]

print('90°:', fk1mod(j1_pos, radians(90), l1_len))
# result (rounded): 30°: [2 3]
```

For the 1 DoF example robot, it is enough to perform a vector addition of the two kinematic segments. However, as we learned in the previous chapter, in the general case we need to combine poses, i.e. take translation and rotation into account.

Let's increase complexity a little by continuing with a 2 DoF robot. The basic approach does not change. We use the robot model to calculate the fixed relations between joints and links. We describe the influence of joint values on the relative relation between adjacent segments in a function. Finally, we combine all this information, starting from base link and advancing segment by segment, until we reach the end effector. As a result we get the EE pose in the robot's coordinate system.[213]

The 1 DoF example robot from figure 5.29 gets another joint and link. The added joint j_2 is a rotary joint with a range of motion of [-90°, 90°]. The additional link l_2 has a length of 1. The end effector is redefined to be the distal end of link l_2. Figure 5.31 shows the newly created 2 DoF example robot.

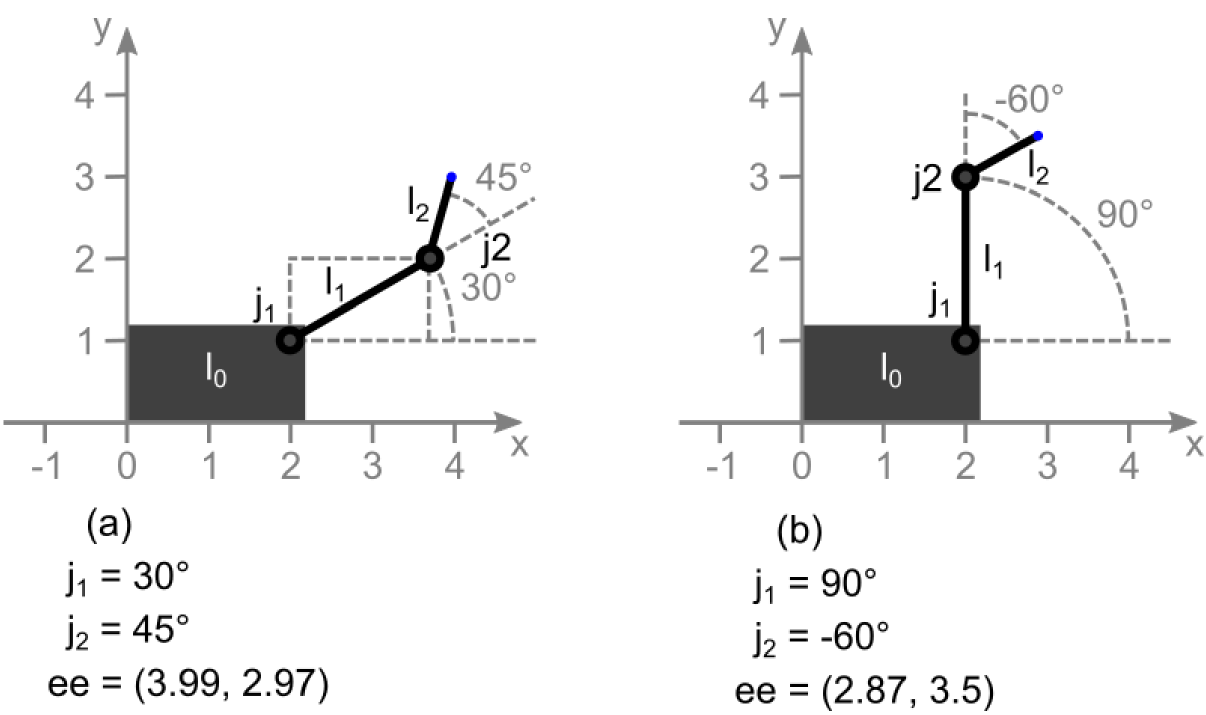


Figure 5.31 Example 2 DoF 2D robot consisting of two rotary joints.

Since the 2 DoF robot is identical with our previous example from base link l_0 to link l_1, the FK of this part is already known. The new joint j_2 is also a rotary joint, hence we can use the already discussed trigonometric method to describe the relation between its joint angle and the resulting position of the link l_2 attached to it. The main difference is that the pose of j_2 in the robot's coordinate system depends on the joint values of all previous[214] joints, in the example specifically on j_1 . Using the approach just discussed, we - fortunately - do not have to deal with j_2 in the robot's coordinate system. Instead we work in the joint's (and link's) *local* coordinate system and then use coordinate transformations to get the global pose. Figure 5.32 highlights the local coordinate system (^{j2}x, ^{j2}y) of joint j_2. This coordinate system is attached to the parent of joint j_2, i.e. link l_1.

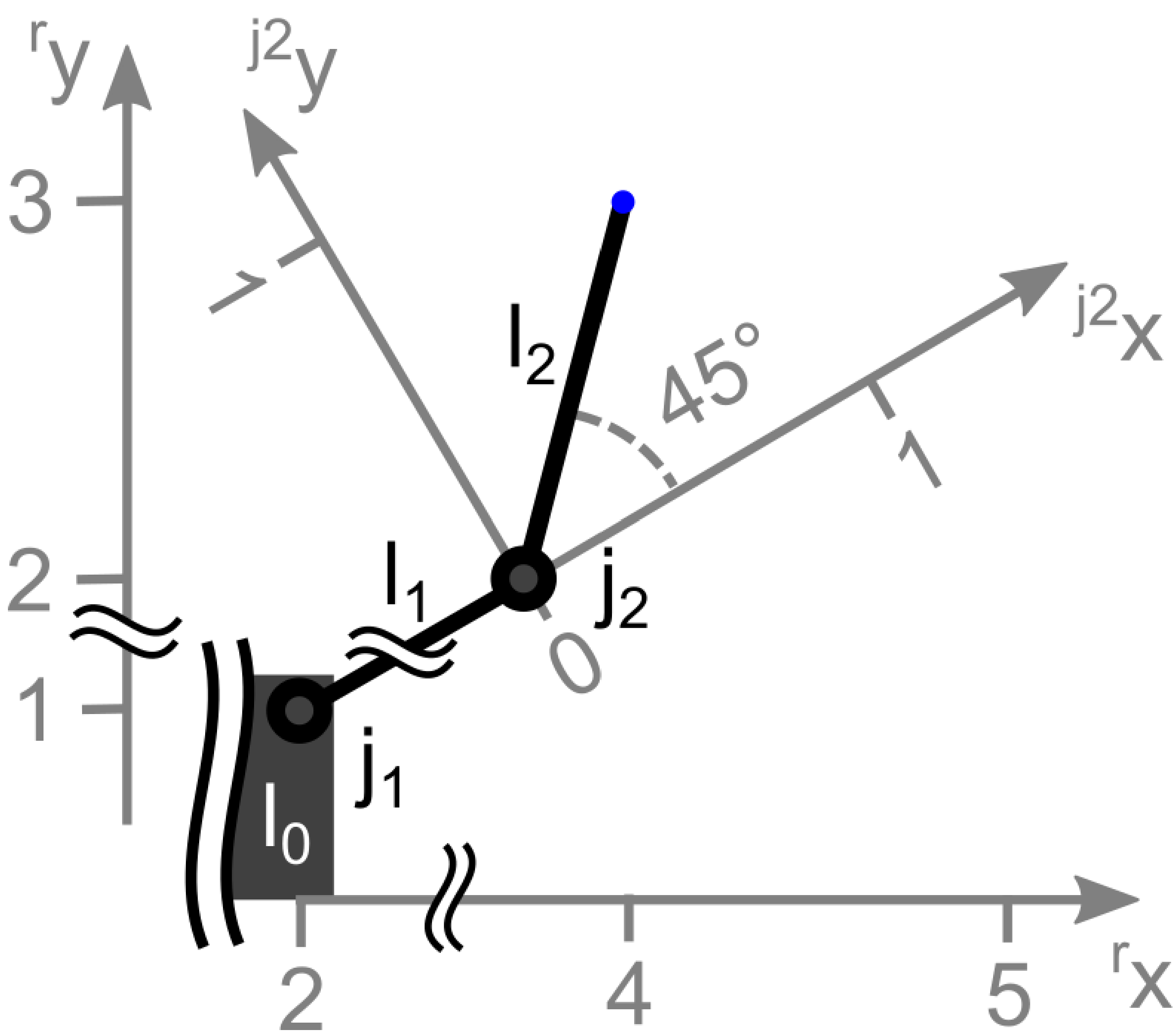


Figure 5.32 Zoom in view of 5.31(a) with the local coordinate system of joint j_2 visualized.

Once we have a function for the relationship between joint value and link pose in the local coordinate system, we use the machinery of coordinate transformations to combine all local descriptions into a complete description. Let's write down each transformation, base link to first joint, joint to next joint[215] and last joint to end effector using (2D) poses:[216]

- Base link l_0 to joint j_1:
  - translation (x, y): (2, 1)
  - rotation (theta/): 0°
- Joint j_1 to joint j_2:
  - rotation: j_1
  - translation: (2, 0)
- Joint j_2 to end effector:
  - rotation: j_2
  - translation: (1, 0)

Expressed in Python:[217]

```
def pose2d(x, y, theta):
  return np.array([[cos(theta), -sin(theta), x],
                   [sin(theta),  cos(theta), y],
                   [0,           0,          1]])

l0_to_j1 = pose2d(2, 1, 0)

def j1_to_j2(j1_val):
  # rotation before translation here
  return pose2d(0, 0, j1_val) @ pose2d(2, 0, 0)

def j2_to_ee(j2_val):
  return pose2d(0, 0, j2_val) @ pose2d(1, 0, 0)
```

Now we can combine the individual transformations `l0_to_j1`, `j1_to_j2` and `j2_to_ee` to get the end effector pose via `fk2()`:

```
def fk2(j1_val, j2_val):
  return l0_to_j1 @ j1_to_j2(j1_val) @ j2_to_ee(j2_val)

# Only prints the position of end effector, not the orientation.
print('j1=30°, j2=45°:', fk2(radians(30), radians(45))[0:2,2])
# result (rounded): [3.99 2.97]

print('j1=90°, j2=-60°:', fk2(radians(90), radians(-60))[0:2,2])
# result (rounded): [2.87 3.5]
```

Adding a third link is a mere repetition of this procedure. Also going from 2D to 3D requires no additional knowledge. The major difficulty in 3D is the practical issue of working with 3D objects on 2D paper/displays. In terms of the trigonometry, we can decompose 3D objects into 2D planes. Figure 5.33 illustrates usage of a 2D triangle in 3D to relate the angle of a 1 DoF rotary joint to the pose of an attached link.

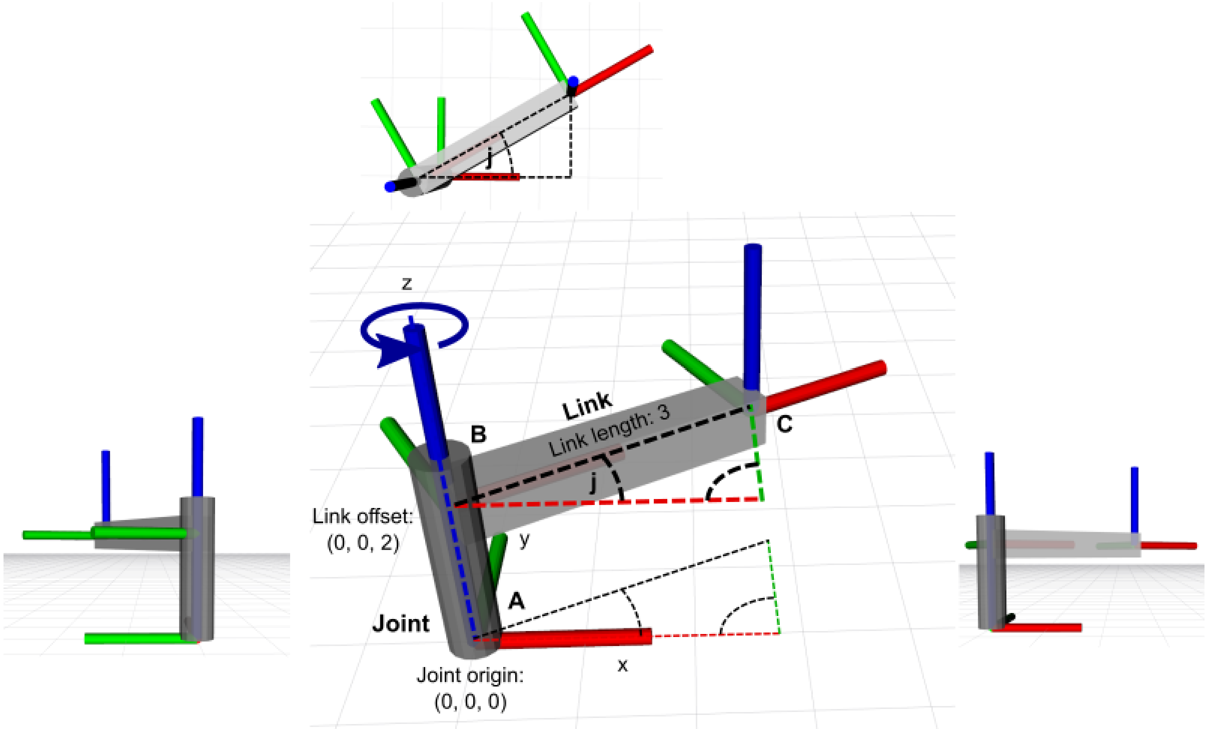


Figure 5.33 Applying trigonometry in 3D by identifying relevant 2D planes.

There will be a sufficient number of 3D robot models in later chapters for you to become accustomed to 3D geometry. Thus, instead of repeating in 3D what we have already done in 2D, let's continue and look at another joint type and other robot types before learning about inverse kinematics.

We create a new 2 DoF robot by replacing the first rotary joint with a *prismatic joint* that extends linearly, like a drawer slide or telescopic pole Prismatic joints are also known as *linear joints* or *sliding joints*. The new prismatic joint j_1 can move between 0.5 m and 2.5 m, hence its joint limits are [0.5, 2.5]. It is attached to link l_0 at (2, 1). The result can be seen in figure 5.34.

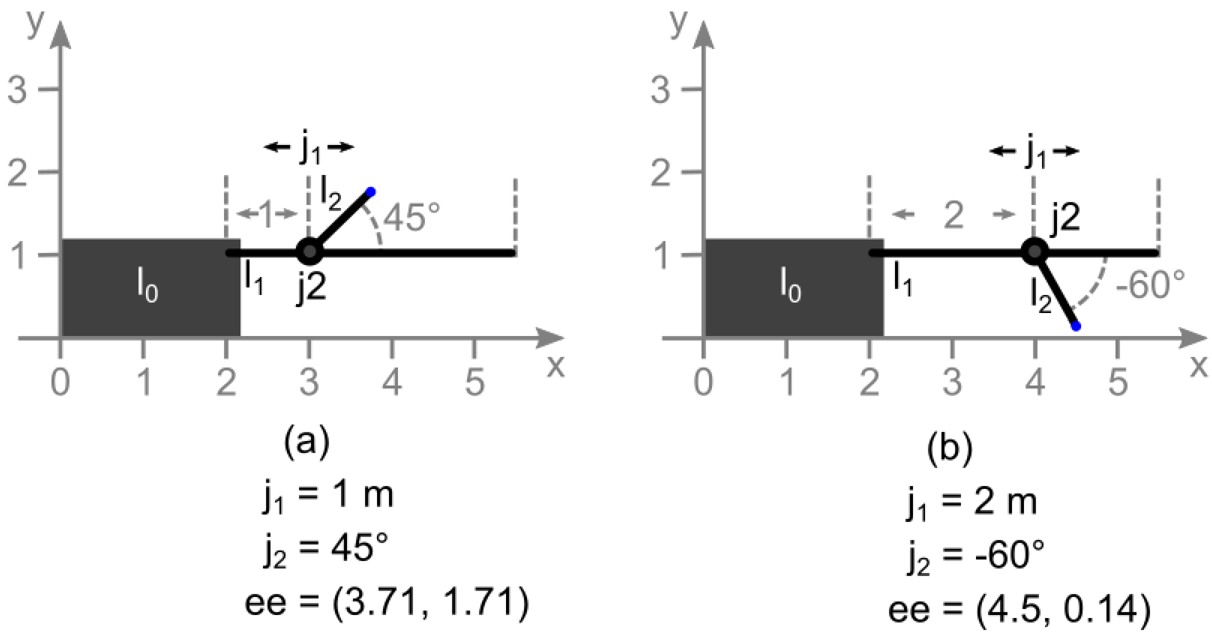


Figure 5.34 2 DoF example robot consisting of one prismatic and one rotary joint.

The only difference between the previous 2 DoF RR robot and the new 2 DoF PR robot is the transformation between joint j_1 and joint j_2.[218] Thus, only `j1_to_j2()` must be adapted. Looking at the effect of different values for joint j_1 in figure 5.34, we observe that it only influences the x value of the j_2 pose. In the local coordinate system of j_1 the exact relation between joint value and pose of the connected j_2 is simply $x = j_1$. This is all the information we need to write our `j1_to_j2()`, which together with the unmodified `l0_to_j1()`, `j2_to_ee()` and `fk2()` gives us the forward kinematics. Note that the joint value of j_1 changed from an angle to a length.

```
def j1_to_j2(j1_val):
  return pose2d(j1_val, 0, 0)

# Reusing the unmodified l0_to_j1(), j2_to_ee() and fk2() from above.

# Only prints the position of end effector, not the orientation.
print('j1=1m, j2=45°:', fk2(1.0, radians(45))[0:2,2])
# result (rounded): [3.71 1.71]

print('j1=2m, j2=-60°:', fk2(2.0, radians(-60))[0:2,2])
# result (rounded): [4.5  0.14]
```

We have now derived the FK solutions for relatively simple (2D) robots from first principles (geometry). Although it is important to understand how these solutions come about, in practice you can work at a higher level of abstraction by reusing known FK solutions for the elementary joint types. For every joint type, there is a known parametrized transformation, which can be combined with information about the links in order to get the overall manipulator FK. We will look at some common ways to model kinematics in section 5.8. The overall FK can then be programmatically derived from such a robot model. There is no need to put together the transformation functions by hand - at least for forward kinematics.

Our approach works for any kind of serial kinematic. What about parallel kinematics? First of all, the difference between serial and parallel kinematics is the existence of joint/link loops between the robot base and the end effector. In *serial kinematics* there is only a single joint/link path from base to end effector. *Parallel kinematics* contain at least one loop, i.e. there are at least two different paths between base and EE. Both types of kinematics have advantages and drawbacks. In general, serial kinematics have a larger workspace, but are less precise and have less payload capacity. Examples of parallel kinematics are Delta robots, Stewart platforms (also known as hexapod) and cable-driven robots. The kinematic loops impose additional constraints on valid sets of joint values. We will not address parallel kinematics in this introductory chapter, but you will encounter them in later chapters.

Let's now discuss the forward kinematics for one type of kinematics for ground-based mobile robots: the differential drive. The *differential drive*, or diff drive for short, consists of two independently actuated wheels. Either the robot balances on these two wheels or there are additional passive caster wheels. While the caster wheels are important to prevent the robot from dipping over, we can ignore them from a kinematics point of view. Figure 5.35 shows the differential drive kinematics and the essential parameters required to create an FK function.

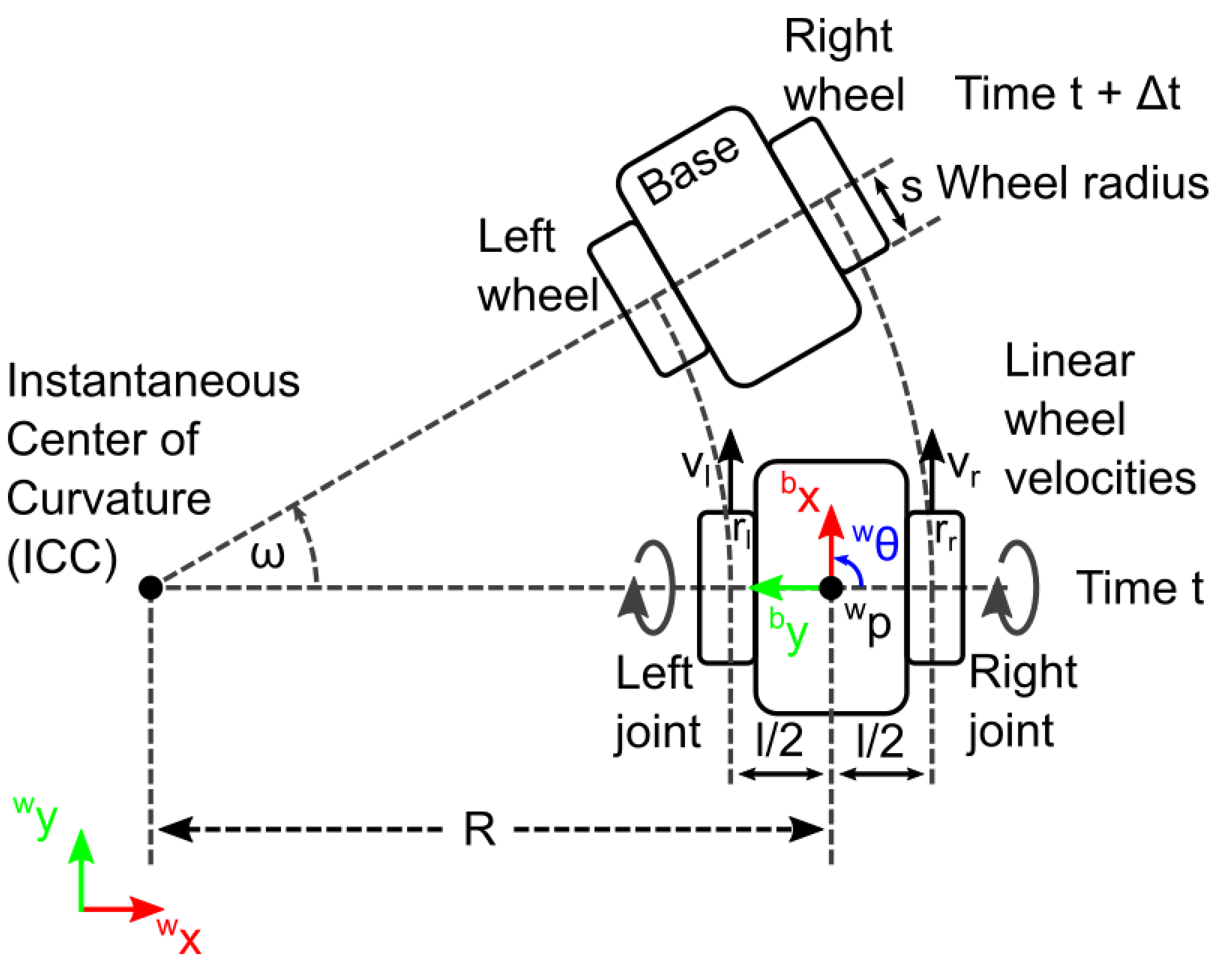


Figure 5.35 Differential drive kinematics of a mobile robot. Fixed coordinate system w (^wx, ^wy, ^{w}) and robot body attached frame b.

There is one crucial difference between kinematics for manipulators and mobile robots that you need to keep in mind. For manipulator joints we assume to know the absolute value of each joint. The mechanical setup of the manipulator ensures that moving to the same joint values always results in the same robot pose.[219] In case of mobile robots, it is not possible to reliably move the robot to the same pose by moving the robot joints, e.g. its wheels, into the same position. Slippage between the robot's wheels or legs can neither be fully controlled nor fully eliminated.

This entails two major consequences for mobile robots. First, mobile robots, in contrast to manipulators, need additional sensor data beyond their joint position sensors in order to control their pose in the environment. Second, the joint position has no stable relation to the environment pose, thus kinematics is calculated based on velocities instead of positions. We are working with *velocity* kinematics.[220] The assumption is that there is a known and sufficiently stable *instantaneous* relation between wheel velocities and the resulting motion of the robot, but that this relation is not stable over time due to slippage. Since velocity is the derivative of position, i.e. the momentary rate of change in position, we can relate wheel velocity to robot velocity, although there is no stable relation between wheel position and robot pose. Closing the loop on the environment with additional sensors to address the first consequence will be a topic for chapter 11.[221] The second topic directly influences how we formulate the differential drive kinematics here. Due to these differences, you will often find a different term used for the forward kinematics of mobile robots: *odometry*.[222]

We start with some intuitions about diff drive kinematics. If both wheels rotate with the same speed, the robot will move on a straight line. If the left wheel rotates slower than the right wheel, the robot will make a left turn. If the right wheel is slower relatively to the left one, the robot will turn right. The turn angle depends on the magnitude of the difference between the wheel velocities.

The insight that greatly facilitates calculating diff drive kinematics is to introduce the *instantaneous center of curvature (ICC)*. The ICC is the point along the wheel axis around which the robot is rotating *instantaneously*. The term "instantaneously" indicates that this center of curvature is only valid for an instant, a snapshot, of the robot motion - it shifts over time. You can think of it in a similar way as about a derivative, e.g. velocity, being the *instantaneous* rate of change of a function, e.g. position.

Looking at the ICC in figure 5.35, we recognize that both wheels rotate around this point on circles of different roll radii. The difference between the radius r_l on which the left wheel is rolling and the right wheel's roll radius r_r is the distance l between the two wheels. The origin of the robot coordinate system is placed in the middle of the wheels. We define the radius R between the ICC and the robot origin as well as the rate of rotation ω (omega) around the ICC. Put differently, ω is the angular velocity about the rotation axis defined by the ICC.

In order for the wheels to roll, as contrasted to them spinning or being dragged along, the following condition must be satisfied: $v_i = r_i \cdot \omega$. The linear velocity v_i of each wheel must be equal to the roll radius r_i multiplied by the angular velocity ω.[223] We can express the roll radius of the left wheel as $r_l = R - l/2$ and as $r_r = R + l/2$ for the right wheel. Putting this together we can write

$$v_l = (R - l/2) \cdot \omega$$
$$v_r = (R + l/2) \cdot \omega$$

Solving the equations for R and ω, we get

$$\omega = (v_r - v_l) / l$$
$$R = l/2 \cdot (v_l + v_r) / (v_r - v_l)$$

If both wheels have the same speed, $v_r = v_l$, then $\omega = 0$, i.e. the robot moves straight. In all other cases, ω is not zero and R has a finite value. An interesting special case is when one wheel's velocity is zero, $v_l = 0$ or $v_r = 0$, then $R = \pm l/2$ and the ICC is at the wheel with zero velocity. The robot then rotates around the wheel standing still.

In world coordinates, the robot is located at ${}^w\mathbf{P} = ({}^wx, {}^wy, {}^w\theta)$. With some trigonometry, we can describe the location of the ICC in terms of ${}^w\mathbf{P}$ and R. The ICC is located at $({}^wICC_x, {}^wICC_y) = ({}^wx - R \cdot \sin({}^w\theta), {}^wy + R \cdot \cos({}^w\theta))$. Now we can formulate a relation between the change in the robot's pose ${}^w\mathbf{P}$ given its linear wheel velocities (v_l, v_r). The wheel radius s directly relates the linear wheel velocities (v_l, v_r) to the joint velocities (j_l', j_r'): $(v_l, v_r) = (s \cdot j_l', s \cdot j_r')$.

The rate of change in pose, i.e. the pose's derivative, is called a *twist* and consists of angular and linear velocities. *Spatial velocity* is another term for twist. We denote twists by the symbol **V**. Poses and twists have the same number of pose parameters, e.g. 3 PP in 2D and 6 PP in 3D. We first write down the relation between wheel velocities (v_l, v_r) and change in pose, i.e. twist $({}^w\theta', {}^wx', {}^wy')$.[224] Afterwards we integrate the twist to get the resulting robot pose $({}^wx, {}^wy, {}^w\theta)$.

For a short time Δt, the robot moves with velocity (v_l, v_r). Thus, a specific ICC at position $({}^wICC_x, {}^wICC_y)$ results and there is also a specific value for R and ω. We express everything in the world coordinate system and hence omit the reference frame w. This motion with velocity (v_l, v_r) for time Δt causes the following twist $({}^w\theta', {}^wx', {}^wy')$:

$$\theta' = \omega \cdot \Delta t$$
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(\omega \cdot \Delta t) & -\sin(\omega \cdot \Delta t) \\ \sin(\omega \cdot \Delta t) & \cos(\omega \cdot \Delta t) \end{bmatrix} \begin{bmatrix} x - \mathrm{ICC}_x \\ y - \mathrm{ICC}_y \end{bmatrix} + \begin{bmatrix} \mathrm{ICC}_x \\ \mathrm{ICC}_y \end{bmatrix}$$

Instead of a step by step derivation of this formula, I'll start the explanation from the final formula. Orientation changes with a rate of ω · Δt, i.e. the angular velocity multiplied by the short duration Δt. The vector (x - ICC_x, y - ICC_y) is the current pose of the robot with the ICC shifted to the origin. The matrix is a rotation matrix and rotates the ICC-shifted-to-origin pose by ω · Δt. The vector added on the right shifts the robot pose back to its previous position. The formula simply describes rotating the robot around the ICC by an amount of ω · Δt. In other words, there is nothing to derive about this formula. It is the generic formula for rotating a body in 2D (3 PP) around a point, here the ICC, by a certain angle, here by an angle of ω · Δt.

SIDEBAR **2D rotation around an arbitrary axis**

A 2D body (3 PP) can be rotated around an arbitrary axis of rotation r as shown in figure 5.36:

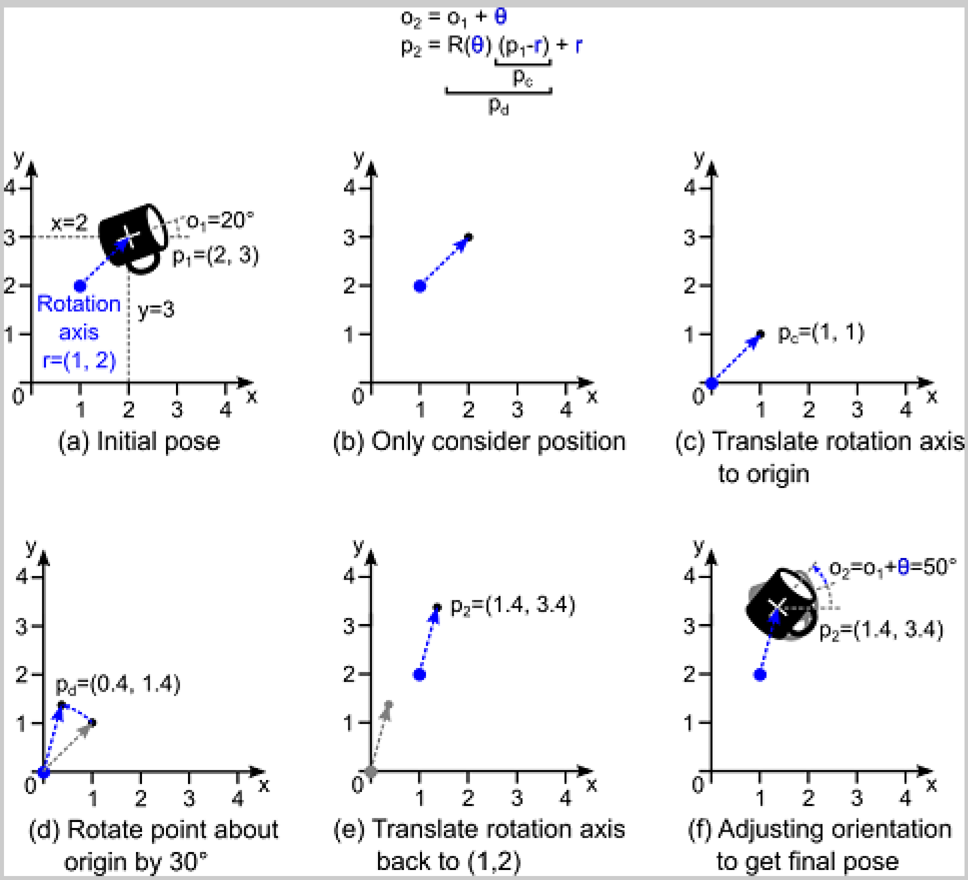


Figure 5.36 Rotation of 2D body (p_1, o_1) = (2, 3, 20°) around rotation axis r = (1,2) by = 30°.

We can also write this in homogeneous coordinates

$$\mathbf{P}_2 = \mathbf{P}_r\mathbf{R}(\theta)\mathbf{P}_r^{-1}\mathbf{P}_1$$

and Python

```
# See definition of pose2d(x, y, theta) above
r = (1, 2)
theta = radians(30)
pr = pose2d(r[0], r[1], 0)  # a point, orientation does not matter
p1 = pose2d(2, 3, radians(20))
pc = np.linalg.inv(pr) @ p1
pd = pose2d(0, 0, theta) @ pc
p2 = pr @ pd

print((p2[0,2], p2[1,2], degrees(acos(p2[0,0]))))
# rounded: (1.4, 3.4, 50)
```

Reading the formula right to left, we can interpret the poses/transformations in the following way:

- P_r^{-1} changes the frame of P_1 from world (${}^wP_1$) to a frame centered on the rotation axis r (${}^rP_1$). P_r can be understood as ${}^wT_r$ and thus P_r^{-1} as ${}^rT_w$. See `pc` .
- R() rotates ${}^rP_1$ in its rotation axis coordinate system. See `pd`.
- P_r transforms the rotated point, which is specified in rotation axis coordinates, back into world coordinates, giving us ${}^wP_2$. See `p2`.

Rewriting the symbols of the formula - the matrices are still the same - with this understanding, we get ${}^wP_2 = {}^wT_r\, R()\, {}^rT_w\, {}^wP_1$.

These two interpretations are equivalent. One interpretation involves a temporary virtual move of the rotation axis to the origin. The other interpretation involves a temporary change of basis to a coordinate system centered at the rotation axis.

We now have a formula for the robot's twist over time. Next we want to know its pose over time. We already learned that integrating velocity gives us position. Expressed using more general terms, we can get the pose by integrating the twist. Our sidewalk robot example was described in 1D (1 PP), the diff drive robot is 2D (3 PP). Fortunately, we can integrate a vector-valued function by integrating each vector dimension separately:

$$\mathbf{f}(t) = \begin{bmatrix} f_a(t) \\ f_b(t) \\ f_c(t) \end{bmatrix}$$

$$\int \mathbf{f}(t)\,dt = \begin{bmatrix} \int f_a(t)\,dt \\ \int f_b(t)\,dt \\ \int f_c(t)\,dt \end{bmatrix}$$

We can apply numeric integration to the calculated twist (θ', x', y') to get the pose (x, y, θ). It is also possible to calculate the integral for diff drive forward kinematics symbolically. However, it is important to keep in mind that the resulting pose will not match the actual pose of the robot due to wheel slippage and skidding.

Over time the calculated pose will deviate more and more from the actual pose. This is a general problem with any kind of *dead reckoning* process. The issue lies neither in diff drive nor the mathematical description of its kinematics. It is an inherent property of integrating changes over time (here twist), because errors accumulate over time. The only mitigation are additional sensor measurements that directly measure the quantity (here pose) instead of only measuring its changes. Having said that, forward kinematics and odometry are a vital building block for mobile robots.[225] They provide exact information when using them instantaneously, e.g. to calculate the next actuator commands. They also remain sufficiently accurate when used locally, e.g. to plan over a short time horizon.

I want to close the discussion of differential drive for this section with a terminological clarification. The term kinematics is consistently used and understood when talking about manipulators. It is always about relating joint space to task space (cf. figure 5.17). When it comes to mobile robots, especially aerial robots, you will often encounter the term kinematics as it is used in physics. In physics, more specifically in classical mechanics, kinematics describes the motion of bodies. This does not necessarily imply creating a connection between joint space and task space. It might as well mean to describe the relation between the trajectory of a coordinate system moving along with the robot to an external inertial coordinate system. Another clarification concerns (forward) kinematics and odometry. For mobile robots, we will refer to the instantaneous relation between joint velocity and twist as forward kinematics. When discussing the pose calculated by integrating the twists over time, we will use the term odometry.

The need to clarify terminologies and to be aware of the specific conventions followed in the book, arises from the many divergences within the field of robotics. Nevertheless, my goal is to describe to you all areas of robotics in a manner that is as consistent as possible. If you work in one specific robotics domain, you will likely adopt the terminology, notation and conventions from this domain. If you work across robotics domains, you will have to compartmentalize terms and conventions or at least treat the various domains as dialects of a unified language of robotics. 226

We finish this section on forward kinematics with a brief look at quadrotor kinematics. The obvious difference with flying mobile robots is that they inherently require a description in 3D, as opposed to the 2D description often sufficient for ground-based robots. Another dissimilarity, especially to manipulators, is the free-floating nature of airborne robots. There is no kinematic chain between a fixed base link and an end effector when it comes to aerial vehicles. We already observed a similar phenomenon with the ground-based mobile robot's (differential drive) kinematics. Coordinate transformations enables us to express the relation between the flying robot's pose described in its local moving coordinate system and in an external stationary coordinate system. By means of differentiation we can also relate the change in the robot's pose, its twist, in different coordinate systems. However, this relates two 3D poses or twists with each other. It does not relate joint space values, in generalized coordinates, to task space poses, in

Cartesian coordinates.[227] Hence the clarification in the two previous paragraphs. With this being stated, let's look at the relation between the moving quadrotor coordinate system and an external stationary one.

Figure 5.37 shows an example quadrotor with the two coordinate systems, body and world, as well as the four rotor joints highlighted.

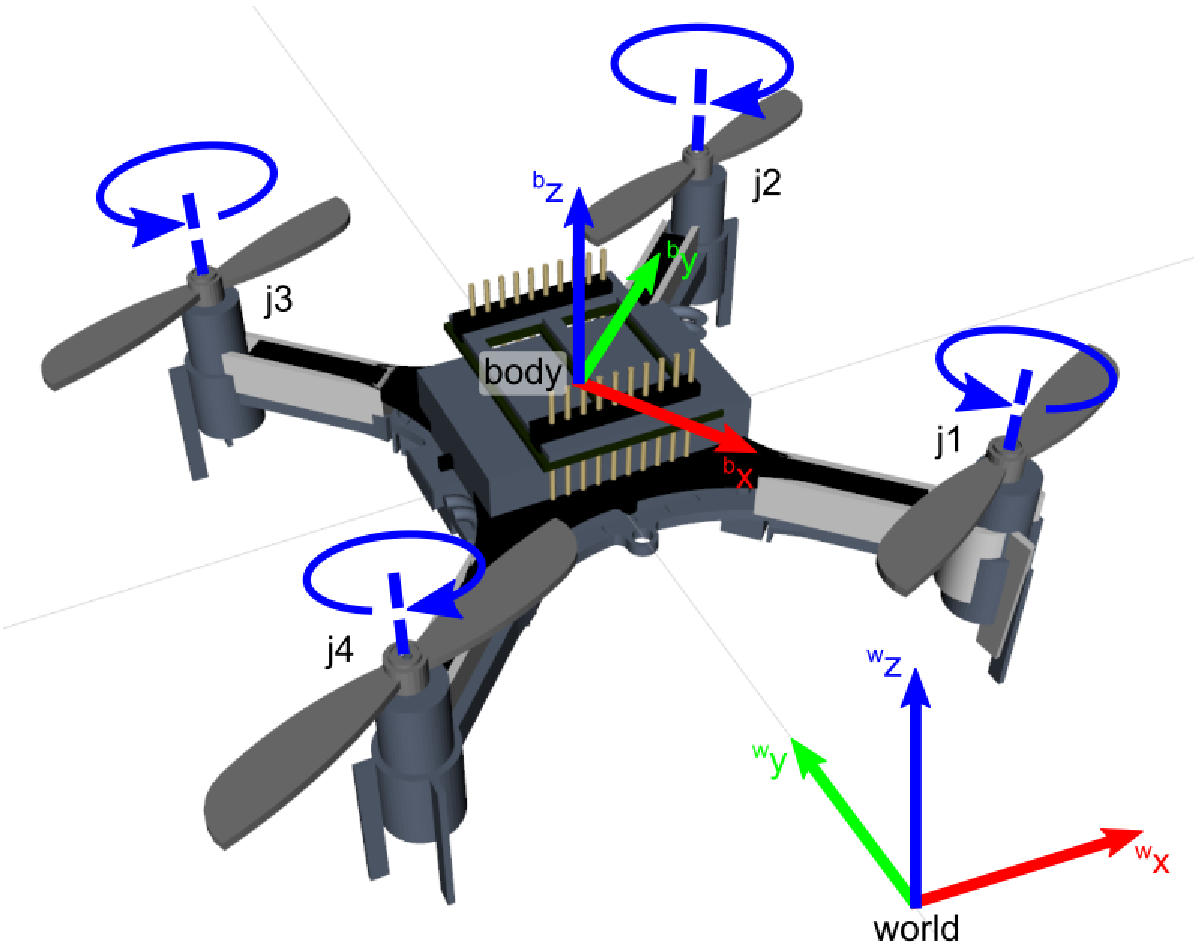


Figure 5.37 A quadrotor with its four rotor joints, its body coordinate system and a stationary world coordinate system.

The rotor joints 1 and 3 rotate counter clockwise (CCW). Rotors 2 and 4 rotate clockwise (CW). Due to different propeller pitches on joint 1 & 3 and joint 2 & 4 all generate "upward" *thrust*, i.e. towards the positive z axis of the body frame ^{b}z. The reason for this arrangement is that each rotor does not only cause thrust, but also torque opposite to its direction of rotation.[228] The opposing torques mostly cancel each other out. Don't worry if you are not familiar with the dynamics terms, e.g. force, torque, thrust, that I just used, we will discuss them in the next chapter. For now it is sufficient to understand the following preliminary description.

Gravity force is acting on the quadrotor and accelerating it downward ($-^{w}z$). Each rotor generates a force along its axis of rotation, accelerating the rotor and attached airframe in the direction of the rotor. When all rotors spin at the same speed, they generate the same amount of thrust. If this overall thrust exactly balances out gravitation, the quadrotor will hover in the air, i.e. keep a constant pose in world coordinates. This assumes there are no other external forces, such as wind, acting on the robot. If the overall thrust is higher than the opposing (gravitational) force, the quadrotor will move along the positive ${}^{b}z$ axis.[229] For lower overall thrust, the resulting motion is in negative ${}^{b}z$ axis. When rotors 2 and 4, both on the ${}^{b}y$ axis, generate different amounts of thrust, the quadrotor will roll around the ${}^{b}x$ axis. Similarly, when rotors 1 and 3, located on the ${}^{b}x$ axis, differ in thrust, the robot will pitch around the ${}^{b}y$ axis. Finally, if the speed of the motor pair 1 & 3 differs from the motor pair 2 & 4, the resulting motion is yaw around the ${}^{b}z$ axis.

Forward kinematics of the quadrotor provides us with the relation between motion expressed in its body coordinate system and the world coordinate system. In aviation terminology, we want to get the change in attitude and altitude given the pose change in body coordinates. *Attitude* is the aerial vehicle's orientation in the world coordinate system. The stationary world coordinate system is often referred to as *inertial frame of reference*. *Altitude* is the aerial vehicle's height above ground, i.e. above the ${}^{w}x$-${}^{w}y$ plane, hence the value of the ${}^{w}z$ coordinate.[230] We can map between a pose given in the body coordinate system ${}^{b}\mathbf{P}$ and and the same pose in world coordinates ${}^{w}\mathbf{P}$ via a transformation ${}^{w}\mathbf{T}_{b}$: ${}^{w}\mathbf{P} = {}^{w}\mathbf{T}_{b}{}^{b}\mathbf{P}$. Inverting the transformation ${}^{w}\mathbf{T}_{b}^{-1}$, we get the inverse correspondence ${}^{b}\mathbf{T}_{w}$: ${}^{b}\mathbf{P} = {}^{b}\mathbf{T}_{w}{}^{w}\mathbf{P}$.[231] Hence, the solution to relate the poses is coordinate transformations, which we already discussed in the previous chapter.

To relate twists instead of poses, all we have to do is calculate the derivative of this equation. While this is conceptually simple, the mathematics are more involved. This complexity is mostly due to the non-linear time-dependent terms in the transformation. A 3D transformation ${}^{w}\mathbf{T}_{b}$ is a function of (x, y, z, roll, pitch, yaw). Writing these parameters in a more detail way as (x(t), y(t), z(t), roll(t), pitch(t), yaw(t)) the dependence on time t becomes much more obvious. We postpone analyzing the time derivative of a rotation matrix until later and finish this section on forward kinematics with an API point of view.

Referring back to figure 5.17 once more, the forward kinematics API has joint space values (generalized coordinates) as input and task space values (Cartesian coordinates) as output. The abstract FK API has the same time unit for inputs and outputs. Positional joint space values are mapped to positional task space values and velocity joint space values are mapped to velocity task space values. Hence the abstract forward kinematics API is simply

```
pose = robot.position_forward_kinematics(joint_positions)
twist = robot.velocity_forward_kinematics(joint_velocities)
```

Depending on the specific robot, `pose` and `twist` will have 3 PP (2D) or 6 PP (3D). The number of values in the `joint_positions` and `joint_velocities` vectors depends on the number of robot joints. I recommend to use self explaining variable names rather than relying on readers of your code to be familiar with the particular symbol conventions of your favorite robotics textbook. It is still good to mentally map the variables to their mathematical symbols. In the conventions of this book, the Python code above would use

- **P** for `pose`. Ideally, $^{r}\mathbf{P}$ to explicitly specify the reference frame is the robot coordinate system.
- **q** for `joint_positions`.
- **V** for `twist`. Again, ideally with a leading superscript to denote the frame of reference.
- **q**' for `joint_velocities`.

Let's now investigate how to go from task space to joint space.

5.7 Inverse Kinematics

Inverse kinematics (IK) maps task space values (poses, twists and further derivatives) to joint space values (joint positions, joint velocities and further derivatives). We again start with manipulator IK in 2D and revisit the 1 DoF (figure 5.29) and 2 DoF RR (figure 5.31) robots from the previous section.

The FK for the 1 DoF robot, reproduced with further annotations in figure 5.38 below, is $^{r}\mathbf{P}_{ee} = {}^{r}\mathbf{T}_{j1} \cdot {}^{j1}\mathbf{T}_{ee}$ when expressed in homogeneous coordinates. Given the generalized joint coordinates $\mathbf{q}$ (here $\mathbf{q} = q = j_1$), we get the end effector (EE) pose (here $^{r}\mathbf{P}_{ee}$ has 3 PP: ^{r}x, ^{r}y, $^{r}\theta$).

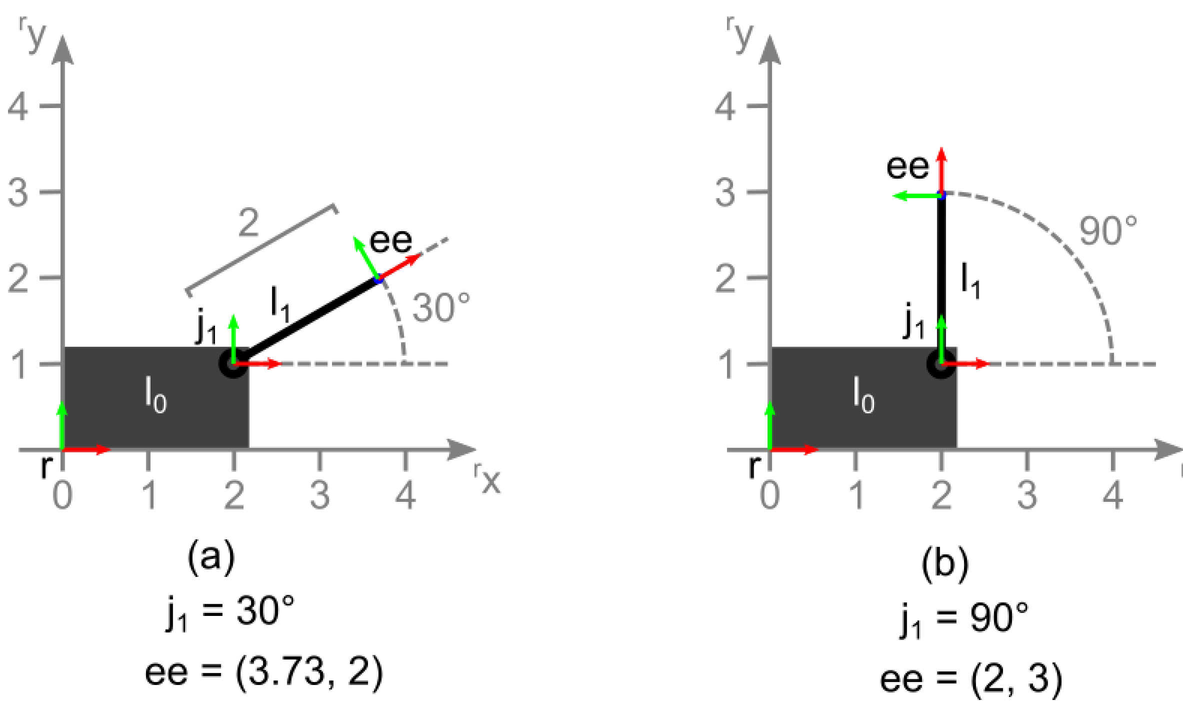


Figure 5.38 1 DoF 2D robot (same robot as in figure 5.29).

Using what we learned in the meantime, this description is more concise compared to the initial one. Apart from the robot coordinate system, we put a coordinate system into each joint and one at the end effector. ${}^{r}\mathbf{T}_{j1}$ is considered constant as it only depends on constant values, such as the size of link l_0 and the attachment point of joint j_1. Constant here means that it does not depend on time, i.e. its derivative is zero. In contrast, ${}^{j1}\mathbf{T}_{ee}$ depends on the position of joint j_1, which can vary over time. To make this explicit, we use a function-like notation and add the time dependent variables as parameters of the transformation: ${}^{j1}\mathbf{T}_{ee}(j_1)$. In this notation, we have

$$
{}^{r}\mathbf{T}_{j1} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{j1}\mathbf{T}_{ee}(j_1) = \begin{bmatrix} \cos(j_1) & -\sin(j_1) & 0 \\ \sin(j_1) & cos(j_1) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
$$

${}^{r}\mathbf{T}_{j1}$ corresponds to `l0_to_j1` and ${}^{j1}\mathbf{T}_{ee}$ to `j1_to_j2` in the previous section. For reasons of readability, we wrote ${}^{j1}\mathbf{T}_{ee}$ using two matrices, but we could multiply them to get a single homogeneous matrix. Now that we have a FK description of our 1 DoF robot, how can we get to the IK?

Given there is only one rotary joint, the only reachable positions are on a circle centered at j_1 with a radius of l_1. For the same reason, the EE orientation has a fixed relation to its position, i.e. collinear with l_1 link and perpendicular to the circle around j_1. Focusing on a position (${}^{j1}x$, ${}^{j1}y$) in the j_1 frame, we can directly apply basic trigonometry (cf. figure 5.28). This gives us $\tan(j_1) = {}^{j1}y / {}^{j1}x$ and hence $j_1 = \tan^{-1}({}^{j1}y / {}^{j1}x)$.[232] You already know how to transform a point given in robot coordinates (${}^{r}x$, ${}^{r}y$) into the j_1 frame using ${}^{r}\mathbf{T}_{j1}$. For reachable positions, we have our IK solution.

However, the above formula also calculates a j_1 value for positions that are clearly not reachable, e.g. (${}^{j1}x$, ${}^{j1}y$) = (1, 1) or (10, 0) or (0, -2). First, we need to make sure the IK "solution" is actually within joint limits. For the 1 DoF robot we specified [0°, 180°] as joint limits of j_1. Second, we must validate that the calculated joint values actually result in the desired pose. Luckily we already know the tool for it: forward kinematics. Calculating the FK for the joint values and comparing the calculated FK pose to the desired pose that was the input to IK is sufficient to validate the IK solution. Let's move to the next kinematically more interesting robot.

When we created the FK for the 2 DoF RR robot[233] we composed it joint by joint, one joint-link segment at a time. There is no similar step by step process when it comes to IK. It is good to understand why this is the case. The modularity of the forward kinematic calculation for serial kinematics hinged on the fact that the pose of joint i only depends on the pose of all previous joints i-1, i-2, …, 1, but not on next joints i+1, i+2, …, n. Therefore, it is possible to start with joint 1, calculate the resulting pose of joint 2 given the joint 1 value and continue the calculation with joint 2. This process continues until reaching the last joint and the EE respectively. We cannot simply reverse this process for inverse kinematics. In general, a given EE pose does *not* allow us to calculate the last joint's value, which would then in turn allow us to calculate the previous joint value and so forth. The value of joint i depends both on previous and next joints. We thus need to solve for all joint values simultaneously.[234]

Not only was the 1 DoF robot kinematically deficient in a 3 PP task space, the same is also true for the 2 DoF robot. Therefore, position and orientation of the EE are not independent. We can only choose two out of three parameters. The third parameter is then determined by the robot's kinematics. We pick the two position parameters. Yet, we could equally well choose one position and the one orientation parameter. Hence, the IK task is to get the joint positions $\mathbf{q} = (j_1, j_2)$ given the EE position $\mathbf{p} = ({}^{r}x, {}^{r}y)$.

Figure 5.39(a) depicts the 2 DoF RR robot along with some helpful geometric relations.

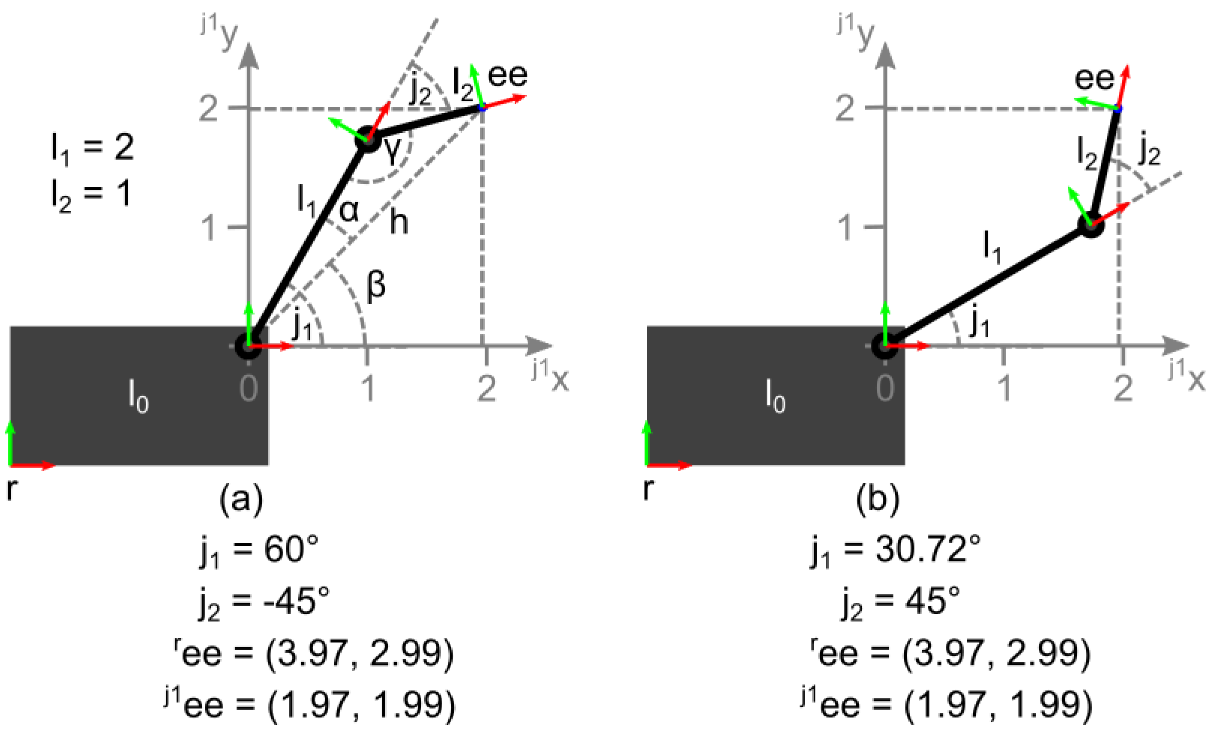


Figure 5.39 Inverse kinematics of 2 DoF RR robot (same robot as in figure 5.31).

We start by writing h in terms of ^{j1}x and ^{j1}y. We leave away the frame identifier for better readability. h is the hypotenuse of a right-angled triangle together with x and y. Using the Pythagorean theorem, we get $h = \sqrt{x^2 + y^2}$. Next we apply the law of cosines to the triangle formed by joint j_1, joint j_2 and the EE.

SIDEBAR **Law of cosines**

The following law of cosines is applicable in every triangle. Note the relation between sides a, b, c and angles (alpha), (beta), (gamma) as shown in figure 5.40: is opposite to a, opposite to b and opposite to c.

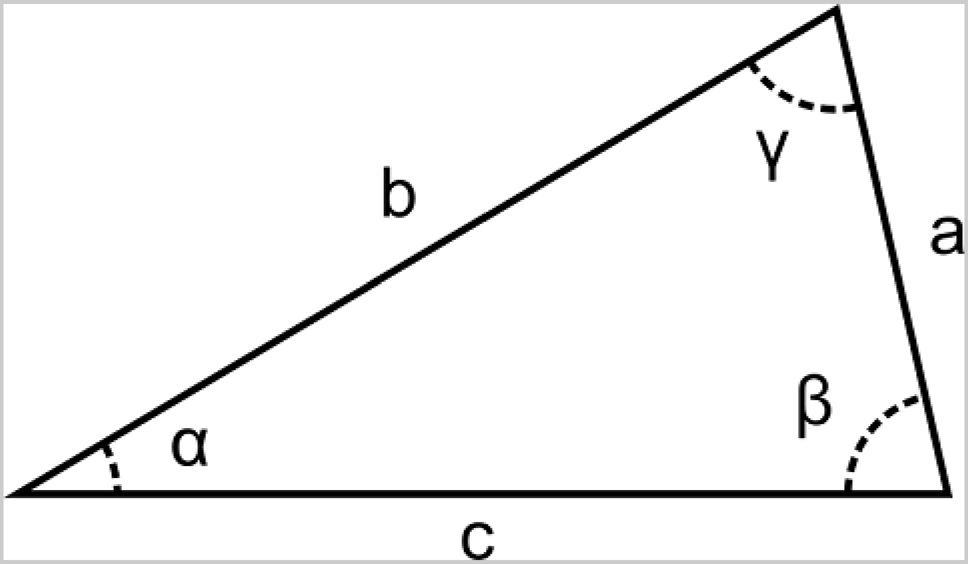


Figure 5.40 Triangle with sides and angles marked for the law of cosines.

$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos()$

The Pythagorean theorem is a special case of the law of cosines for =90°: $c^2 = a^2 + b^2$.

This gives us $h^2 = x^2 + y^2 = l_1^2 + l_2^2 - 2 \cdot l_1 \cdot l_2 \cdot \cos(\gamma)$. Solving this equation for γ results in $\gamma = \cos^{-1}((l_1^2 + l_2^2 - x^2 - y^2) / (2 \cdot l_1 \cdot l_2))$. We can get j_2 from $\gamma + -j_2 = 180°$, because γ and $-j_2$ are complementary angles on the line through l_1, thus $j_2 = \gamma - 180°$. We use the law of cosines again, but this time with angle α. The side opposite of α is l_2, thus $l_2^2 = h^2 + l_1^2 - 2 \cdot l_1 \cdot h \cdot \cos(\alpha)$. Solving for α and substituting h, we get $\alpha = \cos^{-1}((x^2 + y^2 + l_1^2 - l_2^2) / (2 \cdot l_1 \cdot \mathrm{sqrt}(x^2 + y^2)))$. Finally, we relate j_1 with α and β, $j_1 = \alpha + \beta$, and express β in terms of x and y, $\beta = \tan^{-1}(y/x)$, which gives us j_1.

This IK solution also requires us to check that the calculated **q** = (j_1, j_2) is within joint limits as well as to verify that **q** actually results in the desired pose. As long as the input to all mathematical functions is valid[235] and we do not perform other invalid operations, such as dividing by zero, the IK formulas themselves will always produce values - albeit potentially wrong values.[236]

Figure 5.39(b) illustrates that there is another IK solution for the EE position (not pose). We can apply the same geometric analysis to find it. The only difference to the previous solution is $j_2 = 180° - \gamma$ (instead of $\gamma - 180°$) and $j_1 = \beta - \alpha$ (instead of $\alpha + \beta$). More important is the insight that it is usually not obvious whether we have identified all possible IK solutions. Furthermore, as discussed in section 5.5 IK can have no solution, one unique solution or multiple solutions (finitely and infinitely many).

We could also apply the geometric approach to the 2 DoF PR robot (figure 5.34). However, this is more an exercise to train your new IK skills rather than teaching you something new about IK. Going from 2D to 3D makes things more difficult, but it is - surprisingly - not much different from the 2D case. *If* it is possible to find a closed-form analytic solution by geometric analysis, the process is similar to what we just discussed. It can take months, even years of studying a particular type of kinematic to find an IK solution in this manner. Actually, one reason you frequently see the same types of robot kinematics is that there are known analytic kinematic solutions for them, which can then be parametrized for concrete robots.

You might wonder whether solving IK necessitates knowing many geometric laws, identities, rules, theorems, propositions, lemmas, etc. Geometry definitively is an important foundation in robotics. It is always an advantage to have good knowledge about this topic. However, there are other approaches to solve IK than the geometric approach we have taken here. We will discuss other analytical and numerical approaches in section 5.9 below. Before looking at the IK of mobile robots, let's look at the abstract IK API.

The abstract API for position IK is

```
joint_positions_set = robot.position_inverse_kinematics(pose)
```

with `joint_positions_set` being a set of joint positions, i.e. a set of generalized joint position vectors (**q**). Depending on the number of IK solutions, the set can be empty, contain one element or multiple elements. We ignore the case of an infinite number of solutions for now. Furthermore, we will later on extend this basic abstract API by a second parameter to `position_inverse_kinematics()` that allows to ignore a subset of pose parameters for IK calculation. We continue with inverse kinematics for mobile robots.

The differential drive robot from figure 5.35 continues as the example for driving mobile robots. We already discussed in detail the reason for focusing on velocity kinematics when it comes to mobile robots. For non-holonomic drive mechanisms, velocity inverse kinematics brings up an additional topic: path planning. *Path planning* or *trajectory planning* computes a sequence of valid (reachable) poses that move the robot to a given target pose.[237] Path planning is closely related to kinematics, but we consider it to be a higher level robot functionality build on top of kinematics.[238] Trajectory planning for manipulators is the focus of chapter 10. Chapter 11 discusses this topic in detail for mobile robots.

I bring up path planning here because IK for non-holonomic mobile robots only provides very limited capabilities without path planning.[239] You will often find simple forms of path planning, planning in an obstacle free environment, discussed along with IK. In the quest for a coherent account of all robot types, structured along robot software components and subsystems, we will only discuss the IK part here and keep all path planning related topics for dedicated chapters.

Having clarified that we will *not* do path planning under the heading of inverse kinematics, what can diff drive velocity IK actually compute? The input is a 2D twist ${}^{w}\mathbf{V} = ({}^{w}\theta', {}^{w}x', {}^{w}y')$ and the output are the joint velocities $\mathbf{q}' = (j_1', j_2')$ of the robot's wheels. As the diff drive is non-holonomic, there are constraints on the reachable twists.[240] For example, the intended twist, specified in the robot frame, must always have ${}^{r}y' = 0$, because it is not possible to move sideways (along ${}^{r}\hat{\mathbf{y}}$ axis). The only twists possible are determined by the curves that can be created by an instantaneous center of curvature (ICC). Since the ICC is always a point on the axis through the two wheels, these curves are circular segments, see figure 5.41.

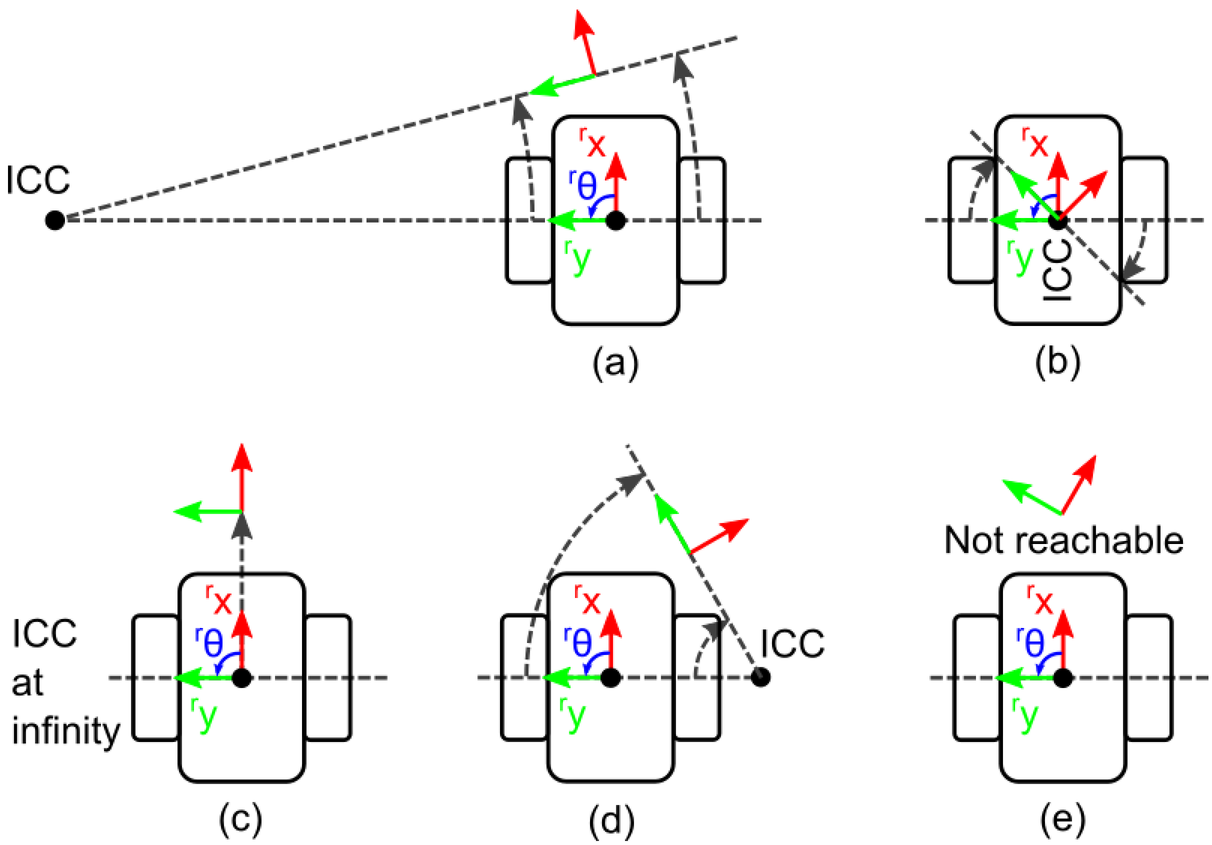


Figure 5.41 Four possible twists for a diff drive robot (a)-(d) and one unreachable one (e). The twist is illustrated as a "small" motion (short time t).

We again use the straightforward relation between scalar linear wheel velocities v_i and angular wheel velocities j_i' via the wheel size s, $v_i = s \cdot j_i'$, and work with v_i. Alternatively to the FK derivation performed in world coordinates in the previous section, we could also derive the diff drive FK in the robot frame and used coordinate transformations to go to the world frame.[241] Using the same definitions as before, the alternative formulation can be written as

$$ {}^{r}\mathbf{V} = \begin{bmatrix} {}^{r}\theta' \\ {}^{r}x' \\ {}^{r}y' \end{bmatrix} = \begin{bmatrix} (v_r - v_l)/l \\ (v_r + v_l)/2 \\ 0 \end{bmatrix} $$

Rearranging the formulas for ${}^{r}\theta'$ and ${}^{r}x'$, we get expressions for v_l and v_r that only depend on ${}^{r}\mathbf{V}$:

$$ v_r = {}^{r}x' + \frac{l}{2} \cdot {}^{r}\theta' $$
$$ v_l = {}^{r}x' - \frac{l}{2} \cdot {}^{r}\theta' $$

A sanity check for motion straight ahead (${}^{r}\theta' = 0$), $v_r = v_l$, and rotation in place (${}^{r}x' = 0$), $v_r - v_l = 0$ agrees with this IK solution.

When transforming a twist in world frame ${}^{w}\mathbf{V}$ into the robot frame twist ${}^{r}\mathbf{V}$, the FK formula makes it clear that the resulting ${}^{r}y'$ must be zero. This condition ensures that the results calculated by the IK formula are actually valid. Thus, unreachable twists can be detected by verifying ${}^{r}y'$ is zero and also checking that the calculated wheel joint velocities are within the joints' (velocity) limits. Let's conclude this introductory IK section with quadrotor kinematics as an example of aerial mobile robots.

The quadrotor, cf. figure 5.37, has a 3D task space with 6 PP, but only 4 DoF, hence it is non-holonomic. As explained before for aerial robots, the abstraction of dynamics (forces/torques) to kinematics (velocities) is more of theoretical than practical relevance. Regardless, we make the assumption that there is a known (momentaneous) stable relationship between rotor joint velocity and the resulting twist to the quadrotor's airframe. Then, the velocity IK provides joint velocities **q'** for a given twist **V** = (α', β', γ', x', y', z'). We assume the twist is given in the body frame, i.e. ${}^{b}\mathbf{V}$, and that the twist is reachable. An example of twists unreachable by quadrotors are movements in the ${}^{b}\hat{\mathbf{x}} - {}^{b}\hat{\mathbf{y}}$ plane. In the strict sense, quadrotors never move forward or sideways. Instead, they roll (γ) and pitch (β) so their ${}^{b}\hat{\mathbf{z}}$ axis is tilted with respect to the world frame ${}^{w}\hat{\mathbf{z}}$ axis and then move along this tilted axis.

Given a reachable ${}^{b}\mathbf{V}$, we want to calculate the corresponding linear velocity v_i along each rotor axis j_i. A difference between v_2 and v_4 results in a roll around x.[242] The origin of the body frame is in the middle of j_2 and j_4 as well as j_1 and j_3. Let l/2 be the distance between origin and the axis of each rotor joint j_i, i.e. l is the distance j_2 to j_4 and j_1 to j_3. Then the same formula results for ${}^{b}\gamma'$ as it did for θ' in the diff drive robot: ${}^{b}\gamma' = (v_4 - v_2) / l$. We get the same result for pitch around the y axis: ${}^{b}\beta' = (v_3 - v_1) / l$. The yaw around z axis is caused by the difference in angular velocity of each rotor pair, we assume this angular velocity ω_i to be proportional with the linear velocity through a factor k. Thus, we can write ${}^b\alpha' = k \cdot ((v_2 + v_4) - (v_1 + v_3)) / l$. Finally, the linear velocity along the z axis is the average rotor velocity: $z' = (v_1 + v_2 + v_3 + v_4) / 4$.

We could solve these FK equations for v_i - in a similar manner as we did for the diff drive - and then get our IK solution **q'** from these linear velocities. However, there is one crucial dissimilarity to ground-based mobile robots. The relation between joint velocities **q'** and task space velocities v_i/ω_i is *not* constant for aerial robots, it is *not* independent of the robot's pose (in the inertial world coordinate system). We cannot avoid to take dynamic quantities, especially gravity, into account. The resulting linear velocity v_i of a propeller spinning at some joint velocity j_i' very much depends on whether the propeller is pointing upward, sideways or downward in the world frame, i.e. with respect to gravity.[243] For this reason, we stop the discussion of flying mobile robots in this chapter on kinematics and continue it at the right place, in the following chapter on dynamics. We wrap up kinematics by looking at common representations and alternative IK solution methods.

5.8 Common Kinematics Representations

So far we have defined joints, links and the corresponding coordinate systems in an ad hoc manner. Even though this is not an issue or a shortcoming in itself, following established conventions is definitively a best practice. In addition, these standardized representations serve as an interface to readily available kinematics libraries. Furthermore, there often is a good reason why these conventions have become standards, e.g. they simplify calculations. There are two kinds of standardization. First are conventions for the definition of coordinate systems together with joint and link parameters. Second are specific file formats that can be used to record robot kinematics in a machine-readable manner, e.g. a particular XML, JSON or YAML schema. These are two independent considerations. One can use non-standard conventions but use a standardized file format to store the information. Similarly, one can adopt standardized conventions but capture them in a non-standard file format.

Let's start with a widely adopted standard to describe (serial) manipulators: *Denavit-Hartenberg (DH) parameters*. DH parameters describe the relative pose between two consecutive joint coordinate systems using 4 parameters. Although there are 6 PP in 3D space and hence also between two 3D frames, the conventions employed by DH eliminate a number of choices and thereby reduce the effectively required number of parameters to describe kinematic chains. I will only give you a summary of the method here, not a detailed account or step by step tutorial.[244] The main reason being that the robot software framework of choice for this book, ROS2, does not make use of them. This will become evident in a moment when discussing the URDF file format.

Referring to figure 5.42, we enumerate the joints with index i. Each joint i has a link i-1 as predecessor and a link i as successor. Link 0 is the base link. We relate a coordinate system $^i\mathbf{O}$ to each joint i in the following manner:

- The $^i z$ axis of $^i\mathbf{O}$ points along the motion axis of joint i.[245]
- The $^i x$ axis points along the common normal of ^{i-1}z and $^i z$ away from the previous joint: $^i x = {}^i z \times {}^{i-1}z$.
- The $^i y$ axis completes the right-handed coordinate system.[246]
- The $^i d$ parameter is the translation distance along ^{i-1}z axis from the the origin of $^{i-1}\mathbf{O}$ to the common normal with $^i z$. In case of a prismatic joint i, $^i d$ varies as the joint moves.
- The ^{i} parameter is the rotation angle around ^{i-1}z to align ^{i-1}x with $^i x$. In case of a rotary joint i, ^{i} varies as the joint moves.
- The $^i a$ parameter (sometimes called $^i r$) is the length of the common normal between ^{i-1}z and $^i z$.
- The ^{i} (alpha) parameter is the rotation angle around $^i x$ to align ^{i-1}z with $^i z$.

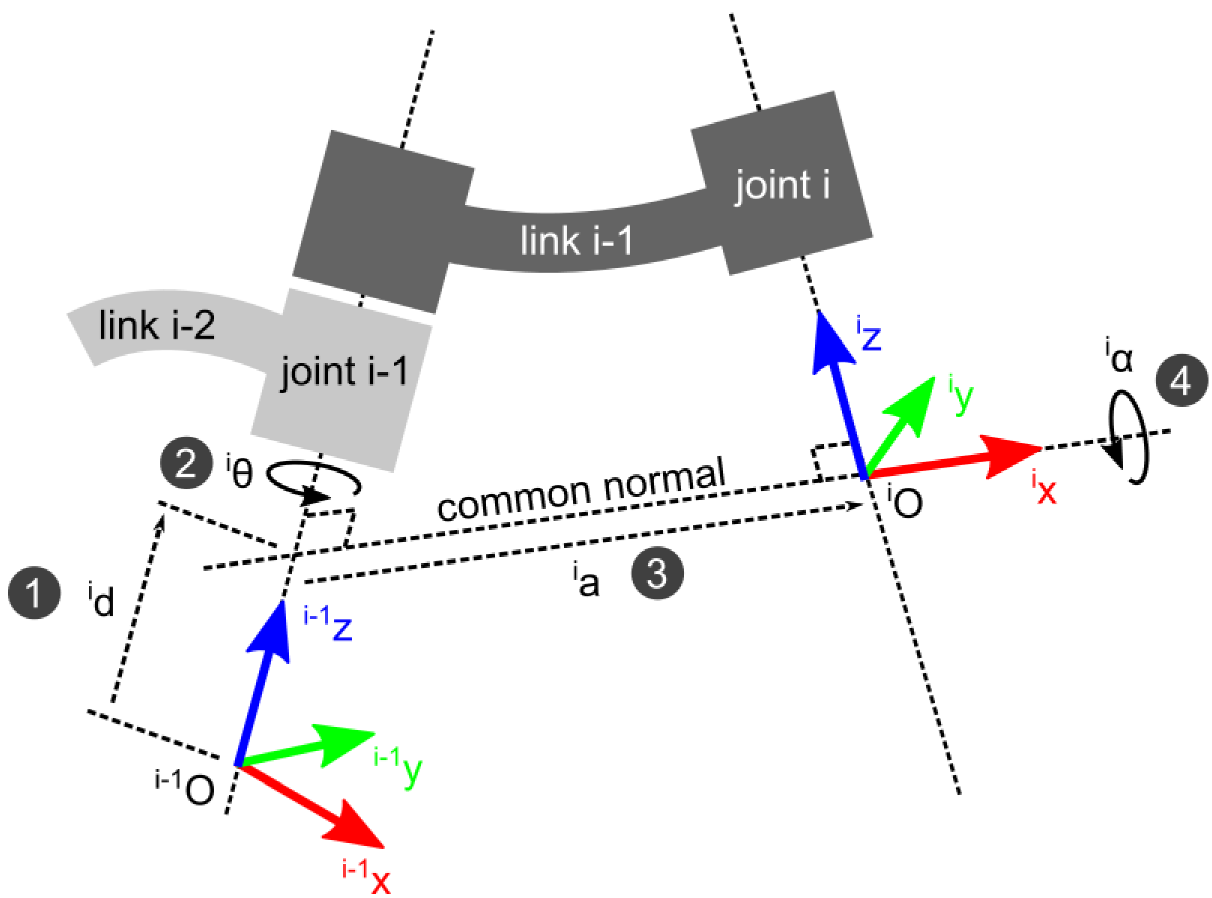


Figure 5.42 Illustration of Denavit-Hartenberg (DH) parameters.

Given these 4 DH parameters, we can create a homogeneous transformation matrix $^{i-1}\mathbf{T}_i$ that describes frame $^i\mathbf{O}$ in frame $^{i-1}\mathbf{O}$:

$$^{i-1}\mathbf{T}_i = \mathbf{Trans}_{z_{i-1}}(^i d)\mathbf{Rot}_{z_{i-1}}(^i \theta)\mathbf{Trans}_{x_i}(^i a)\mathbf{Rot}_{x_i}(^i \alpha)$$

Trans and **Rot** are just homogeneous matrices that translate, respectively rotate, along the axis given in the subscript by the amount specified as the parameter.[247]

Once we have created all **T**, we can chain them together from ${}^{0}\mathbf{T}_1$ to ${}^{n-1}\mathbf{T}_n$ to get the transform ${}^{0}\mathbf{T}_n$ from base link to end effector: ${}^{0}\mathbf{T}_n = {}^{0}\mathbf{T}_1 \cdot {}^{1}\mathbf{T}_2 \cdot \ldots \cdot {}^{n-1}\mathbf{T}_n$. This is also the forward kinematics solution with the ${}^{i}d$ (prismatic joint), respectively ${}^{i}\theta$ (rotary joint), making up the generalized coordinates **q** in joint space.

We now take a look at a popular file format for describing robots and their kinematics: the *Unified Robot Description Format (URDF)*. URDF is an XML schema for describing geometric, kinematic and dynamic properties of robots. It originates from the ROS ecosystem and is an integral part of it, but it is also used independently of ROS. I will introduce some of its XML elements here and further ones in the following two chapters (dynamics and control). When we discuss ROS2 in chapter 8, you will already know URDF.[248] I make the assumption you are already familiar with markup languages such as XML and understand XML concepts such as tags, elements and attributes.

The subset of URDF we are interested in right now consists of the following elements

```
<robot name="ROBOT_NAME">
  <link name="LINK1"> ... </link>
  <link name="LINK2"> ... </link>
  <link name="LINK3"> ... </link>

  <joint name="JOINT1"> ... </joint>
  <joint name="JOINT2"> ... </joint>
  <joint name="JOINT3"> ... </joint>
</robot>
```

The capitalized words, such as `ROBOT_NAME`, are placeholders and have to be replaced by actual values. Each `<link>` element has a distinct name and contains information on geometry (`<visual>`, `<collision>`) and dynamics (`<inertial>`). We will look at them in more detail later on. `<joint>` elements create the kinematic structure of the robot by connecting links. Typical elements in a `<joint>` are

```
<joint name="JOINT_NAME" type="TYPE">
   <origin xyz="X Y Z" rpy="ROLL PITCH YAW"/>
   <parent link="PARENT_LINK_NAME"/>
   <child link="CHILD_LINK_NAME"/>

   <axis xyz="AX AY AZ/">
   <limit effort="MAX_EFFORT"
          velocity="VELOCITY_LIMIT"
          lower="LOWER_JOINT_LIMIT"
          upper="UPPER_JOINT_LIMIT"/>
</joint>
```

The joint `JOINT_NAME` connects the two links `PARENT_LINK_NAME` and `CHILD_LINK_NAME`. Joint types (`TYPE`) of interest are `revolute` (= rotary with position limits), `continuous` (= rotary without position limits) and `prismatic` (= linear). The joint axis is defined by the `<axis>`

element in terms of a unit vector (AX, AY, AZ) in the joint frame. `LOWER_JOINT_LIMIT` and `UPPER_JOINT_LIMIT` specify position limits and `VELOCITY_LIMIT` specifies the maximum joint velocity. Last but not least, the joint's frame in terms of the parent frame is specified in the `<origin>` tag attributes (`X`, `Y`, `Z`, `YAW`, `PITCH`, `ROLL`) corresponding to $\mathbf{P} = (x, y, z, \alpha, \beta, \gamma)$.[249] Figure 5.43 illustrates the relation between parent link frame, joint frame and child link frame.

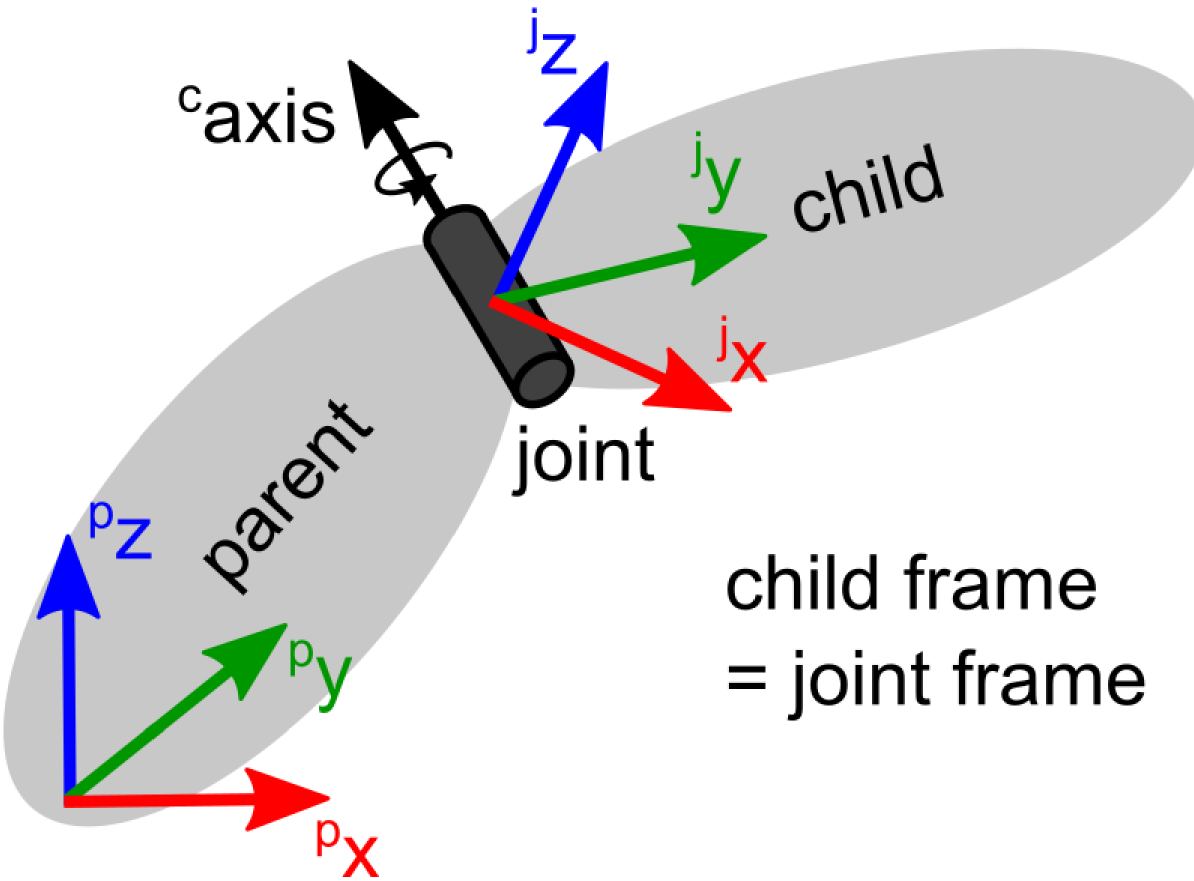


Figure 5.43 Illustration for Unified Robot Description Format (URDF) `<joint>` element.

A complete description of a robot in terms of `<link>` and `<joint>` elements provides all information required to calculate FK given the joint values **q**. As you will learn in the next section, when using numerical IK, the URDF also contains everything required for IK, too.

5.9 Analytical and Numerical Solutions

Inverse kinematics (IK) solutions can be found in different ways. These are commonly categorized into analytical approaches and numerical approaches.

The analytical *geometric approach* disregards existing forward kinematics (FK) descriptions and works directly with the geometric relations of the robot mechanism. It heavily relies on geometric identities and rules, such as trigonometry and the law of cosines. One can see it as a kind of first principles approach to solve IK. It therefore is a highly manual approach. However, if a solution is found, it is usually a closed-form solution that can be efficiently computed.

The analytical *algebraic approach* (or often simply analytical approach) starts from the standard FK transformation and tries to solve this equation for the joint coordinates. The initial equation is simply ${}^{base}\mathbf{T}_{ee}(\mathbf{q}) = {}^{base}\mathbf{P}_{ee}(\mathbf{p}, \mathbf{o})$ with generalized joint space coordinates **q** and the 6 PP variables for a 3D pose, position $\mathbf{p} = (x, y, z)$ and orientation $\mathbf{o} = (\alpha, \beta, \gamma)$. This equation has a 4x4 matrix on the left side, whose elements depend on **q**, and a 4x4 matrix on the right side, whose elements depend on **p** and **o**. Element-wise comparison provides 16 equations. Each equation is of the form $a_{i,j} = b_{i,j}$ with $a_{i,j}$ being the elements in ${}^{base}\mathbf{T}_{ee}(\mathbf{q})$ and $b_{i,j}$ those of ${}^{base}\mathbf{P}_{ee}(\mathbf{p}, \mathbf{o})$. Note that the elements in these matrices are usually not numbers, but rather equations, some of which depend on the coordinate variables. However, 4 of these equations are trivial, as the bottom row of both matrices is (0, 0, 0, 1).[250] Thus, there are actually only 12 equations. There is no universal step by step process to solve these equations for **q**. However, one helpful tool to generate additional equations, which might facilitate solving them, exploits the coordinate transformations in the FK. Assuming ${}^{base}\mathbf{T}_{ee}(\mathbf{q})$ is composed of n+1 transformations, each of them relating the joint frame j_i to the previous joint frame j_{i-1}, i.e. ${}^{base}\mathbf{T}_{ee}(\mathbf{q}) = {}^{base}\mathbf{T}_{j1} \cdot {}^{j1}\mathbf{T}_{j2}(q_1) \cdot \ldots \cdot {}^{jn}\mathbf{T}_{ee}(q_n)$. Then we can left-multiply both sides of the equation by ${}^{base}\mathbf{T}_{j1}{}^{-1}$, which cancels out the first term on the right side, which results in ${}^{base}\mathbf{T}_{j1}{}^{-1} \cdot {}^{base}\mathbf{T}_{ee}(\mathbf{q}) = {}^{j1}\mathbf{T}_{j2}(q_1) \cdot \ldots \cdot {}^{jn}\mathbf{T}_{ee}(q_n)$. This gives us a two new matrices set equal, i.e. 12 additional equations. One usually tries to solve one of these equations for a single variable, which can then also be plugged into other equations to solve them. We can repeat this and left-multiply by ${}^{j1}\mathbf{T}_{j2}(q_1)^{-1}$, moving another term from right to left, which provides us with additional equations. Furthermore, instead of left-multiplications (starting with ${}^{base}\mathbf{T}_{j1}$), we can also perform right-multiplications (starting with ${}^{jn}\mathbf{T}_{ee}(q_n)^{-1}$), which gives us further equations, the first one being ${}^{base}\mathbf{T}_{ee}(\mathbf{q}) \cdot {}^{jn}\mathbf{T}_{ee}(q_n)^{-1} = {}^{base}\mathbf{T}_{j1} \cdot {}^{j1}\mathbf{T}_{j2}(q_1) \cdot \ldots \cdot {}^{jn-1}\mathbf{T}_{n}(q_{n-1})$. Compared to the geometric approach, the algebraic one is more standardized and "mechanical". Computer algebra systems (CAS) can support us in manipulating and solving the equations. Still, this analytical approach has similar advantages and disadvantages.

The *numerical approach* uses a given FK function to calculate IK solutions by means of numerical root-finding algorithms. When making use of the numerical approach, our goal is *not* to find a (closed-form) formula with which we can calculate **q** for a given pose. The goal is to numerically calculate each IK solution using the FK function. The FK is seen as a generic function and we rely on general-purpose methods to find parameters to this function (for IK: **q**) that produce a certain function output (for IK: **P**). These numerical methods are usually known as root-finding algorithms. We can rewrite f(x) = y as f(x) - y = 0 and find the roots of h(x) = f(x) - y, i.e. the zero values of this function (h(x) = 0).

One such root-finding algorithm is *Newton's method*, aka *Newton-Raphson method*. Applying this method to our kinematics function requires multivariable calculus (aka multivariate calculus), because we are dealing with functions of several variables. In particular, we need to make use of concepts such as partial derivatives, gradients and Jacobian matrices. Although these are "only" extensions of what you already learned about differentiation in this chapter to vector-valued functions of several variables, it takes quite some room to introduce them properly. Therefore, I'll just illustrate the basic method for a scalar function of a single variable here. It will be clear how this translates to the multivariant method for IK functions, once we have discussed multivariable calculus in the next chapters.

Newton's method is an iterative algorithm to find the root of a function f(x). In other words, it finds a function parameter x for which the function value is zero, i.e. f(x) = 0. Starting from an initial guess x_0, a new value x_1 is computed with $f(x_1)$ being closer to zero than $f(x_0)$. This is iterated from x_n to x_{n+1} until some $f(x_n)$ is zero or rather is very close to zero. The formula for the iteration step from x_n to x_{n+1} is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Figure 5.44 helps to get a geometric understanding of this formula.

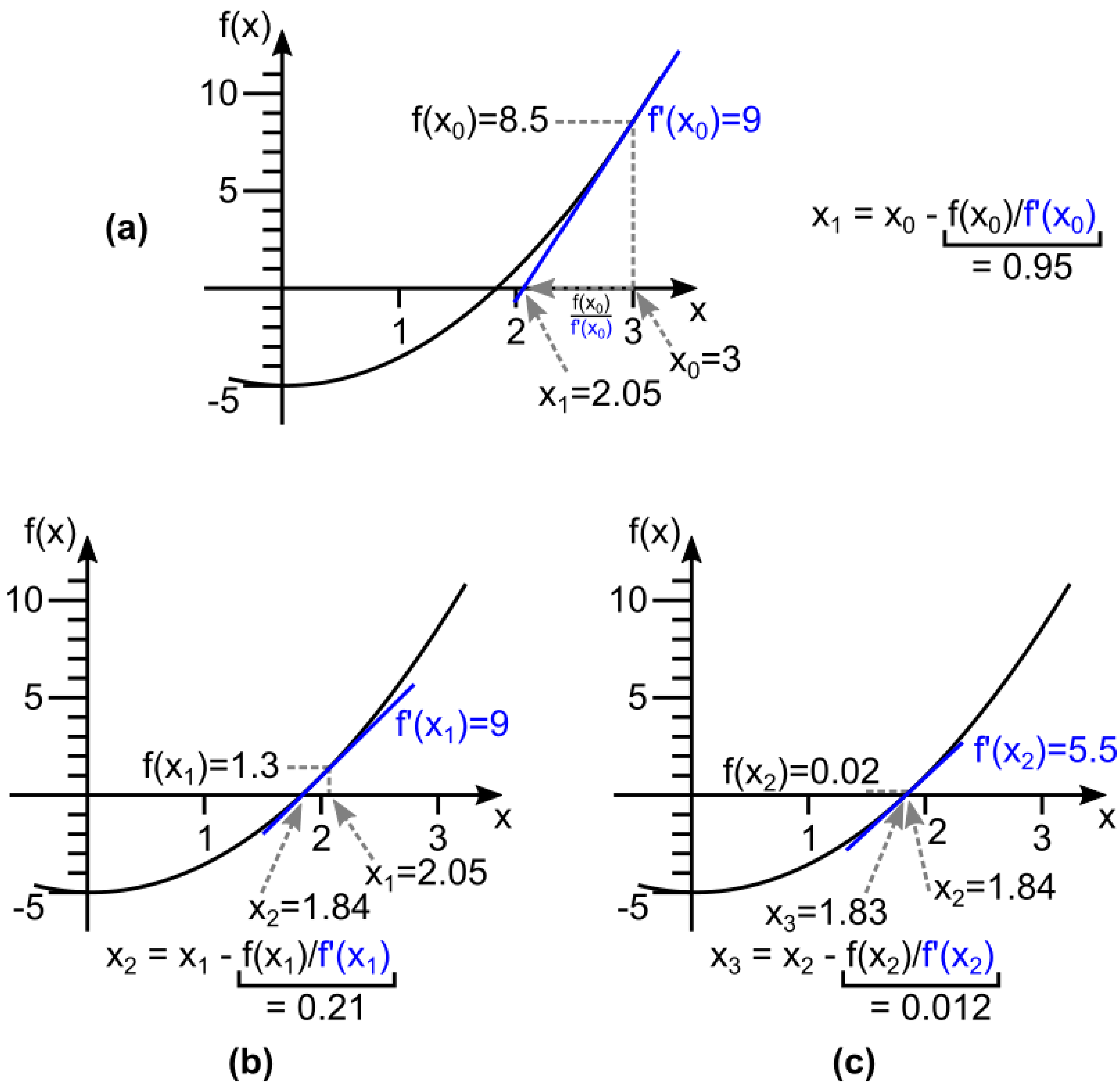


Figure 5.44 Illustration of Newton's method to find a root of $f(x) = 1.5x^2 - 5$.

The formula calculates the function value at x_n, $f(x_n)$, and divides it by the derivative value at x_n, $f'(x_n)$. This fraction is then subtracted from the current x_n in order to get the improved x_{n+1}. The derivative of a scalar function can be seen as the slope of the tangent line at each point. $f(x_n)/f'(x_n)$ is simply the width of the triangle with slope $f'(x_n)$ and height $f(x_n)$. The overall expression, $x_n - f(x_n)/f'(x_n)$, hence calculates the intersection of the tangent line at $f(x_n)$ with the x axis.

Note that the series of x_n values does not always converge to the root from one side as in figure 5.44. It can also go past the root and approach it from the other side. The question whether, under which conditions and how fast this iterative method converges to a root value is an important one, but we will not discuss it here. For now it is sufficient to know that the method is guaranteed to converge for certain categories of functions as well as for certain parts of a function given a close enough initial guess. Unfortunately, the IK functions often do not have the properties that guarantee convergence. Still, Newton's method usually finds a solution when starting from multiple random initial values - which frequently is an acceptable workaround.

I leave it as an exercise to you to translate the above description of Newton's method into Python code (see the following section). The simplicity and elegance of Newton's method makes it a good introduction for root-finding (and optimization, as you will learn later). However, improved algorithms have been derived from it. Thus, as I will not get tired to emphasize, you should understand the methods, but use existing library implementations whenever possible. In case of root-finding in Python, `scipy.optimize.root()` is a good candidate:

```
from scipy import optimize

def f(x):
  return 1.5 * x**2 - 5

x0 = 3.0  # initial guess
sol = optimize.root(f, x0)

print(sol.x)
# returns (rounded): 1.83
```

The obvious advantage of numerical IK is that it works "out-of-the-box" for any kinematic. It only requires an FK function - and an FK function can be automatically derived from the robot model.[251] But there are also a number of drawbacks. First, numerical IK is computationally much more expensive than calculating IK based on an analytical solution. Although computers have become fast, IK often has to be calculated every millisecond (1000 times per second) or many thousands IK calls are made during path planning. Second, numerical IK does not always find a solution, even though one exists, i.e. numerical IK can be incomplete. Third, numerical IK is not stable/deterministic. Different joint values might be returned for the same input pose.

Let's finish this chapter on kinematics with a few words on inverse kinematics in real-world robot software. Various IK libraries exist that employ one of these approaches or a combination thereof and they only require a kinematic description of the robot, e.g. a URDF file, and can automatically generate very good IK solutions. Futhermore, for many classes of robots, roboticists have come up with analytical IK solutions. These kinematic models only have to be instantiated with the respective parameters describing the actual robot conforming to their basic structure. Nonetheless, even though you can and should make use of these readily available robot software building blocks, you still need to understand how to use them properly. Having read this chapter has brought you a long way towards this essential understanding when it comes to kinematics. The next section provides a few exercises to test and strengthen your newly acquired skills.

5.10 Exercises on Kinematics

SIDEBAR

Exercise 5.1: Calculate the velocity of a robot, whose position is defined by p(t) = t^2 - 2t, at times t = 0, 1, 2, 3, 4.

Solution 5.1:

```
def p(t):
  return t**2 - 2 * t

def numerical_derivative(f, x, delta=1e-3):
  return (f(x + delta) - f(x)) / delta

for t in range (0, 5):
  print(f'f(t={t})={numerical_derivative(p, t)}')

# prints (rounded):
# f(t=0)=-2; f(t=1)=0; f(t=2)=2; f(t=3)=4; f(t=4)=6
```

The time derivative of position is velocity, i.e. v(t) = p'(t). (Naive) numerical differentiation, `numerical_derivative()` is used to calculate the derivative of `p()`. Then this derivative function is evaluated at the times t of interest.

SIDEBAR

Exercise 5.2: Calculate the location p(t) of a robot each second in time interval [0, 10], given its acceleration function a(t), which is defined as

- **In time interval [0, 3): a(t) = t**
- **In [3, 5): a(t) = 3**
- **In [5, 8): a(t) = 13 - 2t**
- **In [8, 10]: a(t) = 0**

At t=0 the robot is standing still at position 0, i.e. v(0)=0 and p(0)=0.

Solution 5.2:

First write a(t) as a Python function `a()`:

```
import numpy as np

def a(t):
  if t < 3:
    return t
  elif t < 5:
    return 3
  elif t < 8:
    return 13 - 2 * t
  else:
    return 0
```

Next write a function for naive numerical integration `numerical_integral()`:

```
def numerical_integral(f, a, b, delta=1e-3):
  sum = 0
  x = a
  while x <= b:
    sum += f(x) * delta
    x += delta
  return sum
```

We need to perform integration twice to go from acceleration to position, cf. figure 5.16, i.e. a(t) to v(t) to p(t). However, the `numerical_integral()` only provides us with a definite integral, a number, not an indefinite integral, a function. In order to have a representation of the velocity function v(t) that we can integrate, we create an approximation of v(t). The most simple function approximation is a table with the value of the function at discrete points. For any given function parameter x, the tabular approximation of f(x), simply returns the table entry for the next smaller value of x contained in the table. A plot of such a tabular function approximation looks like a step function (like a staircase). The code below creates this table (array) `vt` for v(t) in the range `t_min` to `t_max` with entries spaced out evenly at a distance of `delta`.

```
delta = 0.01
t_min = 0
t_max = 10
times = np.arange(t_min, t_max + delta, delta)
element_count = len(times)

vt = np.empty(element_count)
i = 0
for t in times:
  vt[i] = numerical_integral(a, 0, t, delta)
  i += 1

def v(t):
  return vt[int(t / delta)]
```

Given `v()`, its numerical integral can be calculated to get `p()`.

```
pt = np.empty(element_count)
i = 0
for t in times:
  pt[i] = numerical_integral(v, 0, t, delta)
  i += 1

def p(t):
  return pt[int(t / delta)]

for t in range(0, 11):
  print(f'p(t={t})={p(t)}')

# prints (rounded):
# p(t=0)=0.00; p(t=1)=0.16; p(t=2)=1.31; p(t=3)=4.50; p(t=4)=10.50;
# p(t=5)=19.50; p(t=6)=31.17; p(t=7)=43.83; p(t=8)=55.50; p(t=9)=65.99;
# p(t=10)=76.47;
```

Note that the tabular function approximation `vt` and `pt` is calculated in a very inefficient manner. Calculation of the definitive integral [0, 2] includes calculating the definitive integral [0, 1]. Calculating it for [0, 3] includes calculating [0, 2], thus again [0, 1], and so forth. An efficient implementation would reuse the already calculated integrals for subintervals, similarly to calculating a cumulative sum (and dynamic programming).

SIDEBAR

Exercise 5.3: Given the following excerpt from an URDF robot model, create a forward kinematics (FK) function and calculate the end effector pose (in robot base frame) ${}^{base}P_{ee}$ for joint values q = (10°, 20°, 0.2).

```
<robot name="ex_5_3_rob">
  <link name="base"> ... </link>
  <link name="link1"> ... </link>
  <link name="link2"> ... </link>
  <link name="link3"> ... </link>
  <link name="ee"> ... </link>

  <joint name="joint1" type="revolute">
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <parent link="base"/>
      <child link="link1"/>
      <axis xyz="0 0 1"/>
  </joint>

  <joint name="joint2" type="revolute">
      <origin xyz="0.53 0 0.1" rpy="0 0 0"/>
      <parent link="link1"/>
      <child link="link2"/>
      <axis xyz="0 0 1"/>
  </joint>

  <joint name="joint3" type="prismatic">
      <origin xyz="0.28 0 0.6" rpy="3.14 0 0"/>
      <parent link="link2"/>
      <child link="link3"/>
      <axis xyz="0 0 1"/>
  </joint>

  <joint name="joint4" type="fixed">
      <origin xyz="0 0 0.7" rpy="0 0 0"/>
      <parent link="link3"/>
      <child link="ee"/>
  </joint>
</robot>
```

The robot is visualized in figure 5.45.

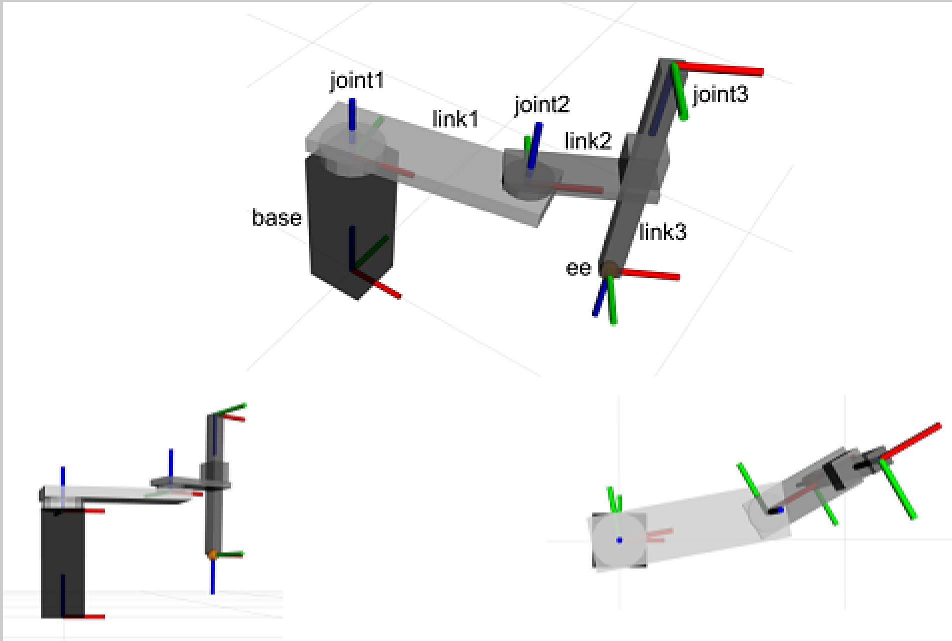


Figure 5.45 Visualization of the 3 DoF RRL robot described by the exercise's URDF.

Use `translation_matrix` and `euler_matrix` from `transfomations.py` to model the parametrized joint transformations. Alternatively, see appendix B.1.3 for the required parametrized transformation matrices / operators, in particular, $\text{Trans}_a(b)$ and $\text{Rot}_c(d)$.

Solution 5.3:

```
from math import radians
from transformations import *

def fk(q):
  base_T_link1 = (translation_matrix([0, 0, 0.5])
                  @ euler_matrix(0, 0, q[0], axes='sxyz'))
  link1_T_link2 = (translation_matrix([0.53, 0, 0.1])
                  @ euler_matrix(0, 0, q[1], axes='sxyz'))
  link2_T_link3 = (translation_matrix([0.28, 0, 0.6 + q[2]])
                  @ euler_matrix(3.14, 0, 0, axes='sxyz'))
  link3_T_ee = (translation_matrix([0, 0, 0.7])
                  @ euler_matrix(0, 0, 0, axes='sxyz'))
  return base_T_link1 @ link1_T_link2 @ link2_T_link3 @ link3_T_ee

print(fk([radians(10), radians(20), 0.2]))
# rounded:
# [[ 0.87  0.50  0.00  0.77]
#  [ 0.50 -0.87  0.00  0.23]
#  [ 0.00  0.00 -1.00  0.70]
#  [ 0.00  0.00  0.00  1.00]]
```

The RRL kinematic structure of this example robot is known as SCARA. Note that the solution does not check for joint limits (nor for self-collisions).

SIDEBAR

Exercise 5.4: Use the information contained in the following forward kinematic equation (${}^{base}T_{ee}$) to create an inverse kinematics (IK) function. Follow the algebraic approach. Use this IK to calculate joint values q = (q_1, q_2, q_3) that put the end effector at pose P = (x, y,) = (0.9787, 0.198, 40°). Although the FK is given as a 3D transform, which in general has 6 pose parameters, the manipulator structure is planar, i.e. all reachable poses are on a 2D plane.

$$
{}^{\text{base}}\mathbf{T}_{\text{ee}}(\mathbf{q}) = \begin{bmatrix} \cos(q_1+q_2+q_3) & -\sin(q_1+q_2+q_3) & 0 & l_1\cos(q_1)+l_2\cos(q_1+q_2)+l_3\cos(q_1+q_2+q_3) \\ \sin(q_1+q_2+q_3) & \cos(q_1+q_2+q_3) & 0 & l_1\sin(q_1)+l_2\sin(q_1+q_2)+l_3\sin(q_1+q_2+q_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

with l_1 = 0.5, l_2 = 0.4, l_3 = 0.15.

Figure 5.46 visualizes the robot matching the above FK structure. The figure is for illustration only. It is not needed for the exercise, because we are using the algebraic analytical IK approach.

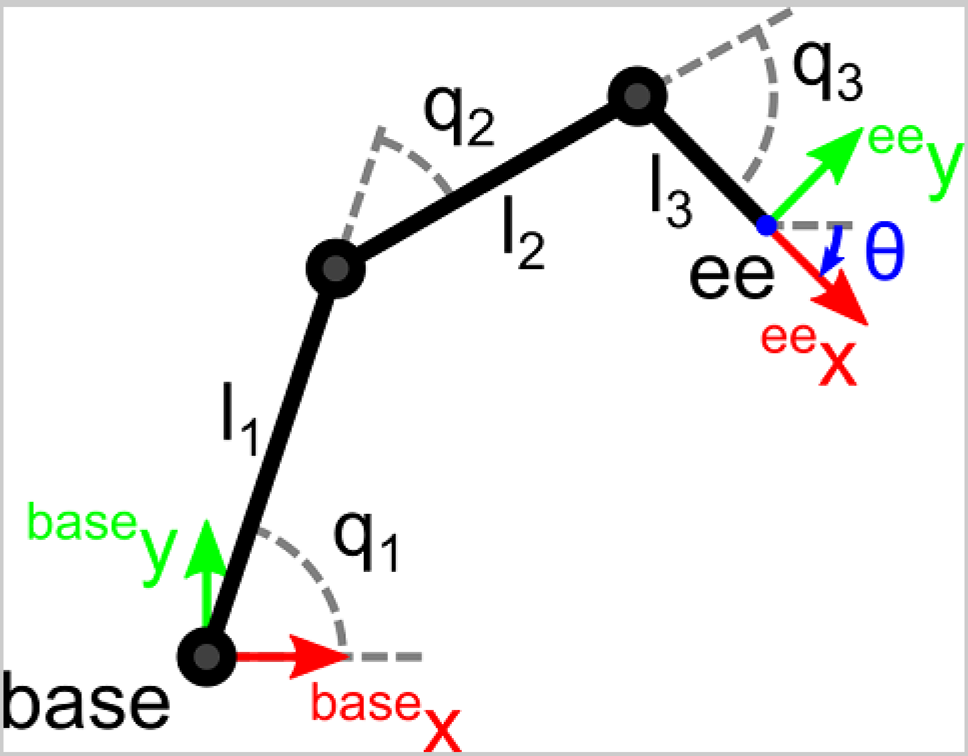


Figure 5.46 Visualization of the 2D 3 DoF RRR robot described by FK in this exercise.

This exercise requires quite some manipulating of equations. If you get stuck, have a look at the solution below. The point of the exercise is not to train solving mathematical equations, rather it is about understanding the algebraic approach to IK. Put differently, I want you to be able to come up with the equations, but you are not expected to be able to solve them on your own.

For the same reason, I strongly encourage you to make use of the following partial solution. Given a, b, d and e, the equation system

$$
\begin{aligned}
a &= b\cos(\alpha_1) + d\cos(\alpha_1 + \alpha_2) \\
e &= b\sin(\alpha_1) + d\sin(\alpha_1 + \alpha_2)
\end{aligned}
$$

has these solutions for $_1$ and $_2$

$$
\begin{aligned}
u &= a^2 + e^2 \\
c_2 &= \frac{u - b^2 - d^2}{2bd} \\
s_2 &= \pm\sqrt{1 - c_2^2} \\
s_1 &= \frac{(b + d\,c_2)e - d\,s_2 a}{u} \\
c_1 &= \frac{(b + d\,c_2)a + d\,s_2 e}{u} \\
\alpha_1 &= \operatorname{atan2}(s_1,\ c_1) \\
\alpha_2 &= \operatorname{atan2}(s_2,\ c_2)
\end{aligned}
$$

A solution exists if c_2^2 is less or equal to 1. In general there are two solutions, created by choosing the sign in the s_2 formula.

This is not a well-known system of equations. It also has no special relation to (inverse) kinematics. It just happens to come up when deriving an analytic inverse kinematics solution for the example kinematics using the algebraic method. There is no point in spending a lot of time and pages to solve this equation system step by step here. Thus, the solution is provided instead of a derivation.

Solution 5.4:

The input is P = (x, y,) and we want to calculate a set of corresponding joint values q = (q_1, q_2, q_3). In homogeneous coordinates, the input is

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

Following the algebraic method, we set T(P) equal to ${}^{base}T_{ee}$(q):

$$
\mathbf{T}(x,\ y,\ \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & x \\ \sin(\theta) & \cos(\theta) & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(q_1 + q_2 + q_3) & -\sin(q_1 + q_2 + q_3) & 0 & l_1\cos(q_1) + l_2\cos(q_1 + q_2) + l_3\cos(q_1 + q_2 + q_3) \\ \sin(q_1 + q_2 + q_3) & \cos(q_1 + q_2 + q_3) & 0 & l_1\sin(q_1) + l_2\sin(q_1 + q_2) + l_3\sin(q_1 + q_2 + q_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = {}^{\text{base}}\mathbf{T}_{\text{ee}}(\mathbf{q})
$$

This gives us 16 equations. One for each entry of the 4x4 homogenous matrices. The equations for the bottom row are always trivial, i.e. 0 = 0 or 1 = 1. The remaining 12 equations are the interesting ones, in particular:

$$\cos(\theta) = \cos(q_1 + q_2 + q_3)$$
$$x = l_1\cos(q_1) + l_2\cos(q_1 + q_2) + l_3\cos(q_1 + q_2 + q_3)$$
$$y = l_1\sin(q_1) + l_2\sin(q_1 + q_2) + l_3\sin(q_1 + q_2 + q_3)$$

From the first equation, we can directly derive = $q_1 + q_2 + q_3$. The remaining task is to solve the two non-linear equations for x and y. This is far from trivial, especially if you have never solved similar equations before. Substituting $q_1 + q_2 + q_3$ with , we can move all known variables to the left:[252]

$$x - l_3\cos(\theta) = l_1\cos(q_1) + l_2\cos(q_1 + q_2)$$
$$y - l_3\sin(\theta) = l_1\sin(q_1) + l_2\sin(q_1 + q_2)$$

In order to make it easier to see the connection to the formula given above, we substitute a = x - $l_3$cos() and e = y - $l_3$sin():

$$a = l_1\cos(q_1) + l_2\cos(q_1 + q_2)$$
$$e = l_1\sin(q_1) + l_2\sin(q_1 + q_2)$$

Using the solution provided above for this system of equations, we get q_1 and q_2. Finally, q_3 follows from = $q_1 + q_2 + q_3$, i.e. q_3 = - ($q_1 + q_2$). Converting these formulas to Python, gives us the IK and hence the joint values for the target pose:

```
from math import radians, cos, sin, atan2
import numpy as np
from transformations import *

def ik(P, sign=1):
  l1 = 0.5
  l2 = 0.4
  l3 = 0.15

  x = P[0]
  y = P[1]
  theta = P[2]

  a = x - l3 * cos(theta)
  e = y - l3 * sin(theta)
  u = a**2 + e**2
  c2 = (u - l1**2 - l2**2) / (2 * l1 * l2)
  if c2**2 > 1:  # No solution
    return None
  s2 = sign * sqrt(1 - c2**2)
  s1 = ((l1 + l2 * c2) * e - l2 * s2 * a) / u
  c1 = ((l1 + l2 * c2) * a + l2 * s2 * e) / u

  q1 = atan2(s1, c1)
  q2 = atan2(s2, c2)
  q3 = theta - (q1 + q2)
  return q1, q2, q3

P = [0.9787, 0.198, radians(40)]
q = ik(P)
print([degrees(qi) for qi in q])
# rounded: (-6.6, 30, 16.6)

q_alt = ik(P, sign=-1)
print([degrees(qi) for qi in q_alt])
# rounded: (20, -30, 50)
```

SIDEBAR

Exercise 5.5: Implement Newton's method and find all roots of f(x) = $x^3 + 3x^2 - 22x - 24$ in the interval [-10, 10].

Solution 5.5:

```
epsilon = 1e-4  # iteration stopping criterion
h = 1e-3  # difference quotient for numerical derivation

def newton(f, x0):
  x = x0
  delta = 1
  while abs(delta) > epsilon:
    df = (f(x + h) - f(x)) / h
    delta = f(x)/df
    print(f"x={x} f({x})={f(x)} f'({x})={df} f({x})/f'({x})={delta}")
    x = x - delta
  return x

def f(x):
  return x**3 + 3 * x**2 - 22 * x - 24

roots = []  # unique roots found
for x0 in range(-10, 11):
  root = newton(f, x0)
  if all(abs(r - root) > epsilon for r in roots):
    roots.append(root)

print(roots)
# prints (rounded): [-6.0, -1.0, 4.0]
```

The implementation of `newton()` follows the description in section 5.9. f'(x) is calculated as naive numerical derivative f'(x) = (f(x+h) - f(x))/h with h as finite difference, i.e. a "small number". The stopping criterion for the iteration is the change of x_n falling below a certain threshold (here epsilon /). If the method does not converge, this becomes an infinite loop. A production-grade implementation should handle such cases more gracefully. In order to find all roots, Newton's method is started with different initial guesses (x_0) and the unique solutions found are collected in `roots`.

5.11 Summary

- Kinematics deals with objects moving in space, i.e. the change of an object's pose over time. Kinematics is not concerned with how this motion comes about in terms of the acting forces/torques.
- A trajectory is a path (a sequence of poses) with temporal information.
- Velocity describes the rate of change in an object's pose.
- Acceleration describes the rate of change in an object's velocity.
- A pose describes both position and orientation. Likewise, spatial velocities (and spatial accelerations) consist of linear velocities (accelerations) and angular velocities (accelerations).
- The derivate of a function is the instantaneous rate of change of the function at each point. Intuitively, the derivative of a function at a given point is the tangent line at this point. The derivative is also a function.
- The process of calculating derivatives is called differentiation.
- Higher-order derivatives are derivatives of derivatives. For example, acceleration is the second-order derivative of position - velocity being the first-order derivative of position.
- Integrals can be seen as inverse operation of derivatives, i.e. antiderivatives. Integrals accumulate all momentaneous changes of a variable within an interval into the resulting overall change of the variable. An intuitive view of integrals is that they provide the area under the curve of a function.
- The process of calculating integrals is called integration.
- Indefinite integrals are functions whose derivative is another function. Definite integrals calculate a value.
- Understanding the relation between position, velocity, acceleration and jerk with respect to time derivatives and time integrals is essential in robotics.
- Task space (aka Cartesian space) describes a robot's configuration in terms of poses (and their derivatives) in Cartesian coordinates.
- Joint space describes a robot's configuration in terms of joint positions (and their derivatives) in generalized coordinates.
- Forward kinematics (FK) translates from joint space to task space, i.e. from joint values to poses.
- Inverse kinematics (IK) translates from task space to joint space, i.e. from poses to joint values.
- For a given pose, inverse kinematics can have no solution, one solution or multiple solutions (incl. infinitely many).
- If there are multiple IK solutions, they are referred to as configurations.
- The workspace is the task space volume, which is reachable by the robot.
- A robot is kinematically deficient or non-holonomic, if it has less degrees of freedom (DoF) than pose parameters (PP), i.e. DoF < PP. A robot is holonomic if DoF = PP. A robot is kinematically redundant if DoF > PP.
- A robot is underactuated if it has less actuators than DoF. A robot is overactuated if it has more than one actuator per DoF.
- Geometry, especially trigonometry, is an essential tool to create FK and IK functions.
- Joints can be seen as imposing motion constraints between two rigid bodies (e.g. the attached links). Joints reduce the DoF of each body from 6 to a smaller number, e.g. to 1

DoF for rotary and prismatic joints.

- In serial kinematics there is only a single joint/link path from base to end effector. Serial kinematics are also referred to as kinematic chains. Parallel kinematics contain at least one loop.
- FK of serial kinematics can be composed and calculated one joint value / one coordinate frame at a time from base to end effector. IK cannot be decomposed in a similar manner. IK requires to calculate joint values simultaneously.
- In contrast to manipulators, there is no stable relation between joint positions (joint space) and robot pose (task space) in mobile robots. Hence, the focus is on velocity kinematics that map joint velocities to task space velocities.
- For mobile robots kinematics is mostly relevant locally, i.e. for short distances / short time horizons. Odometry and path planning are required for larger distances and longer time horizons.
- Denavit-Hartenberg (DH) parameters are a common convention to describe kinematic chains, e.g. serial manipulators. DH requires only 4 parameters instead of the generic 6 transformation parameters in 3D space. This is achieved by standardizing joint coordinate frames.
- The Unified Robot Description Format (URDF) and similar description schemata play an important practical role in modelling robots and their kinematics.
- There are two principal approaches to compute IK. The analytic approach either uses geometry (geometric approach) or algebra (algebraic approach) to find closed-form IK solutions. The numerical approach uses a given FK function to calculate IK solutions by means of numerical root-finding algorithms such as Newton's method.

Notes

1. Nevertheless, if you are interested in a text book definition, this is a concise official one from ISO 8373:2012: robot: actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks.

2. If you come up with the equivalent of the universal Turing machine (UTM) for robots, i.e. a universal physical machine (UPM), please let me know, and expect to become a very famous (and wealthy) roboticist.

3. In case you are not already a Python programmer, but familiar with a C-family programming languages such as C, C++, C#, Perl, Java, JavaScript, PHP, Go, Swift, Rust or others, then you should find it easy to quickly learn enough Python for this book. The large majority of robot software stacks today are actually not written in Python, but Python is a great language to teach robotics and is also seeing increased adoption in real-world robots.

4. This may be funny, but it is no joke.

5. http://ros.org; ROS2 to be more precise, but more about that later on.

6. http://gazebosim.org

7. Most commercially deployed robot software is proprietary.

8. In order to create the data output, complex sensors usually run sophisticated processing algorithms implemented in firmware running on the device itself. We will treat this as an implementation secret of the sensor hardware. Given that a sensor user usually does not have access to the sensor firmware anyway, this is a suitable approach.

9. For many sensor types noise can be modelled as the normal distribution, also known as the bell curve, around the real value. This basically means that most measured values are close to the real value and significant outliers are very very rare. More on this topic in a later chapter.

10. Calculating the robot pose from joint angles is known as forward kinematics and will be discussed in chapter 5.

11. Section 2.2.2 will introduce this type of robot.

12. Learning robotics is not very different from learning a programming language such as Python. You cannot claim to be a Python programmer if you don't know any Python libraries, but you are also not expected to know all or even a large percentage of existing Python libraries. This is a good thing, as of this writing, PyPI alone lists more than 325k libraries.

13. Section 2.2 in the next chapter will describe these and other robot types in more detail.

14. The Robot Operating System (ROS2) framework will be introduced in chapter 7. The Gazebo robot simulator in chapter 8.

15. The term API is used in today's very wide sense of an API being a software interface offering functionality to other software.

16. I'm emphasizing this because there is a common misconception that one must be familiar with embedded software programming, i.e. with reading IC data sheets and manipulating bits in hardware registers, in order to write robot software. This is not the case. Unless you are working on the software layer *directly* interfacing the hardware, i.e. writing drivers. But this is just one of many layers in the robotics stack. More on this topic in chapter 3.

17. In terms of units, industrial robots have been far surpassed by household robots in recent years, such as robotic vacuum cleaners. Depending on what level of autonomy we set as criteria to count a car as a robot car, i.e. an autonomous vehicle, these already have surpassed both types or will surpass them soon.

18. The question of what motions can be performed by different manipulator types and how to perform the necessary calculations is discussed under the heading of kinematics and will be the subject of chapter 5.

19. In practice one would move all motors concurrently. However, this still would *not* result in a straight line.

20. Eventually, some robot needs to go to the fridge to fetch milk and bring it to the stationary pancake robot at the stove.

21. *Discrete* objects, because transportation of liquids and gases through pipelines does not fit here. However, once we consider liquids or gases when put into discrete containers, e.g. tank vessels and fuel trucks, we are back to discrete objects again.

22. The same is also true for stationary robot manipulators. Just compare all the tasks where objects are handled in a bounded volume, e.g. by humans, compared to actual robot deployments. However, for all robot types, there is not only the huge potential, but also the many challenges that still have to be solved for each application.

23. Not that I have anything against entertaining robots.

24. In this book, the word autonomous is used in contrast to manually controlled. While an interesting discussion itself, I will not pose or discuss the question about the nature of autonomy and whether machines can be autonomous in the philosophical meaning of the term. A classical car that a human driver needs to control at any point in time through accelerating, braking and steering has zero autonomy in this definition. A car that can stay in lane without requiring manual steering is seen as partially autonomous. A car in which the driver - or rather passenger? - only has to specify the destination, but does not have to do anything else, would be considered fully autonomous. This kind of highly or fully autonomous car also meets our criteria to be regarded as a mobile robot.

25. According to the view taken here, robotic vacuums and robot lawn mowers are considered payload carrying robots with the vacuum and mower being the payload. One could argue that the robot controls the vacuum and mower by turning them on and off and thus they are actuators. However, I'd argue that control over these components is rather peripheral to the robot system. If control was removed from the robot system entirely, for example by having the vacuum component turn on as soon as the robot is powered up with no control from the acting part whatsoever, the robot would still function as intended. Such a separation of an actuator from the robot system would not result in an operational robot system for true actuators, such as the differential drive. Therefore, it makes sense to consider the robot vacuum as a mobile robot with a vacuum as payload.

26. See figure 2.6 above.

27. Digital outputs will be discussed in 2.5.1

28. The smallest detectable voltage difference for the ADC described here is 1 / 256 * 5 V = 0.0195 V.

29. Whether this code would actually result in the robot following a light or driving around aimlessly depends on many details such as the speed of turning. The code snippet serves its purpose to exemplifying analog inputs, but, as all snippets in this chapter, is definitively not a production ready solution. We will work our way up to the latter step by step.

30. Often you will find frame rate not expressed as frames per second (FPS), but simply as Hertz (Hz). Hertz is the more universal unit of frequency, i.e. cycles per second. 1 FPS is equal to 1 Hz as long as it is clear one is talking about frame rate.

31. It is worth mentioning that the human visual perception system works quite differently from camera sensors. Therefore, the notion of frame rate is not directly applicable to human eyesight. The same is true for resolution.

32. Squinting might do the trick.

33. As our eyes' anatomy obeys the same laws of optics, the image on our retina is also upside down and reversed - it is a point reflection of the environment. It is also inverted back by our brain circuitry.

34. See section 2.3.1 on digital inputs.

35. Just in case your are interested: My strong opinion in this debate is that code comments should always be a last resort. Readability should primarily result from meaningful names of the programming language constructs and their structure, such as types, objects, variables and functions.

36. See figure 2.16(b) for an illustration.

37. Be assured that in later chapters, we will address such details that can be quite tricky to deal with in practice.

38. A valid *approach*, not a valid implementation for production. Among other things, the code lacks proper error handling such as bounds checking.

39. What we have just referred to as sensitivity is closely related to the slope of the sensor's measurement-to-output function. This might look more familiar to what you learned in school math, when you tilt the book (or tilt your head) so that the distance axis points upward. We will come back to and make use of this mathematical insight in later chapters.

40. A statement like this cannot be made to an international audience without a short qualifier: This book only uses units defined by the International System of Units, SI units, together with derived SI units. Therefore, length is measured in meters, weight is measured in kilogram, current in ampere and time in seconds, etc. Imperial units are also defined in SI units, but they will not be used in this book.

41. It also serves to prove the point that classification does often not make use of sophisticated algorithms, but can be something as simply as the mundane `if` statement.

42. If you hear about a *robot* zoo opening up somewhere in the world, please let me know!

43. The reason being the this is *obviously* an elephant fallacy described in our visit to the zoo above.

44. Please note that this is a naive implementation. Although it is correct, it is not very efficient computationally. When it comes to the basic building blocks of computer vision algorithms, it is highly advisable to *not* implement them yourself. Instead, you should rely on computer vision libraries that provide you with optimized implementations. You bring the understanding which method to apply to a problem, the library brings the implementation of these methods. I will not get tired to repeat this over and over in Chapter 9.

45. Chapter 12 will be the place to go into more details about computer vision.

46. I make the assumption that source code is a more familiar notation to most readers, given they book's title is Robotics for Programmers not Robotics for Mathematicians.

47. You will learn more about practical *electrical* considerations later on, such as voltage ranges, push-pull drivers, pull-ups, pull-downs, tri-state logic, current limitations, etc. Strictly speaking these topics fall into the hardware domain, but it is still a significant advantage for robotic Software Engineers to have some background knowledge on them.

48. We will make use of orders of magnitude multiplies, i.e. SI prefixes. For example, instead of specifying a voltage as 2,000 V, we can would use the kilo prefix and write it as 2 kV. For a current of 0,004 A, we would use the milli suffix and write 4 mA.

49. A story quite similar to solid state disks (SSDs) replacing more and more electro-mechanical hard disk drives (HDDs).

50. Similar to our discussion on analog inputs and ADCs in section 2.3.2, the output does not have arbitrary resolution, rather it has only a number of discrete voltage levels. The resolution for DACs is stated in number of bits, same as for ADCs. An 8 bit DAC can output 2 = 256 different voltages that are equally spaced over its operating range. In our abstract API we encapsulate these details behind the floating point value normalized to a range of 0.0 to 1.0.

51. As discussed in the previous section (2.5.1).

52. Once we have learned about pulse-width modulation (PWM), we could use PWM through a digital output as control signal. Today this is actually the more common solution compared to using an analog signal. Still, digital outputs and PWM are not the same thing. Not all digital outputs can be used as PWM outputs, due to the relative high toggling frequencies and timing requirements for proper PWM control.

53. So called outrunner motors are built the other way around. Their stator is inside the rotor, which rotates around it on the outside. They are common in flying robots, e.g. quadcopters.

54. As long it can be guaranteed that the motor’s torque is always sufficient in the application, stepper motors can be run without feedback in actual use. However, if this cannot be guaranteed with confidence, the lack of feedback means that the actual motor position can end up different from the intended one without there being a possibility to detect the issue. In context of stepper motors, this issue is commonly referred to as loosing steps or missing steps. This the reason why there are also stepper motors with position encoders available. The more general topic here is open-loop control in contrast to closed-loop control - something we will come back to in later chapters.

55. Refer back to figure 2.5 for an illustration.

56. Recent high-end cordless drills often use BLDC motors instead of DC motors. By the end of this section, you will understand both why BLDC motors provide improved performance and durability compared to DC motors and also why tools using them are still more expensive and more widely available only since a few years.

57. There is a fourth state: Turning on both transistors at the same time will create a short circuit between positive and negative supply voltage. As this will most likely result in permanently damaged electronics, software must ensure that this state never occurs - not even briefly.

58. In practice we need six PWM outputs, i.e. digital outputs that can be toggled rapidly with high temporal accuracy. More about PWM in section 2.6.

59. So called sensorless control of BLDC motors is possible and provides sufficient control quality for certain use cases. However, the term is a bit misleading as sensors that measure electric quantities are still used, just no additional sensors in or on the motor. The most widely used method of sensorless control measures the back electromotive force (back-EMF) produced by the motor during operation.

60. Refer back to figure 2.1 and 2.2 in section 2.1 for a visual reminder.

61. Going back to our toast example, we neither want cold toast (toaster turned off too early) nor charcoal toast (toaster turned off too late). We want it nicely toasted (toaster turned off at the correct time).

62. PWM frequencies in real system are often on the order of thousands of Hz and thus stated in kHz.

63. Furthermore, both cameras and human eyes perceive flickering lights above a certain frequency not as flickering, but as having a constant average brightness.

64. We can also express this relation by saying that the component's output is the low-pass filtered PWM signal. We will return to this signal and filter terminology in later chapters.

65. The property can also be in the robot system itself instead of the environment, e.g. a motor position.

66. The operating system layer is frequently called Hardware Abstraction Layer (HAL).

67. Instead of talking about more complex robot systems, one could also talk about more advanced robot systems or even modern robot systems.

68. With well-defined places I don't mean that the eggs are always in the fridge and the flour is always stored in the top left cupboard, but that they have an *exact* position. To ensure this we would have to add fixtures for each item, including the consumables. In other words, while this solution is simple when it comes to programming our robot, it might not be an easy overall solution, given the high effort to fix everything in place.

69. The opposite here is not the use of analog computers instead of digital computers, but not using computers at all. A class of entirely analog robots are the so called BEAM robots. BEAM robots rely only on analog electric circuits. While a fascinating subject in itself, they have very limited capabilities (and very limited practical use). More importantly for this book, they don't include software and are thus outside our scope.

70. In this context a computer can be a powerful desktop processor, an embedded system-on-chip (SoC) or a low-power microcontroller.

71. We could also see the robot fleet, or robot swarm, as a (higher level) robot system. However, this would just create confusion and thus we refrain from system recursion here.

72. An arrow between two boxes can mean many different things in software architectural diagrams. It could describe the dataflow between components, but it could equally well describe a (build) dependency. It could also describe composition, inheritance, a physical connection or a myriad of other things.

73. Referring back to section 1.3 and figure 1.3, you have already seen this idea of hierarchical diagrams in action.

74. Beyond their main CPU, modern general-purpose computers usually contain a variety of additional chips that run embedded software and interface with the main CPU through internal communication buses. Examples are the platform chipset (BIOS or UEFI), network interface controllers (NICs), disk controllers and graphics cards.

75. The time is either specified as a deadline the software must react within X milliseconds to event A or as a period (plus acceptable jitter) the control loop must adjust the motor current every X microseconds (+- Y%).

76. We make the assumption that the deadline or periods must *always* be met within defined limits of accuracy. This type of system is also known as a *hard* real-time system.

77. The ECUs in a car are hence also an example of a distributed system. See the next subsection on this aspect.

78. The safety-critical aspect of software controlling brakes or other software that can cause harm when not working properly will be discussed in chapter 17.

79. Using PyPy it is still 10x slower.

80. There are two common workarounds for the comparatively slow speed of executing Python primitives. Both rely on compiling (parts of) the Python code. Either the entire Python program is (just-in-time) compiled or only the libraries providing computationally expensive routines are compiled while the main program is still interpreted.

81. MicroPython is a Python implementation for microcontrollers that addresses some of the challenges mentioned, especially memory usage and performance. However, it cannot solve issues inherent in an interpreted dynamic language that uses a garbage collector either.

82. While this statement is true for batch processing systems, it is an oversimplified statement for interactive systems. Given some app with a GUI, the user will not care whether the app reacts to inputs in 10 ms or 100 ms. However, the users will be very annoyed if the app reacts within 10 ms 19 out of 20 times, but takes two seconds to react on every 20th click, although the average is still an acceptable 60 ms.

83. My recommendation for embedded real-time software is modern C++ (C++14 or newer). Rust is emerging as an interesting alternative.

84. Looking at the machine code doesn’t help either.

85. The exception would be safety-critical parts of the systems where we need to dig deeper and give real guarantees.

86. Priority has nothing to do with importance here. We are not saying task A is more important than task B. By assigning a higher priority to task A, we are saying that whenever both task A and task B could run, we want the scheduler to run task A.

87. See also the discussion on interrupt handling below.

88. This definition is based on the one given in Distributed Systems by M. van Steen and A. Tanenbaum.

89. Think back to our discussion on this topic in the previous chapter.

90. While it is clear that the list originated at Sun Microsystems, the identity of the authors is less well known. See Fallacies of distributed computing in the English Wikipedia for more information. The list is also reproduced in the already mentioned book by van Steen and Tanenbaum.

91. All program operation take time, of course. However, compared to passing data to a function within a program, packaging data into a message and sending it over a network is several orders of magnitude slower. Hence, while it is ok for practical purposes to think of function calls as immediate one must nearly always consider duration when it comes to networks.

92. See chapter 17 for more information on quality characteristics and software engineering in robotics.

93. Physical layer as in layer 1 of the OSI model, i.e. the transmission of bits over a physical medium.

94. Topics are also known as channels or subjects.

95. This is clearly a simplified description. For example, in real HDDs there is not a single platter, but a stack of several platters along with multiple arms and heads. Furthermore, in recent HDDs, the arm positioning described here is still there, but in addition the arm has a second fine positioning stage based on piezoelectric actuators.

96. This is the same type of actuator found in most loudspeakers - hence the name. Like electric motors the basic principle is the Lorentz force, or electromagnetic force, caused by electric current flowing through a cable or a coil in presence of a magnetic field. In contrast to electric motors, voice coil actuators have only a limited range of motion. Their advantage is highly dynamic and precise actuation.

97. In general, in mechanics precision equals cost. Manufacturing parts to higher precision, i.e. lower tolerances, is simply more complex and hence more expensive.

98. So how does positioning work in HDDs? In addition to the data, the platter also contains read-only position information, layed out as a magnetized rays pattern along the platter. This position information is read by the head and used as a position encoder (cf. our discussion on position encoders in chapter 2.3.3). The position information is the feedback needed to create a closed-loop control system (cf. 2.6.2) for fast and high precision arm positioning.

99. S. Legg, M. Hutter: Universal Intelligence: A Definition of Machine Intelligence.

100. This section draws a lot of inspiration from R. Pfeifer and J. Bongard: how the body shapes the way we think.

101. Please note that although this view has become associated to the ambivalent term "Good Old-Fashioned Artificial Intelligence" (GOFAI), it is not at all superseded by the embodied view. It had just been the mainstream view before the new one came to prominence.

102. If this would be a book about embodied intelligence, you would read the term agent here instead of robot. For our purposes these are synonymous.

103. Deterministic behavior of software is what enables us to write (unit) tests. Tests verify the correct and reproducible behavior of our software. We verify the correctness of software under test by verifying that it always produces the same output when given the same input after it has been brought into the same state.

104. The same goes for the intelligence of human beings, actually for every living organism and every system with a goal for that matter. Many accidents are caused by humans or systems whose actions are based on symbols referring to the wrong thing and hence actions leading to unintended results.

105. This is quite different from all *imaginable* situations or environments. However, when it comes to safety, we basically have to consider all reasonably imaginable ones. More on the latter in chapter 17.

106. Try picking up hard objects with two fingers and compare your success to picking up the same objects with chopsticks or barbecue tongs.

107. You can read perception as equivalent to sensor usage and sensing here.

108. We will take a closer look at system identification in chapter 6 and at calibration in chapter 7 and 12.

109. Having a number of robots coordinated by a central instance is rather a robot fleet with the coordinator being the fleet manager, cf. figure 3.4.

110. Borrowing the terminology from statically typed languages, such as C++ or Java, we can say that an AER is a template (C+) or generic type (Java) and that specifying more details corresponds to instantiating (or specializing) the C\+ template respectively to parameterizing the generic Java type.

111. Refer back to chapter 2.2 for a recap.

112. Python always means Python 3.

113. Of course it is a good thing to have a firm grasp of geometry and related subjects. However, for what you are about to learn, it does not matter if you don’t remember Pythagoras theorem or how to calculate the hypotenuse without consulting Wikipedia. I will introduce the concepts needed when we need them. Furthermore, you can refer to appendix B.1 for a minimal math refresh.

114. If `move()` and `navigate()` don’t sound familiar, please refer to section 2.2.

115. The concept “pose” was introduced in section 2.2.1.

116. Throughout the book we follow the convention of specifying 2D positions in the order (x, y) and 3D positions as (x, y, z).

117. We will discuss more about shapes at the end of section 4.4.

118. Independent coordinates means that we are interested in the minimum number of complete position/orientation values. Some representations, e.g. rotation matrices, contain more numbers than this minimum. Dimension and degree of freedom are widely used terms, albeit with different meanings in different contexts and for different people. Pose parameters is not a widely used term, but one that I find very useful to reduce ambiguity - and I hope you will find it so, too.

119. Assuming the object has a distinguishable orientation or, put differently, assuming the object is not rotationally symmetric.

120. This case was used as illustration for the concept of pose in chapter 2.2.1, cf. figure 2.6.

121. When you encounter the notion of "6D (pose)" in robotics, this most likely refers to the pose of a 3D object in 3D space with position and orientation taken into account. I will *not* use the term in this book, but stick to the outlined terminology of referring to the same fact as 6 PP (pose). To demonstrate why things can get very confusing when not following the clear distinction between dimension, pose parameters and degrees of freedom, consider a ground-based mobile robot (on flat terrain). It can be described as a 2D object in a 2D space, it has two position values and one rotation value, i.e. 3 PP. Depending on actuation, it can have two degrees of freedom (2 DoF), e.g. a differential drive, 3 DoF (three omnidirectional wheels), 4 DoF (four Mecanum wheels) or 8 DoF (four active steerable wheels). In either case, the robot pose is 2D and has 3 PP. Imagine the confusion, already in this simple example, if we use dimension, PP and DoF incoherently.

122. You will get to know these and other drive types in later chapters.

123. We will use the term frame synonymously with the term coordinate system. When discussing image processing and coordinate systems at the same time, the term frame is potentially ambiguous. However, in practice the context makes it clear whether we are talking about coordinate systems or images.

124. Derived from "Right hand rule cross product.svg" by Acdx, CC BY-SA 3.0.

125. Refer back to 2.2. We will go in much more detail in chapter 13.

126. This convention is based on ROS (REP 103) and aircraft principal axes. It is equivalent to Z-Y-X Euler angles: $R_{ZYX}(, ,) = R_Z() \cdot R_Y() \cdot R_X()$.

127. You can find the conversion formulas in appendix B.1.

128. The gripper is seemingly floating in the air as the robot arm that it is attached to is not shown.

129. I assume you are familiar with the basic concept and terminology of graphs, such as vertices/nodes, edges/links, adjacency, undirected, directed, cyclic and acyclic. The same goes for trees, i.e. connected acyclic unidirected graphs.

130. Coordinate transformation trees are also known under the term “scene graphs” in computer graphics. Other common names are “transform graphs” and “pose networks”. Moreover, due to the influence of ROS, many people today refer to a coordinate transformation tree as “TF tree” even when not using ROS. More on the latter in the next chapter.

131. Saying one has become used to it doesn’t mean that it isn’t still a major source of errors and severe bugs.

132. These relations are identical to the relations between a time point and a duration. There are 2 hours (a duration) between 13:00 and 15:00 (two time points subtracted from each other) and 4 hours minus 1.5 hours equals 2.5 hours, but 13:00 plus 15:00 is not meaningful (and should not be confused with 13:00 plus 15 h, which is 04:00).

133. See section 4.5 for some practical advice on this topic.

134. You will learn about two additional vector operations, dot product (aka scalar product) and cross product (aka vector product), when we make use of them for the first time.

135. The code below is intentionally not very Pythonic, but rather aims for verbosity.

136. More on matrices soon.

137. The indices in mathematical notation start with index 1, not with index 0 as in most programming languages, incl. Python.

138. Also referred to as free vectors.

139. Also referred to as a bound vector.

140. In this context the absolute position refers to the position in a coordinate system external to the object.

141. Just like a vector is a one dimensional array from this perspective.

142. This is quite similar to how a 2D image is represented as a two dimensional array, cf. chapter 2.3.4.

143. Remember that the index numbers starts with 1 instead of 0 in mathematical notation.

144. While I encourage you to learn more about trigonometry and (applied) linear algebra, neither is a precondition for reading this book or starting out in robotics.

145. Calculating the inverse of a matrix is a key operation nevertheless. Thus, you can expect that every major library that provides matrices also provides matrix inversion operations. In numpy it is `numpy.linalg.inv()`.

146. See for yourself by doing the exercises below.

147. Refer back to section 4.1 if this does not sound familiar.

148. The same is true for positions and translations as well as for orientations and rotations, because they are special cases of poses and transformations.

149. This statement also holds true, if you use mathematical notation instead of (Python) code.

150. In transformations.py use `inverse_matrix()` (instead of the general `numpy.linalg.inv()`).

151. In chapter 24, we will briefly talk about how augmented reality (AR) and virtual reality (VR) can support us.

152. Expressed in mathematical terms, in 3D, translations are commutative, but rotations are not.

153. REP 103. We will come back to it in the next chapter.

154. Please note that this is equivalent to working with mobile axis and reversing the order of operations: First yaw around z axis, then pitch around *moved* y axis, then roll around *moved* x axis.

155. See appendix B.1 for the full formula. An example of such a Python library, transformations.py, is used for exercises below.

156. In your continued robotics journey, it is beneficial to incrementally build up more knowledge about these subjects. However, you can really use them as a black box with a defined API. Most software engineers using compilers, operating systems, databases or processors for that matter are not experts in any of these topics either, but succeed in using these as building blocks in their applications. Feel confident to adopt the same stance towards some of the more involved mathematics we touch upon here. (The majority of *successful* roboticists would actually struggle with the details as well, if you put them on the spot.)

157. The minimal number to describe arbitrary 3D rotations being three.

158. You can still keep them in one data structure / class and define operators for them to hide this fact. There are also *dual quaternions* that can express position/translation and orientation/rotation together, just like homogenous coordinates. However, I rarely see the latter used in practice.

159. It is no accident that this actually describes the ROS(2) geometry_msgs/Pose type, as we will discuss in chapter 8.

160. See vectors above (4.4.1).

161. You can see this as an abstract API, but it is also the specific API of transformations.py.

162. Note that we import `tf_transformations` here, instead of `transformations`. Although both refer to transformations.py, `tf_transformations` is the version included in ROS2 and `transformations` is the PyPI version. There are - unfortunately - some subtle differences between them, e.g. the internal representation of quaternions. We will only use the ROS2 version starting in chapter 8. All exercises in this and the next chapters work with both versions.

163. Note that this is not an efficient solution for drawing circles in computer graphics. See the midpoint circle algorithm, also known as Bresenham’s circle drawing algorithm, as an example for a more efficient approach.

164. “Boolean_union.PNG”, “Boolean_difference.PNG” and “Boolean_intersect.PNG” by Captain Sprite, CC BY-SA 3.0.

165. In general meshes can also consist of polygons with more than three edges (n-gons), such as four sides polygons (quads).

166. The same is true for robotics in general.

167. In ROS(2) this is the tf (resp. tf2) package.

168. Refer back to figure 4.8.

169. I assume you already know how to install Python packages. If not, please read up on it before running `pip install numpy transformations`.

170. See https://numpy.org/ and https://pypi.org/project/transformations/. More detailed documentation for transformations can be found directly in the source file transformations.py.

171. You have already learned everything about how to describe a path in the previous chapter. In the most simple way, you can describe a path as an ordered list of poses.

172. If thinking about a change in pose, i.e. translation and/or rotation, does not (yet) feel intuitive to you, just think “position” here when the more general “pose” is used.

173. Another frequently used symbol for position is **x**, which might have nothing to do with the x axis.

174. Dividing by a scalar s can be expressed as multiplication with the inverse of the scalar 1/s, i.e. it is scalar multiplication.

175. Scalar linear velocity is often simply called speed. However, speed and velocity are often also used synonymously. Thus, I prefer to make it explicit in language and symbols whether we are talking about a spatial quantity or about a scalar quantity.

176. The idiom “to go around in circles” basically summarizes this well. An orientation has a period of 360° (= 2 radians), i.e. 0° and 360° are the same orientation. Any orientation value below 0° and above 360° has a corresponding one in the range 0° to 360°.

177. Units are not mentioned explicitly for readability reasons. Time is measured in seconds [s] and position in meters [m]. Thus velocity has the unit meters per second [m/s].

178. The time interval is written in interval notation here - not to be confused with vector notation. t = [3, 5] simply means the interval between 3 and 5. More specifically the square brackets mean that both 3 and 5 are themselves part of the interval.

179. Whether the approximation is “good enough” depends on many factors, such as mathematical properties of the function and also application demands. Numerical analysis can provide quantitative answers to this question.

180. You can find more in appendix B.1. Please note that the symbol x here has nothing to do with position.

181. We will discuss a similar topic more detailed in section 5.9.

182. Sticking to (derived) SI units, the frequently needed conversion from meters per second [m/s] to kilometers per hour [km/h] is 1 m/s = (1 m/s) · (3600 s/h) / (1000 m/km) = 3.6 km/h.

183. The introductory tutorials available on https://matplotlib.org are a good starting point.

184. Depending on what part of the robot software stack you are working on, knowing these details is either nice to have or crucial for your success. If your focus is motion control, you should - over time - deepen your calculus skills.

185. Similar to the inverse relationship between addition and subtraction or the relation between multiplication and division.

186. We already used the same segment in figure 5.5 above to derive velocity from position.

187. This corresponds to segment t = [15, 45] in figure 5.9. To avoid unnecessary complexity in the description here, we treat it is as its own segment between $t_1 = 0$ and $t_2 = 30$.

188. There are functions that are not integrable or only locally integrable. However, this goes beyond the short introduction here. If we encounter one of these and we have to apply calculus to them, I’ll give you a heads up - same as with non-differentiable functions. Nevertheless, I want to warn you that you can easily calculate a numerical integral (derivative) for a function that is non-integrable (non-differentiable), you will get a value, but obviously it will be wrong as no correct one exists.

189. Please refer back to figure 5.7 and the surrounding description, if you do not yet see the connection.

190. Translating formulas to words and the other way around is often very helpful in robotics. Because the formulas often - but not always - have a straightforward meaning in the real world, which allows us to cross-check them against our world knowledge.

191. For example, `scipy.integrate` for numerical integration and `sympy.integrate` for symbolic integration.

192. Array here refers to any array-like data structure that stores values ordered according to some index or key, e.g. tuples, lists or NumPy arrays.

193. Depending on the robot type and its application, a more realistic temporal resolution than t = 1 s would be 0.1 s (= 100 ms) or 0.01 s (= 10 ms) or even 0.001 s (= 1 ms).

194. If there are fast changes in the robot's velocity, there will be significant errors in the calculated velocity using t = 0.1. This can be mitigated by filtering the calculated velocity signal or interpolating between the position values and calculation with smaller t on the interpolated values. We will discuss filters and interpolation in later chapters.

195. If you want to confirm what you learned before continuing learning new things, I have provided a few small exercises in section 5.10.

196. See section 2.2.1.

197. Due to joint limits and self-collisions there are often less than eight IK solutions. Self-collision means that some of robot's links collide with each other for a set of joint values. Thus, the robot cannot actually move to an IK solution that is a self-collision, even though the IK solution is within joint limits.

198. Why *eight* configurations? The typical 6 DoF manipulator kinematics exhibit the following alternative ways in reaching the same pose: The elbow of the robot can be up or down (2 possibilities), its lower arm can be rotated towards or away from the pose (2 possibilities) and axis 5 can be flipped (change sign) with the other wrist axis compensating for it (2 possibilities). Because these alternatives are independent, possibilities combine in a multiplicative way, resulting in $2 \cdot 2 \cdot 2 = 8$ possible configurations.

199. Another term for changing configurations is re-configuration.

200. In more general terms, it can be a group of joints canceling out the motion of another group of joints.

201. It becomes singular for a 6 DoF robot in 3D space (6 PP).

202. At this point we do not yet take the robot's environment, esp. obstacles, into account. An IK solution colliding with an obstacle in the environment is hence still considered a valid IK solution. We will later on also pay attention to obstacles under the heading of collision-free path planning. The robotics literature is divided on whether IK solutions causing a self-collision (see earlier footnote) are considered valid IK solutions, albeit not actually usable ones, or whether they are considered invalid, i.e. not considered solutions. I follow the convention to only consider sets of joints values as valid IK solutions that a robot can actually reach.

203. The work envelope commonly only takes the position into account, but not the orientation. Hence the term "points" here. In other words, if a point can be reached from any direction, then it is deemed part of the workspace. In practice, it is often not enough to just reach a point, instead one wants to reach the point from multiple directions, too. Manipulability is a measure on the reachability taking orientation into account. We will discuss manipulability and the so called dexterous workspace in a later chapter. Thus, the *useful* volume for a robot application is often smaller than the complete work volume.

204. This way of visualizing the workspace is defined by the standard ISO 9946 "Manipulating industrial robots - Presentation of characteristics".

205. Refer to section 4.1 for the distinction between degrees of freedom (DoF), pose parameters (PP) and dimensions.

206. You can also think of it the other way around: Each orientation can only be reached at certain positions in the workspace. However, most people find it easier to think of position first and orientation second.

207. As shown above, some poses can be reached with multiple set of joint values. Some cannot be reached due to self-collisions of the robot links with each other or due to joint limits.

208. A more mathematically oriented description and definition of non-holonomic robots focuses on the difference between constraints on velocity compared to constraints on position. If one can substitute all constraints on velocity by constraints on pose, the robot is holonomic. If this is not possible, the robot is non-holonomic. In case of a car, it can reach any pose in 2D (given there are no obstacles), but it cannot drive sideways (a constraint on its velocity). There is no constraint on pose, but one on velocity. Hence, it is non-holonomic.

209. The typical rack-and-pinion plus tie rod linkage that mechanically realize the Ackermann steering is not shown. This figure simplifies it into a 1 DoF rotational joint.

210. See section 4.3.

211. We disregard the robot's location is located with respect to a coordinate system external to the robot. This would require an additional 3D pose in both cases, i.e. 6 PP, which represents the transformation from the external coordinate system to the robot's base link.

212. The numbers in the example also apply for other kinds of joints that only allow a single motion, such as linear/prismatic joints. You have to adjust it for joints that allow multiple motions, i.e. that need more than one joint value to describe them, such as ball joints.

213. By convention the robot coordinate system is a fixed coordinate system attached to the robot's base.

214. When talking about "previous" or "next" in this context, we regard the serial kinematic chain from base link to end effector. Things (joints, links) that are closer to the robot's base relative to the current thing are considered "previous". Things closer to the end effector are considered "next". Another type of common terminology is borrowed from transformation trees. For example, one speaks of "ancestor joints", "parent link" or "child link", just as one would speak of ancestor, parent and child nodes in a tree (graph).

215. We do not need to split between joint to next link and link to next joint because the kinematics in the example are a kinematic chain and we only care about the end effector pose.

216. In this notation, j_i is used to denote the joint j_i itself as well as its joint value (angle). It should be clear from the context whether we mean the joint or its value.

217. NumPy uses the symbol `@` for matrix multiplication, cf. section 4.4.2.

218. The letters R and P stand for rotary and prismatic respectively. The first 2 DoF example robot had a kinematic chain consisting of a rotary joint followed by a rotary joint, hence RR. The second one consists of one prismatic joint followed by a rotary joint, hence PR. This is a common shorthand to describe different robot manipulator kinematics. For example, a 6 DoF articulated robot consisting only of rotary joints is also known as 6R. SCARA robots consist of two rotary joints followed by a prismatic joint and their kinematic is known as RRP. Finally, a Cartesian gantry robot consists of three orthogonal prismatic joints and is thus a PPP or 3P kinematic.

219. For now we ignore real world mechanical concerns such as backlash and stiffness - but we will discuss them later on.

220. Velocity kinematics are also known as instantaneous kinematics. Futhermore, they also play an important role for manipulators, not only for mobile robots. However, be aware that the term "velocity kinematics" usually has a much more specific meaning in the context of manipulators than it in the context of mobile robots.

221. The mobile robots joints are usually closed-loop controlled using the joint sensors, just as for manipulators. Although it might be a velocity control instead of a position control. The big difference is not on the level of this control loop, but the need for an additional one, cf. chapter 2.

222. The terms are often used interchangeably when talking about mobile robots. However, there is a subtle difference that guides the terminology for this book as defined below.

223. The conversion from joint rotation velocity of the wheel joint j_i, let’s call j_i', to the linear velocity v_i along the ground is $v_i = s \cdot j_i'$ with s being the wheel’s size expressed as its radius.

224. Please note that we follow the convention to write the angular velocities before the linear velocities in twists. While 2D poses are written as (x, y,), a 2D twist is written as (', x', y'). This has no deeper meaning. It is just to stay closer to the dominant notation in robotics.

225. We will discuss in later chapters how we can fuse this internal sensor information together with additional external measurements and achieve higher accuracy than with either alone.

226. This unified language of robotics - terminology, notation and conventions - does not exist today. Although one can trace everything about robot mechanisms back to first principles, e.g. physics, this still doesn’t address the need for abstractions that are useful for each kind of robot. The reason for the robotics field seeming in disarray is its wide range of applications. Neither naval engineering, nor automotive, nor aerospace nor other fields with a century long history will adopt conventions from robotics just because of unmanned surface vehicles, autonomous cars and unmanned aerial vehicles coming into existence. Even though all these man-made objects clearly are robots according to our definition.

227. In a similar manner to how we related the diff drive wheel speed to the 2D rigid body kinematics, we could relate the linear rotor velocity to the velocity of the quadrotor frame. Yet, this is not done in practice in the velocity domain as it is practically meaningless. The ground robot’s joint rotation has a known direct relation to the linear wheel speed. This is at least true when moving without a lot of slippage, which is a common case for many types of ground-based mobile robots. For aerial robots this is not true to the same extent. Hence, the focus for flying robots rests much more on dynamics (forces, moments and thrust) than on kinematics (velocities).

228. These torques are reaction forces or more specifically reaction torques. This phenomenon is described in Newton’s third law of motion in classical mechanics “If two bodies exert forces on each other, these forces have the same magnitude but opposite directions.”. We will come back to this topic in the next chapter.

229. Gravity always acts in the direction of the negative z axis in world coordinates, $-^wz$. When the robot is not moving relative to the world coordinate system, i.e. when hovering, the z axis of the body coordinate system, ^bz, points in the same direction as ^wz. In other flight states, moving along ^b^z is not the same as moving upward (^wz).

230. Altitude can either be used to denote height above mean sea level or height above local ground level. Obviously, it is highly important to be clear which definition is used - consider flying over mountains. We can safely ignore these complexities for most of the book and will simply refer to the ^wz value.

231. Cf. sections 4.4.3 and 4.5.

232. $\tan^{-1}(y/x)$, more specifically `atan(y / x)`, only returns angular values in range [-90°, 90°], requiring us to manually handle the case when x < 0 by adding a correction offset of 180° to the returned angle. We would actually use `atan2(y, x)`, which returns the correct value in the range of [-180°, 180°].

233. See figure 5.31 above or figure 5.39 below.

234. For specific kinematics it is possible to separate the joints into groups that can be solved sequentially. However, within each group one still needs to holistically calculate a solution for all joints.

235. The so called domain of the function, i.e. the set of all valid inputs to the function.

236. In case of our IK solution for the 2 DoF RR robot, we only need to check for joint limits as all unreachable positions will result in invalid formules, e.g. dividing by zero or passing a number outside of arccos domain [-1, 1] to it.

237. As stated in the beginning of the chapter, a trajectory is a time-parametrized path. Hence, path and trajectory are not the same. Paths do not contain temporal information. Still, the two terms are often used synonymously.

238. As visualized in figure 3.2.

239. For legged mobile robots, the topic of kinematics is more related to kinematics of (parallel) manipulators than other mobile robots. We will discuss them separately.

240. Let me repeat that the constraint is on velocities / twists, not on the pose. In an obstacle-free environment, there always exists a path that allows the diff drive robot to reach any target pose starting from any starting pose.

241. Here, body frame, denoted by superscript b, and robot frame, denoted by superscript r, are used synonymously. Both are the same frame attached to the robot chassis.

242. See figure 5.37 and the surrounding description for the arrangement of rotor joints in the body frame.

243. For the driving robot, it holds that $v_i = s \cdot j_i$, independently of the robots pose in world coordinates.

244. There is an excellent 3 min video that describes how to create DH parameters for a given kinematic: https://www.youtube.com/watch?v=rA9tm0gTln8.

245. It is assumed all joints are either rotary (motion axis = axis of rotation) or prismatic (motion axis = axis of translation).

246. The ${}^iy$ axis can be calculated as a cross product of ${}^iz$ and ${}^ix$: ${}^iy = {}^iz \times {}^ix$.

247. You can find their definition in appendix B.1.3

248. You will also already know many other important robotics concepts, which will greatly ease learning and working with ROS - or any other robot software.

249. Refer back to section 4.4.3 regarding the convention used for 3D rotations (Z-Y-X Euler angles).

250. Cf. section 4.4.3.

251. At least for serial kinematics.

252. The link lengths l_i are known constants. Also, x, y and are given parameters.